

# Qualitative Analysis of Knowledge Transfer in Pair Programming



Dissertation zur Erlangung des Grades  
eines Doktors der Naturwissenschaften (Dr. rer. nat.)

am Fachbereich Mathematik und Informatik  
der Freien Universität Berlin

vorgelegt von

*Franz Zieris*

Berlin  
2020

**Gutachter:**

Prof. Dr. Lutz Prechelt, Freie Universität Berlin, Deutschland

Prof. Dr. Paul Ralph, Dalhousie University, Kanada

**Datum der Disputation:** 01. Oktober 2020

## Abstract

Pair programming (PP) is the practice of two developers working closely together on one computer to solve a technical task. It is used by developers in industry in order to tackle difficult problems, to produce code with better design and fewer defects, and to learn together and from another. The transfer and acquisition of knowledge is central to all these expectations, but whether and how they actually come to pass is an open question.

My goal is to understand the mechanisms of knowledge transfer in PP in order to formulate practically relevant results and advice for software developers.

I perform a qualitative analysis based on the Grounded Theory Methodology and the Base Layer for pair programming research. I analyze 27 industrial PP sessions recorded in ten companies: The developers work on their everyday tasks covering different aspects of software development, with whom and for as long they want, totaling 40 hours of material. I performed supporting field observations and ad hoc interviews in two of the companies.

My results are a detailed bottom-up conceptualization of knowledge transfer processes in pair programming, ranging from individual utterances, over knowledge transfer episodes, to overall session dynamics that are shared across different types of pairings. In particular, I find that:

1. Knowledge is transferred in basically all PP sessions, not just in supposed “expert/novice” constellations.
2. Knowledge regarding the software system is by far the most commonly transferred type, and most developers appear familiar with transferring it to or from the partner and to acquire it together.
3. General software development knowledge is also transferred between partners, but less than system knowledge and only after the pair dealt with its system knowledge needs.
4. Additional types of knowledge, such as application domain concepts, were not explicit topics but merely showed up as identifiers in the source code.
5. Pairs that maintain a shared understanding of the system and software development in general may have short, but highly productive focus phases; others may suffer from a breakdown of the pair process when such a shared understanding is lacking. A missing shared plan, reduced workspace awareness, or language barriers further reduce their togetherness.

I validated the high-level concepts with practitioners from four companies—two from the original data collection and two additional—and developed three ideas for how to put my results to use in everyday software development.



## Zusammenfassung

Die Idee der Paarprogrammierung (PP) besteht darin, dass zwei Softwareentwickler gemeinsam an einem Computer an einer technischen Aufgabe arbeiten. In der Berufspraxis wird sie eingesetzt um schwierige Probleme anzugehen, um bessere Programmwürfe und Programmcode mit weniger Defekten zu erzielen, und damit Entwickler Neues gemeinsam oder voneinander lernen können. Der Transfer und die Aneignung von Wissen ist zentral für all diese Erwartungen; ob und wie sie aber tatsächlich erfüllt werden, ist eine offene Forschungsfrage.

Mein Ziel ist es, zunächst zu verstehen wie Wissenstransfer bei der PP tatsächlich funktioniert, um dann praktisch relevant Ergebnisse für Softwareentwickler zu formulieren.

Meine qualitative Analyse basiert auf der Methode der Grounded Theory und der Basisschicht für Paarprogrammierungsforschung. Ich habe 27 industrielle PP-Sitzungen mit einer Gesamtlaufzeit von 40 Stunden analysiert. Das Material stammt aus zehn Firmen, wobei die Paare bei ihren alltäglichen Softwareentwicklungsaufgaben mit selbstgewählten Partnern und selbstbestimmter Sitzungsdauer aufgezeichnet wurden. Unterstützend habe ich in zwei der Firmen Feldbeobachtungen und Ad-Hoc-Interviews durchgeführt.

Das Ergebnis meiner Arbeit ist eine Bottom-Up-Konzeptionalisierung von Wissenstransferprozessen bei der Paarprogrammierung, angefangen bei einzelnen Äußerungen, über Episoden von Wissenstransfer bis hin zu einer Gesamtdynamik, die Sitzungen verschiedener Paarkonstellationen gemein ist. Meine Erkenntnisse im Einzelnen:

1. Wissenstransfer erfolgt in allen PP-Sitzungen, nicht nur in Konstellationen eines vermeintlichen "Experten" mit einem "Neuling".
2. Wissen über das Softwaresystem wird mit Abstand am häufigsten transferiert. Die meisten Entwickler scheinen den Austausch und den gemeinsamen Erwerb dieses Wissens gewohnt zu sein.
3. Wissen über Softwareentwicklung im Allgemeinen wird ebenfalls in PP-Sitzungen transferiert, allerdings in einem geringeren Maße als System-Wissen und auch erst dann, wenn das Paar seinen Bedarf an System-Wissen geregelt hat.
4. Weitere Wissensarten, wie etwa Wissen über die Anwendungsdomäne, waren keine ausdrücklichen Themen in den Sitzungen, sondern traten lediglich in Form von Bezeichnern im Quellcode in Erscheinung.
5. Paare, die im Laufe ihrer Sitzung ein gemeinsames Verständnis von ihrem konkreten Softwaresystem und von Softwareentwicklung allgemein erarbeiten und pflegen, können kurze, aber sehr produktive Fokus-Phasen haben. Der Paarprogrammierungsprozess kann allerdings auch völlig zusammenbrechen, wenn ein solches gemeinsames Verständnis zu schwach ist und der Zusammenhalt des Paares weiter geschwächt ist durch das Fehlen eines gemeinsamen Plans, durch mangelndes Gewährsein des Arbeitsplatzes (etwa bei räumlich verteilter Paarprogrammierung, aber auch durch zu kleine Schrift), oder durch Sprachbarrieren.

Ich habe die wichtigsten Konzepte meiner Arbeit sowie drei konkrete Ideen zu ihrer Einbettung in den Entwicklungsalltag mit Praktikern aus zwei der ursprünglichen Unternehmen sowie zwei weiteren Firmen erfolgreich validiert.



Every honest researcher I know admits he's just a professional amateur.  
He's doing whatever he's doing for the first time. That makes him an amateur.  
He has enough sense to know that he's going to have a lot of trouble,  
so that makes him a professional.

– Charles F. Kettering



## Acknowledgements

First of all, I would like to thank Lutz Prechelt for many years of encouragement and support, for giving me every freedom to do what felt right to me, for always being open for late-afternoon discussions not only about research, and for providing the most thorough, honest, and valuable feedback I could hope for. Many thanks go to Paul Ralph, who agreed to be the second reviewer for this colossus of a thesis and who gave me excellent feedback in a very short time.

Special thanks go to Stephan Salinger, who not only spent nearly a decade of his life studying pair programming and laying the foundation my work builds on, but also, long ago, supervised my Master's thesis in his freetime and introduced me to the topic of pair programming and to qualitative research in the first place. In many ways, this work would not have been possible without him.

I also want to thank my other former AGSE colleagues Uli Stärk, Julia Schenk, Björn Kahlert, Edna Kropp, Holger Schmeisky, Barry Linnert, Lena Barke, Kelvin Glaß, and Victor Brekenfeld for many challenging, interesting, and fruitful research meetings. Much of the data I analyzed and the know-how that goes into collecting it only exists because of the hard work of Laura Plonka, Stephan Salinger, Holger Schmeisky, and Julia Schenk. Thank you!

None of this research would have been possible without the many unnamed software developers who volunteered to have their pair programming sessions recorded and scrutinized, took part in one of my workshops, or agreed to be interviewed. I greatly appreciate their openness.

Furthermore, I want to thank the many researchers and practitioners with whom I discussed my ideas on many informal occasions and conferences over the years. I thank the Simula Research Laboratory in Oslo, Norway and in particular Leon Moonen for welcoming me for a three-month visit as well as the German Academic Exchange Service (DAAD) for funding it.

Many thanks go to my mother and to my wife whose proofreading skills saved me from making a fool of myself.

I cannot overstate the impact of my family who always supported me through all these years: My parents, who raised me by encouraging openness, curiosity, patience, and perseverance, the love of my life and wife Anna, who sacrificed countless evenings and weekends we could have spent together, and our son Levi, who is simply the best.



# Contents

---

Abstract . . . . .	3
Zusammenfassung . . . . .	5
Selbstständigkeitserklärung . . . . .	463
List of Tables . . . . .	12
List of Figures . . . . .	13
List of Examples . . . . .	14
Notational Conventions . . . . .	18
<b>1 Introduction</b>	<b>19</b>
1.1 A Brief History of Pair Programming . . . . .	20
1.2 Motivation . . . . .	22
1.3 Goal of this Thesis . . . . .	24
1.4 Structure of this Thesis . . . . .	27
<b>I Foundation</b>	<b>31</b>
<b>2 Related Work</b>	<b>33</b>
2.1 Purpose and Structure of this Chapter . . . . .	34
2.2 Knowledge and Software Development . . . . .	34
2.3 Pair Programming . . . . .	40
2.4 Pair Work and Small Groups . . . . .	94
2.5 Summary of Related Work . . . . .	105
<b>3 Qualitative Research Methods</b>	<b>107</b>
3.1 Purpose and Structure of this Chapter . . . . .	108
3.2 Research Methods of the Social Sciences . . . . .	108
3.3 The Grounded Theory Methodology . . . . .	117
3.4 The Base Layer for Pair Programming Research . . . . .	128
<b>4 Research Goal, Method, and Data</b>	<b>139</b>
4.1 Purpose and Structure of this Chapter . . . . .	140
4.2 Goal Definition . . . . .	141
4.3 Data Collection . . . . .	144
4.4 Case Descriptions . . . . .	160
4.5 Analysis Method . . . . .	165
4.6 Discussion of Overall Research Method . . . . .	178
<b>II Results</b>	<b>181</b>
<b>5 Results Overview</b>	<b>183</b>
5.1 Purpose and Structure of this Chapter . . . . .	183
5.2 Pair Programming Process . . . . .	183
5.3 Knowledge Transfer Episodes . . . . .	184
5.4 Pair Programming Session Dynamics . . . . .	185
5.5 A Recurring Example . . . . .	186

<b>6</b>	<b>Process Fluency and Pair Togetherness</b>	<b>189</b>
6.1	Purpose and Structure of this Chapter . . . . .	189
6.2	Dialog Structure in Pair Programming . . . . .	190
6.3	Fluency . . . . .	200
6.4	Togetherness . . . . .	222
6.5	Discussion of Related Work and Summary . . . . .	234
<b>7</b>	<b>Knowledge Conceptualized</b>	<b>237</b>
7.1	Purpose and Structure of this Chapter . . . . .	237
7.2	Knowledge Want . . . . .	240
7.3	Topic and Target Content . . . . .	244
7.4	Summary and Discussion of Related Work . . . . .	254
<b>8</b>	<b>Knowledge Transfer Activities: Asking and Explaining</b>	<b>257</b>
8.1	Purpose and Structure of this Chapter . . . . .	257
8.2	Asking Questions with Explanation Elicitors . . . . .	260
8.3	Providing Explanations . . . . .	272
8.4	Summary . . . . .	277
<b>9</b>	<b>Episodes of Knowledge Transfer</b>	<b>279</b>
9.1	Purpose and Structure of this Chapter . . . . .	280
9.2	Properties of Episodes . . . . .	282
9.3	 Pull Mode . . . . .	289
9.4	 Pioneering Modes . . . . .	291
9.5	 Co-Production Mode . . . . .	295
9.6	 Push Mode . . . . .	297
9.7	Summary and Discussion of Related Work . . . . .	300
<b>10</b>	<b>Patterns of Episodes</b>	<b>303</b>
10.1	Purpose and Structure of this Chapter . . . . .	303
10.2	Anti-Patterns . . . . .	304
10.3	Positive Patterns . . . . .	307
10.4	Summary and Discussion . . . . .	314
<b>11</b>	<b>Session Dynamics</b>	<b>315</b>
11.1	Purpose and Structure of this Chapter . . . . .	315
11.2	Individual Developers' Knowledge Needs . . . . .	316
11.3	Pair Constellations . . . . .	319
11.4	Session Dynamics Prototypes . . . . .	321
11.5	Summary and Discussion of Related Work . . . . .	333
11.6	Grounded Theory of Knowledge Transfer Session Dynamics . . . . .	335
<b>III</b>	<b>Evaluation and Conclusion</b>	<b>339</b>
<b>12</b>	<b>Actual Research Process</b>	<b>341</b>
12.1	Phase 1: Initial Analysis of Base Activities . . . . .	342
12.2	Phase 2: Developing the Episode Concept . . . . .	342
12.3	Phase 3: Analysis of Pull Episodes . . . . .	343
12.4	Phase 4: New Knowledge Transfer Mode: Produce . . . . .	343
12.5	Phase 5: First Round of Data Collection . . . . .	343
12.6	Phase 6: Considering Practitioner Relevance . . . . .	345
12.7	Phase 7: Give Up Naturalistic Approach? . . . . .	345

12.8	Phase 8: Discovery of Second Knowledge Dimension . . . . .	346
12.9	Phase 9: Member Reflection and Selective Coding . . . . .	347
12.10	Phase 10: Finishing the Thesis . . . . .	347
<b>13</b>	<b>Evaluation</b> . . . . .	<b>349</b>
13.1	Purpose and Structure of this Chapter . . . . .	349
13.2	Member Reflection . . . . .	350
13.3	Eight Criteria for Qualitative Research . . . . .	356
<b>14</b>	<b>Conclusion and Further Work</b> . . . . .	<b>361</b>
14.1	☞ Research Contributions . . . . .	361
14.2	☞ Practical Applications . . . . .	362
14.3	Further Work . . . . .	364
	<b>Appendices</b> . . . . .	<b>365</b>
<b>A</b>	<b>Own Publications</b> . . . . .	<b>367</b>
<b>B</b>	<b>Transcription Notation</b> . . . . .	<b>371</b>
<b>C</b>	<b>Pair Programming Sessions</b> . . . . .	<b>373</b>
C.1	Session AA1 . . . . .	373
C.2	Session BA1 . . . . .	385
C.3	Sessions BB1, BB2, and BB3 . . . . .	386
C.4	Session CA1 . . . . .	387
C.5	Session CA2 . . . . .	389
C.6	Session CA3 . . . . .	393
C.7	Session CA4 . . . . .	394
C.8	Session CA5 . . . . .	394
C.9	Session DA2 . . . . .	398
C.10	Session DA5 . . . . .	404
C.11	Session EA1 . . . . .	406
C.12	Session JA1 . . . . .	408
C.13	Session JA2 . . . . .	413
C.14	Session KA1 . . . . .	414
C.15	Session KB1 . . . . .	416
C.16	Sessions KC1 and KC2 . . . . .	416
C.17	Session MA1 . . . . .	417
C.18	Sessions OA1 and OA2 . . . . .	418
C.19	Session OA5 . . . . .	420
C.20	Session OA8 . . . . .	420
C.21	Sessions PA1 and PA2 . . . . .	424
C.22	Sessions PA3 and PA4 . . . . .	424
C.23	Data Mapping . . . . .	426
<b>D</b>	<b>Meta-Analyses</b> . . . . .	<b>427</b>
D.1	Technical Information . . . . .	427
D.2	Pair Programming Effect on Students' Exam Scores . . . . .	428
D.3	Pair Programming Effect on Students' Assignment Scores . . . . .	429
D.4	Pair Programming Effect on Quality . . . . .	431
	Index . . . . .	433
	Name Index . . . . .	441
	Bibliography . . . . .	447

## List of Tables

2.1	Reported pair programming adoption rates in industry . . . . .	43
2.2	Reported percentages of developer time spent working in pairs . . . . .	43
2.3	Selection criteria used by the four secondary studies on pair programming . . . . .	52
2.4	Research topics and methods applied in pair programming research . . . . .	53
2.5	Statistical results of meta-analyses on educational pair programming effects . . . . .	55
2.6	Overview of students' self-reported learning achievements through PP . . . . .	57
2.7	Significance of correlational studies on pair members' skill levels . . . . .	59
2.8	Knowledge-related questionnaire items from PP effectiveness studies . . . . .	61
2.9	Statistical results of three meta-analyses on pair programming effects . . . . .	64
2.10	Overview of recurring problems in qualitative-quantitative PP research . . . . .	74
2.11	Overview of recurring problems in qualitative PP research . . . . .	86
2.12	High-level summary of research results on pair programming . . . . .	90
2.13	Knowledge types mentioned in PP studies . . . . .	92
2.14	Overview of knowledge coordination studies with pairs . . . . .	101
3.1	Common traits of qualitative research . . . . .	114
3.2	Common qualitative research traits mapped to GTM . . . . .	127
3.3	Object classification in the base layer . . . . .	133
3.4	Verb classification in the base layer . . . . .	135
4.1	PP session recording contexts . . . . .	146
4.2	Overview of available data . . . . .	151
4.3	Context and characterization of analyzed PP sessions . . . . .	161
4.4	Mapping of ATLAS.ti elements to uses in my analysis process . . . . .	175
4.5	Common traits of qualitative research in my research process . . . . .	180
6.1	Further differentiation of the base concepts . . . . .	191
6.2	<b>Fluency</b> and its three levels . . . . .	201
6.3	Overview of <b>Focus Phases</b> . . . . .	205
6.4	<b>Togetherness</b> and its three degrees . . . . .	223
8.1	Properties and types of <b>Explanation Elicitors</b> . . . . .	261
8.2	Types of <b>Explanations</b> . . . . .	272
9.1	Properties of knowledge transfer <b>Episodes</b> . . . . .	283
9.2	Characteristics of knowledge transfer <b>Modes</b> . . . . .	288
10.1	Elements and types of <b>Episode Patterns</b> . . . . .	313
11.1	Concepts to characterize programming pairs and their session dynamics . . . . .	323
11.2	Related work on knowledge-relevance in software engineering . . . . .	334
13.1	Member reflection activities . . . . .	352
13.2	Summary of member reflection . . . . .	356
B.1	Transcription notation . . . . .	371
C.1	Mapping of Plonka's data . . . . .	426
C.2	Mapping of Salinger's data . . . . .	426

## List of Figures

1.1	First description of pair programming as a practice, by James Coplien (1994)	22
2.1	Overview of memory systems	36
2.2	“We don’t practice pair programming”, anecdote by Salinger (2013)	42
2.3	Knowledge-related pair programming mechanisms expected by practitioners	45
2.4	Different types of effect sizes and meta-analyses	50
2.5	Overview of secondary studies on pair programming	52
3.1	Reciprocal and layered nature of communicative knowledge	111
3.2	Grice’s Maxims	112
3.3	What is axial coding? Or: The mystery of the “subcategory”.	122
3.4	Timeline of early pair programming publications in my research group	128
3.5	Classes of explicit knowledge in the base layer	137
4.1	Plot of a PP session: execution, knowledge transfer, and decision making	141
4.2	Data collection protocol: overview of data collection activities	147
4.3	Still frame of a session recording	148
4.4	PP session recording questionnaires	149
4.5	Classes of explicit knowledge in my work	174
4.6	Using ATLAS.ti codes for concepts, properties, and property values	176
4.7	Comparison of ATLAS.ti and my own visualization	177
6.1	Code at the beginning of an episode of <b>normal</b> pair programming in <b>OA8</b>	202
6.2	Code before the <b>Focus Phase</b> in <b>CA5</b>	206
6.3	Code after the <b>Focus Phase</b> in <b>CA5</b>	209
6.4	Relevant code changes before the <b>Breakdown</b> in <b>OA8</b>	211
6.5	Code before the <b>Breakdown</b> in <b>OA8</b>	211
6.6	Code after the <b>Breakdown</b> in <b>OA8</b>	217
7.1	Jigsaw puzzle metaphor of knowledge concepts	238
8.1	Overview of knowledge transfer activities	260
9.1	Stages and outcomes of <b>Episodes</b>	285
10.1	Overview of the beginning of session <b>DA2</b>	304
10.2	Overview of the beginning of session <b>JA1</b>	307
10.3	Overview of the beginning of session <b>CA2</b>	308
10.4	Overview of the beginning of session <b>EA1</b>	309
11.1	<b>Initial Constellations</b> of the pairs in the analyzed sessions	320
11.2	Visualizing session trajectories	321
11.3	Three different strategies of dealing with a <b>Primary Gap</b>	324
11.4	Three different strategies of dealing with a <b>Secondary Gap</b>	326
11.5	Trajectory of sessions with a <b>G Opportunity</b>	327
11.6	Trajectory of sessions starting with <b>Complementary Gaps</b>	329
11.7	Trajectory of sessions starting with a <b>Two-Sided G Gap</b>	332
11.8	Grounded Theory of knowledge transfer session dynamics	336
12.1	Example of a memo	344
13.1	A <b>G-S</b> chart	350

13.2	Example of a G-S chart in practice . . . . .	355
A.1	Timeline of my research group’s pair programming research . . . . .	369
C.1	Relevant excerpts of the Java code before Focus Phase #4 . . . . .	373
C.2	Relevant excerpts of the Java code after Focus Phase #4 . . . . .	374
C.3	Relevant excerpts of the Objective-C code in the middle of Focus Phase #5 . . . . .	375
C.4	Relevant excerpts of the Objective-C code at the end of Focus Phase #5 . . . . .	375
C.5	Relevant excerpts of the Java code before Focus Phase #1 . . . . .	394
C.6	To-scale representation of 60-second Focus Phase #1 . . . . .	395
C.7	Relevant excerpts of the Java code after Focus Phase #1 . . . . .	396
D.1	Meta-analysis of PP effects on exam scores . . . . .	428
D.2	Meta-analysis of PP effects on assignment scores . . . . .	429
D.3	Meta-analysis of PP effects on assignment scores of Zacharis (2011) . . . . .	430
D.4	Meta-analysis of PP effects on quality . . . . .	431
D.5	Meta-analysis of PP effects on quality of Zacharis (2011) . . . . .	432

## List of Examples

3.1	Coding with Base Concepts (DA5, 22:50–24:30) . . . . .	136
4.1	Good or Bad Behavior? (AA1) . . . . .	142
4.2	Knowledge Reconstructed From Interactions (CA2, 10:07–20:21) . . . . .	171
4.3	Expressing Opinions (CA4, 34:25–34:41) . . . . .	173
5.1	The “Raw Data” of the Recurring Example (JA1, 02:29–06:15) . . . . .	186
6.1	Agreeing to Proposal Through Action (CA2, 10:14–10:47) . . . . .	192
6.2	Reacting to Question With Answer (CA2, 19:54–20:00) . . . . .	193
6.3	Reacting to Question With ‘Improper’ Answer (CA2, 28:16–28:33) . . . . .	193
6.4	Self-Referential Activity without Partner Involvement (CA2, 31:17–31:57) . . . . .	194
6.5	Self-Referential Activity With Partner Involvement (CA2, 32:41–33:07) . . . . .	194
6.6	Corrective Activity (CA2, 35:21–35:39) . . . . .	195
6.7	Corrective Activity (CA2, 37:52–38:18) . . . . .	195
6.8	Expected Common Ground (CA2, 37:15–37:30) . . . . .	196
6.9	Unrelated Proposal (CA2, 43:02–43:26) . . . . .	197
6.10	Non-Actions (CA2, 55:41–55:49) . . . . .	197
6.11	Following One’s Own Initiative (CA2, 1:14:07–1:14:42) . . . . .	197
6.12	Not Clearing Up (CA2, 16:45–17:05) . . . . .	198
6.13	Five Types of Base Activities (JA1, 04:09–06:15) . . . . .	199
6.14	Normal Pair Programming (OA8, 49:10–51:19) . . . . .	201
6.15	A Focus Phase (CA5, 19:12–20:11) . . . . .	205
6.16	Breakdown with Seemingly Unhelpful Partner (OA8, 13:42–41:42) . . . . .	210
6.17	Breakdown Due to Huge Knowledge Gaps (OA1, 59:36–1:08:53) . . . . .	219
6.18	Asking for Intention (DA2, 1:14:25–1:14:44) . . . . .	224
6.19	Clarifying Intentions (CA5, 1:19:58–1:20:16) . . . . .	224
6.20	One Shared Plan (CA5, 17:28–18:20) . . . . .	226
6.21	High Togetherness (CA2, 28:14–28:23) . . . . .	227
6.22	Splitting Up (DA2, 40:03, 1:24:25, 1:35:42, & 1:41:17) . . . . .	228
6.23	Reading Documentation (KC2, 14:12–15:04 & 53:54–54:24) . . . . .	229

6.24	Maintaining Workspace Awareness (JA1, 53:56–54:34) . . . . .	231
6.25	Maintaining One Shared Plan (AA1, 25:39–28:00) . . . . .	232
6.26	Dealing With Conflict (CA5, 23:20–24:20 & 43:34–43:57) . . . . .	233
7.1	Knowledge Wants, Topics, and Target Contents (JA1, 02:29–06:15) . . . . .	239
7.2	Internal Knowledge Want During Implementation (AA1, 16:20–16:42) . . . . .	241
7.3	Incidental Internal Knowledge Want (AA1, 1:05:29–1:05:38) . . . . .	241
7.4	Explicit Topics and Target Contents (MA1, 05:13–11:16) . . . . .	242
7.5	From Internal to Collective Knowledge Want (AA1, 08:58–11:08) . . . . .	243
7.6	Uncovering Task Requirements (DA2, 09:23–18:57) . . . . .	245
7.7	Clearing Up Architectural Misconception (CA2, 19:30–20:37) . . . . .	245
7.8	Rationale for Data Type (KB1, 15:54–16:27) . . . . .	246
7.9	Inquiring about GUI Technology Stack (DA2, 01:54–02:46) . . . . .	246
7.10	Getting to Know Relevant Classes (KA1, 51:17–53:37) . . . . .	247
7.11	Open Bugs? (KA1, 59:23–1:00:01) . . . . .	247
7.12	How to Start a Manual Test (CA2, 44:08–45:52) . . . . .	248
7.13	Understanding Failed Network Calls (BA1, 01:33–04:00) . . . . .	248
7.14	Template Method Design Pattern (DA2, 1:36:35–1:37:58) . . . . .	249
7.15	Inner Classes in Java (DA2, 1:29:55–1:30:25) . . . . .	250
7.16	Deleting Folders Under Version Control (CA2, 26:11–27:01) . . . . .	250
7.17	OSGi Class Loading (DA2, 1:30:49–1:35:15) . . . . .	251
7.18	Talking About Developer Backgrounds (DA2) . . . . .	252
7.19	Talking About the Company (DA2) . . . . .	252
7.20	Domain Knowledge as Background Information (JA1, 05:03–05:19) . . . . .	253
7.21	Application Domain Knowledge Explained (KA1, 54:00–54:29) . . . . .	253
7.22	Domain Concept as Identifier (KA1, 1:00:05–1:01:24) . . . . .	253
7.23	Hypothesis to Satisfy Internal Knowledge Want (JA1, 06:00–06:12) . . . . .	254
8.1	Knowledge Transfer Activities (JA1, 04:15–06:15) . . . . .	258
8.2	Intentional Elicitation or Incidental Trigger? (CA2, 47:10–47:21) . . . . .	262
8.3	Frustrating Improper Asking (JA1, 08:27–09:19) . . . . .	263
8.4	Ignored Improper Asking (DA2, 12:52–13:21) . . . . .	263
8.5	Insufficient Improper Asking With Low Togetherness (JA1, 28:44–29:18) . . . . .	265
8.6	Direct Asking with Open Questions (JA1, JA2) . . . . .	265
8.7	Direct Asking With Possible Answers (DA2, 01:54–02:06) . . . . .	266
8.8	Asking to Show Application (DA2, 09:23–09:37) . . . . .	266
8.9	Asking to Show Source Code (CA2, 10:07–10:47) . . . . .	266
8.10	Refer to Common Ground as Context for Question (JA1, 05:00–05:48) . . . . .	267
8.11	Imply Question by Referring to Common Ground (JA2, 13:41–14:25) . . . . .	267
8.12	Translating German Modal Particles (JA1, 05:48–05:53) . . . . .	268
8.13	Simple Step with Almost Revealed Conclusion (DA2, 18:17–18:57) . . . . .	268
8.14	Conclusion Drawn From Simple Step (JA1, 09:32–10:01) . . . . .	269
8.15	Conclusion Not Drawn From Simple Step (JA1, 28:57–29:47) . . . . .	269
8.16	Misunderstood Optimistic Proposition (OA8, 10:35–10:49) . . . . .	270
8.17	Propositions to Demonstrate Understanding (JA1, 27:47–28:42) . . . . .	271
8.18	Pessimistic Proposition in Disbelief (CA2, 11:32–11:55) . . . . .	271
8.19	Present New Fact After Improper Asking (AA1, 59:12–59:25) . . . . .	273
8.20	Present New Fact After Direct Asking (KB1, 02:39–02:58) . . . . .	273
8.21	Present New Fact to Correct Understanding (AA1, 28:26–28:33) . . . . .	273

8.22	Refer to Common Ground for Obvious Explanation (CA2, 10:58–11:26)	274
8.23	Conclusion Not Drawn From Simple Step (JA1, 14:35–14:58)	275
8.24	Entice to Simple Step After Presenting New Fact (JA1, 24:16–24:44)	275
8.25	Present New Fact and Refer to Common Ground (MA1, 13:26–13:41)	276
8.26	Entice to Simple Step as Indirect Criticism (OA8, 50:12–52:52)	276
8.27	Entice to Simple Step in Direct Criticism (AA1, 19:44–20:13)	277
9.1	Illustrating Episodes (JA1, 04:15–06:15)	281
9.2	Start Episode With Oblivious Interrupt (CA2, 10:46–11:56)	283
9.3	Start Episode With Careful Interrupt (CA1, 00:55–01:22)	284
9.4	Start Pushing at the Right Moment (CA1, 12:10–12:47, 13:57–14:32)	284
9.5	Ignored Episode (JA1, 40:29–40:42)	285
9.6	Resigned Episode With Explicit Topic (DA2, 10:11–10:26)	286
9.7	Resigned Episode With Tacit Topic (CA2, 1:03:46–1:04:12)	286
9.8	Unnecessary Episode (DA2, 15:02–15:15)	286
9.9	Postponed Episode (CA2, 1:05:29–1:05:38)	287
9.10	Pull Episode Switching to Pioneering (CA2, 52:26–53:44)	287
9.11	Pull Episode Switching to Co-Production (JA1, 22:59–23:48)	287
9.12	Asking for Code, Non-Verbal Reaction (CA2, 10:42–10:54)	290
9.13	Pulling for Guidance (CA1, 06:17–07:05)	290
9.14	Peril of not Sharing Pioneering Intention (CA2, 1:15:59–1:16:43)	292
9.15	Silent Pioneer Who Talks (CA1, 19:57–21:01)	292
9.16	Irritating Silent Pioneer with Low Togetherness (JA1, 19:13–19:39)	293
9.17	Necessary Pioneering w/o Knowledgeable Partner (AA1, 13:10–14:13)	293
9.18	Necessary Pioneering Despite Knowledgeable Partner (AA1, 49:53–52:16)	294
9.19	Dealing with Interrupts Seamlessly (AA1, 1:43:40–1:47:42)	295
9.20	Co-Production with High Togetherness (AA1, 11:19–11:45)	296
9.21	Together or Not? (AA1, 1:23:27–1:25:13)	297
9.22	Socratic Issue Push (JA2, 18:38–19:43)	298
9.23	Why Push? (PA3, 29:53–31:37)	299
10.1	Discussion of Recurring Example: Multiple Wants (JA1, 04:15–06:15)	303
10.2	Long, Winding Way to Understanding Requirements (DA2, 01:54–18:57)	305
10.3	Return After Finished Sub Pull (JA1, 04:08–06:33)	308
10.4	Return After Resigning a Catalyzed Pull (CA2, 10:07–11:58)	308
10.5	Easy Return from Short Sub Pull (EA1, 04:23–13:45)	309
10.6	Negotiating the Scope (JA1, 13:15–13:43)	310
10.7	Limiting Scope for Focus (CA5, 19:12–20:11, 47:11–47:22, 1:21:57–1:22:20)	311
10.8	Limit Scope of Partner's Pioneering (BB1, 16:47–19:06)	312
10.9	Trying to Limit Scope of Partner's Push (JA1, 58:23–59:32)	312
11.1	Recurring Example: Foreshadowing the Session Dynamics (JA1)	316
11.2	Greenfield Development (BB1, BB2, BB3)	322
11.3	Bringing Partner Into Ongoing Work (EA1)	324
11.4	Prepared Interview Mode (MA1)	325
11.5	Closing the Primary Gap Painfully (CA2)	325
11.6	Pairing-Up Throughout (AA1)	326
11.7	Initially Misunderstood Teaching (PA3, PA4)	328
11.8	G Opportunity Not Seized (CA1)	328

11.9	Embracing a Difference (JA1)	330
11.10	Easy-Task Jump-Start with G Opportunity (DA2)	330
11.11	It's not easy! (KC2)	331
11.12	Breakdown (OA1, OA2)	332
C.1	Focus Phase #4 (AA1, 1:53:20–1:54:56)	373
C.2	Focus Phase #5 (AA1, 1:55:45–1:57:04)	375
C.3	Focus Phase #6 (AA1, 1:57:38–2:00:55)	376
C.4	Coding Surprises (AA1, 27:20–32:32)	377
C.5	Coding with Base Concepts (DA5, 22:50–24:30)	404

## Notational Conventions

- Typewriter font: Source code snippets or constructs from programming languages, e.g., for-loop.
- SMALL CAPS: Informal concepts that are used in literature or which I introduce myself as shorthands for some ideas, e.g., PAIR PRESSURE or NATURALISTIC INQUIRY.
- Blue sans-serif font: Theoretical concepts that are the result of my qualitative analysis of knowledge transfer in pair programming, e.g., Focus Phase.
- Blue italic serif font: Theoretical concepts that are the result of qualitative analyses performed by other researchers, e.g., *explain\_knowledge*.
- Light-red sans-serif font: Identifiers of analyzed pair programming sessions, e.g., CA2.
- Green-ish/purple-ish sans-serif font: Identifiers of pair programmers, e.g., C1 and C2.

I use different types of quotation marks:

- “*Italics in double quotes*”: A direct quotation from a concrete source such as literature, an interview, or a pair programming session, such as developer C5 who said “*M-hm, we will see.*” See Appendix B for the full transcription scheme.
- ‘*Italics in single quotes*’: A pseudo-quotation, something that someone *could* have said or thought, such as developer C5 who might have thought ‘*I don’t want to deal with this now!*’.
- “Upright text in double quotes”: A reference to a commonly used term without a specific source, as in saying that there are developers say they are “pair programming”.
- ‘Upright text in single quotes’: Figurative use of speech, as in saying that people ‘store’ information in memory.

# Chapter 1 Introduction

---

*It is not once nor twice but times without number that the same ideas make their appearance in the world.*

– Aristotle

<b>1.1 A Brief History of Pair Programming</b> . . . . .	20
1.1.1 Collaboration from the Very Beginning. . . . .	20
1.1.2 Programming Groups . . . . .	20
1.1.3 Programming with a Partner . . . . .	21
1.1.4 Pair Programming as a Practice . . . . .	21
<b>1.2 Motivation</b> . . . . .	22
1.2.1 Knowledge and Knowledge Transfer in Software Development . . . . .	23
1.2.2 Common Research Questions. . . . .	23
1.2.3 State of Research . . . . .	23
<b>1.3 Goal of this Thesis.</b> . . . . .	24
1.3.1 Goal Formulation and Characterization . . . . .	25
1.3.2 Scope . . . . .	25
1.3.3 Research Approach and Initial Definitions . . . . .	25
1.3.4 Is this Software Engineering?. . . . .	26
<b>1.4 Structure of this Thesis.</b> . . . . .	27

The term “pair programming” refers to two distinct ideas. The first idea is probably as old as programming itself and did not have a name for a long time: Tackling a difficult programming task with a partner makes it easier to solve. I myself experienced this in eleventh grade, together with a friend for CS homework assignments. Nobody suggested this to us and I had never thought about it until recently—it just felt natural. Perhaps most software developers can tell such a story: Similar anecdotes reach as far back as Fred Brooks in the 1950s (and beyond), and are now retrospectively called “pair programming”. This work mode sparked scientific interest driven by an economic question, which to this day has no conclusive answer: Is the code quality produced by a pair higher and the total time spent less as to justify paying two developers to do the work of one?

The second idea dates back to the 1990s: Pair programming (PP) was no longer only an ad hoc *work mode* chosen for individual tasks. It became a named *practice* that is integral to a larger software development process and has effects reaching far beyond a single programming task, potentially changing the way how development teams produce software systems. But let us start at the beginning.

## 1.1 A Brief History of Pair Programming

### 1.1.1 Collaboration from the Very Beginning

Ada Lovelace is generally considered to be the first ‘programmer’. In 1843, she translated and—more importantly—commented a French description of Charles Babbage’s *Analytical Engine*.<sup>1</sup> The resulting “Notes” remain the most complete description of the engine today, and by adding her ideas, she not only surpassed Babbage’s original plans but was “*the first person to have crossed the intellectual threshold between conceptualizing computing as only for calculation on the one hand, and on the other hand, computing [through] symbolic substitution*” (Fuegi & Francis, 2003). In Note D, Lovelace characterized the difficulties of ‘programming’:

“ It must be evident how multifarious and how mutually complicated are the considerations which the workings of such an engine involve. There are frequently several distinct *sets of effects* going on simultaneously; all in a manner independent of each other, and yet to a greater or less degree exercising a mutual influence. To adjust each to every other, and indeed even to perceive and trace them out with perfect correctness and success, entails difficulties whose nature partakes to a certain extent of those involved in every question where *conditions* are very numerous and inter-complicated [...].

Lovelace, written in 1843, cited in Babbage (1889, p. 36, emphasis in original)

In today’s terms, Lovelace and Babbage were writing a program to calculate Bernoulli numbers and designing the hardware to run that program at the same time: “*She was programming the machine. She programmed it in her mind, because the machine did not exist*” (Gleick, 2011, p. 119). Although Lovelace is clearly the author of the Notes, they are considered to be the result of collaborative efforts of her and Babbage (Toole, 1996). The two “*sent letters by messenger back and forth across London at a ferocious pace [...] and met whenever they could*” (Gleick, 2011, p. 115). It appears that programming has always been complicated, and from the very beginning, it was a collaborative effort.

### 1.1.2 Programming Groups

Well over one hundred years later, Jerry Weinberg framed programming as a social activity in his book *The Psychology of Computer Programming*:

“ Programmers do not ordinarily work in isolation. Although an individual programmer may find himself assigned the task of writing a program, even then he has other programmers to whom he may turn for help—and who, at the same time, may be turning to him.

Weinberg (1971, p. 45)

As Weinberg later remembered on multiple occasions (e.g., in interviews from 2011 and 2016), he learned how to program from Bernie Dimsdale in the late 1950s, who in turn learned it from John von Neumann. According to Weinberg, Dimsdale and von Neumann already “*pair program[med]*” in the 1940s—on paper (cited by Coplien, 2015). However, Weinberg (1971) originally described von Neumann’s programming style in the context of *egoless programming*—i.e., detaching programmers from their work for better judgment—as something that sounds more like a *code review* than actual programming as a pair:

---

<sup>1</sup>The story of Babbage’s plans, his trouble with British authorities, and how he ended up meeting Ada Lovelace is convoluted. Fuegi & Francis (2003) and Gleick (2011) shed some light on it.

“ A programmer who truly sees his program as an extension of his own ego [...] is going to be trying to prove that the program is correct—even if this means the oversight of errors which are monstrous to another eye. [...] John von Neumann himself was perhaps the first programmer to recognize his inadequacies with respect to examination of his own work. [...] [H]e was constantly asserting what a lousy programmer he was, and [...] he incessantly *pushed his programs on other people to read for errors* and clumsiness.

Weinberg (1971, pp. 55–56, emphasis added)

Educator Paul H. Cheney (1977) was probably the first to use the term “pair programming”—in an experiment comparing its effect on exam scores to that of individuals. However, Cheney’s pairs were modeled after Weinberg’s programming groups, meaning each programmer would write a program of her own, then exchange it with the partner, and have it checked for errors.

### 1.1.3 Programming with a Partner

The earliest account of a true collaborative effort of programmers working on a *joint* task is probably an industrial experiment by Randall W. Jensen (2003) conducted in 1975. Jensen compared the productivity and error rates of “*two-person programming teams*” to historical data of individuals. However, Jensen did not publish his report until 2003, so it is P.J. Plauger who is sometimes said to have ‘*invented*’ pair programming as a professional technique in the late 1970s. Larry Constantine visited Plauger’s company Whitesmiths, Ltd. and observed programmers routinely working in pairs, though it is not clear which role Plauger played in establishing this culture:<sup>2</sup>

“ At each terminal were two programmers! [...] The room buzzed with a steady stream of questions about the algorithm or whether an initial value was correct, suggestions about how to break out of a loop, or drawing attention to a syntax error or test done in the wrong order or a missing case. [...]

Plauger assured me that this was *their chosen mode for working*. [...] I came to think of this model for programming teamwork as the “Dynamic Duo.”

Constantine (1995, p. 118, emphasis added)

Other developers also chose to work with a partner: Reenskaug & Skaar (1989) report on a Smalltalk system comprising 75,000 lines of code for which they had “*found that two persons working together on one workstation are very productive since they challenge each other’s clear thinking and immediately document the results of this thinking*”. Williams & Kessler (2002) collected a number of historical pair programming anecdotes, including that of Richard Gabriel who recalls that “[*p*]air programming was a common practice at the M.I.T. Artificial Intelligence Laboratory when I was there in 1972–73 [...] [*W*]e’d sit next to each other in front of his or my terminal” (*ibid.*, pp. 11–12).

### 1.1.4 Pair Programming as a Practice

So far, the idea of programming with a partner had remained a *work mode* that programmers occasionally engaged in on a per-task basis, but was not yet an institutionalized development *practice*: A named activity that goes beyond ad hoc usage, for which there is an agreement of when and how it is supposed to be done. This began to change in the 1990s. Coplien (2015) recalls talking to Ward Cunningham and Paul Chisholm at the OOPSLA conference in 1993 about programming with a partner, after which he decided to formulate the pattern *Developing in Pairs* (see Figure 1.1) and presented it at the PLoP conference in 1994.

<sup>2</sup>I was not able to find a first-hand report from Plauger, but there are second-hand mentions by Jeffries et al. (2001, p. 88), Constantine (2011), and Coplien (2015).

**Pattern: Developing in Pairs****Problem:** People are scared to solve problems alone.**Context:** Code ownership has been identified and development is proceeding.**Forces:** People sometimes feel they can solve a problem only if they have help. Some problems are bigger than an individual. Too many people can't sit in front of a keyboard and screen. Effort goes up nonlinearly with number of people.**Solution:** Pair compatible designers to work together; together, they can produce more than the sum of the two individually.**Resulting Context:** A more effective implementation process. A pair of people is less likely to be blindsided than an individual developer.

**Figure 1.1:** First description of pair programming as a *practice* as opposed to a mere work mode, presented by James Coplien at the PLoP conference in 1994, printed in Coplien (1998, p. 294).

When Kent Beck (1999) first formulated Extreme Programming (XP), he made *pair programming* one of the twelve practices. Beck introduces it as *code reviews* taken to an “*extreme level*” (ibid., p. xv). The idea is, that “[a]ll production code is written with two people looking at one machine, with one keyboard and one mouse”. He describes two roles in this practice: The developer with keyboard and mouse thinks about the implementation, while her partner thinks strategically about the overall approach (ibid., p. 58). While this describes programming pairs as rather asymmetric, Beck offers a second, more nuanced characterization that emphasizes the communicative nature of the practice later in the book:

“ It isn't one person programming while another person watches. [...] *Pair programming is a dialog* between two people trying to simultaneously program (and analyze and design and test) and understand together how to program better. It is a conversation at many levels, assisted by and focused on a computer.

Beck (1999, p. 100, emphasis added)

To Beck (ibid., p. 97), pair programming is *the central XP practice*, as it “*ties the whole [XP] process together*”. He mentions a number of benefits that affect different aspects of a team's development process, including product quality, information flow, discipline, and developer satisfaction (Beck, 1999, pp. 30, 67, 102; Beck & Andres, 2004, p. 42). Beck is also aware of a number of caveats and preconditions that should be met in order to reap the full potential of the practice: Pair programming “*is a subtle skill*” and some developers may refuse to pair altogether or with certain partners (Beck, 1999, pp. 100–101); developers need to understand their system, should have effective coding standards, and should be well-rested to avoid unfruitful discussions and slow progress (ibid., p. 67).

To summarize, *pair programming* (PP), at its core, is the idea of two software developers working together on a technical task. This idea is very likely as old as programming itself, and was popularized as an XP practice in the early 2000s.

## 1.2 Motivation

In this section, I briefly discuss the roles of knowledge and pair programming in software development, a number of common research questions, and the current state of research regarding these questions. Based on this, I formulate the goal for my thesis in Section 1.3.

### 1.2.1 Knowledge and Knowledge Transfer in Software Development

In a sense, software development “is the progressive crystallization of knowledge into a language that can be read and executed by a computer” (Robillard, 1999). It involves a lot of knowledge: Programming languages, design patterns, algorithms, system architecture, requirements, procedures for debugging and testing, and much more. In practice, usually not all of the relevant knowledge is readily available to the developers. As Armour (2000) puts it: “the hard part of building systems is not building them, it’s knowing what to build—it’s in acquiring the necessary knowledge. [...] [S]oftware development [...] is a knowledge-acquiring activity.”

One way for practitioners to acquire the relevant knowledge is to work in pairs. Asking industrial software developers from different companies why they pair program, Plonka et al. (2012a, Sec. VII) found *knowledge transfer* to be an important reason. All-pair-programming companies such as Pivotal Labs use pair programming as the essential practice to remove and avoid knowledge silos where only one team member knows about some system or technology (Sedano et al., 2016). Begel & Nagappan (2008) surveyed practitioners at Microsoft and report a number of perceived pair programming benefits, many of which relate directly or indirectly to knowledge and knowledge transfer, such as spreading of code understanding, learning from the partner, or fewer bugs. Put differently, many of the reasons why developers choose to pair-program relate to the following expected effects:

- To make use of their combined knowledge to work on tasks which would be more difficult for either of them alone.
- To make use of their combined competence to acquire any lacking knowledge faster and more reliably than they would alone, which is helpful for debugging situations and catching defects in the making.
- To learn together and from another, e.g., in training scenarios and to avoid knowledge silos in mature teams.

### 1.2.2 Common Research Questions

Although all these expectations seem plausible, empirical evidence remains to be presented. It is safe to assume that no two software developers are equally and perfectly knowledgeable in all regards. Consequentially, *some* knowledge transfer and knowledge acquisition can be expected to occur in any pair programming session.

Questionnaires about developers’ reasons to work in pairs (Plonka et al., 2012a, Sec. IV) and perceived PP effects (Begel & Nagappan, 2008, Sec. 4) cannot accurately capture actual events, but indicate that combining existing knowledge and the ability to acquire new knowledge in pair programming has a positive effect on the technical outcome *and* on the pair members’ abilities to work on future tasks.

In describing pair programming, Beck (1999, pp. 30, 102) addresses the question of *how* such effects come to be only implicitly: He expects knowledge transfer to just happen because PP is “*conversational*” in nature, i.e., it cannot be done without communicating. As a consequence, developers would talk about many different things and knowledge will spread in the team, in particular knowledge about the code, the overall software system, and about development practices.

### 1.2.3 State of Research

I discuss the existing body of research on pair programming and other related work in detail in Chapter 2. Here, I merely provide a shorter overview to motivate my work.

In the early 1990s, researchers began to publish studies on pair programming with professional software developers. These studies were motivated by the expectation that *collaboration* may have a positive effect on problem-solving. They addressed PP from an economic perspective: Is the cost of two developers working on one job compensated by higher quality and/or less time needed?

Typically, these studies were designed as controlled experiments. A meta-analysis by Hannay et al. (2009) showed that for single tasks, overall, programming in pairs appears to have a small *positive* effect on quality, and a medium *negative* effect on required effort. More importantly, there is significant between-study variance, i.e., individual studies reported different, sometimes even contradicting effects, indicating that additional factors—such as expertise, task complexity, amount of training in PP, or motivation—play a role and were not accounted for.

Despite the apparent practical relevance of knowledge transfer in pair programming, there are only few studies which address this topic directly. There is only a handful of studies considering a learning or knowledge-sharing effect of pair programming *beyond* a single task. Some gave questionnaires to students (e.g., Cockburn & Williams, 2001) or software developers (e.g., Palmieri, 2002) who report *perceived* learning and knowledge-spreading effects. Others tried to determine directly whether knowledge was transferred or acquired (e.g., McDowell et al., 2003) by comparing end-of-term test results of students who worked on their assignments either in pairs or alone (they found no significant difference). I am not aware of any industrial study in this vein.

To understand *how* knowledge transfer in pair programming works, actual sessions need to be observed or recorded and then analyzed, which only few researchers did. Bryant et al. (2008) looked at the abstraction levels of individual utterances in industrial sessions and demonstrated that both partners, overall, tend to talk on all the levels, high and low. Plonka et al. (2015) identified six “*teaching strategies*” which professional software developers use in pairs with members who have different levels of knowledge, such as “*nudging and physical hints*” or “*gradually adding information*”. These strategies, however, are limited to the particular excerpts that Plonka et al. selected for their detailed analysis—in particular to sessions with (a) a clear expert-novice constellation, and (b) the explicitly declared purpose of transferring knowledge, and (c) to episodes in which the expert tries to teach the novice. Considering the supposed importance of knowledge in software development, such a focus cannot be expected to cover all relevant knowledge-related behavior.

In summary, the existing body of software engineering literature acknowledges the importance of knowledge in software development and expects knowledge transfer to occur between the members of a programming pair, but has yet to provide a coherent description of how this works and how practitioners may improve their knowledge transfer.

### 1.3 Goal of this Thesis

Software developers and managers alike expect pair programming to be a means to avoid knowledge silos and spread skills in the team (see Section 1.2.1). Two decades of research on pair programming focused almost exclusively on the economic aspects, and could neither provide a conclusive economic answer nor an explanation of how knowledge actually gets transferred in pair programming (see Section 1.2.3). In practice, however, it seems that software developers do not wait for an economic answer, but choose to pair program anyway (see the long history of pair programming in Section 1.1).

In Chapter 4, after discussing related work, I will formulate my goal, scope, and research approach in detail. What follows is the executive summary.

### 1.3.1 Goal Formulation and Characterization

The goal of my thesis is shaped by a lack of understanding in research on the one hand, and on the other hand by the practical relevance for software developers who use or consider using pair programming in their professional environments. It is hence twofold and consists of a  $\mathbb{A}$  *knowledge-seeking* part and a  $\mathbb{Q}$  *solution-seeking* part (Stol & Fitzgerald, 2018, Table 3):

$\mathbb{A}$  **Goal 1:** Understand how knowledge transfer works in pair programming in industrial settings, in particular how developers deal with what they individually and collectively know and do not know, i.e., what the underlying mechanisms are of the exchange of existing knowledge and the acquisition of new knowledge.

$\mathbb{Q}$  **Goal 2:** Formulate results in a way that is comprehensible and relevant for software developers, allowing them to reflect on their own process and identify which mechanisms work well and which are problematic.

### 1.3.2 Scope

I focus on pair programming as it occurs in **industrial settings**. My research is concerned with pair programming as a work mode as opposed to a practice, i.e., the phenomenon of two software developers working together on a task and *not* the strategic decision to use this work mode routinely or based on some other criteria. I do not attempt to answer the question whether PP is worth doing, whether its benefits outweigh its costs. Instead, I start my investigation at a point where developers already made the decision to pair program and study how knowledge transfer then happens.

Consequentially, the unit of interest for me is the individual **pair programming session** with professional software developers who chose to work on some task in the context of an industrial project. I do not study the pair, their team, nor the project as such. The pair's reason to engage in a PP session, the developers' history, the constraints and peculiarities of their project may echo in their session, but are not my research subject, and neither are any effects of the PP session on the pair's abilities, the product's quality, or the project's success.

I exclude contrived settings such as homework assignments, coding katas, or recruiting sessions. These settings all have a make-believe or game character which, even if source code from actual software projects is involved, may affect the developers' motives.

### 1.3.3 Research Approach and Initial Definitions

Given how little is known about how pair programming actually works, my research is *empirical* and *exploratory* in nature. Consequentially, there is no fixed research question, but more an area of interest: How do pair programmers deal with what they know and what they do not know? To this end, I collect data in industrial settings, with professional software developers working in pairs on their everyday tasks. Moreover, my approach is *qualitative* and *theory-building* in nature, rather than quantitative or theory-testing.

However, even an exploratory empirical investigation needs at least a rough understanding of the phenomena of interest to begin with. The notion of *knowledge transfer* is not easy to define adequately, in part because defining *knowledge* is deeply philosophical matter.

An important distinction here is between (a) knowledge as *the condition of knowing something* and (b) knowledge as *the object being known*. Epistemology is concerned with what it means to “know something”, i.e., with knowledge in the first sense. The common—though not undisputed—notion of knowledge as “*justified true belief*” (see, e.g., BonJour, 2010, pp. 23–24) takes this stance. In contrast, knowledge definitions such as “*permanent structure of information stored in memory*” (Robillard, 1999) point at the close relationship between knowledge and information. For my research, which is limited to the world of software engineering, I do not need a universally accepted definition of knowledge, but an appropriate characterization of what “knowledge” and “knowledge transfer” means in the context of pair programming. For my research, a distinction between knowledge and information is not necessary.

**Definition**

**Knowledge** is *information* (i.e., a proposition pertaining to a part of the reality such as an object or an event) that the pair members consider *relevant* for their software development context, including their current task and the software project it is embedded in.

In this sense, knowledge is an object, something which a developer can possess. When the according information is not available to her, there is a gap in knowledge. Examples for knowledge in this sense include:

- Long-lived knowledge that is (potentially) relevant beyond a single session, including information about the source code, its constraints, specific technologies, usage of development tools, but also about specific tasks, requirements and requirement prioritization.
- Short-lived knowledge such information about the current session’s goal, what has been done already, possible solutions (product- and process-wise), what the developers know about these, and the developers’ attitudes regarding these solutions.

**Definition**

**Knowledge Transfer** is any attempt of the developers to close a gap in knowledge, either by exchanging existing knowledge or by acquiring new knowledge.

Note that, for my research, these definitions are mere starting points which are based on practitioners’ colloquial use of the phrase *knowledge transfer* in the context of pair programming. Finding more appropriate notions and operationalizations is part of this thesis.

### 1.3.4 Is this Software Engineering?

Software Engineering as a discipline is concerned “*with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production*” (Sommerville, 2007, p. 7). Pair programming is a part of how software developers work in industry, either because it is a *practice* of their software development process, a planned activity, which developers engage in strategically, or it is a spontaneous work mode, e.g., after asking a colleague for help.

Enabling software developers to take more informed decisions when it comes to whether or not to employ PP, and if so, how to improve, is therefore a software engineering goal. Since the idea of pair programming is universal, any such result may have practical relevance for a large number of contexts independent of particular software domains and technologies.<sup>3</sup>

---

<sup>3</sup>Some conditions apply: Developers who do not physically share a workspace, for example, cannot directly employ the ‘*on one machine*’ idea. Distributed Pair Programming, then, requires some non-trivial technical setup and affects the development process (see, e.g., Schenk, 2018).

## 1.4 Structure of this Thesis

I divide my thesis into three parts. Chapters 2 to 4 are the foundation with a discussion of related work and methodology, as well as my own research method and data:

- Ch. 2** Pair programming (PP) has been studied since the early 1990s. Most studies regard PP as an alternative to programmers working alone and have a quantitative interest in comparing these two work modes in terms of time spent or resulting quality. Qualitative studies ask *how* pair programming works, and what its consequences are, e.g., for teams or companies trying to introduce it. My discussion of their findings and limitation shows that quantitative studies fail to provide a conclusive answer as to which work mode is superior, while most qualitative studies either lack a rigorous research method or are concerned only with a narrow aspect of the practice without providing starting points for further studies. To complete the picture, I also briefly discuss the notions of *knowledge* and *knowledge transfer* as well as the research of cognitive and social psychology regarding small groups and pair work in general.
- Ch. 3** To overcome the methodological shortcomings of existing PP research, Salinger & Prechelt (2013) propose the *base layer*, a framework for qualitative research on pair programming. The base layer facilitates the analysis of recorded pair programming sessions, which are the data source of choice if the inner workings of the pair programming process are to be understood. I explain the base layer after discussing the basis of qualitative research in the social sciences in general and the Grounded Theory Methodology in particular.
- Ch. 4** Here, I describe my own implementation of this methodology. My analysis is based on full-length recordings of everyday PP sessions from industrial settings. The recordings comprise the pair's dialog, screen content, and webcam video. Part of the analyzed sessions stem from earlier studies (Plonka, 2012; Salinger, 2013; Schenk, 2018); others I collected during research stays in three different companies, during which I also did interviews and field observations of everyday work.

Chapters 5 to 11 contain my results, mostly in the shape of *concepts* (set in blue sans-serif font) to characterize different aspects of pair programming processes:

- Ch. 5** My concepts are highly interconnected, which hinders a linear write-up. This chapter presents the most important concepts in a nutshell. Readers interested in only some of the full-fledged results of the later chapters should read this first.
- Ch. 6** The first area of results concerns the general pair programming process. How well two software developers work *as a pair* at any moment is described by the pair's **Fluency**. Arguably, high **Fluency** is more desirable than low **Fluency**—the first potentially culminating in highly productive **Focus Phases**, the latter being susceptible to a **Breakdown** of the pair process. A pair's **Togetherness** describes how 'in tune' its members are, and determines how **fluent** their process can be. Two factors influencing a pair's **Togetherness** are the members' *shared understanding of the software system* and of *software development in general*. To avoid low **Togetherness**, pairs have to **Maintain** it; transferring knowledge about the system and about software development is one way to do this.
- Ch. 7** Here, I clarify the meaning of *knowledge* the context of PP sessions. I distinguish three knowledge concepts: (1) The **Knowledge Want**, which arises from a perceived knowledge gap and is the driving force for starting explanations, asking questions, or examining source code or artifacts; (2) the **Topic** of a knowledge transfer, which

needs to be understood by both developers—at least to some degree—for a knowledge transfer to be successful; and (3) the **Target Content**, which is the very piece of information which is (or would be) able to fill the knowledge gap that gave rise to the **Knowledge Want**. I distinguish two types of knowledge which characterize most **Topics** and **Target Contents** in PP sessions: System-specific **S knowledge** and general software development knowledge, or **G knowledge**.

- Ch. 8** Each instance of knowledge transfer in a pair programming session is comprised of a number of individual activities, the *atoms*, which together constitute the knowledge transfer. Roughly speaking, there are *questions* and *answers*, or more conceptually, **Explanation Elicitors** and **Explanations**. In practice, however, these two categories are not mutually exclusive and contain a number of peculiar types.
- Ch. 9** On a higher level, all knowledge transfer in PP sessions occurs in **Episodes** which each pertain to a single **Topic**. An **Episode** is **propelled** by one pair member in a specific **Mode**: In  **Push** and  **Pull Mode**, existing knowledge is transferred from and to the **propelling** pair member, respectively; in **Production Mode**, new knowledge is acquired, either by one pair member alone ( **Pioneering Production**) or with both partners engaged in the **Topic** ( **Co-Production**).
- Ch. 10** In a complex environment such as software development, there may be many knowledge gaps that need to be addressed, some of which the pair only becomes aware of along the way. Some pairs may get into trouble because they **Branch Wildly** by starting new **Topics** without finishing off open ones. Others keep their slate clean by **Returning Explicitly** when a **Sub-Episode** is done and through **Scope Limiting** where they do not get sidetracked in the first place.
- Ch. 11** Then, I integrate all these pieces to a grounded theory of knowledge transfer in pair programming. I consider the PP session as a whole and identify six **Initial Constellations** of the pair regarding its members' task-specific **Knowledge Needs**. Although the details vary and pairs may have different **Target Constellations** (such as solving a problem fast or educating the partner), all analyzed pairs follow the same overall session dynamic: First, they address their **Primary Gap**, which is their relative difference in system understanding, then they address their **Secondary Gap**, which is what they both do not yet understand about the task-relevant parts and aspects of the system. Only then, with **S Knowledge Needs** out of the way, may pairs actively engage in transferring **G knowledge**. I call a relative difference in this regard “the **G Opportunity**” since not all pairs use it. If *both* pair members lack relevant **G knowledge**, however, progress becomes much more difficult and the pair may struggle with a **Breakdown**.

In Chapters 12 to 14, I evaluate my results and discuss implications for further work:

- Ch. 12** I arrived at these results after a long research process during which I amended my analysis focus multiple times. In this chapter, I reiterate the process and key decisions I made along the way.
- Ch. 13** I discuss and evaluate my research based on eight quality criteria for qualitative research (Tracy, 2010). Most of them (Worthy Topic, Rich Rigor, Sincerity, Significant Contribution, Ethics, and Meaningful Coherence) are addressed at various places throughout this document. I summarize them again and also discuss how I addressed the criteria of Resonance and Credibility through *member reflection*, through workshops, interviews, and hands-on evaluation with practitioners from companies.
- Ch. 14** I conclude my thesis with a summary and discussion of further work.

I provide additional information in the following appendices:

- Appendix **A** summarizes my publications and their relation to parts of this document.
- Appendix **B** explains the transcription notion used in the examples throughout this thesis.
- Appendix **C** contains the original transcripts of the analyzed pair programming sessions for all the excerpts used in the examples throughout this thesis along with additional information on the developers, their tasks, and their companies.
- Appendix **D** provides technical details on statistical meta-analyses I performed as part of my literature study.

There is also an index for concepts and important terms (page **433**) and a name index (page **441**).



— **Part I** —

# Foundation



## Chapter 2 Related Work

---

*It is well to read everything of something  
and something of everything.*

– Henry Brougham

<b>2.1 Purpose and Structure of this Chapter</b>	34
<b>2.2 Knowledge and Software Development</b>	34
2.2.1 Epistemology: Philosophy of Knowledge	35
2.2.2 Knowledge Concepts in the Cognitive Sciences	35
2.2.3 Knowledge in Software Engineering	36
<i>Two Notions of Expertise in Software Development • Types of Knowledge Relevant for Software Development</i>	
2.2.4 Summary and Definitions	39
<b>2.3 Pair Programming</b>	40
2.3.1 Practitioner Perspective	40
<i>What is Pair Programming? • Industrial Adoption Rates • Expected Effects and Mechanisms • Practitioners' Observations on Knowledge Transfer • Summary</i>	
2.3.2 Overview of Pair Programming Research	48
<i>"Industrial" vs. "Educational" Studies • Properties of Different Research Designs • Approaching the Body of Research on Pair Programming</i>	
2.3.3 Pair Programming in Education	54
<i>Learning Effectiveness of Pair Programming • Pair Compatibility • Pair Process • Summary of Pair Programming in Education</i>	
2.3.4 Pair Programming with an Industry Focus	62
<i>Controlled Experiments on Pair Programming • Studying Pair Programming as a Practice • Qualitative-Quantitative Studies on the Pair Programming Work Mode • Qualitative Analyses of the Pair Programming Work Mode • Summary of Industrial PP Research</i>	
2.3.5 Summary of Pair Programming Research	89
<i>Effectiveness • Types of Knowledge • Task Suitability • Pair Constellations • Pair Process</i>	
<b>2.4 Pair Work and Small Groups</b>	94
2.4.1 Pair Work on Distinct Tasks	94
<i>Joint Decision for Visual Perception Task • Understanding a Complex System • Understanding a Simple System</i>	
2.4.2 Small Groups and Knowledge Processing	97
<i>Coordination and Shared Cognition • Effectiveness • Exemplary Studies</i>	
2.4.3 Summary of Psychological Research on Pair Work	103
<b>2.5 Summary of Related Work</b>	105

## 2.1 Purpose and Structure of this Chapter

The goal of my research is to understand how knowledge transfer works in pair programming in order to advise practitioners in industry. In this chapter, I discuss related scientific literature—mostly, but not exclusively from the field of software engineering.

To clarify what I mean when saying “knowledge” and “knowledge transfer”, I first discuss notions used in philosophy, the cognitive sciences, and software engineering in Section 2.2.

Section 2.3 then covers the body of literature on pair programming, including practitioners’ experience reports as well as scientific studies from industrial and educational settings. I summarize what is already known and what remains to be found out to motivate my research goal, but also characterize the pros and cons of the different employed research methods to inform the choice of my research approach.

For an additional perspective, I discuss studies on pair work from social and cognitive psychology in Section 2.4. These studies use pairs either as the convenient extension of a more difficult-to-study individual or as the smallest of all groups. This research is more theory-oriented than software engineering research and has identified *shared cognition* as a central concept of what makes groups effective. However, it is less concerned with the mechanisms of realistic pair work and more with the outcomes of artificial experimental tasks and falls short of providing detailed explanations of what actually happens when pairs work together.

## 2.2 Knowledge and Software Development

What do I mean when I say “knowledge” and “knowledge transfer”? The purpose of my work is to advise software developers. A common manner of speaking among practitioners is that of *transferring knowledge through pair programming*. Here, *knowledge* appears as some sort of object that can be passed on: Prior to a PP session, only developer A had some knowledge; after the session, it has been transferred to developer B who now also has that knowledge. In addition to possessing it, software developers also acquire and make use of their knowledge. In the following subsections, I briefly survey the literature of three of the many fields that use the term “knowledge”—philosophy, the cognitive sciences, and software engineering—to formulate a preliminary *knowledge*-definition that captures the intuitive notion used by software developers.

A side note: The term “knowledge transfer” is also used in economy and business administration literature but refers to an organizational level, as in transferring knowledge from one department to another. A metaphor used by Davenport & Prusak (2000) is that of a “*knowledge market*” with buyers, sellers, and brokers. Knowledge transfer then occurs in the form of transactions for which managers may establish an environment with opportunities (a marketplace) as well as cultural values and rewards for sharing knowledge (incentives to buy and sell):

“ How can an organization transfer knowledge effectively? The short answer, and the best one, is: hire smart people and let them talk to one another. [...] Organizations often hire bright people and then isolate them or burden them with tasks that leave no time for conversation and little time for thought. [...] [V]arious knowledge transfer issues and strategies [...] come down to finding effective ways to let people talk and listen to one another.

Davenport & Prusak (2000, p. 88)

The focus in the according literature is on establishing such an environment, but not on the interpersonal mechanisms that make up the actual transfer of knowledge—which is precisely my research interest.

### 2.2.1 Epistemology: Philosophy of Knowledge

Philosophers take great care when speaking about *knowledge*. Epistemology is the branch of philosophy concerned with the question what it means to truly *know* something. A common definition of knowledge is based on the three Cartesian conditions (after René Decartes) and commonly abbreviated as “*justified true belief*” (see, e.g., BonJour, 2010, pp. 23–24): Software developer A may be convinced that some class Foo extends the class Bar (belief), because she recently used both classes (justified), and class Foo actually happens to extend Bar (true).

For my purpose, that is, to understand how industrial software developers deal with what they know and do not know during pair programming, the notion of knowledge as justified true belief is impractical as it emphasizes aspects of *knowledge* that are irrelevant to me. In particular, there are the following four reasons why I do not build upon this definition:<sup>1</sup>

- Developer A may not actually have such a justification, but still be convinced (true belief). For knowledge transfer in a pair programming session, developer B may not ask for a justification, and still believe the fact, e.g., because developer A is more experienced. From a researcher’s perspective, distinguishing the cases of not having a justification and not providing one could only be done through asking developer A, which is impractical, especially when data is analyzed a long time after it was collected. For the developers, such a distinction may not be relevant at all.
- Developer A may be less than absolutely certain (i.e., she is not really maintaining a belief), which might be nevertheless good enough for her for practical purposes.
- Knowledge is not neatly compartmentalized as a collection of isolated facts which are independently true or false. Instead, some aspect of what developer A believes may be objectively false, which does not necessarily make what she believes to be true useless for her software development task.
- In addition to explicit knowledge developer A is able to verbalize, there is also know-how, which can be tacit knowledge (see next section). This is arguably relevant for software development but would not be appropriately characterized as “belief”.

The above definition is mostly about the properties that make the difference between ‘truly knowing’ and ‘just being convinced’ that Foo extends Bar. It is *not* about the difference between ‘knowing about Foo’s and Bar’s relationship’ and ‘having no idea’, and not about the practical implications of this difference.

### 2.2.2 Knowledge Concepts in the Cognitive Sciences

The cognitive sciences are concerned with cognition, i.e., how human beings perceive, make sense of, and behave in their environment. Since their inception with the “cognitive revolution” in the 1950s and 60s—accompanied by the advent of computers in academia—the cognitive sciences have been framing the human mind as an information-processing unit, comprising memory, programs, and central processor to run them (Weisberg & Reeves, 2013, pp. 13–19). In modern understanding, all cognitive processes are knowledge-based, including ‘lower’ processes such as pattern recognition and attention (*ibid.*, pp. 38–40). Where to direct one’s attention depends on pre-existing knowledge about the world, as does reading letters and words from a piece of paper. Here, I discuss some of the developed terminology which has also been adapted in other fields (including software engineering) and which I will use to delimit my notion of knowledge.

<sup>1</sup>This does not even cover the “*Gettier Cases*” (named for the three-page paper by Gettier, 1963) where the justification is wrong but accidentally leads to a true belief (see BonJour, 2010, pp. 39–45).

Memory Systems				
Declarative or Explicit Memory		Non-Declarative or Implicit Memory		
Episodic Memory	Semantic Memory	Procedural Memory	Priming	Conditioning

**Figure 2.1:** Overview of memory systems (based on Weisberg & Reeves, 2013, Figure 2.19)

In the cognitive sciences, there appears to be no clear-cut distinction between *knowledge* and *[a] memory*: Both are used as labels for information that is retained in a memory system (sometimes also called *[the] memory*) where it can affect future perception, decision making, or behavior (Weisberg & Reeves, 2013, pp. 90 & 178). There are a number of distinctions of memory systems, which are empirically backed-up by studies with patients suffering from different types of memory loss, including the following two (*ibid.*, p. 85, see also Figure 2.1):

- **Declarative** or explicit vs. **non-declarative** or implicit memory: The content of declarative memory can be consciously accessed and verbalized, whereas non-declarative memory can affect perception, behavior, and decision-making, too, but is not directly accessible.
- **Semantic** vs. **episodic** memory: Both are part of the declarative memory, one comprising general facts (not necessarily on a syntactic level, but more meaning-oriented, thus *semantic*), the other ‘storing’ one’s own personal historical events or *episodes*.

The different memory systems serve different purposes and the stored information is labeled accordingly, e.g., *semantic knowledge* or *procedural knowledge* (*ibid.*, pp. 83–84). The knowledge from non-declarative memory is also sometimes called *tacit knowledge* as it cannot be verbalized by the one who possesses it. Overall, the notion of *knowledge* from the cognitive sciences is rather broad. It covers semantic knowledge such as that Paris is the capital of France, episodic knowledge such as how one ate breakfast this morning, and procedural knowledge such as how to tie your shoelaces, how to recognize your spouse’s voice or the letter K. It also includes other types of implicit knowledge such as the amnesiac patient who cannot explain why she would not shake the hand of the doctor who pinched her the day before (conditioning, *ibid.*, p. 91) or who has forgotten she just read the word “attach” but still completes the stem “at \_\_\_\_” accordingly (priming, *ibid.*, p. 90).

A common notion to describe knowledge organization in memory are **schemata**, which represent abstract forms of concrete experiences that allow to improve cognitive abilities (*ibid.*, pp. 101–103). Widely known are the chess masters who are good at recalling rule-conforming chess piece positions, but not random placements. Other experiments show that Scrabble players identify letter combinations as words faster, musicians recall longer sequences of notes, and visual artists recall more picture details (*ibid.*, pp. 105–107). In all cases, the “experts” are said to *know* basic patterns from their domain and only need to remember the differences to the current stimulus.

### 2.2.3 Knowledge in Software Engineering

It appears to be common understanding in software engineering that software development heavily relies on knowledge. Curtis (1984) sees the most important individual differences between developers that affect overall productivity in their “*knowledge bases*”. Robillard (1999) and Armour (2000) argue that all software development is, in essence, acquiring and codifying knowledge. However, the exact nature of this so-important knowledge remains less clear.

Robillard (1999) distinguishes two types of knowledge and argues that students of computer science mostly possess semantic knowledge from textbooks and courses but lack practical experience in a real-world application domain.<sup>2</sup> A similar distinction can be found in the grounded theory of software developer expertise by Baltes & Diehl (2018) who distinguish *knowledge* and *experience* as the two main components of expertise. Considering the terminology from the cognitive sciences and the examples given in their article, I would roughly map these two to explicit *semantic knowledge* and to non-verbal *procedural knowledge*, covering programming language details, algorithms, data structures, and programming paradigms, etc., and the mental remnants of, for example, having built both small and large systems, having worked on shared code, respectively (*ibid.*, Sec. 3.3).

Thus, the notion of *expertise* appears to be tightly coupled to that of *knowledge*. As Sonnentag et al. (2006, p. 375) explain in the *The Cambridge Handbook of Expertise and Expert Performance*, two notions of programming or software development expertise are used in the literature: Expertise in terms of long years of practical experience and expertise as high performance.

### 2.2.3 a) Two Notions of Expertise in Software Development

One conception of expertise relates to the idea that developers accrue more and more knowledge over the years. In this mindset, asking “*What is it that expert programmers know that novice programmers don’t?*” (Soloway & Ehrlich, 1984) is a reasonable question. One type of such knowledge appears to include *schemata* (see Section 2.2.2). Experiments by Shneiderman (1976) demonstrated advantages of more experienced developers in recalling source code line by line, the effects of which were less pronounced when the original statements were shuffled. Soloway & Ehrlich (1984) then later hypothesized some of these schemata and called them “*programming plans*” and “*programming discourse rules*”. They designed fill-in-the-blanks programs which experienced programmers could more easily complete correctly if the program was schema-conforming while their performance almost dropped to the level of the novices for non-conforming programs. The common explanation goes like this: If a developer encounters source code that matches one of the learned schemata, she can use top-down reasoning and memorize larger, semantic chunks; without a matching scheme, she has to resort to bottom-up reasoning and syntactic memorizing.

While studies with the mindset of expertise-as-experience appear to be concerned more with programming and isolated algorithmic tasks, the conception of expertise as high performance entails a perspective which pays more attention to software development involving larger systems and richer contexts. It emphasizes the specificity of the situation a software developer finds herself in. As Sim & Holt (1998) discuss, new team members or “*software immigrants*”, even if they have good general software development experience, first need to “*naturalize*” in their new environment and learn the peculiarities of the project, e.g., by working on narrow tasks first or getting mentored. Zhou & Mockus (2010) define “*developer fluency*” as the ability to work on every task in a project “*accurately and rapidly*”. Across ten projects they

<sup>2</sup>Robillard (1999, p. 88) explicitly employs the vocabulary of the cognitive sciences but is mistaken at one point. His characterization of practical experience (e.g., “*reusing a function*” or “*defining objects from specification requirements*”) as “*episodic knowledge*” is inappropriate, since episodic memory is part of explicit/declarative memory, whereas practical know-how is mostly part of (implicit) procedural memory—as Robillard himself explains earlier on the very same page. Remembering a concrete *episode* of how one reused a function in the past (i.e., *episodic knowledge*) is not as relevant as having done it multiple times and possessing the *procedural* knowledge to do it again or the *semantic* knowledge to explain it.

showed that it takes about three years until a software developer is fluent in a project, i.e., able to work on even the most difficult and critical tasks.

So, software developers become *experts* (one type or the other) at least in part through their knowledge. But what kind of knowledge is that and how do they acquire it?

### 2.2.3 b) Types of Knowledge Relevant for Software Development

Programming schemata appear to be tacit knowledge (Soloway et al., 1982), meaning that they cannot be easily verbalized, which implies that they are acquired through practice rather than explanation. Other strands of research allow to circle in on the question what other types of knowledge, possibly explicit knowledge, developers need to possess.

Zhou & Mockus (2010, Sec. 4.1) are not explicit about what knowledge developers need to acquire over the three-year period until they are *fluent* in their project, but some ideas can be reconstructed from what the interviewed managers said makes tasks in their project difficult and critical: Technology, domain/application, necessary interaction with or impact on other developers and/or customers, impact on other parts of the system or the system's architecture.

A survey by Li et al. (2015) sheds a bit more light on what types of knowledge are relevant for being a “*great software engineer*”. The respondents—59 (very) experienced engineers from 10 different Microsoft divisions—would agree with Robillard's separation of semantic and non-declarative knowledge as they felt that “*book knowledge*” is not sufficient. The list of additional knowledge areas corroborates what Zhou & Mockus' interviewees emphasized: Great software engineers need to be knowledgeable about the people and the organization around them, about their technical domain, product, and competitors, about customers and business values, about their tools, and about development processes and practices (*ibid.*, Sec. IV.B).

The interview-based studies (Zhou & Mockus, 2010; Li et al., 2015; Baltes & Diehl, 2018) do not explicitly discuss to which degree the knowledge they refer to might be characterized as *semantic knowledge* (which can be verbalized) or as non-verbal *tacit knowledge*. In all these studies, the *effects* of possessing knowledge were more important—developer fluency, the ability to make effective decisions, and better quality of source code, respectively—which can be achieved by semantic and tacit knowledge alike. Additionally, the interviewees in the above studies provided personal accounts of what they *felt* was relevant, which does not necessarily correspond with what actually matters in everyday software development.

In this vein, Sillito et al. (2008) analyzed what precisely software developers want to know while they perform a change task. They recorded 12 sessions of students working in pairs for 45 minutes on assigned change tasks in an unfamiliar open-source project and 15 sessions of (mostly solo) professionals who were asked to think aloud while working for 30 minutes on their normal tasks. The researchers focused on “*questions targeting the code base*” and distilled a catalog of 44 question types. Among these questions are both static concerns such as #8 *Where does this type fit in the type hierarchy?*, which a *fluent* developer could be expected to know, and dynamic concerns such as #31 *Which execution path is being taken in this case?*—strictly speaking not properties of the code base, but of the running system with specific inputs—which even an expert would not usually be expected to know right away all of the time. Consequently, Sillito et al. discuss for each of the question types how well existing tooling supports these information needs, concluding almost full support for the isolated and static question types and only partial support for the complex and dynamic question types (*ibid.*, Table 9).

Overall, Sillito et al. (*ibid.*) focused on information which is already encoded in the code base and which tools may help extract. But, as Armour (2000) puts it, the software is only a medium for storing knowledge which developers had to acquire in the first place. The tool of Fritz et al. (2010) consequently goes beyond what it is already ‘in the code’ and helps identifying

a knowledgeable colleague by automatically quantifying their respective knowledge levels based on fine-grained code authorship and interaction information, thus acknowledging that an important source for developers' everyday information needs are other developers.

To come back to the original question on the notion of knowledge: There appears to be no established conception of *knowledge* in software engineering that goes beyond the terminology from the cognitive sciences. In software engineering, *knowledge* is something that gets the job done. It may be explicit or tacit knowledge; it may be already 'in the code' and one might provide tools to get it out, or there is another individual who carries it around in her head who may get asked for explanations or for help.

My research is primarily about the latter case where a knowledgeable colleague is the main source of information. However, both developers 'extracting' knowledge from existing source code together is also an important aspect of many pair programming sessions. In Chapter 9, I will call these cases **Push/Pull** and **Produce**, respectively. Regarding the types of knowledge that are actually relevant in pair programming sessions, I identified two main types of explicit knowledge: System-specific **S knowledge** and generic **G knowledge** (see Section 7.3.1).

#### 2.2.4 Summary and Definitions

In philosophy, knowledge is (a certain type of) awareness of information. In the cognitive sciences, knowledge is information retained in memory such that it can affect future behavior and perception (which the knowing person is neither necessarily able to verbalize nor necessarily aware of). In software engineering, knowledge is what is relevant for working on software development tasks—be it tacit know-how or factual information.

Considering my goal of advising practitioners, I limit my research to explicit knowledge, i.e., knowledge that can be verbalized. What developers do (or can) explain during pair programming is explicit knowledge. What developers (can) do, e.g., coming up with design ideas or having hunches where to look during debugging, is at least in part enabled by tacit knowledge. Any rationales developers provide for their decisions and behavior are also explicit knowledge (even if they are post-hoc rationalizations), but I do not dig deeper into what enables developers, thus excluding tacit knowledge.

My definition of *knowledge* therefore builds on a notion of *information* that is closer to computer science than to the cognitive sciences. I exclude information that is retained in a software developer's memory but which she cannot consciously access and verbalize.

In a strict sense, knowledge as information *stored in one's memory* cannot be transferred—only information in the form of an utterance or a signal can, which might end up as knowledge once another person understands it. To simplify terminology, I gloss over the difference between knowledge and information from here on:

##### Definition

**Knowledge** is *information* (i.e., a proposition pertaining to a part of the shared reality such as an object or an event) which the pair members consider *relevant* for their software development context, including their current task and the software project it is embedded in. Knowledge in this sense may be lacking by either of the pair members.

Acquiring knowledge from a knowledgeable partner or acquiring it from source code—together or alone—goes hand in hand in actual pair programming sessions (see Chapter 9). My broad definition of knowledge transfer reflects that:

**Definition**

**Knowledge Transfer** is then any attempt of the developers to close a gap in knowledge, either by *exchanging* knowledge they already possess or by *acquiring* new knowledge they still lack.

Note that in this sense, knowledge transfer refers to a process between developers and not—e.g., as Sonnentag et al. (2006, p. 378) use the term—to the application of something learned in one context to another one.

## 2.3 Pair Programming

I approach the topic of pair programming by taking a practitioner’s perspective first (Section 2.3.1): What does *pair programming* actually mean to software developers, how do they think it works, what benefits do they expect? In Section 2.3.2, I give a primer on PP research by characterizing the different contexts in which and the methods with which it is generally studied. I then discuss the results and methodology of two largely separate areas of research: PP in education and PP with an industry focus (Sections 2.3.3 and 2.3.4).

### 2.3.1 Practitioner Perspective

Software developers pair-program. Although reliable numbers are difficult to get by, it is safe to say that one third of software developers at least sometimes works in pairs (I discuss industrial adoption rates below). There are a number of assumptions and expectations regarding the effects of pair programming and how they come to be. Much of the practitioner literature lacks a clear terminology, so I attempt to establish one in this section.

#### 2.3.1 a) What is Pair Programming? – Work Mode and Practice

For my discussion, I distinguish two different meanings of “pair programming” (PP). First, there is PP as a *work mode* (sometimes also called a *programming style* or *technique*): Two developers work closely together on a single technical task. Pair programming in this sense is a decision made for an individual task. This idea can be traced back to the very beginnings of programming (see Section 1.1). The second meaning of “pair programming” pertains to a *practice* which is part of a software development process. This notion was established by Beck (1999) in the context of Extreme Programming (XP) and it is based on the idea that the routine application of the PP work mode has effects beyond single tasks.

Not long after XP was introduced, Williams & Kessler published their book *Pair Programming Illuminated* (2002) which builds on Beck’s ideas and blends research results with practical advice on how to pair-program. In their discussions, Beck (1999) and Williams & Kessler (2002) mix aspects pertaining to pair programming as a work mode and as an established practice in a process. In this section, I separate these two notions more clearly. (For brevity, I refer to the two editions of Beck’s book as *XP1* and *XP2* below.)

When I refer to pair programming *as a practice*, I refer to the strategic decision of developers, teams, and/or managers to work in pairs on some or even all tasks. The pair programming *work mode* then is any occasion where two software developers actually sit down together as the result of a tactical decision, either in an ad hoc manner or as the concrete form of the practice. Note that the two notions are somewhat independent as teams may have agreed on the practice without ever actually engaging in the work mode.

**Work Mode: How and Why to Work as a Pair?**

Beck describes two roles in a programming pair: One developer with keyboard and mouse who thinks about the current implementation, the other thinking about the overall approach (XP1, p. 58). Williams & Kessler (2002, pp. 3–4) went on to call the roles “driver” and “navigator”<sup>3</sup> with distinct responsibilities of writing code and looking for defects as well as strategic problems, respectively. As empirical research has later shown, this notion is misleading (see discussion on pages 72, 73, and 83). In fact, Beck himself already characterized the pair programming work mode as a “dialog”: “It isn’t one person programming while another person watches” (XP1, p. 100). Pair programming could not be done without communicating (XP1, p. 30), it is a “subtle” but “learnable skill” (XP1, pp. 100 & 141).

Beck expects this work mode to have positive effects on the **quality** of the source code (fewer defects and better design) and on the **effort** required to complete a task, as well as on the developers’ **knowledge levels** and **satisfaction**. Behind these effects, he sees a number of mechanisms which he alludes to throughout his XP books (although he does not make an explicit *mechanism-effect* distinction). Williams & Kessler (2002, pp. 21–30) later provided a number of labels which happen to map to Beck’s ideas quite well:

- **PAIR REVIEWS:** Pairs are less likely to make mistakes. This saves debugging time while completing the task, and it increases the source code quality (XP1, pp. 66–67; XP2, pp. 35, 94, & 98). In fact, Beck introduced the idea of pair programming in XP as *code reviews taken to an extreme level* (XP1, p. xv).
- **PAIR NEGOTIATION:** Pairs produce more ideas, and in the process of agreeing on one way to proceed they are forced to clarify uncertainties, which leads to clearer ideas, better design, and ultimately higher quality (XP2, p. 42).
- **PAIR COURAGE:** Pairs are more courageous when it comes to difficult refactorings, again leading to a better design (XP1, pp. 66–67; XP2, p. 35).
- **PAIR PRESSURE:** Pairs are less likely to abandon good development practices and they stay focused on the task, improving the quality and reducing the time spent until the task is completed (XP1, pp. 67 & 102; XP2, p. 42).
- **PAIR DEBUGGING:** Pairs are less likely to get stuck on a problem, which lowers frustration and makes PP satisfying (XP2, p. 42).
- **PAIR LEARNING:** Working on a task together results in conversations and consequentially learning about the particular software system and its parts as well as software development in general (XP1, p. 102).

Note that while the expected effects have some empirical support in scientific studies (which I discuss in detail in Sections 2.3.3 and 2.3.4), the underlying mechanisms are mostly plausible ideas based on observations of reflective practitioners, but were not subject to scientific scrutiny. My own research supports the existence of the PAIR LEARNING mechanism as characterized above (see Chapter 7).

**The Practice: Embedding the Work Mode in a Project**

The above mentioned effects relate to pair programming as a work mode. Pair programming as a practice is then a strategic decision for *when* and *how* to employ the work mode in a project. For Extreme Programming, Beck proposes the following rules:

- **When?** Pair programming is the default work mode in which all long-lived code is produced in order to routinely get the mentioned benefits (XP1, p. 54; XP2, p. 57).

<sup>3</sup>Earlier names were “driver and non-driver” (Williams, 2000, p. 3) and “driver and observer” (Williams, 2001).

- **How?** Pairs are not statically assigned, but rotate frequently in the team—after fixed intervals or at natural breaks in the process—and developers chose partners based on recent experience in task-relevant areas (XP1, p. 59; XP2, pp. 42–43).

The effects of such a PP practice build on the work mode effects: Through knowledge being transferred between individuals in PP sessions (work mode level), knowledge may spread in the team when the pairs rotate (practice level). Or as Beck puts it, “*the right communications*” in the team keep flowing such that the developers learn about the system’s code as well software development in general (XP1, pp. 30 & 102). To Beck, pair programming is the central practice in XP as it “*ties the whole process together*” (XP1, p. 97).

### **An Important Difference**

Beck’s XP books are not about how to actually implement the practices (see XP1, p. xvi). However, Beck’s formulation to write all long-lived code in pairs stuck, and through the popularity of Extreme Programming, the label “pair programming” received the connotation of ‘*for everything, all the time*’. Some criticize that XP “*mandates*” pair programming for everyone (e.g., Stephens & Rosenberg, 2003, p. 137). Here is what Beck actually has to say to skeptics of pair programming:

““ In my experience, pair programming is more productive than dividing the work between two programmers and then integrating the results. [...] All I can say is that you should get good at it, then try an iteration where you pair for all production code and another where you program everything solo. Then you can *make your own decision*.”

Beck (XP1, p. 101, emphasis added)

When talking to practitioners about “pair programming” in industry, it can make a difference whether the framing is ‘*Do you practice pair programming?*’ or ‘*Do you occasionally sit on one machine to work together on certain tasks?*’. These can be two very different things as the anecdote in Figure 2.2 and survey results on PP adoption rates (see below) illustrate. Scientific studies also have an implicit focus on either PP as a work mode (e.g., in controlled experiments, see Section 2.3.4a) or PP as a practice (e.g., in observational field studies, see Section 2.3.4b).

#### **“We don’t practice pair programming”**

We once visited a software development company to explore possible ways to conduct empirical research in an industrial setting. The head of development assured us beforehand that they *do not practice pair programming*. At the company, we had to wait for some time before the scheduled meeting, and sat down among the developers. After five minutes, one developer called a colleague for help with some problem. The colleague interrupted his work and moved over. Soon enough, both developers engaged in a vivid discussion and took turns on the keyboard. After about 25 minutes, the head of development was ready for our meeting—the developers were still working together on one machine.

**Figure 2.2:** Anecdote by Salinger (2013, p. 21, original in German). The head of development here is not necessarily oblivious of what happens in his department. It is quite possible that he referred to “pair programming” as a prescribed practice, and not an occasional work mode.

### **2.3.1 b) Industrial Adoption Rates**

While the above mentioned mechanisms and effects appear plausible, an empirical validation is still necessary to determine the value of pair programming. One way to assess actual effects rather than just expected benefits is to turn to economic settings, expecting that poor ideas

Survey	Subjects	$n =$	Scope	Scale	Rate
Cusumano et al. (2003)	“industry contacts”	104	project	binary	36%
Begel & Nagappan (2007)	developers, testers, managers	487	team	rating	36%
Salo & Abrahamsson (2008)	developers, managers	35	project	rating	43%
Schindler (2008)	developers, managers	61	project	binary	46%
StackOverflow (2018)	developers	57 075	individuals	binary	28.5%

**Table 2.1:** Reported pair programming adoption rates in industry. The surveys differ in who was asked (subjects) about pair programming usage on which level (scope) with what kind of question (scale). Adoption rates coming from rating scales represent the share of those who answered *sometimes* and above—Begel & Nagappan (2007) used levels *yes, sometimes, planning to, no, never will*; Salo & Abrahamsson (2008) used the levels *systematically, mostly, sometimes, rarely, no*.

will not be widely adopted and eventually discarded by developers. Possible questions could be: How prevalent is pair programming in the industry? How often do developers employ it, and why?

For individual teams, reported percentages of developer time spent working in pairs range from 8% to 90% (see Table 2.2). But these numbers only refer to teams who work in pairs at all. What about software development industry as a whole?

In a number of surveys, pair programming adoption rates in industry range from 36% to 46%. In the *2018 Stack Overflow Developer Survey*, the number is as low as 28.5% (see Table 2.1). These numbers, however, are difficult to interpret for two reasons:

First, the pair programming adoption rate is operationalized differently in these surveys.

Some surveys considered whole projects or teams as a unit, while others asked for individual usage. Some surveys asked developers who can give first-hand accounts, others included responses of managers as well. Some asked for a binary value as in ‘*Do you use pair programming?*’ which could mean anything from daily PP to once a month, others used a rating scale for more nuanced answers. The resulting numbers may roughly represent a share of developers who sometimes pair program, but not a proportion of development time actually spent in pairs, leaving the question of ‘*How much PP, overall?*’ more or less unanswered.

Second, since the distinction between pair programming as an occasional work mode and as an established practice is not commonly made—at least not explicitly—the framing of the survey questions is important for interpreting the answers (again, see Figure 2.2). Some particular survey results might be explained through a conflation of work mode and (mandatory) practice:

- In the questionnaire used for the Stack Overflow 2018 survey, pair programming was framed as a “*methodology*” among other options such as “*Scrum*”, “*Lean*”, or “*ISO 9001 or IEEE 12207*”, evoking the idea of systematic usage as a practice and thus introducing a bias

Source	Ratio
Coman et al. (2008, Sec. 3.1 & 4)	< 8%
DeMarco & Lister (2013, p. 61)	50%
Vanhanen & Korpi (2007, Tab. 7)	55%
Hulkko & Abrahamsson (2005, Fig. 3)	40–90%*

**Table 2.2:** Reported percentages of developer time spent working in pairs. I calculated the ratios for Vanhanen & Korpi (2007) and Coman et al. (2008) based on information provided in their papers; see page 70 for details. Hulkko & Abrahamsson (2005) only report the pair portion of *programming* time, not overall work time.

against counting the occasional work mode. A hypothetical developer who pair-programs two hours a week in an ad hoc manner (i.e., not as a planned practice) may be discouraged to say ‘Yes, we use pair programming’ in such questionnaires. (More recent Stack Overflow developer surveys from 2019 and 2020 did not ask for pair programming.)

- Around 63% of Microsoft developers with recent PP experience state that “*pair programming is working well*” for them and their partner, but only 48% said so with regard to their team, and only 39% for their larger group (Begel & Nagappan, 2008, Fig. 3). The researchers see this as an indication of a “grassroots” phenomenon with individuals who adopt it, but who face difficulties convincing their management “to spread the practice”. Possibly, the respondents thought of an ad hoc work mode for themselves and their partner (high agreement), and of a mandatory practice for their team and larger group (lower agreement). A valid interpretation of these numbers, however, is not possible without understanding what it even means (to the respondents) that “*pair programming is working well for my team*”/“*my larger group*” (ibid., Fig. 3).
- Among the reasons why pair programming is “not adopted” in some companies, Schindler (2008, Table 10) lists “only with complex code”. One way to interpret this (paradoxical) statement is that programming in pairs is indeed an employed work mode—even if “only with complex code”—but it is “not adopted” as a systematic practice.

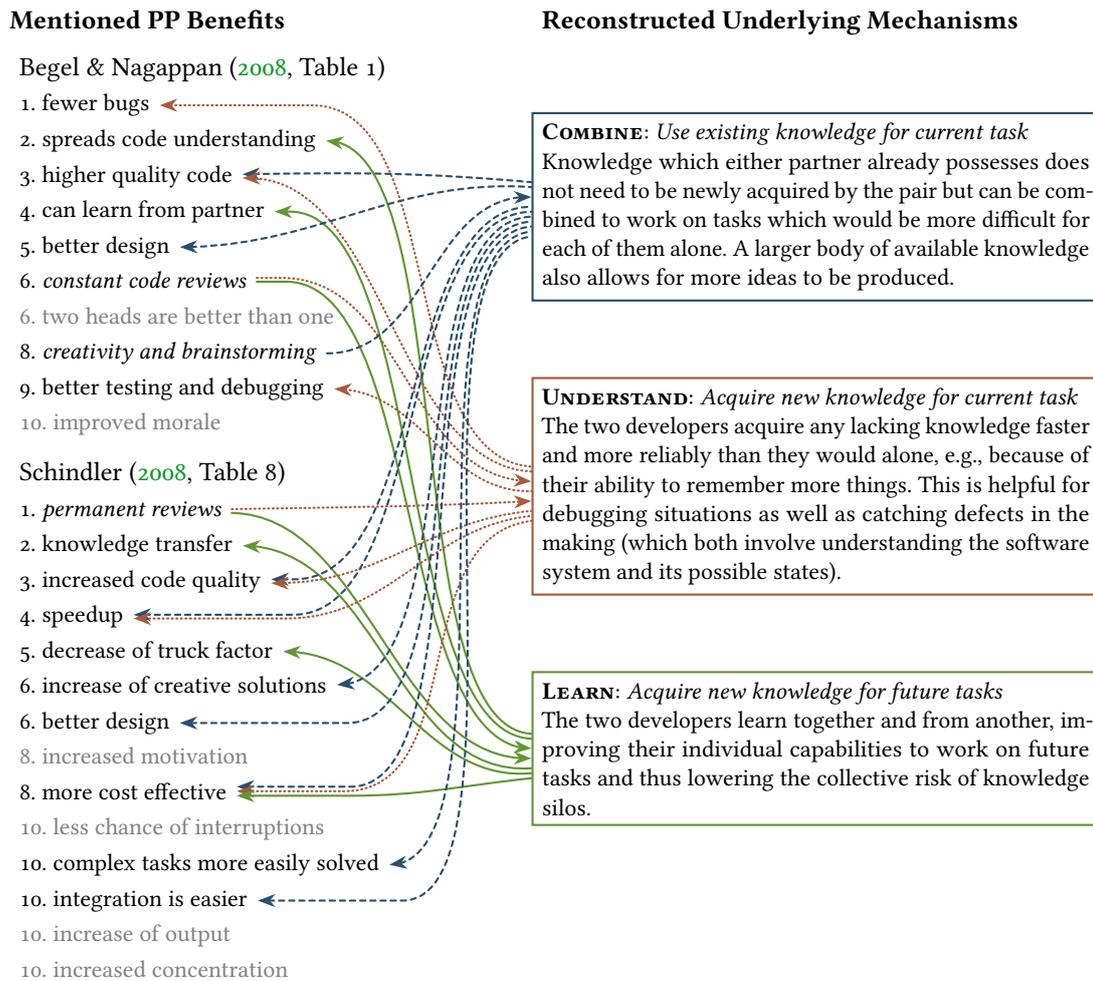
Considering that some survey respondents may have understood “pair programming” as a systematic practice and hence stated that they do *not* use it even though they do occasionally engage in a pair work mode, the reported adoption rates can be regarded as a conservative estimate of the actual adoptions rates: Roughly **one third** of professional software developers appears to **work in pairs** at least sometimes. This is reason enough to assume that there is at least some actual positive effect to this work mode, and that finding ways to improve PP can help *many* developers.

### 2.3.1 c) Expected Effects and Mechanisms

Some surveys cover the developers’ reasons to pair program, which puts relative weights to the expected benefits mentioned by Beck (XP<sub>1</sub>, XP<sub>2</sub>) and Williams & Kessler (2002). Begel & Nagappan (2008) and Schindler (2008) both report similar results with “fewer defects”, “knowledge transfer”, and “higher quality” at the top of their respective lists of PP benefits mentioned by practitioners. The relative importance of these aspects, however, is not conclusive: Entries such as “permanent review” and “higher quality” are mentioned in both lists as separate and independent PP benefits, indicating that the authors did not differentiate between *mechanisms* and their *effects*. In this section, I make this distinction clear and identify the mechanisms that practitioners and researchers expect to underlie the pair programming work mode.

Among all the nice-to-have benefits ascribed to pair programming, some appear to be the prime reasons why developers choose to work in pairs. As reported by Hulkko & Abrahamsson (2005, Sec. 3.2.1) who studied four teams for 5 to 8 weeks, developers found pair programming to be especially useful for spreading system understanding, for complex tasks with many dependencies, and for demanding tasks which no individual could do alone. Companies such as Pivotal Labs have been employing *continuous pair programming* (i.e., 100% pair programming) and *overlapping pair rotation* for two decades to spread knowledge, in particular about the software systems, to remove and avoid knowledge silos in their projects (Sedano et al., 2016). Plonka et al. (2015) showed that transferring knowledge from an “expert” to a “novice”, as opposed to, e.g., completing some functionality, is the main purpose for some industrial PP sessions.

Overall, knowledge and knowledge transfer appear to be of major importance to practitioners. Indeed, many of the expected pair programming benefits directly or indirectly depend on the pair members' individual and collective knowledge. This may be obvious for benefits like “*can learn from partner*” (Begel & Nagappan, 2008, Table 1), but also holds for others. In the following Figure 2.3, I reconstruct three knowledge-related mechanisms which survey respondents and researchers seem to assume. Software developers appear to pair-program because they expect one or more of these to yield positive effects: COMBINE their existing knowledge, UNDERSTAND the rest together, and LEARN together and from another.



**Figure 2.3:** Reconstruction of three knowledge-related mechanisms (right side) based on corresponding positive pair programming effects that practitioners report in surveys (left side, inward arrows). The mechanisms capture how the respondents (and the researchers) apparently think that pair programming works. Some mentioned benefits are actually part of the mechanisms rather than effects (outward arrows, text set in *italics*); others are vague or do not correspond to a knowledge-related mechanism (no arrows, *grayed out*).

LEARNING from another is what is commonly called *knowledge transfer* in the context of pair programming. Since LEARNING new things together as well as the COMBINE and UNDERSTAND mechanisms also involve the exchange, acquisition, and integration of information and ideas, I consider them **knowledge transfer** in a broader sense. I will use the above identifiers as shorthands to refer to the expected mechanisms throughout my discussion of related work.

### 2.3.1 d) Practitioners' Observations on Knowledge Transfer

Before I dive into scientific studies, I first turn to what practitioners have to say in surveys and experience reports. Surveys are suitable for capturing respondents' beliefs and opinions, but less for facts. Asked about development practices, some may refer to their own current project while others may think of something a colleague told them. Similarly, experience reports may document interesting phenomena, but often lack methodical rigor and/or details in their writing. Nevertheless, for understanding how pair programming is actually employed, these two formats can help identify relevant phenomena and research questions.

Belshee's (2005) experience report contains two compelling anecdotes of knowledge transfer through pair programming. His team's PP rules are: All the time and "promiscuously", i.e., with frequent partner changes. The first anecdote pertains to a tiny piece of knowledge: In the morning, one pair discovered a keyboard shortcut for accessing a paste stack (a history of recently copied text fragments), which spread across the whole team of 11 developers by the afternoon, supposedly through pair rotation only (*ibid.*, Sec. 2.7). The other anecdote describes the integration of a newly hired developer who—despite not knowing the programming language or even the programming paradigms—lowered the team's velocity for only one week, was able to work alone on any task within three weeks, and even taught the next new-hire in the fourth week (*ibid.*, Sec. 3.1).

Like Belshee's, there are more *informal reports of practitioners*<sup>[a-f]</sup> who reflected on their process in individual projects, typically not long after they introduced pair programming for the first time, but only some which provide details about knowledge or knowledge-transfer related topics. *Surveys among practitioners*<sup>[g-i]</sup> which ask for general perceptions of pair programming also contain relevant observations. Together, these sources point to five practically relevant aspects of knowledge transfer in pair programming:

#### Practically Relevant Aspects of Knowledge Transfer in Pair Programming

- A1 Effectiveness:** Is pair programming effective in the sense that it produces the expected effects? In other words: Do pairs COMBINE their existing knowledge and UNDERSTAND missing parts together to produce code with fewer defects and better design? Do the developers LEARN together and from another?
- A2 Types of Knowledge:** What do the developers COMBINE, UNDERSTAND, and LEARN while pair programming?
- A3 Task Suitability:** Are there some types of tasks for which pair programming is more or less suited?
- A4 Pair Constellations:** Are there some types of pairs for whom pair programming is more or less suited?
- A5 Pair Process:** How do pairs actually COMBINE, UNDERSTAND, and LEARN?

Here is what practitioners report regarding these five aspects:

- A1 Effectiveness:** Experience reports<sup>[a,b,c,f]</sup> and surveys<sup>[g,h,i]</sup> agree that PP is helpful for transferring knowledge both in terms of individual learning and spreading knowledge in a team. In Belshee's paste-stack anecdote, one pair discovered and LEARNED the feature at one point. This is effectiveness of pair programming as a work mode. For effectively spreading knowledge in a team, partner changes should be frequent,<sup>[a,e]</sup> which is why

**Key:** <sup>a</sup>Belshee (2005) <sup>b</sup>Pandey et al. (2003) <sup>c</sup>Rasmusson (2003) <sup>d</sup>Sharifabdi & Grot (2002) <sup>e</sup>Tessem (2003) <sup>f</sup>Wood & Kleb (2003) <sup>g</sup>Begel & Nagappan (2008) <sup>h</sup>Fitzgerald et al. (2006) <sup>i</sup>Vanhanen & Lassenius (2007)

Belshee’s team made ‘promiscuity’ part of their practice. Code quality in terms of defect count, readability, and maintainability is said to be better<sup>[f,g,h,i]</sup>—which can be interpreted as observable effects of COMBINE and UNDERSTAND.

- A2 Types of Knowledge:** PP appears suitable for transferring different types of knowledge including simple tips and tricks (e.g., paste stack), understanding of programming languages, tools, and the software system (e.g., new-hire), as well as domain knowledge.<sup>[a,b,c,f,i]</sup>
- A3 Task Suitability:** Some reports draw a line between tasks that involve system understanding (e.g., those with complex code with many dependencies and/or old code) and rote tasks which can be done as fast as typing.<sup>[h,i]</sup>
- A4 Pair Constellations:** Two pair constellations appear particularly valuable, and both of them indicate the importance of knowledge transfer in pair programming: Junior-senior constellations and pairs with complementary skills with regard to their task.<sup>[b,i]</sup>
- A5 Pair Process:** Some reports provide lists of things to avoid in PP sessions—probably based on reflecting on bad first-hand experience—including not setting a clear objective for the session, making and taking criticism personal, being always right, always agreeing, or not sharing one’s ideas.<sup>[d,e,h]</sup> Belshee (2005, Sec. 1.5) does not see any problems: “[w]hile two people are paired, they share knowledge [...] [K]nowledge transfer is automatic”.

I will use these practically relevant aspects as a structuring element throughout my discussion of the results of empirical studies on pair programming.

### 2.3.1 e) Summary

The purpose of this section was to establish a pair programming vocabulary based on the voices of practitioners. I reviewed and discussed experience reports and surveys, and introduced an array of terminology to make tacit notions explicit:

- The term “pair programming” can refer to a *work mode* which is the (possibly ad hoc) decision to work as a pair on a software development task, or to a *practice* which is a strategic decision for when and how to employ the work mode (see Section 2.3.1a).
- On the one hand, there are *effects* expected by practitioners (including faster progress, finding and preventing defects, better quality, and knowledge spreading) such that about one third of them occasionally engages in the work mode, possibly as a manifestation of a practice. On the other hand, there are purported *mechanisms* behind these effects: COMBINE and UNDERSTAND which affect the technical outcome, and LEARN which affects developers’ ability to work on future tasks.
- There are five practically relevant aspects of the pair programming work mode, which can be considered individually: Its effectiveness, the types of situations (pairs and tasks) for which it is suited, the types of knowledge that are relevant, and the underlying processes that make it actually work.

Researchers by-and-large appear to share the practitioners’ expectations while designing their studies. I hope making these distinctions explicit will prove useful in structuring the discussion of related work and in understanding the status quo of research in this field.

In the next sections, I work through the scientific literature on pair programming in order to inform my research interest: Understand how knowledge transfer actually works in pair programming in industry (see Section 1.3.1). From here on, I focus primarily on pair programming as a work mode, regardless of whether it is done as a tactical decision for only one task or results from a strategic process-level decision.

### 2.3.2 Overview of Pair Programming Research

Pair programming has been a research subject in both industrial settings and computer science (CS) education for many years. These two areas set different priorities regarding the five aspects **A1** to **A5** introduced above. I discuss the relevant differences between them in Section 2.3.2a.

Across the field, researchers use different research designs to study (different aspects of) pair programming. Depending on the research question, different designs are more or less suitable, affecting the credibility of the results. In my discussion of related work, I do not just cite conclusions but also discuss the employed methods for a richer characterization of the state of research. As a primer, I briefly discuss the characteristics of the research methods commonly used in pair programming research in Section 2.3.2b.

In Section 2.3.2c, I describe how I organized my literature study and give an overview of the body of research before I discuss concrete findings in Sections 2.3.3 and 2.3.4.

#### 2.3.2 a) “Industrial” vs. “Educational” Studies

First of all, the notion of “industrial” and “educational” studies needs to be discussed. These two domains are not neatly separated from another, and where exactly to draw the line depends on the *research goal*, the *subjects*, and their *context*.

1. The **research goal** determines the type of research questions which individual studies address. *Educationally* motivated studies are concerned with finding better ways to educate students. They focus on learning achievements, social aspects, and student satisfaction. *Industrially* motivated studies want to inform professional software development. They are interested in gains in product quality and savings in development time and cost.
2. The study **subjects** are sometimes *students*, who are easier to access for many researchers, but may have less experience, e.g., with complex code bases and development processes. Other subjects *professional software developers*, who are more difficult and/or expensive to recruit, but may possess more study-relevant experience.
3. For practical research questions, the subjects need to *do* something for which there needs to be a **context**. On the one end of the spectrum are make-believe environments, such as homework assignments or small projects in *educational* settings (but also coding katas and recruiting sessions in companies). Here, working towards a technical goal is a means, but not the actual purpose (which may be gaining practical experience with theoretically learned concepts, or improving routine programming, or assessing job applicants). The same applies to environments which are specifically designed by a researcher for a study with small and isolated tasks. At the other end of the spectrum are contexts which existed before a researcher entered the scene. Here are *industrial* and other projects with a bigger time frame, more involved people, and more dependencies.

In theory, studies could be positioned anywhere along these dimensions. In practice, these dimensions are not completely independent from another. There might be a natural fit, e.g., for an *educational research goal*, it makes sense to use *students as subjects*. Studies with an *industrial research goal* often rely on student subjects as a “convenience sample”, which is debated in the SE community (e.g., in Salman et al., 2015). Whether or not a particular “industrial” or “educational” study is relevant for my research depends on the three dimensions as follows:

- Although my own research **goal** is industrial, there is no reason to exclude studies with an educational goal—in particular since studies on students’ learning achievements presuppose the LEARN mechanism, i.e., pair programmers learning together and from one another, improving their capabilities to work on future tasks.

- I expect no fundamental differences between student and professional **subjects** for my research. After all, individuals can assume both roles, e.g., as students working during their studies or experienced professionals studying part-time.
- I expect artificial and industrial **contexts** to differ in two relevant ways: The subject's motivations differ (e.g., because they expect to work on similar tasks again in the near future) and the amount and diversity of knowledge necessary to work on isolated tasks is smaller than in large projects. Hence, I focus on studies in realistic contexts, where different knowledge types and the effectiveness of the COMBINE mechanism—instead of just UNDERSTAND—can be studied (aspects **A2** and **A1**). However, I do not categorically exclude artificial tasks, for which the researchers are possibly omniscient. This enables them to assess the effectiveness of the LEARN and the UNDERSTAND mechanism.

### 2.3.2 b) Properties of Different Research Designs

There are different ways to collect data from which to draw conclusions about reality:<sup>4</sup>

- In **controlled experiments**, a situation is described by a number of variables, one or more of them are varied as independent or input variables, while the others are kept the same to measure an effect on one or more dependent or output variables. Quantitative data and statistical analysis then allow to test for causal relationships between the variables. If the measured difference *between the groups* is large enough to become visible against the variation *within* the groups, a statistically significant result may be reported, often in the form of an *effect size*. In my discussion of quantitative studies, I will cite (or, if possible, calculate) relative differences using intuitive scales as *10% faster* or *5% higher exam score* rather than just the standardized effect size which makes studies with different scales comparable, but is less intuitive (see Figure 2.4 on *effect sizes*).

Some studies set the work mode (pair vs. solo) as the input variable, give the same task to all subjects, and measure differences in product quality or time to completion, which could then be attributed to the application of the pair work mode. Other studies let all subjects work in pairs and vary the difficulty of the tasks to test its influence. Either way, controlled experiments can only show causal relationships and effect sizes for already understood and operationalized phenomena (such as time spent, lines of code written), but do not tell which other variables to consider as moderators (such as task complexity, development experience, pair programming skill, personality attributes, office environment, etc.). With relevant moderator variables left uncontrolled, otherwise rigidly controlled experiments yield inconclusive results.

- In **surveys**, subjects are asked for their personal experiences, beliefs, and opinions on a matter. Correlational statements can be made based on such data, but causal relationships cannot be established. There is no guarantee that subjects report truthfully and accurately, so surveys are not suitable to determine facts about the events that actually take place in pair programming sessions, but are sometimes the only practical proxy. However, if the reporting subjects are able to reflect on their practice, they may provide some ideas for relevant aspects (see Section 2.3.1d above).
- In **observational studies**, researchers observe subjects directly or indirectly and attempt to record facts about events as they take place. Such studies may use tools to automatically collect data, e.g., all interactions with a computer program, or rely on a researcher to take fieldnotes. (See Lethbridge et al., 2005, for an extensive overview of many more ways to collect data in the field.) The resulting data tends to be detailed, and allows for

<sup>4</sup>See Stol & Fitzgerald (2018) for a comprehensive overview. I focus on methods used in PP research.

An **effect size** is the strength of the relationship of two variables, say, between developers working either alone or in pairs and the quality of the produced software. There are several ways to express an effect size. The **mean difference  $D$**  is an unstandardized effect size. Such a measure is accessible to intuition if the scale is commonly known. In contrast, a **standardized effect size** is scale-free, which allows comparison across studies. A commonly used measure is Hedges's  $g$ : It is the ratio of the mean difference and the pooled standard deviation corrected by a factor for small sample sizes. The idea is that a large mean difference is less meaningful for noisy data. A third type is the **means' ratio  $R$** , which can be used if the data is on a ratio scale with a natural zero point.

In a hypothetical study comparing assignment scores between a solo and a pairing group of sizes  $n_1 = n_2 = 10$ , with means  $\bar{X}_1 = 9$  and  $\bar{X}_2 = 11$  points, and with standard deviations  $s_1 = 3$  and  $s_2 = 2.5$  points, the three effect sizes pan out as follows:

$$D = \bar{X}_2 - \bar{X}_1 = 2 \quad g = \frac{D}{s_p} \cdot J = \frac{D}{\sqrt{\frac{(n_1-1)s_1^2 + (n_2-1)s_2^2}{n_1+n_2-2}}} \cdot \left(1 - \frac{3}{4(n_1+n_2-2)-1}\right) \approx 0.69 \quad R = \frac{\bar{X}_2}{\bar{X}_1} \approx 1.22$$

The mean ratio  $R$  of 1.22 can also be expressed as a **relative mean difference  $D\%$** : A change of +22% (calculated as  $D\% = (R - 1) \cdot 100$ ). The  $g$  value, however, is less intuitive: It merely states that the two groups' means are 0.69 standard deviations apart. Kampenes et al. (2007) compared 284 effect sizes reported in software engineering research and propose the following categories: An effect with  $0.00 \leq g \leq 0.376$  is *small*,  $0.378 \leq g \leq 1.000$  is *medium*, and  $1.002 \leq g \leq 3.40$  is *large*.

Either type of effect size, generically referred to as  $Y$ , can be reported as a **point estimate  $\bar{Y}$** , which is a single value that comes close to the true effect size, or as a **confidence interval (CI)** that very likely contains the true effect size. The width of the interval depends on the standard error of the effect size, which in turn is the square root of the effect size's variance:  $SE_Y = \sqrt{V_Y}$ . (The formula for the variance depends on the type of effect size.) Assuming the effect size is normally distributed, the boundaries of the 95%-CI are given by:  $\bar{Y} \pm 1.96 \cdot SE_Y$  (with 1.96 being the area under the standard normal distribution between the 2.5 and 97.5 percentile). In the hypothetical study, the confidence intervals would be as follows:

$$95\% \text{ CI}_D = [-0.42, 4.42] \quad 95\% \text{ CI}_g = [-0.17, 1.56] \quad 95\% \text{ CI}_R = [0.96, 1.56] \text{ or } [-4\%, +56\%]$$

When the CI does not contain zero, the difference it said to be **statistically significant** at a certain level (here  $\alpha = 1 - 95\% = 0.05$ ). Hence, the difference in the hypothetical study is not significant.

**Meta-analyses** combine the results from multiple studies. The summary effect size  $M$  is the weighted mean of the individual effect sizes  $Y_i$ . If the studies are functionally identical, one may assume that they all estimate the same underlying effect in a **fixed-effect model**. Here, studies with narrow confidence intervals—typically the studies with more subjects—get a larger weight, which is just the inverse of their effect size variance  $W_i = \frac{1}{V_{Y_i}}$ .

If the studies had different populations, followed different procedures, etc., assuming one common effect is not reasonable. A **random-effects model** assumes a (normal) *distribution* of true effect sizes. To include information on different effect sizes, the weights  $W_i^*$  are more balanced, increasing the impact of small studies with wide confidence intervals (and decreasing the impact of large studies with narrow intervals). The CI around a random-effects model summary effect  $M^*$  is always wider than around a fixed-effect model summary effect  $M$ .

**Heterogeneity** is a property of the set of primary studies which gets higher when the primary studies' CIs overlap less. In a fixed-effect model, it can be used to assess the assumption of a common underlying effect; in a random-effects model, it is used to set the strength of the weight balancing, giving far-off studies more impact. Some heterogeneity is to be expected in any meta-analysis, e.g., due to differences in experimental setups and subject selection criteria (Higgins et al., 2003). The  $I^2$  statistic measures heterogeneity as a ratio (the ratio of the not-expected, the so-called *excess variance*, to overall variance) and thus makes different meta-analyses comparable.

**Figure 2.4:** A brief introduction to effect sizes and meta-analyses based on Borenstein et al. (2009), *Introduction to Meta-Analysis*, in particular chapters 4, 8, 11–13, and 16.

qualitative-quantitative analysis (i.e., manually coding phenomena in order to perform statistical analyses on the annotations) and in-depth, qualitative analysis to understand *how* some phenomenon such as pair programming unfolds.

Both industrial and educational studies could, in theory, employ all these approaches, but there are practical considerations involved. Designing a controlled experiment in the context of a large project would be more expensive than an experiment with small isolated tasks, as there have to be (at least) two experimental conditions, such as two otherwise identical projects—or better: groups of projects—which can be compared.

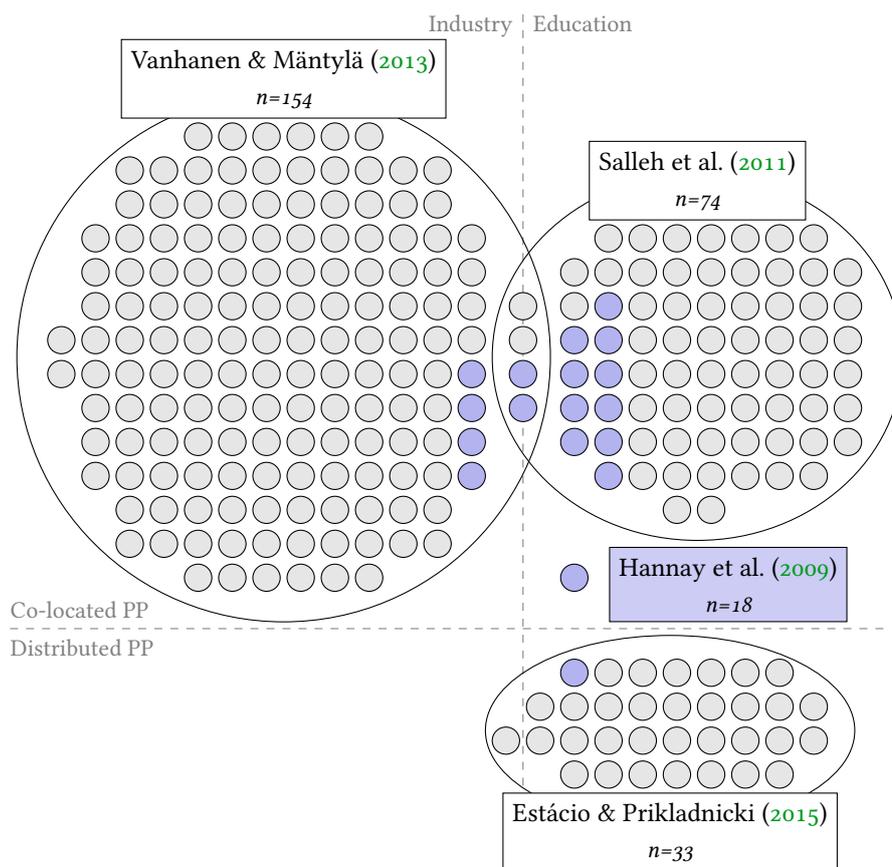
The research approaches differ in how close they are to their phenomena of interest. For studies with an industrial research goal, *controlled experiments* might be tempting because of their seemingly clear results, but at the same time they might be far removed from anything professional software engineers encounter in their daily lives, because of isolated tasks, short time frames, developers not having (or needing) much pre-existing knowledge, etc. (which may be more or less of a problem, depending on the research topic). Here, the experiment is not the real world. In educational settings, however, the setup of a controlled experiment can be quite ‘natural’, e.g., if the students either work in pairs or alone for one semester on their normal homework assignments. *Surveys*, among professionals as well as students, may have a connection to real-world events (e.g., recent experiences of the respondents), but are better at recording beliefs and opinions rather than objective facts about these events. Responses that look like facts may or may not pertain to real events—either way, the researcher usually cannot validate them. *Observational studies* aim for more rigor at capturing aspects or the full breadth of specific real-world events, such as the *cases* in case studies (Yin, 2014, pp. 31–35).

### 2.3.2 c) Approaching the Body of Research on Pair Programming

The existing PP literature is mostly covered by four secondary studies: In terms of the number of primary studies, there are two large ones, Salleh et al. (2011) and Vanhanen & Mäntylä (2013), and two smaller ones, Hannay et al. (2009) and Estácio & Prikladnicki (2015), which together cover 258 different primary studies (see Figure 2.5).

Roughly speaking, Salleh et al. (2011) focused on pair programming in education, whereas Vanhanen & Mäntylä (2013) looked at industrial software development. As discussed in Section 2.3.2a, there is no canonical way to separate these two areas. As can be seen in Figure 2.5, there is in fact a small overlap between the two studies. Hannay et al. (2009) looked exclusively at quantitative results regarding pair programming effectiveness; Estácio & Prikladnicki (2015) reviewed studies on *distributed pair programming* (DPP), i.e., the idea of two software developers working closely together without sharing a physical workspace. The area of DPP were explicitly excluded by the two large secondary studies. The explicit and implicit selection criteria for each of the four secondary studies on pair programming are summarized in Table 2.3. Table 2.4 summarizes the research topics and methods commonly used to investigate the five aspects.

Salleh et al. (2011) provide a systematic literature review (SLR) of studies on pair programming as an educational tool in higher education, which are mostly concerned with the effectiveness of LEARNING through PP and with pair compatibility. Here, effectiveness (A1) is often studied in experimental designs. Salleh et al. provide a quantitative synthesis which I extend with the data from two additional studies, showing an overall positive effect of PP. Pair compatibility (A4) is addressed through experiments and surveys. Here, however, the authors’ qualitative synthesis is not convincing (details follow in Section 2.3.3b), so I searched for further primary studies and provide one of my own. The aspects of relevant knowledge types (A2) and task types (A3) remain implicit: Most of the time, students worked on their homework



**Figure 2.5:** Overview of the 258 primary PP studies surveyed by the four secondary studies. Vanhanen & Mäntylä (2013) focused on PP in industry, Salleh et al. (2011) on PP in education. Because of their hybrid nature, four publications are cited in both overview articles. Estácio & Prikładnicki (2015) focused on distributed pair programming, which the other two large reviews explicitly excluded. Hannay et al. (2009) performed a meta-analysis of experimental studies on PP (colored in purple). Table 2.3 lists their respective selection criteria. This figure is based on the secondary studies’ bibliographies. Salleh et al. (2011) excluded one of their 74 primary studies, but do not mention which one; Estácio & Prikładnicki (2015, Sec. 5) refer to “34 papers” throughout their manuscript, but list only 33 “Papers extracted from the SLR” in their Appendix A.

Secondary Study	Industry/Education			PP Mode
	Research Goal	Subjects	Context	
Salleh et al. (2011)	educational	(students)	(artificial)	co-located
Vanhanen & Mäntylä (2013)	industrial	professionals	both	co-located
Hannay et al. (2009)	industrial	both	(both)	(both)
Estácio & Prikładnicki (2015)	both	(both)	(both)	distributed

**Table 2.3:** Selection criteria used by the four secondary studies on pair programming. The three dimensions to distinguish industrial from educational research are discussed in Section 2.3.2a. Implicit criteria (effective, but not explicitly mentioned by the authors) are put in parentheses.

Aspect	PP in Education	PP with an Industry Focus
Effectiveness (A1)	COMBINE & UNDERSTAND: Code quality [Exp] Duration/effort [Exp] LEARN: Quiz/assignment/exam scores [Exp] Perceived learning/satisfaction [Sur]	COMBINE & UNDERSTAND: Code quality [Exp] [PF] Duration/effort [Exp] LEARN: Perceived knowledge levels [PF]
Knowledge Types (A2)	–	Knowledge relevant for PP [PF] [QQ] Knowledge acquired through PP [PF] Knowledge used/transferred in PP [Q]
Task Suitability (A3)	–	Task complexity [Exp] PP usage per task [PF] PP usage throughout project [PF]
Pair Constellations (A4)	Personality [Exp] [Sur] Skill level [Exp] [Sur]	Personality [Exp] Experience [Exp] [PF] [Q] Knowledge level [PF]
Pair Process (A5)	Communication amount [Exp] [QQ]	Communication amount [PF] Type of communication [QQ] [Q] Pair activity patterns [Q] Cognitive processes [Q] Conflict handling [QQ] Driver/navigator roles [QQ] [Q]

**Table 2.4:** Research topics and methods applied by the publications cited in this chapter which investigate the five practically relevant aspects of knowledge transfer in pair programming. Research designs: [Exp] (controlled) experiment, [Sur] survey, [PF] project-level field study (which considers PP as a practice), [QQ] qualitative-quantitative and [Q] purely qualitative study (both considering PP as a work mode).

assignment and learned about whatever their CS courses were about. The pair programming process itself (A5) was not a topic in the SLR, so I looked for (and found) information in their cited papers and in additional sources. The details of all this I report in Section 2.3.3.

**Vanhanen & Mäntylä (2013)** did a *mapping study* of pair programming in industry which contains no synthesis of the primary studies' findings. Instead, the researchers (1) rated each individual result from each study, (2) grouped the studies under six topics and 18 subtopics, and (3) characterized the quality of what is known concerning each of the (sub)topics. Across all 154 primary studies, Vanhanen & Mäntylä (*ibid.*, Sec. 4.2) identified 608 instances of relevant information concerning these topics. However, 430 (or 71%) of those instances were only rated as “fair”, i.e., consisting of descriptive data from experience reports as opposed to more rigorous case studies or experiments (*ibid.*, Table 12), leaving only few studies with credible findings. I already discussed experience reports and practitioner surveys in Section 2.3.1d, which led to the five aspects (A1 to A5) relevant for my work. There are studies addressing all of them, some referenced and already evaluated by Vanhanen & Mäntylä, others I found myself. The effectiveness of pair programming (A1) is mostly studied through controlled experiments. In contrast to educational experiments which focus on LEARNING effects, industrial experiments are concerned with technical outcomes such as code quality and development effort, which can be interpreted as effects of COMBINE and UNDERSTAND. In most experiments, however, the context is artificial which diminishes the effect of pre-existing knowledge thus effectively robbing experienced developers of any effect from the COMBINE mechanism.

The third secondary study on my list, **Hannay et al. (2009)**, is a meta-analysis of such experiments, from both industrial and educational settings. I extend their meta-analysis with data from additional studies.<sup>5</sup> For aspects **A2** to **A5**, which are mostly addressed through observational studies (using an array of different methods), no synthesis has been done yet, so I used Vanhanen & Mäntylä's mapping study to (1) identify studies which are decidedly on-topic and (2) looked for possibly relevant studies which contain credible observations on any of the five aspects without having made it their main topic. Additionally, (3) I looked for further material published more recently. I present the results of my literature work grouped by research method in Section **2.3.4**.

The SLR by **Estácio & Prikladnicki (2015)** is concerned with distributed pair programming (DPP), a variant of the work mode where the two developers do not physically sit in front of the same computer, but use some tool to collaborate from different desks, offices, or even countries. Both large secondary studies explicitly excluded DPP studies from their literature search (Salleh et al., 2011, Sec. 2.4; Vanhanen & Mäntylä, 2013, Sec. 3.3). Many DPP studies are concerned with bridging the communication gap between the two separated partners and focus on tool support—Estácio & Prikladnicki (2015, Table 5) list no fewer than 11 tools which were specifically developed for DPP. Empirical studies (with one exception) took place in educational contexts, often in the form of experiments and sometimes surveys comparing either distributed with co-located pair programming or DPP with solo work regarding code quality, productivity, and academic performance (**A1**). I discuss the individual studies in the appropriate places in Sections **2.3.3** and **2.3.4**. The studies discussed by Estácio & Prikladnicki do not address the other aspects **A2** to **A5**.

### **2.3.3 Pair Programming in Education**

Salleh et al. (2011) surveyed 74 studies on PP, which they selected based on their research goal to inform educators in higher education. All of these studies—naturally—were conducted in educational settings and used students as subjects.

According to Salleh et al., PP research in education has two main topics. The first concern is determining the *effectiveness* of PP as a pedagogical tool. These studies compare pair programming against solo programming in the context of homework assignments or some larger project. They are ultimately interested in the effectiveness (**A1**) of the LEARN mechanism, i.e., effects of PP on the individuals' knowledge. These effects are measured either directly through quizzes, assignment scores, exam grades, or through students' self-assessments and satisfaction in comparison with non-pairers. Roughly speaking, knowledge is regarded as an *outcome* of pair programming in this type of studies. I discuss them in Section **2.3.3a**.

In Section **2.3.3b**, I discuss the second topic, which is that of *pair compatibility* (**A4**) and its influence on the effectiveness. These studies do not compare pair programming with solo programming, but focus on the influence of students' skill levels, and their learning and communication styles on their pair programming sessions in terms of session effectiveness, satisfaction, and self-assessed pair compatibility. Here, at least for studies interested in the role of student's skill levels, knowledge is regarded more as an *input* of pair programming sessions.

The aspect of the actual pair programming process (**A5**) is not explicitly addressed in the SLR. In Section **2.3.3c**, I discuss what researchers in education tacitly appear to expect to happen in pair programming and the few studies which actually looked into it.

---

<sup>5</sup>Note that I did not conduct a structured literature search to include all quantitative PP studies that were published after Salleh et al.'s and Hannay et al.'s periods (1999–2007 and 1998–2007, respectively). Rather, I pragmatically extended their respective meta-analyses with quantitative results from studies which I came across in my general search for related work.

### 2.3.3 a) Learning Effectiveness of Pair Programming (A1)

There are basically two ways how researchers try to determine whether students learn in educational pair programming sessions: One is through experimental designs with students' quiz answers, assignment scores, or exam grades as dependent variables, the other is through surveys asking for a personal account from the students' perspective.

#### *Experimental Designs*

Canfora et al. (2004, 2005) and Bellini et al. (2005) conducted a number of controlled experiments. The researchers designed a software system and issued system understanding quizzes before and after students worked for two hours with UML diagrams as well as the system's specification either alone or in pairs. They show that, overall, "pair designing" led to a better understanding of the system's design.

Such a specifically designed system allows researchers to develop quizzes to test the students and to directly measure a change in knowledge levels. Most experimental designs only look at the outcome in terms of assignment and exam scores, assuming that if paired students get better grades than solo workers, some LEARNING must have happened along the way. Although some studies found no significant effect and others even reported significant negative effects of pair programming, the meta-analysis by Salleh et al. (2011, Sec. 3.4) yielded a *small positive effect on final exam scores* (Hedges's  $g = 0.16$ ) and a *medium positive effect on assignment scores* ( $g = 0.67$ ). I amended their meta-analyses with the data from two additional studies and more intuitive measures; see Table 2.5 for the results and Appendix D for the technical details.

Variable	Studies	Subjects	Rel. Mean Diff.		Het.
	$k$	$n$	$D\%$	95% CI	$I^2$
Exam Score	12	2 067	+4.2%	[+0.3%, +8.2%]	72%
Assignment Score	7	1 263	+11.3%	[+5.0%, +18.0%]	83%

**Table 2.5:** Statistical results of two meta-analyses on educational pair programming effects, based on the data collected by Salleh et al. (2011, Sec. 3.4, Fig. 4 & 5) and two additional studies (see Figure 2.4 for an explanation of  $D\%$  and Appendix D for calculation details). The effects are all in favor of pair programming: about 4% higher scores in exams and more than 11% higher scores in assignments. Heterogeneity is high in both cases, meaning that the effect sizes vary a lot between the individual studies. In the *exam score* studies, the subjects were individuals; in the *assignment score* studies, some subjects were pairs, others were individuals.

This first additional study is a controlled experiment conducted by Cheney (1977), which is not discussed in any of the secondary studies on pair programming. To the best of my knowledge, Cheney was the first to use the term "pair programming", although with a slightly different meaning. He randomly assigned 120 students to two conditions: Those in the "individual programming" group had to write six programs alone, while students in the "pair programming" group started alone but then exchanged the first versions of their respective programs with their partner who would check the code for errors and repeat the process with as much communication as needed. The pairs were allowed to hand in just one solution for both partners. The scores of the individually taken final exams showed a significant difference in favor of the "pair programming" group.

The second addition is an experiment by Zacharis (2011) in which he compared 65 solo-working students with 64 who worked in distributed pair programming (DPP) mode on four homework assignments. He found only small and not statistically significant effects on the

students' assignment (in favor of pair work) and exam scores (in favor of solo work). (Zacharis *did* find significant differences with regard to productivity and quality, which I discuss along with other experiments interested in these economic factors in Section 2.3.4a.)

Considering all these studies together, pair programming has a positive effect on exam and assignment scores, about 4% and 11%, respectively (see Table 2.5 and Appendix D). This difference potentially helps explaining the findings of Hanks (2008) who compared the assignment and exam scores of 116 students who worked either in co-located or in distributed pairs. The only significant effect was a negative effect of distributed working on the assignments scores for one out of three classes; there were no significant differences in exam scores. Since there was no group of solo programmers, there can be no 1-to-1 comparison to the above meta-analysis. However, one interpretation would be that DPP imposed too many difficulties on the students (negative effect in assignments), which did not affect the exam scores since pair work during the semester does not affect the final exam scores much anyway (small effect on exam scores).

### ***Self-reported achievements***

Salleh et al. (2011, Sec. 4.5) summarize that PP is satisfying for students in part because it helps them increase their knowledge. Evidence for this conclusion comes from surveys which a number of studies either used as their main instrument for collecting data or as an auxiliary source. Salleh et al. did not conduct a detailed analysis in this regard, so I provide one of my own, covering the studies cited by Salleh et al. (2011) and Estácio & Prikładnicki (2015).

Some studies simply recorded *that* knowledge transfer supposedly happened. With regards to the share of students who report such a knowledge transfer effect, there is a notable difference between surveys in which the students bring up the topic of increased knowledge by themselves as an answer to an *open question* and those in which there were *closed questions* asking students to agree or disagree, which (depending on the framing of the question) might induce a bias towards a conforming answer. In questionnaires with open questions asking for general benefits of pair programming, between 30% and 47% percent of students explicitly list effects related to knowledge transfer (see Table 2.6 for details). In questionnaires with closed questions such as '*I learned more/faster/better about X because I worked with a partner*', those with a binary scale (agree/disagree) got around 85% agreement rates by the students, whereas the rates from studies using scales with more degrees in between range from as little as 48% to as much as 94% of students who '*agreed*' or '*strongly agreed*' (see Table 2.6 for details). Overall, at least half of the students report that they learned more about their courses' subjects in pairs than they would have working alone. Of course, these surveys do not record matters of fact, but only perceived effects at best.

Other studies went beyond treating knowledge transfer as a binary property and looked at the breadth and depth of transferred knowledge. Vanhanen & Lassenius (2005) asked students for their perceived knowledge levels after having worked in four-developer project teams either with or without having pair-programmed. The authors correlated students' self-reported understanding across software modules with self-reported involvement in development and found differences between the pairing team and the solo team: Pair programmers, on average, understood 4.5 out of 10 packages at least "*quite well*", as opposed to 3.4 packages for the solos. However, due to the small sample size (one team per condition, four developers each), this difference is not statistically significant.

### ***Summary***

Studies with an experimental design and those capturing students' own experiences point in the same direction: The LEARN mechanism is effective (A1) in that students feel they learned

Study	Scale	Agreement	Subjects ( $n =$ )
Williams & Kessler (2000, Sec. 4)	binary	84%	20
Freeman et al. (2003, Table 4)	binary	86%, 86%	106
Janes et al. (2003, Sec. 3.2)	binary	80%	15
Chaparro et al. (2005, Sec. 5.4)	rating	48%, 55%, 65%	58
Hanks (2006, p. 114)	rating	50%	115
Howard (2006, Table 2)	rating	78%, 88%	74
Zacharis (2011, Table VII)	rating	94%	64
Xu & Rajlich (2005, Table V)	rating (0–5 †)	$\bar{x} = 3.8$	4
Zin et al. (2006, pp. 167–168)	rating (1–4 †)	$\bar{x} = 2.06$	147
Edwards et al. (2010, Table 2)	rating (1–4 †)	$\bar{x}_1 = 3.16$ $\bar{x}_2 = 2.79$	$\approx 100$
VanDeGrift (2004, Table 1)	rating (1–5 †)	$\bar{x}_1 = 3.2$ $M_1 = 3$ $\bar{x}_2 = 3.5$ $M_2 = 4$	293
Freeman et al. (2003, Fig. 2)	open	47%	106
VanDeGrift (2004, Table 2)	open	30%	293

**Table 2.6:** Students' agreement rates from surveys with close-ended questions in the form 'I learned more/faster/better about X because I worked with a partner' and open-ended questions asking for PP benefits in general. Agreement rates for rating scales represent '(strongly) agree' responses. More than one agreement rate indicates multiple similar items.

Edwards et al. (2010) did not report the exact number of subjects. Four studies mapped their rating scales († = higher is more agreement; † = lower is more agreement) on interval scales and only report the median ( $M$ ) and/or mean values ( $\bar{x}$ ).

something and that it has a positive impact on how much students learn—at least in terms of assignment scores, not so much exam scores.

### 2.3.3 b) Pair Compatibility (A4)

Instead of comparing pair programmers with solo programmers, some educators compared different *types of pairs*, expecting that not all constellations work equally well (A4). Salleh et al. (2011, Table 4) list 14 possible factors of a pair's compatibility which may influence their effectiveness. Most of these factors were subject to three or fewer primary studies, leaving *personality* and *actual skill level* as the most studied ones. I discuss findings regarding both factors below as they may be relevant for understanding knowledge transfer in pair programming.

#### *Personality Types*

Most primary studies looking at the pair members' personality types did not produce significant effects, with only a few indicating that pairs with different personality types are more productive (*ibid.*, Sec. 3.3). As of 2011, there had been no agreement on whether homogeneity or heterogeneity of personalities is better (*ibid.*, Sec. 4.4).

Salleh et al. (2014) consequentially performed a series of four experiments investigating the effect of three of the Big Five personality factors (*conscientiousness*, *neuroticism*, and *openness*) on the academic performance (in terms of assignment scores and exam results) as well as satisfaction and confidence of students working in pairs. (Note that pair programming is part of the context here, not an independent variable.) Their first experiment compared conscientiousness-wise heterogeneous pairs to those with a similar elevation, which did not

show a significant difference in academic performance. The other three experiments only considered homogeneous pairs, i.e., with both partners scoring low, medium, or high on conscientiousness, neuroticism, or openness, respectively. Since students cannot be randomly assigned to a trait level, these last three were quasi-experiments, each of which was designed to study the effects of one personality trait. The only significant difference the authors found was a better academic performance of students who scored high on openness compared to medium or low scorers.<sup>6</sup> However, since a higher openness level is considered beneficial for academic performance anyway—as the authors themselves note (Salleh et al., 2014, Sec. 8.2)—the observed effect is not necessarily related to pair programming. Although the authors considered studying pairs comprising a more and a less open student (*ibid.*, Sec. 6), to the best of my knowledge, no such study has been done yet.

Findings from a different study suggest an advantage of different personalities (Sfetsos et al., 2013). The 20 pairs with different MBTI temperament types achieved significantly higher design and correctness scores in their lab assignments than the 20 homogeneous pairs.<sup>7</sup> Furthermore, there was a strong positive correlation ( $r^2 = 0.839$ ) between the pairs' scores and the *amount of communication* in their session which was self-recorded by the students during their sessions on paper by counting their “*communication transactions*”. However, it is not clear what to make of this result: The granularity of these “transactions” appears to be rather coarse as the average session with a length of 105 minutes had only about 16 of them (*ibid.*, Table 5). The authors do not provide detail on the pairs' actual interaction, so it is not clear which circumstances led to more or less recorded transactions and what the students learned along the way beyond technically solving their task.

Overall, the personality (dis)similarity of student pairs appears to have *some* effect on how they work together.

### **Skill Level**

The pair members' respective skill level—in terms of programming experience or academic performance—is another common research interest for educators. Intuitively speaking, a too large difference between pair members may pose a problem as a more skilled student might get bored or frustrated by her less skilled partner who might in turn be overburdened. Salleh et al. (2011, Table 4 & Sec. 3.3) see a “*consensus*” in the form of a “*significant positive effect [of] actual skill level [similarity on] effectiveness and/or pair compatibility*”. This characterization, however, cannot be justified by the findings of the primary studies they cite, because:

- a) Not all cited studies employed quantitative methods to yield ‘*significant effects*’.<sup>8</sup>
- b) Not all cited studies actually considered (1) pair *similarity*. Other studies had different *skill level* notions as they analyzed the effect of (2) a *single* pair member's experience (which may point to compatibility-related effects indirectly), or of (3) the *sum* of the pair

---

<sup>6</sup>Calculated from Salleh et al. (2014, Table 11) using Borenstein et al. (2009, Eq. 4.30–4.36): relative mean differences between pairs with low and high openness is +9% in assignments (95%-CI: [−2%, +22%], Hedges's  $g = 0.34$ ) and +18% in exams (95%-CI: [+4%, +34%], Hedges's  $g = 0.54$ ).

<sup>7</sup>Calculated from Sfetsos et al. (2013, Table 5) using Borenstein et al. (2009, Eq. 4.30–4.36): relative mean difference between homogeneous and heterogeneous pairs for scores is +150% (95%-CI: [+100%, +213%]). Hedges's  $g = 3.09$  is *very large* (in the 99-percentile of software engineering effect sizes, see Kampenes et al., 2007).

<sup>8</sup>Cliburn (2003, Sec. 5, aka S15) closes with an informal observation on his introductory programming classes that “*pair programming works best [...] when partners are at the same ability level*”, after assigning partners based on their grades. Van Toll et al. (2007, Sec. 5, aka S58) report on a *single* student who worked in four different pair constellations, for which he experienced some learning in constellations of slightly different prior experience, no learning with comparable prior experience, and constant frustration with much more experience. I was not able to obtain the full-text of Cao & Ramesh (2004, aka S11), but according to the paper's abstract, this is an “*exploratory study*” that “*collected qualitative data*” rather than a quantitative one with a statistical analysis.

Study	[ID]	Subjects ( $n =$ )	Skill Level Notion	Dependent Variable	Sig. Corr.?
Canfora et al. (2005)	[S8]	28	(1) similarity of educational background	quiz score	(✓)
Williams et al. (2006)	[S63]	1 320	(1) relative skill levels (midterm, SAT)	perceived compatibility	(✓)
Gevaert (2006)	[S74]	28	(2) solo performance	pair satisfaction	✗
Müller & Padberg (2004)	[S42]	19/19	(2) experience of less/more experienced pair member	implementation time	✗/✗
Müller & Padberg (2004)	[S42]	19	(3) total pair experience	implementation time	✗
Madeyski (2006)	[S68]	35/31	(3) mean pair experience	external code quality (with/without TDD)	✗/✓

**Table 2.7:** Characteristics of correlational studies (IDs from Salleh et al., 2011) looking into pair members’ skill difference (1), individual skill levels (2), and collective skill levels (3). Subjects for notions (1) and (2) are individuals; for notion (3), the subjects are pairs.

Symbols: ✓—one study reports a significant correlation; ✗—most studies do not report significant correlations; (✓)—two studies have mixed results: Canfora et al. (2005) report inconsistent rank-sum statistics making it impossible to validate their claims, Williams et al. (2006) report significant correlations only for a part of one out of three courses.

member’s individual experience (which completely masks any differences between, say, a high-low and a medium-medium pair).

- c) The cited studies which *did* employ statistical means to test for significant correlations between some notion of skill level and some measure of effectiveness overwhelmingly *did not* report significant effects (see Table 2.7).

Since the secondary study cannot be relied on in this regard, I discuss the primary studies individually, grouped by their notion of “skill”.

**Skill in Terms of (1) Pair Similarity** Canfora et al. (2005) compared the levels of system understanding of pair members in different constellations of students with “*scientific*” and “*non-scientific*” educational backgrounds based on individually taken quizzes. The authors claim that mixed pairs performed worse than pairs of students with the same background (*ibid.*, pp. 1480–1481). According to the reported rank-sum statistics (*ibid.*, Table 2), individuals from the 5 scientific pairs indeed performed significantly better than individuals from the 4 mixed pairs and the 5 non-scientific pairs. The reported statistics for the comparison of mixed pairs and non-scientific pairs, however, are inconsistent,<sup>9</sup> so the authors’ conclusion cannot be validated. And even if the statistic results are correct, the experiment did not compare *actual skill levels* (as implied by Salleh et al., 2011) but *educational backgrounds*. The reported  $p$ -value (0.020), however, indicates some difference between the groups (assuming it was calculated correctly). Quite possibly the different educational backgrounds of the pair members in the

<sup>9</sup>According to Canfora et al. (2005, Tables 1 & 2),  $n_1 = 8$  individuals came from mixed scientific/non-scientific pairs and  $n_2 = 10$  from non-scientific pairs, for which the authors report rank-sums of  $R_1 = 135$  and  $R_2 = 75$ . This cannot be true because for a total of  $N = n_1 + n_2 = 18$  ranks to be filled,  $R_1 + R_2$  should equal  $N(N + 1)/2 = 171$ , but is actually  $135 + 75 = 210$ : There are not enough ranks to get rank-sums this large. This is possibly a copy-paste error, as the rank-sum values are the same as in the table row above which compares two larger groups.

mixed pairs made their communication more difficult. The actual process (A5), however, was not looked at by the researchers.

Williams et al. (2006)<sup>10</sup> analyzed pair programming in the context of a number of university courses and relied on 1,320 students' assessments of whether they felt "very compatible", "ok", or "not compatible" with their assigned partner.<sup>11</sup> The authors tried to predict compatibility through a number of variables such as learning styles, self-esteem, and also actual skill level. They did find a small number of significant correlations for some instances of one out of three courses, but overall conclude that compatible pairs cannot be proactively created based on differences in actual skill levels (*ibid.*, Sec. 3.3.2). On the flipside, each of the over 1,300 students was assigned to three to four different partners over time and only 7% of the pair constellations were self-assessed as "not compatible" in the first place (*ibid.*, Table 3), so it does not appear particularly urgent to find a way to *create* compatible pairs.

**Skill in Terms of (2) Individual and (3) Collective Skill Levels** While Canfora et al. (2005) and Williams et al. (2006) considered the (dis)similarity of pair members with regard to their educational background and skill levels, other correlational studies looked at individual and collective skill levels, which are not suitable for answering questions about pair compatibility. These studies are nevertheless relevant for understanding knowledge transfer as they deal with the pair members' exhibited skills which can be considered a result of their individual and collective prior knowledge.

Gevaert (2006, Sec. 4.1.1 & 5.5) found no significant correlation between individual performance scores of 28 students and the satisfaction they attributed to their 60 minutes of pair work. Müller & Padberg (2004) conducted two experiments on the influence of prior experience on development performance with a total of 19 programming pairs. They neither found a statistically significant correlation between total pair experience and time taken to complete implementation tasks (*ibid.*, Sec. 4.1), nor between the individual experience levels of either the less or more experienced pair member and implementation time (*ibid.*, Sec. 4.3). Madeyski (2006) performed an experiment with 66 programming pairs and found a significant correlation between a pair's mean programming experience in years and external code quality (but not for pairs who employed test-driven development).

**Summary** Overall, there is only little and only inconclusive evidence for assessing the effect of pair members' skill levels on their effectiveness and satisfaction. For the experimental tasks given to student pairs, prior knowledge or solo performance of the pair members may or may not have an effect on how well the pair performs. Although intuition and anecdotal evidence suggest that large differences in pair members' skill levels may hamper the pair process, reliable evidence is lacking.

The results of a study by Wilson et al. (1992), which is not referenced by any of the secondary studies, indicate that having a programming partner lessens the impact of each pair member's capabilities. The authors correlated the scores of 10 student pairs with 14 solo students on a 60-minute programming task with their respective university grades. While the solos' grades significantly correlated with their programming scores ( $r = 0.46$ ,  $p < 0.1$ ), the correlation was not significant and weaker for the pairs ( $r = 0.26$ ,  $p > 0.1$ ).

---

<sup>10</sup>Although Salleh et al.'s SLR references Katira et al. (2004, 2005) and Williams et al. (2006) as separate studies (S28, S29, and S63), the last publication includes all data and findings from the earlier ones.

<sup>11</sup>Although Williams et al. (2006, Sec. 3) speak of compatibility data of 1,350 students, 30 students in the OO class never actually worked in pairs (*ibid.*, Sec. 2.2.3 & Table 2).

### 2.3.3 c) Pair Process (A5)

Bellini et al. (2005) noted on their pair design experiment, “[w]orking in pairs could be seriously affected by the way pair’s members achieve cooperation” (see discussion on page 55). Starting with the first pair programming experiment (Cheney, 1977, my discussion on page 55) and continuing with most of the above mentioned studies, the authors do not discuss the actual process of the programming pairs. In principle, to communicate “as much as they desire” (ibid., p. 2) could amount to as little as exchanging pieces of paper with source code, having very little resemblance with “pair programming” in the common sense. Asking (or allowing) students to work in pairs is not necessarily the same as students actually working in pairs—and even if they do, their work styles may differ drastically.

Sfetsos et al. (2013, my discussion on page 58) report a strong positive correlation between a pair’s amount of communication and its programming score. While none of the above mentioned studies attempted to explain what actually happens when two students pair program, some allude to an internal process—which becomes visible, e.g., in the wording of the closed survey questions which point to some mechanisms the educators probably had in mind: VanDeGrift (2004) specifically asked for the importance of giving explanations, while Freeman et al. (2003) make a distinction between *explaining* and *receiving an explanation* (see Table 2.8). Xu & Rajlich (2005, p. 11), who did not directly observe their students’ pair programming, state that “*knowledge is constantly exchanged between partners, which include [sic] those concepts learned in the classroom, tool usage tips, programming language, design skill, and debugging techniques*”. It is not clear, however, whether this is a hypothesis or a finding from their analysis of students’ survey answers (in which case it would also be a partial answer to A2).

Study	Questionnaire items
Freeman et al. (2003, p. 6)	<i>I learned more from explaining my work to my partner.</i> <i>I learned more because my partner explained their work to me.</i>
VanDeGrift (2004, pp. 3–5)	<i>I gained more understanding of concepts in the course by explaining them to my project partners.</i>

**Table 2.8:** Items from questionnaires used in studies on pair programming effectiveness in education which hint at internal mechanisms suspected by the administering educators.

Overall, the primary studies surveyed by Salleh et al. (2011) do not provide insight into what the PP process looks like in educational settings and how precisely learning effects come to be. A study from the late 1980s, however, did exactly that. In two individual studies, Webb & Lewis (1988) followed a qualitative-quantitative approach: They recorded and analyzed the conversation of students in groups of three and two who worked on beginner level tasks in the programming languages LOGO and BASIC. Webb & Lewis categorized the individual interactions, aggregated the counts to different “*peer interaction variables*” and correlated these with learning achievement measures acquired through a post-test. They summarize their findings as follows:

“ The specific behaviors in these studies that were found to be positively related to some achievement outcomes were *giving explanations, giving input suggestions, receiving responses to questions, receiving input suggestions, verbalizing planning and debugging strategies to peers*, and to some extent, *receiving explanations*. These behaviors should be encouraged in the classroom. The behavior found to be negatively related to achievement was *receiving no help when needed*. This behavior should be discouraged.

Webb & Lewis (1988, p. 198, emphases added)

In other words, students who exchanged explanations, communicated their plans, and followed each other's computer interactions learned more; students whose partner did not provide help when necessary learned less. Simply put, for knowledge transfer to happen, students need to communicate.

The results from a more recent study point in the same direction even though its setup severely inhibited the collaboration of the pair members. Rodríguez et al. (2017) randomly assigned 54 students to pairs and gave them a one-hour task with a graphical programming language. The pair members were seated in separate rooms, one of them being the designated driver who would share her screen with her partner. A text chat was their only means of two-way communication. The researchers annotated the chat logs and correlated counts of different message types and technical success. They conclude that “*feedback of any kind*” as well as “*meta-comments*” were properties of highly successful pairs' communication (*ibid.*, Sec. 6).

### 2.3.3 d) Summary of Pair Programming in Education

For CS students, working in pairs on small tasks such as their homework assignments and programming exercises in lab sessions appears to have a positive impact on assignment scores and exam grades, indicating that pair programming is effective in terms of increasing the understanding of concepts taught in class (A1, A2). Subjective accounts support this conclusion, since more than half of the students who worked in pairs report such an effect in surveys. Knowledge transfer in the sense of learning together and/or from another generally appears to occur during pair programming.

Pair constellations (A4) in terms of students' personalities and actual skill level were considered by a number of studies which, overall, did not show a clear effect on pair programming effectiveness—a formal meta-analysis has not yet been conducted. Both heterogeneous and homogeneous pairs appear to function more or less. In the single largest study involving over 1,300 students in over 5,000 pairings only 7% of the pairs rated themselves as incompatible.

What actually happens when students are told (or allowed) to work in pairs is not understood. In particular, the extent and kind of collaboration (A5) is not looked at by most studies. The medium positive effect of PP on assignment scores, for example, could also be explained by the phenomenon of *disjunctive group tasks* in which the best individual solution counts for the pair, which only requires minimal communication. Indeed, the effect of pairing on the exam scores, for which students are assessed individually, is only small in comparison. How the overall knowledge transfer effect comes to be in educational settings, is by-and-large an open question except for the findings of Webb & Lewis (1988) which indicate that working closely together (i.e., exchanging explanations, communicating one's plans, following each other's computer interactions) is more learning-effective than not interacting with one's partner.

### 2.3.4 Pair Programming with an Industry Focus

The mapping study by Vanhanen & Mäntylä (2013) makes an assessment of the state of industrial PP research regarding six overarching themes, some of which are mostly concerned with pair programming as a practice, others focus more on pair programming as a work mode. However, the authors do not provide a synthesis of individual research results. I used their mapping study as one source to identify possibly relevant studies. The relevant criterion for this section was to include studies on the pair programming work mode that used data from industrial *contexts* or at least pursued an industrial research *goal* (as opposed to studies with make-believe settings and the goal of improving students' learning, see Section 2.3.2a).

In the next sections, I discuss relevant pair programming studies grouped by their research design, each with different strengths and weaknesses (see Section 2.3.2b). A controlled experiment is the only research design which, in principle, allows to show causal relationships. I discuss pair programming experiments in Section 2.3.4a. For industrial software engineering research goals concerning process-related topics such as pair programming, however, controlled experiments are difficult to design such that they resemble real-world settings: Context and/or subjects are often not *industrial*.

Other empirical approaches, however, work well with industrial contexts and subjects. I already discussed the practitioner perspective through experience reports and surveys in Section 2.3.1d. Observational field studies dig deeper into what actually happens instead of relying on subjective accounts alone. Some of these try to count, measure, and analyze already understood aspects of software development processes in a quantitative fashion either on a project-level, considering PP more as a practice (Section 2.3.4b), or in more detail on a work mode-level (Section 2.3.4c). Others dissect the software development process, investigate the details of what actually happens, and formulate qualitative theories (Section 2.3.4d).

### 2.3.4 a) Controlled Experiments on Pair Programming

In most experiments on pair programming, developers work on specifically designed tasks either alone or in pairs (independent variable) and the effects on economic metrics for *quality*, *duration* (i.e., time passed), or *effort* (immediate cost) are measured. These experiments are essentially programming contests between pair programmers and solo programmers. Unlike studies from educational settings (Section 2.3.3a), industrial experiments are less interested in the effects of the LEARN and COMBINE mechanisms, since what the developers bring to the table and what they take from a session is usually less important than the technical outcome. In a sense, experiments that start with an existing system are more interested in the effects of the UNDERSTAND mechanism.

Some experiments also go beyond mere effectiveness (A1) and consider moderating variables such as *task complexity*, *programmer expertise*, and *personality* (hinting at A3 and A4), expecting that pair programming does not have uniform effects under all conditions.

#### *Quality, Duration, & Effort*

Hannay et al. (2009) performed meta-analyses of the results from a total of 18 experiments on the effectiveness of pair programming with regards to quality, duration, and development effort.<sup>12</sup> To make the individual results comparable, the authors considered the *standardized effect sizes* for which they report both a point estimate and a 95% confidence interval (see Figure 2.4). To measure the degree of inconsistency between the studies, they also calculated the  $I^2$  statistic (again, see Figure 2.4).

I extended Hannay et al.'s original meta-analysis regarding *quality* with the statistical results of two additional primary studies. The first study is by Wilson, Nosek, et al. (1992) who performed controlled experiments involving a small number of professionals and students who worked on a programming task for 60 minutes showing a positive effect of pair work on quality. This experiment is functionally similar to Nosek (1998), whose data from an experiment with a 45-minute time cap was already part of the meta-analysis.

The second study is by Zacharis (2011) who compared the technical outcomes of four homework assignments of 64 students working in pairs with those of 65 students working alone (see also my discussion on page 55). He found no significant difference in *duration*, a

<sup>12</sup> Although some of these experiments were conducted with students (see Figure 2.5), they all share the (industrial) goal of informing practitioners.

Variable	Subjects	Studies	Std. Effect Size		Het.
	<i>n</i>	<i>k</i>	<i>g</i>	95% CI	<i>I</i> <sup>2</sup>
Quality	698	17	0.53	[ 0.15, 0.90]	87%
Duration	403	11	0.53	[ 0.13, 0.94]	70%
Effort	361	11	-0.52	[-1.18, 0.13]	85%

**Table 2.9:** Statistical results of three meta-analyses on pair programming effects based on the data collected by Hannay et al. (2009, Sec. 3.2 & Tab. 2). I included the statistical results of Wilson et al. (1992) and Zacharis (2011) in the meta-analysis for *quality* (see Appendix D for the technical details). Subjects in the primary studies were individuals, pairs, or whole teams who solo- or pair-programmed (Hannay et al., 2009, Tab. 1 & Fig. 1). The primary studies used different scales, so Hannay et al. (2009, Sec. 2.5) report standardized effect sizes and no raw mean differences. Here, a positive effect size is also ‘positive’ in an economic sense, e.g., a positive effect on *duration* means less wall-clock time. The confidence intervals for the summary effect sizes are wide and between-study variance (or *heterogeneity*) is very large.

significant 57% increase in *effort*, and a significant 50% decrease in defect count, i.e., an increase in *quality* (Zacharis, 2011, Sec. IV). The validity of these numbers is questionable since (a) the explicit goal was for the students to learn how to write clean object-oriented code (rather than being as fast as possible), and (b) all data (including defect count and timing) was collected by the students themselves (*ibid.*, Sec. II). As the author himself states, it is not even clear whether the students discovered the defects before or after testing (*ibid.*, Sec. IV.B). I included Zacharis’ quality-related data in my meta-analysis nevertheless (see Appendix D for details) because these validity threats affect both solos and pairs. I was not able to include the time-related data in my meta-analysis, because Zacharis (*ibid.*, Table II) only reports mean values for pairs and solos, but no variance or standard deviation.

Overall, pair programming appears to have a positive effect on quality and on duration (elapsed wall-clock time), but a negative effect on effort (person-hours). However, the confidence intervals for the respective effect sizes are wide, indicating quite some uncertainty regarding the size of the effects and—in the case of *effort*—even direction (see Table 2.9). Heterogeneity measures are high, indicating that this variance likely comes from differences in the real effect size rather than random error. Hannay et al. (2009, Sec. 4) conclude that moderating factors (such as general programming expertise, complexity of the worked-on tasks, prior pair programming experience, the developers’ motivation, whether they worked in a team context) might be relevant. I agree with their assessment: The inclusion of the quality-related data of Wilson et al. (1992) and Zacharis (2011) led to a larger summary effect size and a wider confidence interval compared to the original analysis, meaning that, overall, pair programming appears to have a positive effect on quality, but the extent varies a lot between contexts.

In contrast to the experiments focusing on the economic questions, Beck does not explicitly talk about the immediate costs of pair programming in his XP books. This is likely because he does not see PP as a *work mode* that competes against solo work in programming contests. To him, PP is a *practice*, an integral part of a software development method that pays off through “*reduce[d] project risk, improve[d] responsiveness to business changes, improve[d] productivity throughout the life of a system, and add[ed] fun to building software in teams*” (XP1, p. xvi). As such, pair programming has effects that reach far beyond a single programming task. Hannay et al. (2009, Table 2) attempted a subgroup analysis separating results from experiments where pair programming was done on isolated tasks and where it was part of a larger, long-running project. However, there was only a small number of experiments with a project context and

the meta-analysis did not show significant effects of pair programming on quality, effort, or cost for this subgroup.

Hannay et al. (*ibid.*, Sec. 4) conclude that a simple comparison of pair programming and solo programming is not meaningful as the usefulness of this work mode apparently depends on the context. Two context properties considered relevant for characterizing software development situations in general and pair programming in particular are the *expertise* of the developers and the *complexity* of the tasks they are working on.

### ***Expertise & Task Complexity***

The single largest industrial PP experiment by Arisholm et al. (2007) involved 295 professional software developers and was designed to determine the moderating role of *task complexity* and *developer expertise* on the effectiveness of pair programming. The developers (hired consultants) performed three incremental changes to an existing system unknown to them. The researchers prepared two software systems with different architectures to test two levels of task complexity (a simple centralized vs. a more complicated delegated control-style). The pairs were formed with similar expertise based on the consultants' pay grade (junior-junior, intermediate-intermediate, and senior-senior). Dependent variables were the *correctness* of the produced changes (as a binary variable), as well as the *duration* and *effort* needed to complete the tasks measured in minutes (*ibid.*, Sec. 3.6.1). Results from individually taken pre-tests allow to adjust these variables to account for individual differences which were not part of this experiment (*ibid.*, Sec. 3.6.3). The statistical analysis and calculation of effect sizes is based on the *adjusted means* (i.e., accounting for individual differences), which makes the comparisons of, say, pairs and solos or juniors and intermediates more fair but less accessible to intuition.

- **Moderating role of task complexity:** The pairs were significantly faster than the solos only in the simple system, and produced significantly higher quality only in the complex system (*ibid.*, Sec. 4.1).

It should be noted that the pairs' *unadjusted* mean performance (i.e., actually delivered quality and wall-clock time) was more or less the same regardless of the task complexity (80% vs. 83% correctness and 64 vs. 62 minutes duration, *ibid.*, Tables 9–11), while complexity *did* affect the solos, who worked slower but more correct in the simple system (68% vs. 51% correctness and 98 vs. 72 minutes, *ibid.*, Tables 9–11).

- **Moderating role of developer expertise:** Expertise did not moderate the effect of pair programming on *duration* in a consistent way: Pair programming appeared most beneficial for intermediates and least for juniors (*ibid.*, Sec. 4.1). But, again, the unadjusted means (*ibid.*, Table 9) make clear that this effect is mostly due to the solos whose intermediates were *slower* than their juniors (104 vs. 92 minutes), whereas senior pairs were faster than intermediate pairs who were again faster than junior pairs (51 vs. 61 vs. 81 minutes). With regard to *correctness*, juniors benefited most from pair work, while the effects for intermediates and seniors were not significant (*ibid.*, Sec. 4.1). The unadjusted means (*ibid.*, Table 11) yet again show more consistency in the pairs than the solos (80–84% mean correctness for pairs across all pay grades opposed to 48–75% correctness for solos).
- **Interaction of expertise and complexity:** There appears to be an interaction between the two moderators, e.g., with junior pairs producing a correct solution with 84% (adjusted mean) probability compared to just 34% when working alone (*ibid.*, Table 11). However, this interaction could not be formally tested (*ibid.*, Sec. 4.1).<sup>13</sup>

<sup>13</sup>This has to do with the nonrandom assignment of developers to the two conditions of solo and pair work, which ultimately precludes some kinds of analyses. See Arisholm et al. (2007, pp. 72 & 75) for details.

Overall, despite the large sample size, the statistical analysis did not produce conclusive results on the moderating role of either task complexity and developer expertise (Arisholm et al., 2007, Sec. 4.1).<sup>14</sup> However, even if not formally shown, it appears that the solo developers' performance depends far more on their expertise and task complexity than the pair programmers' and that pair programming reduces the overall variance of the outcome. This is similar to the lack of a significant correlation between university grades and programming scores in a pair situation reported by Wilson et al. (1992) (my discussion on page 60).

As Arisholm et al. (2007, Sec. 5.3.3) note, task complexity is not absolute but depends on the experience of the developer. In this sense, Lui & Chan (2003, 2004, 2006) combined task complexity and developer experience and studied the effectiveness of pair programming under increasing routineness of the task in their "repeat programming" experiments: Solo and pair programmers implemented solutions for the *same task* multiple times from scratch. In the first round, pair programmers completed their project considerably faster than the solo programmers (410 vs. 635 minutes). In both work modes, the time to completion decreased over the rounds, but the pairs' advantage diminished by the fourth round (272 vs. 285 minutes). Although such a setting is completely artificial, these results suggest that working in pairs can be beneficial when there are requirements yet to be understood and a viable design yet to be found (the UNDERSTAND mechanism).

### ***Pair Member's Personalities***

Hannay et al. (2010) analyzed additional data collected in the experiment by Arisholm et al. (2007). In addition to the independent variables *task complexity* and *developer expertise*, they also considered the developers' *personalities* (in terms of the pair's mean elevation and its members' difference regarding each of the Big Five traits) and their *country* of location. Along with the dependent variables used in the first analysis (*correctness* and *duration*), additional ordinal variables were included to characterize the technical quality of a pair's work result (e.g., adherence to object-oriented principles, extensibility, and cost effectiveness).

The univariate analysis (i.e., considering one independent variable at a time) showed only small effect sizes of personality traits compared to those of expertise and task complexity. Even the multivariate analysis involving all independent variables showed only few significant effects and "*considerable unexplained variance*", which can indicate both missing and included, but disturbing variables (Hannay et al., 2010, Sec. 6.2). In an exploratory analysis, the researchers applied a decision tree analysis in which they iteratively split the data points according to a threshold value on one independent variable in order to maximize the significance of the effect on a dependent variable (*ibid.*, Sec. 6.3). Among all personality variables, the difference in a pair's *extraversion* was the "*most general significant predictor*" (*ibid.*, Sec. 6.3), i.e., in many decision trees, splits along this dimension were significant. On the one hand, Hannay et al. do not characterize the effect of a larger intra-pair extraversion difference, i.e., whether homogeneous or heterogeneous pairs are "better" in any way—there is just *some* effect. The researchers appear to have no idea what the effect might be either, since *extraversion difference* was the only personality variable for which their univariate model did *not* contain a hypothesis (*ibid.*, Fig. 1). On the other hand, *extraversion difference* only ranks forth behind *task complexity*, *country*, and *expertise* anyway (*ibid.*, Table 7), so the researchers conclude it is not worth allocating resources to personality-based match-making in industry (*ibid.*, Sec. 7.1).

---

<sup>14</sup>Arisholm et al. (2007, Tables 9–11) *did* find overall effects of pair programming: a significant effect on effort in favor of solo programming (on average 84% more effort with pairs, 95%-CI: [61%, 110%]) along with nonsignificant effects on duration and correctness in favor of pair programming (on average 8% less duration with pairs, 95%-CI: [–19%, +5%] and 28% higher chance of correct solution with pairs, 95%-CI: [–44%, +192%]). These results are part of the meta-analyses by Hannay et al. (2009).

**Discussion: Limitations of Pair Programming Experiments**

In the experiments discussed above, two groups of subjects were asked to work on identical tasks, with the developers in the experimental group working in pairs and those in the control group alone. With subjects being randomly assigned to the groups and all other aspects being equal, any difference in the dependent variable—such as code quality or time spent—could then be attributed to the pair programming work mode. Arisholm et al. (2007) conducted a well-designed experiment, and their writing contains abundant information. I therefore refer to their experiment during my discussion of three categories of problems which such research designs for this particular research interest have.

**Problem Category 1: Number of Unknown and Uncontrolled Factors** The meta-analyses of Hannay et al. (2009) have not shown clear effects of pair programming, but mere tendencies with a lot of variance. To explain this variance, moderating variables first need to be identified before they can be systematically addressed in an experimental setup. Even the large (and presumably very expensive) experiment by Arisholm et al. (2007) which was specifically designed to understand two (possibly) moderating variables could not provide a conclusive answer.

After more moderating variables have been identified, systematically addressing them the same way would require a large number of experiments or many more subjects. For comparison: Arisholm et al. had twelve groups (2 work modes  $\times$  2 complexity levels  $\times$  3 experience levels) and expected a medium-sized effect which led them to a minimum of 14 observations per group, or 168 subjects in total, to get a minimum statistical power of 0.8 (*ibid.*, Sec. 3.1). For more moderating variables and smaller effects, the number of necessary observations grows even larger.

**Problem Category 2: Unrealistic and Unfair Comparisons** If Beck (1999) is right and pair programming is indeed a “*skill*”, prior exposure to this work mode is a relevant moderator for pair programming effectiveness. In the experiment by Arisholm et al. (2007, Sec. 3.3), developers in only *five* out of 98 pairs had any PP experience, which is a type of difficulty not to be expected among industrial development teams who at least sometimes pair. Additionally, the pairs in the experiment were formed by the researchers such that both developers were on the same level of *expertise*. This is something industrial teams *could* aim at, but does not appear to be a natural limitation.

Most experimental tasks do not take long and are based on small programs. The ‘system’ the developers in the large experiment worked in was quite small: Even the ‘complex’ version had only 12 classes with 287 lines of code and the average pair completed their three tasks in little over an hour (*ibid.*, Tables 9 & 12). Such studies can only account for the short-term effects of pair programming, but not for, say, building up shared knowledge in a team. Quality is commonly measured in number of defects or even reduced to a binary value (*ibid.*, Sec. 3.6.1).

The software system in the experiment was unknown to the subjects, meaning they had to rely exclusively on their software development skills to understand it from scratch—which, again, is a type of task difficulty that is not representative of everyday industrial software development. Any effect that might result from one partner being already familiar with at least some parts of the system (the COMBINE mechanism) cannot arise in such setups.

Yet, even within the confined setup created by the researchers and the large number of subjects recruited, the authors conclude that they “*are still far from being able to explain why we observe the given effects*” (*ibid.*, Sec. 6.5).

**Problem Category 3: Pair Programming as a “Black Box”** Even if more elaborated (and expensive) experiments were conducted which controlled for more moderating variables, used

larger systems, and covered longer time frames: The design of such studies still assumes that PP is something that developers can simply *apply* when told to do so, like being exposed to either one or another stimulus.

However, even with similar expertise, task difficulty, training, etc., there is no canonical way how to *do PP*: Do the pair members first brainstorm on a number of possible ideas, discuss them in detail, and then follow an agreed upon plan? Or do they follow the ad hoc idea of one developer until they hit a dead end and then switch? Or does one developer basically work alone, while the other watches silently? It is not realistic to assume that all pairs follow the same process (or at least processes with similar effects on time, quality, etc.).

Based on what is (not) known about the moderating variables that make pair programming more or less effective, it is not clear how a fair benchmark would look like. Without such understanding, programming contests between pairs and solos are of limited scientific value. To gather more insights, it appears more reasonable to study pair programming under natural conditions, where and when it used by the people who eventually decide whether or not to apply a practice anyway: the developers.

### 2.3.4 b) Studying Pair Programming as a Practice

Unlike controlled experiments, experience reports and observational studies pertain to real software development events—with all the phenomena that naturally occur, but that would not be easily thought of (and then adequately recreated) in a laboratory setting. Before I discuss observational field studies that followed a rigid method for collecting and analyzing data, I first come back to the experience report by Belshee, from which I already cited the two anecdotes of knowledge transfer in a pair-programming-only team (the viral paste stack and the quickly naturalizing new-hire, see my discussion on page 46).

#### *On the Benefits of Fieldwork*

Belshee (2005) performed an informal quasi-experiment in his software development team. The whole team pair-programmed all the time and switched partners often according to one of two switching styles: In the *fixed style*, one developer stays on the task until it is completed and gets a new partner every switch cycle; in the *alternating style*, each developer stays on a task for two switch cycles, starting as a beginner with an expert partner, then becoming the expert with a new beginner partner, and finally going on to the next task as a beginner again. The team tried both styles with various rotation rates ranging from less than an hour to three days. They made two key observations with regard to their overall productivity in terms of story points per week and proportion of unanticipated work items such as bugs: (1) for rates less than an hour, the *fixed style* is better; (2) the *alternating style* is consistently better for slower rates, with peak productivity at 90 min, and slowly flattening out with longer times between the switches.

Belshee does neither explicitly mention the number of data points nor concrete productivity measures. However, it is not the quasi-experiment *per se*, the precise point of break-even of the rotation styles, nor the gradient of the productivity curve that are interesting; rather, it is Belshee's interpretation: The overall effectiveness of a pair gets smaller the longer they work together on a task, because they lack "*the beginner's mind*" of a fresh pair member. However, developers need some time to build up enough understanding of the task and must not switch tasks too early. For short cycles, having one developer become (and stay) an expert on the task (*fixed style*) is the only way for task-relevant knowledge *not* to get lost between pair switches. Short cycles in the *alternating style* led to the 'new pair' having to interrupt the developer who just left the task too often. With intervals larger than one hour, however, enough knowledge

transfer from the expert partner to the beginner partner has taken place to increase the latter's level of expertise sufficiently such that a designated expert is no longer needed.

Belshee's findings illustrate the importance of studying naturalistic settings (a point I will come back to in Section 3.2.2): He observed his colleagues dealing with the different rotation styles and rates and ultimately 'discovered' the two competing mechanisms of (a) the need to build up task-relevant knowledge in order to be productive and (b) "*the beginner's mind*" which increases a pair's efficiency.

The study by Chong & Siino (2006) is another case in point: In development teams which do or do not employ pair programming, they analyzed the interrupt behavior—which is unlikely to be observed in laboratory settings. They report that both solo and pair programmers get interrupted by others three to four times per hour (*ibid.*, p. 31), but pairs have the advantages of (a) providing their surrounding with cues on when an interrupt might be minimally disruptive through their ongoing conversation and (b) being able to carry on the work with one developer while the partner deals with the interrupt (*ibid.*, p. 34). Furthermore, in comparison to solo developers, pairs appear to get interrupted more with technical questions than by social interactions such as jokes and chit-chat (*ibid.*, p. 33).

### ***Observational Field Studies on the Project Level***

Some researchers collect data in software development teams who employ pair programming as a practice, usually relying on more than one data source. Although the pair process itself (A5) is not much of an issue here, some analyses may allow to learn something about the effectiveness of the PP work mode (A1) and the kind of situations it is used in (A2 to A4).

The research group around Silliti and Succi published six studies (2008, 2009, 2011, 2012, 2013, 2014) in which they analyze different aspects of software development based on 14 months worth of automatically collected data from a team of 17 software developers who occasionally employ pair programming. Their data collection tool records developer activities in terms of timestamp, currently focused class or method in the integrated development environment (IDE), and window title outside the IDE. In the beginning of a programming session, the developers would enter into the tool the issue they intend to work on, whether they work as a pair, and if so, with whom; after 25 minutes, a notification would pop up, asking whether the developer(s) want to take a break, continue the task, or work on another one (Coman et al., 2014, Sec. 3.2). In the individual analyses, the researchers look at the *frequency* and *amount* of pair programming over time in general and per task (Coman et al., 2008, 2014), frequency of expert/novice pair *constellations* over time (Fronza et al., 2009), developer *attention* (Sillitti et al., 2012), and *defect density* (Phaphoom et al., 2011; di Bella et al., 2013, who also incorporate information from the version control system). For their particular team, which consists of 2 newly hired and 15 experienced team members, the researchers report the following findings pertaining to PP:

- **Effectiveness (A1):** To assess the effect of pair programming on source code quality in terms of defect density (corresponding to the UNDERSTAND mechanism), Phaphoom et al. (2011) and di Bella et al. (2013) calculate a time-based 'pair programming ratio' for methods that were changed in the context of user story implementations and defect removals.<sup>15</sup> The authors consider methods with *any* ratio greater than zero to be the result of "*pair work*". On average, such methods had lower defect rates than purely solo-programmed methods. The difference, however, was small and not statistically significant (di Bella et al., 2013, Sec. 6).

<sup>15</sup>Despite their automated data collection, the researchers could extract the necessary data (i.e., which developer (pair) touched which specific methods when and for how long) for only 8% percent of the defects and 9% of the user stories. In absolute terms, these are 39 defect corrections and 144 user story implementations touching a total of 377 and 1,904 methods, respectively (di Bella et al., 2013, Table 13 & Sec. 3.6).

- **Task Suitability (A<sub>3</sub>):** On days where developers pair-programmed—which they did less than once every five workdays—they did so for roughly 40% of their development time, leading to an overall PP ratio of less than 8% (based on data from the first three months of the observation period, Coman et al., 2008, Sec. 3.1 & 4). About five out of six issues were worked on completely alone; for the rest, the issue-owning developer worked together with a partner at least once (Coman et al., 2014, Sec. 4.2), for which the authors describe different solo/pair patterns, such as *mainly solo with few pair sessions* or *first pair then solo sessions* (*ibid.*, Sec. 4.3).<sup>16</sup>

This could be interpreted as an indication that the developers, by far, did not deem all their tasks suitable for pair programming. Furthermore, as there are different patterns of pair programming usage per task, *suitability* may not be a static binary property. However, the researchers neither analyze the characteristics of different types of tasks nor the developers' motivations to (not) work in pairs, which could depend also on external factors and just as well have little to do with the tasks as such.

- **Pair Constellations (A<sub>4</sub>):** The two newcomers to the team appear to have went through a four-phase process (Fronza et al., 2009, Sec. 6): For the first month, they pair-programmed half of their time, mostly with pre-existing team members. Then they worked mostly alone for two months, before working in pairs again most of their time for five months—this time mostly among themselves. Afterwards, the newcomers' pairing frequency and choice of partners became indistinguishable from the original team members. The researchers interpret these four phases as (1) training sessions for knowledge transfer, (2) consolidation of knowledge, (3) maturation, and (4) complete integration.

Although this interpretation appears plausible, no additional data was collected which could support it (such as records of the developers' actual intentions). Different pair constellations (in terms of the developers' knowledge of the project) may have different properties which make developers seek or avoid them. But, again, no additional data to go beyond a simple newbie/expert dichotomy was analyzed to answer this question.

Although the research group recorded large quantities of data *during* pair programming sessions, none of it is particularly helpful for understanding *how* pair programming actually works (A<sub>5</sub>) and which types of knowledge are important (A<sub>2</sub>). Even when assuming the collected data is correct in terms of pair constellations and session duration, the developers' particular goals and interactions in each of these sessions were not recorded.<sup>17</sup>

Two other (at least partially) industrial studies included developer interviews to capture their views as well as first-hand observations of PP sessions: Hulkko & Abrahamsson (2005) describe four projects (three of which with commercial interest) with a duration of 5 to 8 weeks. The developers were a mix of students and professionals who were encouraged to work in pairs. The researchers collected data through “*pair programming sheets*” which they asked the developers to fill out and through team interviews at the end of the projects. PP sessions were also observed in one of the four cases. In the other study, Vanhanen & Korpi (2007) followed a team of four developers for a little more than 14 weeks. Their developer interviews, time-tracking, and everyday observations were mostly concerned with PP as a practice, i.e., the high-level decision when and how to work in pairs. Additionally, they asked

---

<sup>16</sup>In these studies, a “session” refers to a stretch of data collection of 25 minutes at most.

<sup>17</sup>The only analysis in this publication series which looked at what actually happens during a PP session focused on developer attention: Pair programmers spent more time in the IDE and less in the e-mail client & web browser; they also switched programs less often (Sillitti et al., 2012, Sec. IV.A). Pairs stayed almost twice as long on a task (*ibid.*, Sec. IV.C), averaging close to the team's pop-up window enforced “session limit” of 25 minutes.

the developers for their subjective knowledge levels per module over the observation period.<sup>18</sup> Their findings, grouped by the five aspects, can be summarized as follows:

- **Effectiveness (A<sub>1</sub>):** While a lower defect density was expected for code written by pairs (a possible effect of the UNDERSTAND mechanism), no such effect could be consistently observed: In one case, there was virtually no difference, while in another project relative defect densities differed by a factor of six in favor of pair programming (Hulkko & Abrahamsson, 2005, Sec. 3.2.2).  
Pair programming appeared to be an effective means for knowledge transfer in the team (an effect of the LEARN mechanism): Despite a growing system, the subjective knowledge levels showed little changes to the team averages while the variance (characterizing intra-team differences) decreased over time (Vanhanen & Korpi, 2007, Sec. 6.3).
- **Types of Knowledge (A<sub>2</sub>):** Both studies agree, based on first-hand observations and interviews, that code knowledge or system understanding was the most relevant type of knowledge for the developers (Hulkko & Abrahamsson, 2005, Sec. 3.2.1; Vanhanen & Korpi, 2007, Sec. 5.3). All developers felt their system knowledge improved due to pairing, two out of four felt they learned their tools better (Vanhanen & Korpi, 2007, Sec. 6.3).
- **Task Suitability (A<sub>3</sub>):** About three quarters of the programming time was spent in pairs (72% in Vanhanen & Korpi, 2007, Sec. 5.1; 40–90% in Hulkko & Abrahamsson, 2005, Fig. 3). The PP ratio was lower for easy activities (Vanhanen & Korpi, 2007, Sec. 5.1), and also lower in later iterations (Hulkko & Abrahamsson, 2005, Sec. 3.2.1). Based on their data (mostly the interviews), Hulkko & Abrahamsson (*ibid.*, Sec. 3.2.1) conclude that pair programming is especially useful in the beginning of the project for gaining understanding of the system, for tasks which are too complex for any individual developer, and for tasks involving code with many dependencies.
- **Pair Process (A<sub>5</sub>):** Vanhanen & Korpi (2007, Sec. 5.3) noted continuous communication during the sessions: If one developer knew her way around the problem-relevant code already, she took the lead and would explain her actions.

The findings from observational field studies on knowledge-related aspects of pair programming can be summarized as follows: Pair programming appears to be effective for knowledge transfer (for perceived knowledge levels, at least; effects on technical outcomes appear similarly elusive as in the controlled experiments discussed in Section 2.3.4a); there are different types of knowledge that may be transferred (in particular about the system, possibly also about tools); communication is crucial in pair programming; and not all task types and pair constellations may be suitable for pair programming (knowledge intensity of the task and knowledge distribution in the pair may play a role).

### 2.3.4 c) Qualitative-Quantitative Studies on the Pair Programming Work Mode

With the exception of a few reported first-hand observations of PP sessions in the field studies discussed in Section 2.3.4b, pair programming is often treated as a “black box” (Plonka, 2012, pp. 5–6; Salinger, 2013, p. 42): Many studies consider the effects of pair programming (e.g., on quality or effort), either in comparison with solo programming or based on external factors such as task complexity and developer expertise, but disregard the developers’ interactions that make up a pair programming session and eventually produce these effects.

In the next two sections, I discuss studies which focus on the inner workings of pair programming sessions, i.e., mostly considering A<sub>5</sub>, but other aspects as well. Such studies all

<sup>18</sup>This design is similar to an earlier educational study by the same first author (Vanhanen & Lassenius, 2005) in which pair programming had a non-significant positive effect (my discussion on page 56).

start with a *qualitative analysis*, which means that they deal with unstructured data (such as audio or video recordings), annotate (or “code”) fragments thereof, and thereby assign some *meaning* to it. The studies differ in what comes next: *Qualitative-quantitative studies* (discussed in this section) aim a complete coding of the data in order to be able to treat it as *quantitative data* which can then be statistically analyzed. In contrast, purely *qualitative studies* (discussed in Section 2.3.4d) are geared towards the *discovery* and deep *understanding* of relevant phenomena.

### **Conflict Handling**

Domino et al. (2003) studied how conflict handling styles reflect on pair performance. They recorded seven pairs of part-time students with industry experience who worked on three programming tasks (length between 20 and 45 minutes). The researchers used a rating form to capture the number of “*conflict episodes*”, whether these pertained to the task or to the pair’s relationship, which conflict handling style was employed (*integrating*, *obliging*, *dominating*, *avoiding*, or *compromising*), and whether and how the conflict was resolved. Other measures were the “*faithfulness to the pair programming method*” (‘measured’ as amount of interactions on a 5-point scale), the pair’s overall work pattern (*read-plan-work*, *read-separate-combine*, or *split-work*), and the individuals’ “*general cognitive ability*” (*ibid.*, Sec. 4.1–4.2).

The statistical analysis led to no significant results. However, based on an informal discussion of the best and the worst pair, the authors note that the two extreme pairs had comparable cognitive abilities, but differed in their style of collaboration: One pair member of the worst pair dominated the sessions more and more over time, conflicts were not resolved but avoided, until the other pair member completely withdrew (*ibid.*, Sec. 4.3). I analyzed similar, but less extreme cases and call this a **Breakdown** (see Section 6.3).

### **Driver and Navigator Roles**

Freudenberg (née Bryant) et al. (2004, 2006, 2008; see also the first author’s PhD thesis, Freudenberg, 2006) performed qualitative-quantitative analyses of industrial pair programming, i.e., professional software developers working on their everyday tasks. A common theme of these analyses were the roles of *driver and navigator* which are sometimes used to “explain” pair programming (see Section 2.3.1a). The researchers developed different coding schemes, coded individual utterances in session transcripts, and then counted occurrences across developers and sessions.

In the first study, Bryant (2004) analyzed 14 one-hour sessions from one company for which she coded all individual utterances with one of eleven codes such as *question* or *suggestion*. The researcher notes differences between developers with and without pair programming experience: The utterance type frequencies of “*expert pair programmers*” do not depend on whether they were driving or not, while having the keyboard *does* have an effect on pair programming novices (*ibid.*, Sec. 5.2). Bryant et al. (2006, 2008) therefore excluded developers with less than six months of pair programming experience from their further analyses.

In the second study, Bryant et al. (2006) analyzed 23 one-hour sessions from four different companies with regard to the degree of collaboration concerning different types of tasks such as *refactor* or *configure environment*. Overall, 93% of all tasks were dealt with collaboratively, i.e., *both* partners verbally contributed some new information (*ibid.*, Sec. 4). The analysis also yielded the relative frequencies of the twelve task types. The most prevalent type (with 613 out of 2,735, or 22% of all tasks) was *comprehension [of the code]* (Freudenberg, 2006, p. 143), which relates directly to knowledge transfer in the sense of the UNDERSTAND mechanism.

Their last study (Bryant et al., 2008) covered 24 one-hour sessions for which they coded all 14,886 utterances according to their level of abstraction (*syntax*, *detailed*, *program blocks*,

*bridging, real world, and vague*) to finally put the *driver-navigator* metaphor to test.<sup>19</sup> Their analysis revealed no significant difference between the drivers' and the navigators' respective histograms (*ibid.*, Sec. 5.1). The authors interpret this as evidence that these two roles do *not* generally talk (and think) at different levels of abstraction (*ibid.*, Sec. 5.3). Purely qualitative studies would later investigate this further (see Section 2.3.4d, page 83). Note, however, that Bryant (2004, Sec. 5.2) explicitly excluded unexperienced pair programmers *because* of their different abstraction level histograms. In other words: There may be role differences for PP beginners.

Plonka et al. (2011) analyzed 21 industrial pair programming sessions recorded in four different companies.<sup>20</sup> They found that one pair member had the *driver* role for the vast majority of the duration of most sessions. However, the labels *driver and navigator* were not meaningful for one third of the pair programming time during which the developers mostly communicated verbally without any computer interaction. Role switches occurred, on average, about 15 times per hour, and most of the time they were initiated by the *navigator* and without verbal communication. Plonka et al. also employed purely qualitative methods on the same data; I discuss that research on pages 81 and 84.

### **Pair Members' Personalities**

Walle & Hannay (2009) investigated the effect of the pair members' personalities on their communication, based on the audio-recordings of 44 PP sessions<sup>21</sup> from the large experiment done by Arisholm et al. (2007, my discussion on page 65). The researchers used a Big Five personality test to determine the mean elevation of the pair per trait as well as their difference (similar to Hannay et al., 2010, discussed on page 66). Walle & Hannay (2009, pp. 204–205) coded the sessions on two levels: Based on the pair's discourse, they segmented a session into *interaction sequences* which they categorized as one out of ten *task focus* types, such as *Code Comprehension, Compile and Test, Programming, or Other relevant tasks*. Since they focused on problem-solving during programming, they selected all *Code Comprehension* and *Programming* sequences to be then coded in detail according to how they began, their interaction pattern, their cognitive level, how they ended, and their result.

The researchers aggregated these codings into 44 data points (one per session) each with 10 dimensions characterizing the pair member's personalities and 60 dimensions characterizing their communication (so-called "*collaboration categories*").<sup>22</sup> They conducted a decision tree analysis (*ibid.*, pp. 209–210, again similar to Hannay et al., 2010, discussed on page 66), which iteratively splits the data points according to a threshold value on one of the independent variables (here: personality dimensions) to maximize the significance of the effect on some dependent variable (here: collaboration category). They tested a number of hypotheses and made the following observations:

- **Personality does affect the type of cooperation.** All five personality factors significantly influence the time spent on one or more collaboration categories; three personality factors

<sup>19</sup>An earlier version of this article was also published at a conference (Freudenberg et al., 2007).

<sup>20</sup>These recordings are part of a larger repository of PP sessions to which multiple researchers contributed and which I describe in Section 4.3.

<sup>21</sup>The sessions' duration is not reported, but is probably between 20 and 50 minutes: Arisholm et al. (2007, Table 9) report a mean work duration of 63 minutes for all three tasks—two warm-up tasks and the main task—the last of which was the only one analyzed by Walle & Hannay (2009, p. 204).

<sup>22</sup>The coarse coding led to ten dimensions (one for each task focus type), the detailed coding added another 20 (that is how many different predefined values there were). For each of these 30 dimensions, the researchers considered relative frequencies and absolute time spent, bringing the total to 60 communication dimensions. The 10 personality dimensions come from the Big Five dimensions, each represented with pair mean value and pair difference.

also significantly influence the frequency of one or more collaboration categories (Walle & Hannay, 2009, Table 2). However, given that 10 personality measures were put against 60 collaboration categories, *some* statistically ‘significant’ influence (with  $p < 0.05$  or *1 in 20*) is to be expected.

- **Variability in personalities increases the amount of communication-intensive collaboration.** Pairs with different personality scores had significantly more communication-intensive collaboration, e.g., an *elaborative* or *responsive interaction pattern* as opposed to *nonresponsive* or *stonewalling interaction pattern* (*ibid.*, pp. 210–211). Differences in extraversion had the biggest impact on the occurrence of communication-intensive collaboration (*ibid.*, p. 211), which appears to be mostly due to more time spent on *off-topic sequences* and *cross-purpose interaction pattern* (i.e., both developers talking about different things at the same time), and lower frequency of *nonresponsive interaction pattern* (*ibid.*, Table 3). This makes the *extraversion difference* effect detected by Hannay et al. (2010, discussed on page 66) more tangible.

The other tested hypotheses could not be supported by the data: Collectively high extraversion of the pair does neither lead to more communication-intensive collaboration nor to more metacognitive statements; furthermore, higher agreeableness of the pair does not lead to more off-topic communication (Walle & Hannay, 2009, p. 211).

#### **Discussion: Problems of Qualitative-Quantitative Approaches**

The qualitative-quantitative studies presented above share a number of problems: Their research process is more or less *intransparent* in that their respective coding schemes and/or coding scheme development processes are not discussed in detail and are *inadequate* for understanding the pair programming process.

Qualitative-Quantitative PP Studies	Type	Possibly Problematic Areas			
		1: Scheme Development	2: Rich Codes	3: Rich Data	4: Pair Process
Bryant et al. (2004, 2006, 2008)	M K	(✓)	(✓)	(✓)	✗
Domino et al. (2003)	M U	✗	(✓)	✓	✗
Walle & Hannay (2009)	M U	(✓)	(✓)	(✓)	✗
Plonka et al. (2011)	M K	(✓)	✗	✓	✗

**Table 2.10:** Overview of recurring problems in qualitative-quantitative PP research. Here, all studies involved multiple pairs (M) who either work in existing projects/context known (K) or unknown to them (U). See main text for detailed description of the problem areas. In the respective areas, the studies have ✓ – no problems, (✓) – some problems, or ✗ – considerable problems.

**Problem 1: Coding Scheme Development Process** Domino et al. (2003, p. 48) used a “*pre-established rating form*” with five conflict handling styles, five levels of interaction amount, and three work patterns. The conflict handling styles correspond to five scales of an established inventory for which the participants filled out a questionnaire (*ibid.*, p. 47). It is unclear, however, how the researchers related a participant’s general self-assessment to her concrete behavior exhibited in a specific conflict episode. The three work patterns appear out of thin air.

The coding schemes used by Bryant et al. (2004, 2006) and Walle & Hannay (2009) are at least in part assembled from various literature. The coding scheme presented in Bryant (2004, Table 2), for example, is described as an extension of a list of “*typical pair interactions*” observed

by practitioner Wake (2002, pp. 72–73), but without an explanation which amendments were made for what reason. Incidentally, this point is elaborated on in the author’s PhD thesis (Freudenberg, 2006, p. 37): One change was motivated by making the researcher’s work easier, such that the coding “*did not require any attempt on the part of the coder to ascertain the implicit motivation behind the question (i.e. whether it was a request for help, or, for example, a test of knowledge)*”. The coding schema used in Bryant (2004) therefore appears to address the surface structure of the developers’ utterances, but not necessarily their meaning.

The coding scheme used by Walle & Hannay (2009, p. 205) is the result of combining two preliminary coding schemes, the first of which was assembled from various sources, while the second scheme was “*developed on the basis of samples of [the] audio recordings*”. Large parts of the resulting coding scheme were inherited from the existing sources, with only a few amendments based on their own data. It is not clear, however, what the driving force behind amending the existing schemes was: Having enough labels to code whole sessions *exhaustively* or having a *rich* set of concepts to adequately capture and distinguish relevant phenomena. This leads directly to the second problem.

**Problem 2: Inadequate Coding Scheme** In the studies, it is not clear whether the distinctions manifest in the coding schemes are actually relevant for the data at hand. To give an example from Bryant et al. (2008, Sec. 5.1) which was concerned with the levels of abstraction of the pair members’ utterances: Of the almost 15,000 utterances, 57% were classified as *vague* and only 43% could be assigned to an abstraction level. The majority of those is on the mid-range level *program block* while the extreme low- and high-levels are rare. This indicates that the coding scheme is *exhaustive* in that it could cover all utterances, but not *rich* as a single or very few categories serve as a default label for most phenomena, which is not helpful in explaining *how* pair programming actually works.

Domino et al. (2003) and Walle & Hannay (2009) do not provide enough information on their coding process and/or result (e.g., in the form of descriptive statistics) to make an assessment in this regard.<sup>23</sup> The coding scheme used by Plonka et al. (2011) is presented in detail and it is rather technical (with codes such as *D1.isDriving*, *DriverInitiated switch*, or *verballyInitiated switch*), which is not rich either.

**Problem 3: Limited Accuracy** A minor problem pertains to the limited recording of the actual events. Bryant et al. (2008) and Walle & Hannay (2009) worked exclusively with audio-recordings or transcripts thereof, not knowing what the developers looked at or pointed to on their screen. Without this crucial context, understanding what the developers talk about is (even more) difficult for the not-involved researcher. Bryant et al. (2008, Sec. 5.1) explicitly state that they classified some utterances as *vague* simply because they could not determine their abstraction level based on the transcript alone.

**Problem 4: Disregard Process** The three lines of research of Domino et al., Walle & Hannay, and Bryant et al. all aimed at a statistical analysis, which means that their conclusions depend on the consistent application of their coding schemes in order to produce reliable data to analyze. Consequentially, the researchers who used more than one coder were concerned with inter-agreement (Domino et al., 2003, Sec. 4.2; Walle & Hannay, 2009, Sec. 3.4), which is only a measure of how consistently a coding scheme can be applied (e.g., by staff exclusively hired for coding), but does *not* speak of its relevance.

<sup>23</sup>Literally the *only* piece of concrete information on the contents of the analyzed sessions is that the pairs, on average, spent 18.9% of their time *Programming Aloud* (Walle & Hannay, 2009, Fig. 3).

Ultimately, all these studies condensed whole pair programming sessions into single data points (7, 24, and 44 data points, respectively) to perform statistical analyses on, but did not actually characterize the underlying processes.

### **Summary of Qualitative-Quantitative Studies**

Studying the pair programming process with a predefined perspective, i.e., by focusing on the *application* as opposed to the *development* of a coding scheme, and then statistically analyzing the resulting codings appears to either produce no significant differences where some were expected (Domino et al., 2003; Bryant et al., 2008; Walle & Hannay, 2009, p. 211) or to produce so many of them to not see the forest for the trees (Walle & Hannay, 2009, p. 210). Nevertheless, a number of qualitative results on pair constellations and the pair programming process (A4 and A5) can still be summarized:

- Program comprehension is a large part of what happens during pair programming sessions (Bryant et al., 2006; Walle & Hannay, 2009).
- Pair performance does not depend on individual cognitive abilities alone; instead, a dominating partner and the avoidance of conflicts may be problematic (Domino et al., 2003). The pair members' personalities affect their collaboration; in particular, differences in their personalities appear to lead to more verbal communication (Walle & Hannay, 2009). Also, whether or not developers have experience with the pair programming work mode affects how they work together (Bryant, 2004).
- The roles of “driver and navigator” are generally not evenly distributed among the pair members, i.e., one developer controls keyboard and mouse more than 60% of the time. However, for one third of the time, nobody touches the input devices, so the two roles do not apply here (Plonka et al., 2011).
- Both pair members verbally contribute to the vast majority of the tasks dealt with in a session (Bryant et al., 2006). The levels of abstraction pair programmers actually talk at do not correspond to who is “driving” and who is “navigating” (Bryant et al., 2008).

Due to the research design that brought them forth, neither of these results explain *how* pair programming actually works, but merely point to directions that full qualitative analyses (i.e., those not aiming at the testing of hypotheses but the development of theories) may investigate.

### **2.3.4 d) Qualitative Analyses of the Pair Programming Work Mode**

As I will discuss in Section 3.2, qualitative research methods aim at discovering relevant phenomena, at developing theories instead of testing pre-formulated hypotheses. They develop rich theoretical concepts to describe or explain phenomena of interest. Apart from a number of core features, qualitative research methods differ in the very basic assumptions they make about their subject (see Section 3.2 for details). The schools of qualitative research used to study pair programming include *Distributed Cognition*, *Ethnomethodology* and *Ethnography*, and the *Grounded Theory Methodology*. In this section, I discuss the according studies, which mostly look at the pair programming process (A5) and, to a lesser degree, pair constellations and knowledge types (A2 and A4). First, however, I discuss so-called *Protocol Analysis* studies.

#### **Method: Protocol Analysis**

In psychology, protocol analysis is a method for studying cognitive processes that relies on a *protocol* which “provides a running series of responses that can be used to infer the sequence of mental states”, e.g., in the form of eye movements or computer interactions (Anderson, 1987, p. 472). Typically, though, these protocols record verbal utterances of subjects who were asked to “think aloud” while working on some task. In software engineering research, the appeal of studying a *pair* of programmers lies in that the two subjects, at least in principle, express their

respective thoughts anyway, making their think-aloud natural. Psychological experiments that use a *protocol* of events contrast with studies which are only concerned with the outcome of a cognitive process (Crutcher, 1994, p. 242). “Protocol analysis” is hence a slight misnomer as it refers to the way data is *collected*, and not to how it is *analyzed*.

**Maintaining a Body of Concepts** Xu et al. (2005) compared the cognitive processes of two student pairs with programming experience (“*intermediates*”) and one “*expert*” pair who all worked on the Bowling Game kata.<sup>24</sup> In the transcripts, the researchers identified and counted domain concepts the developers considered, discussed, and included in or removed from the source code. While the experts revised their solution, by adding concepts *and* removing obsolete ones, the students mostly only added new concepts. In their code, the experts deleted methods and classes, while the students adhered to their initial design and tried to fit in new concepts (*ibid.*, Sec. 4.1). Xu et al.’s “knowledge” notion is a bit peculiar, as they do not differentiate between the concepts in a pair’s discourse and those embedded in the source code. To them, a proposal such as “*We have to delete the old method for getScore(!)*” decreases “*the number of concepts in the knowledge of [the] pair*” (*ibid.*, Fig. 2 & 3). Apart from this idiosyncrasy, I see two major threats to the validity of the comparison with the experts:

- **Invalid Data:** The expert pair’s session was not recorded by the researchers. Instead, the transcript comes from the book *Agile Software Development: Principles, Patterns, and Practices*, where it is introduced as “*a pretty faithful reenactment of a programming episode*” (Martin, 2002, p. 43). Its faithfulness probably refers to the rough outline of the design process but not to the accuracy of the verbal utterances.<sup>25</sup> As Xu et al. (2005, Sec. 4.1) themselves note, the experts’ “*dialogue is edited from the original recorded data [...] [p]hrasing is more mature and better articulated*”. But at the same time they characterize the expert’s discourse as “*more coherent and richer*” than the students’, who in turn “*often referred to more than one concept in one phrase*”—which affects how Xu et al. calculate their concept counter. As the many verbatim examples in my thesis will illustrate, incoherent utterances in spontaneous speech are not unusual, even for experienced developers.
- **Unfair Comparison:** The students had no prior experience with either pair programming, refactoring, test-first development, or the “*domain*” of scoring a bowling game (*ibid.*, Sec. 3.3 & 3.4). It is not clear whether the students were tasked with just meeting the specification of counting bowling scores, or whether they were also required to refactor their code (to whatever end) and write tests first. Since the students were equipped with reading material on all three practices (*ibid.*, Sec. 3.5), I assume they had to do all at once. But this is an unfair comparison with the “*experts*”, who *decided* during their session what the scope should be (scoring a single game rather than a whole league of players), to write tests first, and to refactor along the way—and with at least one pair member who “*used to be a pretty good bowler*” (Martin, 2002, pp. 44 & 47).

Overall, this study does not add much to the understanding of the pair programming process (A5)—which is not all that surprising given the authors were primarily interested in the underlying cognitive processes for which pair programmers were just a means to an end.<sup>26</sup>

**Activity Patterns** A study by Cao & Xu (2005), in contrast, had an explicit PP focus. They video-taped pair programming sessions of six pairs during a nine-week project of students with three years work experience average. The developers were assigned to a high, medium,

<sup>24</sup>See <http://codingdojo.org/kata/Bowling/>

<sup>25</sup>As Martin (2002, p. 43) writes: “*We made lots of mistakes while doing this. Some of the mistakes are in code, some are in logic, some are in design, and some are in requirements.*”

<sup>26</sup>This characterization was also independently made by Salinger (2013, p. 41).

or low competence level (based on their performance in assignments, exams, and “*interaction with instructor*”) which I abbreviate as HI, MED, and LO. Overall, they observed three pair constellations HI-HI, MED-MED, and HI-LO, with two pairs each.<sup>27</sup> Cao & Xu (2005, p. 3) mention “*protocol analysis*” in their paper, but neither provide any details nor reference a method paper or book. Their report does not reflect a particular interest in cognitive processes, so it is unlikely they borrowed this term from psychology. Cao & Xu (*ibid.*, pp. 4–8) discuss five “*activity patterns*” (A<sub>5</sub>) which differ in their details depending on the pair constellation (A<sub>4</sub>):

- **One Leader & Summarizing Results:** There appears to be always one developer dominating the session, depending on the competence and personality of the developers. In MED-MED pairs, the leader is concerned with details; in HI-HI and HI-LO pairs, with the goal and strategy for the session. Only HI-HI pairs frequently summarize their current status mid-session and possibly adjust their goal.
- **Ask for Opinions & Critique Partner’s Approach:** HI- and MED-developers ask their partners for opinions on how to proceed along the way. In HI-HI pairs, longer discussions ensue; MED-MED pairs have simpler questions and fewer discussions. Both HI-HI and MED-MED pairs frequently point out problems in each other’s proposals and work together to refine them.

In HI-LO pairs, in contrast, the LO-partner cannot provide valuable responses and always agrees, so the HI-partner eventually stops asking.

- **Explanatory Activities:** All developers try to make sense of what they are doing (UNDERSTAND mechanism). HI-HI pairs frequently add details to their current understanding, correct misunderstandings, and fill respective knowledge gaps, whereas the MED-MED pairs are not always able to fill the gaps.

The HI-partner in the HI-LO pairs explains most of her actions, as well as the design and the code—she fills her knowledge gaps on her own. The LO-developer enjoys the session (unlike their HO-partners), presumably because she learns a lot through the explanations of her partner (LEARN mechanism).

It appears as if knowledge transfer in the sense of UNDERSTAND (i.e., faster comprehension together) only occurs in HI-HI pairs, whereas LEARN (i.e., leaving the session more fit for future tasks) is limited to HI-LO pairs.

### **Method: Distributed Cognition**

For an individual, *cognition* refers to the set of information processing abilities which includes perception, memory, and reasoning (see Section 2.2.2). *Distributed Cognition* is the extension of this metaphor to larger “*functional systems*” comprising multiple actors and artifacts working together to achieve a common goal by propagating and transforming *knowledge representations* (Rogers & Ellis, 1994, pp. 121–122). Examples for such “*systems*” include a crew navigating a ship (Hutchins, 1989) and pilots in a cockpit flying an airplane (Hutchins & Klausen, 1998). A basic assumption of Distributed Cognition is that such a system has ‘cognitive properties’ that differ from those of the involved individuals: Knowledge is distributed and partially redundant, so individuals need to constantly work on matching their individual knowledge with shared knowledge through communication (Rogers & Ellis, 1994, p. 123).

**Maintaining Common Ground** Flor & Hutchins (1991, 1998) performed a detailed analysis of a single pair programming session. From the perspective of Distributed Cognition, such a

---

<sup>27</sup>Three additional HI-LO pairs and one MED-LO pair dropped out of the project. It is not clear from the paper how often each pair was recorded and how long their sessions lasted. At one point, the authors speak of “*the video tape of the two [HI-HI] pairs*”; later, it is “*the videos of [MED-MED] pairs*”; eventually they say that they “*taped one working session of two [HI-LO] pairs*” (Cao & Xu, 2005, pp. 4, 6, & 7, emphases added).

programmer pair and their computer(s) form a single “system”. The pair had to add a *whisper* command to an existing game unknown to them. In total, the pair performed 23 code changes over 2:40 hours. Flor & Hutchins (1991) cover the first code change (with 71 analyzed utterances) where the developers duplicate and amend a case block of three statements; Flor (1998) covers code changes #9 and #10 (22 analyzed utterances).<sup>28</sup>

Many of the 71 analyzed utterances during code change #1 are mono-thematic (they deal with keyboard commands for copying and pasting lines in the *vi* text editor), while many others are incomplete (e.g., “well”, “well how do we”, “so we’ll call it”), so they are not particularly representative of all programming activities. Nevertheless, Flor & Hutchins (1991) identify a number of properties of the system *pair programmers and their computer*:

- One property is inherent to programming and not specific to the pair situation: The developers copied and modified a piece of existing code. In Distributed Cognition terminology, this is **Reuse of System Knowledge** since the source code in a computer file is a representation of knowledge within the *system*, which another *part of the system* (here: a programmer) retrieves and reuses.
- Three additional properties are not specific to programming situations: (a) The two developers have a **Shared Goal**, both want to reuse an existing case block. (b) Many statements of the pair members are not fully specified, but still understood by the partner. This **Efficient Communication** is possible because the physical setup and their shared goal provide enough context for interpretation. (c) Both pair members engage in a **Joint Production of Course of Action**, meaning that the development process is not determined by either of the two alone.

From the analysis it is not clear how the pair arrived at its shared goal—it just appears. Assuming the pair had some negotiation prior to the recorded session, these three properties are known aspects of conversations in general (Fehler, 2005, pp. 1231–1232): (a) Negotiating how to advance a conversation, (b) understanding each other in spite of incomplete utterances, and (c) the overall course not being determined by any one participant alone.

- Three final properties are specific to pair programming. In particular, they hint at auxiliary mechanisms that appear more basic than COMBINE, UNDERSTAND, and LEARN.

First, the pair as a whole is **Considering more Alternatives**, i.e., more than one plan for their shared goal (manually copying the case block by typing vs. copying multiple lines into a buffer and pasting them afterwards). Second, the pair has a **Shared Memory for Old Plans**, which enables them to remember more chunks of information together than either of them alone. Finally, there is a **Division of Labor**: Not typing appears to allow for more free mental capacity.

It is unclear whether these findings are based only on the 71 utterances analyzed in the paper or on the whole PP session of 2:40 hours.<sup>29</sup> Although these properties are grounded in observations and are therefore *true* in the sense that additional observations would not invalidate them, it is not clear how relevant these findings are to understanding pair programming in general:

<sup>28</sup>It was disturbingly difficult to extract these facts, as Flor & Hutchins (1991) use “change” for three (!) different but related concepts: The task involved “*ten required changes*” (*ibid.*, p. 43), meaning that **ten program routines** needed to be written or amended. The pair did not conduct these logical changes *en bloc*, but as **23 fragments** which are listed in the appendix as “*change #1 ... #23*” (*ibid.*, p. 60). The main article discusses only the first of these fragments by rhetorically dividing it into **seven subsections** which are called “*Change #1 ... #7*” (*ibid.*, pp. 45–53 & 62), but should have been properly named #1.1 to #1.7.

<sup>29</sup>Both Flor & Hutchins (1991) and Flor (1998) reference an unpublished technical report which supposedly covers the whole session. It is not clear whether said report merely conducts the reconstruction of what happened during the session on a technical level or whether this reconstruction also served as the basis for deriving the above mentioned findings.

After all, the whole analysis in Flor & Hutchins (1991) is based on the conversation of two developers mostly fiddling with *vi* editor keyboard shortcuts to copy-paste five lines of code.

Flor (1998) covers an even shorter segment of the session in his second study: Based on 22 utterances, he identifies four types of relevant knowledge (task decomposition, code modifications associated with the (sub-)tasks, the system's compositional structure, and the program's behavior). With two developers, however, discrepancies in the individuals' representations (and/or with the representations manifest in the source code) will occur and can be detected and repaired. According to Flor, reconciling such differences or *maintaining common ground* is a large part of what happens during pair programming. However, his concepts of transferring representations between different "*media*" (the shared screen or "*the verbal medium*" consisting of the developers' utterances) through listening, talking, reading, and writing merely describe what needs to happen in any Distributed Cognition system. Hence, they do not offer much insight into pair programming as such.

### **Method: Ethnomethodology and Ethnography**

*Ethnographic* studies assume that a group of humans develops a 'culture' over time. This term is meant in a broad sense: it simply pertains to a set of behaviors and possibly also beliefs that are deemed "normal" within the group. The key to understanding these behaviors as a researcher is to immerse oneself in the natural setting and consider everything as 'strange'. A related type are *ethnomethodological* studies, which are concerned with the 'methods' used by human beings to make sense of their surroundings in their everyday life.

**Reading** Rooksby et al. (2006) did a seven-week ethnographic study in a software company on the role of *reading* during programming which the authors see as a special kind of reading: The reading itself is *occasioned* (i.e., every reading episode pertains to a specific purpose), its subject (the source code) is *orderly* and *analyzable*. Further, they claim that programmers need to learn this special way of reading as part of learning how to program (in the terms of ethnomethodology, it is a set of 'methods' used by programmers to make sense of their everyday world). Two short excerpts from pair programming sessions are to support this point. The pair members appear to immediately understand *why* their partner started reading certain parts of the source, indicating that programmers not only share these 'methods' (a type of knowledge, A2), but also rely on their partner to follow them. Apart from this general idea, however, the authors introduce no concepts and discuss no general mechanisms; just like Xu et al. (2005) and Sillito et al. (2008) (my discussion on pages 77 and 82, respectively), they used pair programmers as a means to investigate programming in general.

**Social Dynamics** Chong et al. (2006 and 2007) performed ethnographic observations in teams with a focus on how they pair-program. The researchers visited different teams once per week for several months, wrote down observations, and recorded conversations of PP sessions. They collected about 20 hours worth of material per team. Their first study on interrupt behavior in a development team (my discussion on page 69) was concerned with pair programming as a practice. In their second study, Chong & Hurlbutt (2007) describe a number of distinct in-session behaviors of pair programmers. In particular, they report large behavioral differences (A5) between pairs of developers with comparable levels of expertise and pairs with a larger gap (A4). In equal pairs, the process is a "*cohesive stream of discourse*" in which both partners "*moved together between the various levels of strategic thinking and implementation detail*", while a role separation in the style of *driver and navigator* would only be observed during "*short bursts of implementation*" (*ibid.*, Sec. 5.1). In contrast, when there was a large gap in expertise in the pair, the more experienced developer dominated the session, while the partner's behavior depended on the context: New team members would ask many questions in their first sessions,

but under time pressure, the less experienced pair member would become passive to not hamper the technical progress by asking too many questions (*ibid.*, Sec. 5.1). Chong & Hurlbutt (*ibid.*, Sec. 6.2) summarize the types of contexts they deem suitable for pair programming (A3): To ramp-up new team members, and when working with a relaxed schedule. They also list which types of knowledge might be transferred best (A2): Design patterns, tool features, language features. Their reported examples, however, only feature instances of code base explanations so their statement may not be evidence-based but a conjecture.

**Expert/Novice Behaviors** Plonka et al. (2015) revisit the same material used in other studies (which I discuss on pages 73 and 84) and employed Interaction Analysis, which is a form of ethnographic analysis that relies on video recordings (Jordan & Henderson, 1995, p. 35). Applying a number of heuristics, Plonka et al. (2015, Sec. 3.2.1) cut down the material from a total of 37 hours to five excerpts of 4 to 6 minutes to analyze developer behavior in sessions with a self-declared “expert/novice” constellation and the explicit goal of transferring knowledge. They identified behaviors which are (a) employed by the expert when the novice is driving and (b) employed by either developer when the expert is driving. In scenario (a), the expert may *nudge* (make a suggestion instead of telling), *prepare the environment* beforehand (e.g., by opening a useful file), *point out problems* instead of telling the solution, *gradually add information* if pointing out the problem does not work, or eventually *give clear instructions* by dictating what to type or naming shortcuts (*ibid.*, Sec. 4.1). In scenario (b), the novice may *ask for explanations* (which may then be given verbally or by showing), while the expert may *verbalize her activities* which might help the partner to understand the thought process (*ibid.*, Sec. 4.2). However, even for experts, understanding code and thinking about the partner’s suggestions at the same time can be challenging: Providing explanations takes some effort and becomes slower when they are unrelated to the expert’s current activity (*ibid.*, Sec. 4.4).

Plonka et al. (*ibid.*) do not explicitly discuss the types of knowledge that are transferred (A2). However, in her PhD thesis, Plonka (2012, pp. 197–198) distinguishes the three following types of topics: *Programming techniques* (coding standards and debugging techniques), *IDE and tools* (keyboard shortcuts and how to use debugger), and *existing code*. They did not, however, evaluate whether or not the novice does actually LEARN something from the expert. Rather, Plonka (*ibid.*, p. 196) speaks of “*learning opportunities*”. In fact, at least for scenario (a), when the novice is driving, all six given examples seem to illustrate how the expert does *not* explicitly explain something, but makes sure the novice develops the ‘right’ design idea, does not introduce defects, or does not forget to write a test, or she gives clear instructions without any explanation. In terms of *explicit* knowledge transfer, i.e., one in which both developers are consciously involved and can appreciate its effectiveness, Plonka et al. (2015, Ex. 7 & 8) distinguish but two cases, which are each triggered by the novice asking a question: The expert may provide a verbal explanation (about why some test cases do not adhere to the current style policy) and the expert may show something (how to use the debugger).

### **Method: Grounded Theory Methodology**

Many qualitative research approaches include assumptions on the phenomena of interest which ultimately shape the findings (e.g., the notions of *knowledge representations* in Distributed Cognition or *culture* in Ethnography). In comparison, the Grounded Theory Methodology (GTM) imposes less structure on the results of a study. Its hallmark is the rigorous application of a number of practices to structure the process of developing *theories* (consisting of concepts and their relationships) which are entirely *grounded* in the underlying data. I discuss the GTM in Section 3.3.

**Expert Behaviors** Zarb et al. (2012, 2013) analyzed 31 videos of a single pair of professional software developers (which were available on the online video platform *Vimeo*), and additional 11 sessions from industrial contexts. They conceptualized the professional pairs' activities in order to eventually teach students how to pair-program like experienced developers. They identified three behavioral patterns: *Restarting*, which consists of deliberately unfocusing when stuck and then coming back to make a strategic decision; *Planning*, in which it is important to clarify each partner's suggestions and to look things up, if need be; and finally *Action*, i.e., turning a plan into concrete changes, during which the driver should voice her thoughts while the navigator should listen to the muttering of her partner.

Zarb et al. (2014) then conducted an experiment with students to test the usefulness of these patterns: Seven pairs were equipped with the above mentioned patterns as guidelines, six other pairs worked without further instructions. All pairs had to find and fix one logical defect in as many given programs as possible within 45 minutes. Statistically, there was no significant difference in the technical results, but a closing survey found higher “*ease of communication*” and “*perceived partner contribution*” for the pairs equipped with the information material. The researchers also asked the students whether they actually used the guidelines provided (they said ‘yes’ for *Restarting* and *Planning*, less so for *Action*), but the students' in-session behavior was not recorded, so it is not possible to discern the effects of *reading* about the guidelines from those of *acting* according to the guidelines.

**Types of Information Needs** Sillito et al. (2008) recorded a number of pair programming sessions (as already discussed on page 38): 12 pairs of students working on assigned issues in an open-source project for 45 minutes and one pair of professionals working on their everyday task for 30 minutes. The researchers analyzed the types of questions developers ask about the code base by employing pair programming as a vehicle to make the subjects' thinking aloud more natural. Sillito et al. (*ibid.*, Sec. 3) characterize their analysis method as “*a grounded theory approach*”, but do not offer more insights into what this actually meant in their case and neither does the first author's PhD thesis (Sillito, 2006, p. 4). I suspect their catalog of 44 generic question formats results from open coding only (which I explain in Section 3.3.3a). Nevertheless, it provides a partial answer regarding aspect *A2* (relevant knowledge types): When it comes to source code, pair programmers want to know a lot. To give two examples: There are no fewer than six question types pertaining to class hierarchies (types #6–#11) and another six concerning control flow (types #12, #13, #24, #29, #30, and #31).

Two things, however, should be noted: First, the student pairs (who worked in a system unknown to them) asked, on average, 20 questions about the code base per session (or once every two minutes), whereas the professional pair (who also had only six months of experience, but worked in *their* system) only asked a total of 5 questions about the code base, that is, one question every six minutes (Sillito et al., 2008, Tables 3 & 4). Although one single pair is not representative of industrial PP, it still appears that (a lack of) code knowledge was not a dominant issue in this situation. Second, although the researchers attempted to limit their analysis to questions concerning the code, the boundary gets blurry at times and touches matters of design and possibly requirements, as question types such as #38 *Where should this branch be inserted or how should this case be handled?* or #44 *Will this completely solve the problem or provide the enhancement?* show. This indicates that during pair programming, just as in software development in general, more types of knowledge than just code knowledge may be relevant. In Section 7.3.1, I discuss additional knowledge types which I identified to be relevant in industrial pair programming sessions.

**Explicit Knowledge Transfer** Jones & Fleming (2013) recorded pair programming sessions of seven pairs of students who worked 110 minutes on fixing a bug in the *jEdit* software. They too limited their use of Grounded Theory practices to open coding (*ibid.*, Sec. III.D). In their analysis of the types of transferred knowledge (*ibid.*, Sec. IV, addressing A2), the researchers identified and categorized 43 episodes of one pair member teaching her partner about something. As their examples make clear, they focused on *explicit* knowledge transfer in which both developers are involved (unlike Plonka et al., 2015, discussed on page 81), making this the first study which actually pinpoints the moments in PP sessions where the developers explained, understood, or learned something (A5). Across most sessions, both developers taught their partner *general development knowledge* (12× regarding development tools, 6× the programming language) and *project-specific knowledge* (16× how to reproduce the bug, 9× about the existing code structure). The researchers do not provide details on how they developed these four categories. From their educator’s perspective, they especially highlight the *tools* category as such knowledge is practically relevant but not generally taught in classes (Jones & Fleming, 2013, Sec. IV.C).

**Driver/Navigator and Process Disruptions** In their next analysis, Jones & Fleming (*ibid.*, Sec. V) revisited the *driver and navigator* roles as analyzed by Bryant et al. earlier (see my discussion on page 72), thus looking at aspects A4 and A5. Similar to Bryant et al. (2006) they found that developers contribute ideas to the process regardless of their role. Unlike Bryant et al. (2008), who found most utterances to be on a medium level of abstraction, Jones & Fleming mostly observed a pattern they call *backseat driving*: The navigator proposes specific actions rather than goals or strategies. In the majority of cases, the driver acted upon these proposals without any discussion, which Jones & Fleming interpret as an indication for how closely the partners worked together, allowing the navigator to make suggestions fitting right into the driver’s course of action. Extreme forms of this have been reported earlier both by practitioner Belshee (2005, Sec. 1.2) as “*pair flow*” and by researchers Chong & Hurlbutt (2007, Sec. 5.1.2) as a “*mode of [being] exceptionally in sync*”. This interpretation is supported by their analysis of *disruptions* in the pair process (Jones & Fleming, 2013, Sec. VI): Within 14 hours of pair programming, there was only *one* instance of a pair member signaling that his partner disrupted his train of thought: The partner pushed forward while he simply needed some time to think.

In their analysis, however, Jones & Fleming (*ibid.*, Fig. 2, Tab. IV) only categorized navigator ideas as either “*discussed*” or “*not discussed*”, but not whether and how the driver’s ideas were discussed. There is no example in their paper to illustrate how a discussion on the content level (i.e., beyond “*Say that again*”) looks like. An alternative interpretation of their observation would be that the respective driver simply did as told, without actively agreeing with her partner, which would not be an indication of close collaboration. In my own research, I analyze the pair programmers’ discourse in more detail, e.g., by distinguishing content-level discussion from mere clarifications (see Section 6.2).

**Other Pair Programming Roles** As Bryant et al. (2008) found no quantitative difference between the *driver’s* and the *navigator’s* abstraction level of speech, Salinger, myself, and Prechelt (2013, prior to my PhD research) started a qualitative analysis of industrial PP sessions from Plonka’s data set (discussed on page 73; see also Section 4.3). We developed a role meta-model in which a *role* consists of multiple *facets* each of which corresponds to observable *actions*. In this terminology, *having the keyboard* would be a facet that belongs to the postulated *driver* role. However, we found other facets to be more relevant for understanding how a pair process unfolds. We identified a number of facets and grouped them to three relevant roles:

- The **Watchman** *recognizes hazards* (detecting and mentioning issues) and *sets priorities* (insisting that something is to be done now).
- The **Task Expert** *passes on task knowledge explicitly* (to her partner) and *turns task knowledge into proposals*.
- The **Spokesperson** *opens a dialog* (about some concern beyond the pair), *carries the dialog forward* and *rounds off the dialog* (by not accepting an end without resolution).

It should be noted that we analyzed only one pair of programmers. Here, one member assumed the *task expert* role and the other was both *watchman* and *spokesperson*. This initial role catalog was not developed further, but the knowledge advantage of a *task expert* would later resurface in my PhD research as a common pair constellation (the **Primary Gap**, see Chapter 11).

**Disengagement** Plonka et al. (2012a) analyzed episodes of *disengagement* in industrial PP sessions where one pair member was temporarily less involved in the process, i.e., where the pair did *not* work as closely together as was reported by others (Belshee, 2005; Chong & Hurlbutt, 2007; Jones & Fleming, 2013). This is the same data set they analyzed for other publications (discussed on pages 73 and 81). Plonka et al. (2012a, Sec. V) used five indicators for different levels of engagement from literature (e.g., mirroring the partner indicates basic engagement while modifying the partner’s contribution indicates high engagement) to identify episodes which *lacked* these indicators. The authors do not put a label on their method, but analysis steps (investigation of the circumstances of such episodes, systematic comparison with similar contexts without disengagement) resemble the GTM’s axial coding and theoretical sampling (see Section 3.3).

Plonka et al. (*ibid.*, Sec. VI) identified five circumstances leading to such disengagement: (1) external *interruptions* which are dealt with by one pair member who then may have trouble getting back into the process; (2) a *division of work according to expertise* where the less knowledgeable developer zones out because the ‘expert’ took over; (3) *simple tasks* of which the complexity is so low that the non-typing developer has nothing to look out for; (4) *social pressure* which makes a less knowledgeable developer take herself back as to not slow down the ‘expert’; and (5) *time pressure* due to which some developers may decide not to spend much time on explanations but to focus on the technical task instead. While the authors discuss concrete *disengagement episodes* excerpted from the recorded sessions for cases (1) and (2), and describe how the ‘novice’ in case (4) avoided driving and stopped asking questions at some point during the session, the empirical basis for the other two cases appears to be developer interviews only: The practitioners stated that they *could* have split up the simple task (case 3), and *would* have explained more if it were not for the time pressure (case 5). The researchers do not discuss how the pair programming process in these cases actually unfolded, e.g., whether one pair member was actually useless during supposedly ‘simple tasks’ or whether the ‘expert’ actually explained too little under time pressure for the ‘novice’ to learn something.

**Process Patterns in Distributed Pair Programming** Schenk (2018)<sup>30</sup> analyzed video recordings of a three-day period of one pair of professional software developers working together in a distributed setting totaling 16 hours of material. The pair used the IDE-plugin Saros which allows both developers to navigate and edit freely on a shared code base in real-time. Her analysis relied mostly on open coding, little axial, and no selective coding (*ibid.*, p. 143), and focused on two topics addressing what is *possible* for a good distributed pair (as opposed to what the *average* pair can expect):

---

<sup>30</sup>Earlier results were published as a conference article (Schenk et al., 2014). Since the concepts reported there were reworked significantly, I summarize the more recent state of research from Schenk’s PhD thesis here.

First, Schenk explains how the pair deals with its lack of awareness, which is more or less granted in a co-located case. Contrary to the expectation of seeing many related problems, Schenk (*ibid.*, Sec. 4.7.6) reports a close communication with little verbal synchronization effort and concludes that the shared source code provides the pair with enough concrete identifiers and line numbers which they can seamlessly embed in their dialog to make up for any lack of awareness. She calls this “*the magic of source code*” (*ibid.*, Sec. 5.3).

Second, Schenk discusses how the pair makes use of its freedom of independent navigation and editing, and how they organize their activities beyond the postulated roles of *driver and navigator*. Schenk (*ibid.*, Sec. 4.5) reports five patterns of how the ‘navigator’ becomes active as well: (1) *Direct Fix* is fixing a small issue without disrupting the partner’s train of thought; (2) *Jump In* is putting an idea of the driver into action; (3) *Check* is reassuring oneself by looking something up relating e.g. to the driver’s action; (4) *Contribution* is performing a code change that relates to the driver’s action (such as adding a comment line to the method currently under work); and (5) *Local Solution* is completing a sub-task of the driver’s work (such as implementing a comparator for a sorting logic). All of these patterns could be more or less feasible for co-located pairs as well, but would require more explicit handover of cursor control. They can therefore be understood as *advantages* of distributed pair programming—if the appropriate tools are employed and the developers are skilled enough.

### **Results of Qualitative Analyses of Pair Programming**

Most qualitative studies are rich in detail in that they include many different aspects which cannot be neatly compressed to a scalar such as an *effect size* in a controlled experiment. Nevertheless, there are a number of themes which resurface in multiple reports. Combining the studies discussed above, the scientific understanding from qualitative studies regarding the five aspects of knowledge transfer pair programming can be summarized as follows:

- **Process (A5):** Communication is key if two software developers are to actually work together on a problem. In good pairs, there is a constant stream of conversation, and their collaboration occasionally gets very close (Chong & Hurlbutt, 2007; Jones & Fleming, 2013; Schenk, 2018). In order to stay closely together with their partner, pair programmers do not stop communicating, not even when writing code or performing other actions (Zarb et al., 2013; Plonka et al., 2015; Schenk, 2018). In particular, this means that they establish and maintain their *common ground*, which includes both a mental model of the system and a shared goal and plan (Flor & Hutchins, 1991; Flor, 1998; Zarb et al., 2013; Schenk, 2018). This closeness may temporarily loosen under certain conditions when pair members *disengage* (Plonka et al., 2012a).

The often mentioned roles of *driver and navigator* are only meaningful terms when someone is actually typing, which is not the case throughout the complete sessions (Chong & Hurlbutt, 2007; Salinger et al., 2013). In distributed PP, the line between the roles is blurred even during coding with the many ways how the navigator can become active in a non-obtrusive way (Schenk, 2018). Overall, both developers contribute to the programming process in terms of ideas and discussions (Jones & Fleming, 2013).

- **Pair Constellations (A4):** While *driver and navigator* roles do not appear to bear much weight, in many sessions, there still appears to be one dominating partner, e.g., the more experienced pair member (Cao & Xu, 2005; Chong & Hurlbutt, 2007). The exchange and consolidation of ideas appears to work best for high-competent pairs and not for low-competent pairs (Cao & Xu, 2005).
- **Types of Knowledge (A2):** First of all, there is tacit *how-to* knowledge that relates to programming in general (such as how to read source code) which becomes visible in pair programming (Rooksby et al., 2006). Regarding explicit knowledge, which can

be verbalized, there are different types of knowledge that are actually transferred in pair programming sessions, including general programming knowledge and system-specific/task-relevant knowledge (Sillito et al., 2008; Jones & Fleming, 2013; Salinger et al., 2013).

- **Effectiveness (A<sub>1</sub>):** Knowledge transfer does happen in pair programming, e.g., in episodes of knowledgeable developers explicitly (Jones & Fleming, 2013) or implicitly teaching each other, for which there are a number of different strategies (Plonka et al., 2015), or through fresh team members (or otherwise inexperienced developers) who ask their partner many questions—at least unless the pair is under time pressure (Chong & Hurlbutt, 2007; Plonka et al., 2012a, 2015).

Although the above list may appear like a cohesive body of knowledge on how pair programming works, it should be noted that I was only able to write the above summary *after* I conducted my own pair programming research for years and filter and consolidate pieces from the, in fact, quite isolated studies.

### *Critique of Qualitative Analyses of Pair Programming*

Among the qualitative studies discussed above there are a number of recurring problems pertaining to their concrete research method, the nature of the results, and the conclusions for further research and possibly practitioners. See Table 2.11 for an overview.

Qualitative PP Studies	Type	Possibly Problematic Areas			
		1: Research Method	2: Theory Building	3: Allow Building On	4: Practical Insight
Flor & Hutchins (1991), Flor (1998)	S U	✗	(✓)	✗	✗
Xu et al. (2005)	M N	✗	✗	✗	✗
Cao & Xu (2005)	M K	✗	✗	✗	(✓)
Rooksby et al. (2006)	M K	✗	✗	✗	✗
Chong & Hurlbutt (2007)	M K	✗ <sup>a</sup>	✗	✗	(✓)
Sillito et al. (2008)	M U	✓	✗ <sup>b</sup>	✗ <sup>b</sup>	✓
Zarb et al. (2012, 2013, 2014)	M K	✓	(✓)	✗	✓
Jones & Fleming (2013)	M U	✓	✗	✗	✗
Plonka et al. (2011, 2012a, 2015)	M K	✓	✗	(✓)	(✓)
Salinger et al. (2013)	S K	✓	(✓)	✓	(✓)
Schenk (2018)	S K	✓	(✓)	(✓)	(✓)

<sup>a</sup>An earlier paper (Chong et al., 2005, Sec. 4) might give a clue: In their pre-study, they supposedly performed open coding on their transcripts (from the Grounded Theory Methodology, see Section 3.3).

<sup>b</sup>Problem areas 2 and 3 pertain to PP, which was not a particular interest of Sillito et al. (2008).

**Table 2.11:** Overview of recurring problems in qualitative PP research. Studies involved either a single (S) or multiple pairs (M) who worked in an existing project/context known (K) or unknown to them (U), or started a new one (N). In the respective areas (see main text), the studies have ✓ – no problems, (✓) – some problems, or ✗ – considerable problems.

The four possibly problematic areas are:

1. **Research Method:** Generally little information is provided about the actual research process. Most qualitative studies use a *coding scheme* of some sort, but these are not specific for pair programming or their generation is not described.

2. **Theory Building:** Studies may be detailed in their examples (e.g., by providing verbatim excerpts), but do not present rich concepts that have internal structure and variability, and are connected to others. A coding scheme—after all, a collection of labels—is not a theory.  
For example, the novice-guiding “*strategies*” are all isolated and listed one after the other, usually illustrated with one example. The only variation is in *giving clear instructions*, which may come with and without explanations (Plonka et al., 2015, see also discussion on page 81). A richer theory would address questions like *Do experts try more than one strategy?* or *Are there any constraints, or are the strategies up to personal taste?*<sup>31</sup>
3. **Allow Building On:** Studies are concerned with narrow and isolated aspects and not with understanding PP as a whole. Consequentially, these studies do not provide obvious starting points for further research.
4. **Practical Insight:** Studies do not provide leverage points for formulating *practical advice* for professional developers on how to improve their pair programming, i.e., being able to achieve the expected benefits of this work mode discussed in Section 2.3.1c.

### ***Starting Pair Programming Research from Scratch***

Salinger (2013, p. 42) made similar observations in his discussion of the state of qualitative PP research and set out to address these problems in his own research. Based on the Grounded Theory Methodology, he performed a (very) detailed analysis of pair programming sessions from both industrial contexts and from an experiment with students. He made three key observations each of which led to a strategic decision concerning his research result:

- **Observation 1: Pair Programming is Complex → Layered Research Strategy**

Pair programming features several different phenomena which could be studied, too many for any single study or even researcher, such that a naive approach without a particular research focus led to outright “*drowning*” in concepts (Salinger et al., 2008).

Salinger’s conclusion was to not aim for a complete analysis of pair programming, but to do the groundwork to structure an overall research process that is to advance in stages. Starting from an utterance level analysis covering all that pair programmers do on its ‘atomic’ level, later studies could focus on only a certain type of phenomenon, certain aspects, and/or coarser granularity. These ‘atoms’ of a pair programming process are the *base activities* which are characterized by one or more *base concepts*. The set of all *base concepts* and the rules when to apply them constitute the *base layer*.

- **Observation 2: Flexibility in Verbal Expression → Reveal Intentions**

Verbal communication constitutes a large part of pair programming sessions while interaction with the computer (e.g., to look up things and to modify source code) is often only the culmination point of a prior discussion.

Base activities have meaning to the pair programmers, but the observable utterances can have many shapes. Instead of addressing the dialog’s surface structure (as Bryant, 2004 did, e.g., by treating an utterance as a *question* regardless of whether the speaker already knows the answer, see page 74), Salinger’s base concepts attempt to capture the primary intention of the developers and thus *do* make a difference between, say, proposing to the partner to click somewhere and actually explaining the rationale behind it (unlike Plonka et al., 2015, discussed on page 81).

- **Observation 3: Progression of Ideas → Analyze Discourse**

---

<sup>31</sup>It is important to note that I do not intend to diminish their efforts. Unlike many publications, their work simply contains enough details to provide a concrete example.

Base activities have an *episode* character in that they pertain to some discourse object (such as a design proposal, or a proposal regarding the next tactical step) which the pair members collectively deal with, e.g., by bringing up such proposal, dis-/agreeing to it, amending it, challenging it, etc.

Salinger figured that this is an important property of a pair programming process, and thus structured the set of base concepts accordingly. Overall, he identified 14 classes of discourse objects and for each between one and seven *verbs* used by pair programmers to operate on them, leading to an overall of 59 verb-object combinations, such as *propose\_step* or *disagree\_hypothesis*.

It is important to note that the base layer is *not meant to be used as a coding scheme* (Salinger, 2013, p. 125; Salinger & Prechelt, 2013, p. 35). Apart from an existence proof of 67 different base activities (that is, 59 human-human interaction activities mentioned above, plus 8 human-computer interaction activities) and their internal taxonomy (e.g., *propose\_design* has three subtypes; *explain\_finding* has seven), the base layer is first and foremost a methodological contribution. The base layer is not meant to be ‘applied’ for a complete coding of PP sessions—as would be necessary for a qualitative-quantitative study—but is a *framework*, a starting point for qualitative pair programming research that avoids problem areas (1) to (3). I discuss the base layer as such in Section 3.4 and my application of it in Section 4.5.2.

### 2.3.4 e) Summary of Industrial PP Research

Pair programming in industrial settings has been studied from a quantitative perspective (Section 2.3.4a), where it is often compared with solo programming in time- and/or quality-based contests (the latter is then reduced to a single scalar or even just a binary measure), which show rough tendencies in favor of quality and against time but also indicate that moderator variables are at play. Developer expertise and task complexity have been explicitly tested as moderators but without yielding clear results. Overall, controlled experiments on pair programming can only cover a small fraction of the software development world as they rely on small programs and isolated tasks (with 300 lines of Java code already being ‘complex’), let developers work in pairs who have no experience in this regard and who could not decide for themselves whether they deem the pair work mode helpful for their particular task at hand.

Project-level field studies in industrial settings (Section 2.3.4b) could get around these limitations, but tend to consider pair programming as a practice that is part of a software development process and not record what actually happens during PP sessions.

Quantitative-qualitative analyses (Section 2.3.4c) condense a whole PP session into a single data point, thus disregarding the actual process of pair programming and any differences therein. Pretty much the only constructive advice for practitioners is that the metaphor of *driver and navigator* is questionable because it is (a) not applicable for all activities that happen during a session and (b) there is no significant difference in the levels of abstraction pair programmers speak at while assuming these roles.

Qualitative analyses (Section 2.3.4d) produced a number of insights: There are recurring activity patterns including explanatory or teaching episodes; there are activity bursts, where the two pair members are exceptionally in sync; pair programmers may develop a shared plan which incorporates ideas from both; communication is important to maintain common ground throughout the session; there are differences in behavior depending on the developers’ competence levels and context (e.g., new to the team, time pressure, or task difficulty). However, the theoretical concepts are rather coarse, come from isolated studies, which, at least in part, are not as transparent with regard to their research process as to inspire much trust, and

mostly lack obvious starting points for further research and ideas for how to eventually come to results which are relevant for practitioners.

### 2.3.5 Summary of Pair Programming Research

I started my discussion of PP literature with a practitioner perspective: What benefits do software developers ascribe to PP? A consolidation of practical books and surveys led to a distinction between (potentially) observable *effects* and (sometimes tacitly) assumed *mechanisms* that bring them forth: Pair programmers COMBINE their existing individual knowledge to work on more difficult tasks while producing better designs, UNDERSTAND a given situation better together which catches defects in the making thus lowering defect counts, and LEARN together and from another for future tasks thereby spreading knowledge in the team.

In addition to the overall *effectiveness* of pair programming with regards to the above mentioned mechanisms and effects (A<sub>1</sub>), practitioners are also interested in more nuanced aspects: Are there differences in the types of knowledge, task types, or pair constellations that affect the mechanisms (A<sub>2</sub> to A<sub>4</sub>) and how do these mechanisms actually work (A<sub>5</sub>)?

Researchers in industry and education ask similar questions, but with different emphases (Sections 2.3.3 and 2.3.4): Although it is plausible for all three mechanisms to have relevant effects in both domains, educational research is more interested in the effectiveness of the LEARN mechanism, while the industry's prime concern is economic effectiveness coming from COMBINE and UNDERSTAND. A high-level overview can be found in Table 2.12.

Below, I summarize what is already understood about knowledge transfer in pair programming by combining both domains, what the open questions are, and summarize which of them I contribute to in this thesis.

#### 2.3.5 a) Effectiveness (A<sub>1</sub>)

**Question** *Is pair programming effective in the sense that it produces the expected effects?*

**Results** Overall, students and practitioners who pair program *report* positive effects. Meta-analyses of actual data show positive trends for economic aspects such as quality and duration, and students' short-term learning. The results per expected mechanism are as follows:

- **The COMBINE Mechanism:** Case studies indicate that developers might choose their programming partners based on their respective individual knowledge levels (see Section *Observational Field Studies on the Project Level* on page 69). Whether or not pairs then actually combine their individual knowledge in educational or industrial practice is not clear. I am not aware of any study that investigates effects of developers pre-existing knowledge levels in pair work, e.g. on design quality.

In general, in many experimental tasks, developers either start from scratch or in a system unknown to them, so there is little relevant pre-existing knowledge that could be COMBINED anyway (see Section *Unrealistic and Unfair Comparisons* on page 67).

- **The UNDERSTAND Mechanism:** For investigating the UNDERSTAND mechanism, working in an unknown system or starting from scratch is less of an issue. Both educational and industrial meta-analyses show positive effects of pair work: Students get better assignment scores (see page 55), professional developers are faster (in terms of wall-clock time, not person hours) and produce higher quality code (see page 64). However, both industrial experiments and more realistic project-level field studies show mere trends and struggle with unexplained variation (see pages 64, 69, and 71).

While the effectiveness of PP for code understanding is not clearly demonstrated in experiments and project-level field studies, qualitative studies at least show that the need

Aspect	PP in Education	PP with an Industry Focus
Effectiveness (A <sub>1</sub> )	<p>COMBINE &amp; UNDERSTAND: Mixed neutral to positive effect on code quality, and mostly positive effect on time spent. But: Little relevance in educational settings.</p> <p>LEARN: Neutral to medium positive effect on assignment scores and less on exam scores. At least half of students report more learning due to pairing.</p>	<p>COMBINE &amp; UNDERSTAND: Experiments show positive effect on quality and duration, but negative on effort. But: A lot of unexplained variance in experiments and project-level field studies. Either insignificant or huge quality effects in industry.</p> <p>LEARN: Over time, developers' perceived knowledge levels increase. Knowledge transfer happens through teaching (explicitly and implicitly) and asking (if time pressure allows).</p>
Knowledge Types (A <sub>2</sub> )	–	Code knowledge is necessary for doing PP and is acquired during PP; general programming knowledge also gets transferred.
Task Suitability (A <sub>3</sub> )	–	In experiments, PP appears to dampen effects of task complexity on effort and quality, making it less beneficial for simple tasks. In practice, PP is less used for easy tasks and more for complex tasks, those involving system understanding and many dependencies, in particular early in projects.
Pair Constellations (A <sub>4</sub> )	In general, incompatible pairs are rare. Regarding personality, neither homogeneous nor heterogeneous are consistently better.	Personality has <i>some</i> effects, but less than differences in task complexity and expertise have. In experience-wise homogeneous pairs, junior and intermediate developers appear to benefit more from PP than seniors; large differences in experience make PP difficult. New team members first pair with experienced colleagues.
Pair Process (A <sub>5</sub> )	More communication, feedback, and meta-communication correlate with learning achievements. But: Only few studies.	There is constant communication in PP sessions. Driver and navigator do not talk (and likely: think) at different levels of abstraction, and both contribute to almost all topics. Nevertheless, one developer appears to dominate. Cognitive abilities alone do not make PP work if the pair does not handle conflicts.

**Table 2.12:** High-level summary of research results on pair programming (as a work mode) with regard to the five practically relevant factors. See Sections 2.3.3 and 2.3.4 for details.

to understand the code base is indeed addressed in pair programming sessions: All pairs appear to try to UNDERSTAND, even though not all succeed (page 78); some student pairs asked questions about the code base about once every two minutes (see page 82); overall, *code comprehension* is the most prevalent topic in industrial PP (page 72).

- **The LEARN Mechanism:** Positive learning effects of pair programming can be measured in students' assignment, quiz, and exam scores (see page 55); learning achievements attributed to PP are self-reported in educational surveys (see Section *Self-reported achievements* on page 56), as well as industrial experience reports, surveys, and project-level field

studies (see Sections 2.3.1c and 2.3.1d and page 71). I am not aware of a truly industrial experiment showing an effect of pair programmers' increased knowledge levels.

Yet again, qualitative studies do not directly address the effectiveness, but do show that pairs use learning and teaching opportunities: The main purpose of some industrial pair sessions is *knowledge transfer* (see page 81). In pairs with knowledge gaps, the more knowledgeable partner provides explanations (see pages 77 and 83) as well as more subtle learning opportunities (page 81), while the less knowledgeable partner such as a new team member asks questions (page 80).

**Open** The economic effects have much unexplained variation. Additionally, measurements in realistic settings are rare, fair comparisons of lab settings and reality are difficult.

**Contribution** I do not address the question of overall pair programming effectiveness. Instead, I characterize the mechanisms underlying knowledge transfer in pair programming based on in-depth analyses of industrial PP sessions (see Chapters 8 to 11).

### 2.3.5 b) Types of Knowledge (A<sub>2</sub>)

**Question** *What do the developers COMBINE, UNDERSTAND, and LEARN while pair programming?*

**Results** Table 2.13 lists five knowledge types that have been reported by various publications. By far the most often mentioned type is knowledge about existing code and the software system itself. Developers do indeed feel that this type of knowledge is the most relevant for pair programming sessions (see page 71). Studies that looked into actual PP sessions found that having code or system knowledge appears relevant for being a valuable pair member: If only one pair member has such an advantage, she appears to take the lead in the session (see pages 71, 78, 80, and 84).

**Open** Are there additional types of knowledge? Are there actual (as opposed to perceived) differences in the relevance of the knowledge types in the context of a PP session?

**Contribution** I distinguish two main types of knowledge that are relevant in pair programming sessions, system-specific *S knowledge* and more general *G knowledge* (see Section 7.3.1), with gaps in *S knowledge* playing the bigger role in most analyzed PP sessions (see Chapter 11).

### 2.3.5 c) Task Suitability (A<sub>3</sub>)

**Question** *Are there some types of tasks for which pair programming is more or less suited?*

**Results** Teams do not use PP for all tasks (see page 70), which may indicate some, possibly tacit, selection criterion. Although most studies are not explicit as to what the contents of the pair sessions actually were, the reported task properties that motivate developers to work in pairs all emphasize the importance of knowledge about the existing system, e.g., ramping up new team members, being early in a project, or working with complex tasks involving many code dependencies (pages 70, 71, and 80).

Actual task suitability, however, may be broader than is expected by practitioners. In an experimental comparison of two architectural styles, pairs performed similar in both conditions in terms of time to completion and average correctness of solutions, whereas the solos were slower in the simple system and produced less correct solutions in the complicated system (see page 65). I am not aware of a further distinction or characterization of tasks that has an empirical basis and is relevant for pair programming.

Knowledge Type	Good Evidence	Vague Evidence
Programming language	Jones & Fleming (2013) . . . . . 83	Xu & Rajlich (2005) . . . . . 61 Chong & Hurlbutt (2007) . . . . . 80
Tool usage	Plonka (2012) . . . . . 81 Jones & Fleming (2013) . . . . . 83 Vanhanen & Korpi (2007) . . . . . 71	Xu & Rajlich (2005) . . . . . 61 Chong & Hurlbutt (2007) . . . . . 80
Task decomposition & bug reproduction	Flor (1998) . . . . . 80 Jones & Fleming (2013) . . . . . 83	
Existing code and system	Flor (1998) . . . . . 80 Jones & Fleming (2013) . . . . . 83 Sillito et al. (2008) . . . . . 82 Plonka (2012) . . . . . 81 Vanhanen & Korpi (2007) . . . . . 71 Canfora et al. (2005) . . . . . 55 Chong & Hurlbutt (2007) . . . . . 80	
Design patterns & programming techniques	Plonka (2012) . . . . . 81	Xu & Rajlich (2005) . . . . . 61 Chong & Hurlbutt (2007) . . . . . 80

**Table 2.13:** Knowledge types mentioned in PP studies with either good or vague evidence. Page numbers refer to my discussion of these studies.

**Open** Are there actual (as opposed to perceived) differences between different types of tasks? What is the relation of task type and relevant knowledge?

**Contribution** I do not categorize software development tasks as such, but pair’s **Target Constellation** in terms of **Knowledge Needs** that the pair wants to meet by the end of the session and the session dynamics that develops from this (see Chapter 11).

### 2.3.5 d) Pair Constellations (A4)

**Question** *Are there some types of pairs for whom pair programming is more or less suited?*

**Results** Neither individual developer experience, skill level, cognitive abilities, nor personality traits explain a pair’s effectiveness well. In educational settings, there is only little evidence for pair programming effectiveness being significantly affected by the pair members’ respective skill levels (see Section **Skill Level** on pages 58 to 60) while studies involving cognitive abilities (see page 72) and studies on personality types are inconclusive (see Section **Personality Types** on pages 57 to 58). The results of an industrial experiment—which only had homogeneous pairs, i.e., both junior, intermediate, or senior—might indicate that working in pairs leads to more consistency in produced quality regardless of developer expertise (see page 65). Again, personality traits do not explain the pairs’ performances (see page 66).

Studies that look into the pair process also consider an *expert/novice* distinction (sometimes with an *intermediate* level in between). Here, the constellation of two experts appears to be beneficial, as it is the only one with longer discussions on how to proceed, proactive asking for opinions and critiquing, consolidation of knowledge and filling the gaps, and summarizing the status mid-session and readjusting the goal if need be (see Section **Activity Patterns** on page 77). This observation begs two questions: First, there is the pragmatic aspect of whether there is

nothing non-expert developers can do about it? Second, what is an ‘expert’ anyway? Is it the academic performance and “*interaction with instructor*” as in the study that gave rise to these characterizations? Or is it the pay grade as in the controlled experiments (see page 65)? The literature on expertise in software development paints a multifaceted picture (Section 2.2.3a) which is not reflected in the one-dimensional conceptions used in PP research.

**Open** What are relevant dimensions to characterize pair constellations, if developer experience, personality traits, and others are not helpful?

**Contribution** I do not characterize individual developers or pairs of developers, but a pair’s **Initial Constellation** in terms of its members’ individual **Knowledge Needs** in the context of the task at hand. Depending on the **Target Constellation** the pair wants to reach, their respective knowledge gaps can pose challenges or opportunities (see Chapter 11).

### 2.3.5 e) Pair Process (A5)

**Question** *How do pairs actually COMBINE, UNDERSTAND, and LEARN?*

**Results** For PP to work and have positive effects, the pair members need to communicate. Although this is merely postulated by most educators, the few who looked into it indeed found a positive correlation of communication amount and learning effects (see Section 2.3.3c). In industrial PP sessions, there is constant communication (see pages 71, 80, and 83) during which the pair members maintain their *common ground* (page 80). Not resolving conflicts can lead to one pair member withdrawing altogether (see page 72).

The only wide-spread model used to “explain” the pair programming process are the roles of *driver* and *navigator* (see Section 2.3.1a). However, these role labels are only meaningful for two thirds of the time when one developer actually touches mouse or keyboard (see page 73). Even when one developer controls the keyboard, the other may “drive” the session forward (see page 83). In practice, both pair members contribute to almost all of the topics dealt within a PP session regardless of their “role” and there is no significant difference in the levels of abstraction the two talk and, presumably, think on (see Section *Driver and Navigator Roles* on page 72). Further analyses of what the communication of a pair looks like are scarce, and only coarse patterns are described (see Section *Expert Behaviors* on page 82).

More detailed studies on knowledge transfer in pair programming are incomplete in that they are limited to ‘implicit teaching’ but do not cover explicit knowledge transfer (see page 81), focus on information needs and questions but do not address the answers (page 82), or focus only on explanations but do not address what the needs or questions were (page 83).

**Open** What does the pair’s communication look like on a process level? How does what the pair does and does not know affect them in their session, and how do they deal with it?

**Contribution** I distinguish different levels of pair programming process **Fluency** and analyze five aspects of the partners’ **Togetherness** that enable a fluent process and productive collaboration: A shared understanding of the software system and of software development in general, one shared plan, good workspace awareness, and no language barrier (see Chapter 6). The main chapters of this thesis pertain to two knowledge-related aspects: How do pair members deal with what they (do not) know about the software system and about software development in general, which I call **S knowledge** and **G knowledge**, respectively (Chapters 7 to 11).

## 2.4 Pair Work and Small Groups

Software engineering researchers are not the only ones interested in how people work together. Psychologists and sociologists both contribute to the field of “small group research”. Although there is some debate on whether a pair of two people—or *dyad*, as it is often called in this discipline—should already be considered a *group*, I follow the argument of Kipling D. Williams (2010): “*Dyads Can Be Groups (and Often Are)*”. While other researchers often define a *group* through attributes such as shared past and goal, cohesion and interaction, he sees these aspects not as *defining* properties but as interesting factors that may *affect* a group, which to him is just “*two or more people*” (*ibid.*, p. 269). He further argues that dyads can be used to study a wide range of group phenomena (*ibid.*, pp. 270–271):

- **Social facilitation:** The mere presence of just one additional person increases the likelihood of the *dominant response*, which may result in a better performance or in making more errors.
- **Social loafing:** Individuals may reduce their efforts whenever they believe their work is combined with that of others.

Williams’ list of group phenomena that are applicable to pair situations goes on: “*conformity, imitation, contagion, deindividuation, social comparison, compliance, obedience, bystander intervention and related prosocial behaviors, social inhibition, stage fright, and even crowding*” (*ibid.*, p. 271).

I do not cover all possible group/pair phenomena here since I am only interested in the aspect of knowledge transfer. The particular phenomenon of *pair programming*, however, does not appear to be a common subject for psychologists. I already discussed all examples I know of (e.g., Hannay et al., 2010, who looked for effects of individual personality traits on pair performance, see page 66). Here, I therefore look at psychological studies that are close to my area of interest, either because they consider the pair as an easier-to-study extension of the individual with otherwise internal processes becoming observable (see Section 2.4.1) or because they study knowledge transfer in groups of which the *dyad* is the smallest form (see Section 2.4.2).

### 2.4.1 Pair Work on Distinct Tasks

There are a number of studies for which pairs worked on well-defined tasks with limited freedom of action, at least in comparison to the breadth of software development activities. The empirical studies I discuss in this section all involve pairs who work on some task. Although they are all meticulous, none of them is particularly helpful for understanding how pair programming works or how to study pair programming.

#### 2.4.1 a) Joint Decision for Visual Perception Task

Bahrami et al. (2010) showed that interpersonal communication is rich enough to transfer the subtle degrees in one’s confidence which are necessary to make an optimal decision together.

##### *Setup*

Two brief visual stimuli are presented to two subjects individually. The pair needs to reach a joint decision on whether the first or the second image had a slightly higher contrast in one of six areas. The subjects can communicate freely and for as long as they like. After they announce their decision, the setup provides them with feedback and commences with the next set of stimuli with a new random contrast difference until the subjects have worked through

256 decisions (*ibid.*, p. 2 of supplementary material). Overall, 51 (all male) dyads were subjected to four different conditions: Experiment 1 as described above, experiment 2 with extra noise on some stimuli (and 768 overall decisions to be made), experiment 3 without communication, and experiment 4 without feedback (*ibid.*, pp. 1–3 of supplementary material).

### **Results and Discussion**

Their first observation is that the pairs perform better than either subject would have alone. To explain this effect, the researchers did not analyze the *actual* communication but instead used the individual responses to reconstruct each subject's sensitivity and then fed this data into different models to determine which of these predicts the actual joint decisions best. They found that the actual pairs were better than simply flipping a coin on disagreement or learning which pair member is better over time. Rather, they appear to communicate their *confidence* and thus make better joint decisions. Through experiments 3 and 4, the researchers also found that while allowing communication is necessary to achieve the pair effect, providing feedback *is not*. However, since the *actual* interaction was not analyzed, it is still unclear *how* the communication of each other's confidence works.

### **2.4.1 b) Understanding a Complex System**

Miyake (1986) studied the process of understanding a mechanical sewing machine.

#### **Background and Proposed Model**

The researcher had an existing understanding of the machine, and formalized it in six levels of understanding: (Black box) *mechanisms* are realized by (white box) *functions*, which in turn are realized by (black box) *mechanisms* one level below.

For the process of understanding, Miyake (*ibid.*, pp. 156–157) postulates a sequence of six standard moves that repeats for each level of understanding: (1) identify function  $F$  on level  $n$ , (2) question function, (3) search for mechanism  $M$  on level  $n + 1$  below, (4) propose mechanism, (5) criticize mechanism, and (6) confirm mechanism. She predicts that moves in that order should happen often (e.g.,  $F(n) \rightarrow M(n + 1)$ ), that the reverse order should happen less often (e.g.,  $M(n + 1) \rightarrow F(n)$ ), and that other moves should not happen at all (e.g.,  $F(n) \rightarrow F(n + 1)$  which would skip the mechanism). Overall, she characterizes the understanding process as a back and forth between a stable state of *understanding* (steps 1, 4, and 6) and an unstable state of *non-understanding* (steps 2, 3, and 5).

#### **Setup and Results**

Just like a number of studies on programming used pairs to make their think-aloud process observable, Miyake (*ibid.*, pp. 158–159) also relied on pairs to study the process of understanding. She video-taped three pairs who approached the understanding task in three sessions without a set time limit, with an overall duration of 60–65 minutes per pair (*ibid.*, pp. 159–160).

Based on coded transcripts, Miyake (*ibid.*, p. 166) concludes that understanding indeed proceeds as predicted by her model. She explains the 17% non-predicted moves with limitations of her data collection method: Moves back to already understood levels are due to communication issues (e.g., making sure the partner understands the point) and the skipping of levels represents non-verbalized thinking (*ibid.*, pp. 166–167).

#### **Discussion**

Miyake appears to see the interaction of the pair as representative for the thought process of an individual: She chose pairs because they make “*a usually invisible process visible*” (*ibid.*, p. 159). Her model of how understanding progresses only speaks of the “*subjects' state of*

*mind*” in the singular form as if the *pair as a whole* goes through the steps of identifying, questioning, etc. (Miyake, 1986, pp. 156–157). However, starting in the *Results* section, she speaks of “*visualiz[ing] the subjects’ moves*” (seeing them as two individuals) and “*the subjects’ thinking processes*” (plural), i.e., she postulates mental states for each pair member *individually* (*ibid.*, pp. 162–167). Put differently, Miyake does not characterize the type of situation her setup of *pair understands sewing machine* should represent: A situation of an individual trying to understand something, or a pair situation? In fact, the article does not feature a *Conclusion* section, but only a *Discussion* of miscellaneous other observations. I have three more concerns:

- For the analysis, each pair’s three sessions were considered as one continuous process, but they do not reflect how understanding would naturally evolve: The pairs started with pen and paper only, later were provided with a physical sewing machine, but did not have a thread until the third session (*ibid.*, p. 160). The researcher was also present during all sessions to ask questions to clarify what was happening and to intervene and suggest new ways to proceed if a pair was not making progress (*ibid.*, p. 160).
- The researcher was able to come up with the particular levels because she knew the subject well (which is not the case for understanding industrial pair programming). She admits that her six levels are not canonical and adds “*I use six levels simply because this is all that is required by the data from my studies*” (*ibid.*, pp. 154–155). It is this sentence that makes me wonder what came first: The model or the data?
- To me, it appears that the conceptualization that is most transferable from this study to other contexts is the notion of the pair repeatedly switching back and forth between understanding a bit, becoming aware of some non-understood part, and then working towards a more stable understanding again. However, the proposed sequence of six steps is generic (since both directions are ‘allowed’) and the analysis is too coarse: The three consecutive steps of *proposing*, *criticizing*, and *confirming a mechanism* alternate between *stable* and *unstable*, but all operate on the same *level* and are thus aggregated to the same category of moves ( $M(n) \rightarrow M(n)$  in Miyake’s terms). It comes as no surprise then that this is the most populated category with 99 out of 287 total moves (*ibid.*, Table V), which effectively masks the supposedly crucial aspect of how a pair’s understanding progresses between stable and unstable states.

#### 2.4.1 c) Understanding a Simple System

Unlike Miyake (1986), Okada & Simon (1997) are explicit about the real-world situation their study is to mimic: Collaboration in the context of scientific discovery.

##### ***Background and Setup***

They compare the performance of undergraduate science students working alone or in pairs on an understanding task that sets off the subjects in the wrong direction and requires them to come up with an alternative hypothesis. The 27 male subjects<sup>32</sup> had to identify a regulatory mechanism of three ‘genes’ in a computer simulation. The sessions were audio- and video-taped and the subjects were asked to make their thoughts, hypotheses, and justification explicit to the camera as to ‘convince’ a viewer that they indeed solved the puzzle. The pairs had to reach a consensus.

The researchers took the time until completion and rated the subjects’ final explanations on a scale from 0 to 4, from *not discovered the relevant effect at all* to *correctly identified two genes as two types of inhibitors*. Counting all possible input parameters, the subjects could

---

<sup>32</sup>Not only was it difficult to “*get female subjects to participate*”, but pilot studies also suggested that females had “*different discussion styles*”, which the researchers left as further work (Okada & Simon, 1997, p. 115).

perform 120 different ‘experiments’, five of which were strictly necessary to see the outcomes that allow to formulate the correct solution and reach a perfect score of 4 points.

### Results

Pairs reached significantly better average scores (2.89 vs. 1.67, *ibid.*, Table 1). Differences in individual performance, spent time, number and quality of conducted experiments, and number of formulated (alternative) hypotheses *did not* explain the observed difference between solo and pair performance (*ibid.*, pp. 120–126). (Note that, similar to the qualitative-quantitative studies on pair programming, the researchers aggregate all process information of a solo or pair session into a single, high-dimensional data point, see page 75). The authors conjecture that the pairs’ interaction itself is important, i.e., that pairs may be more effective in using the available information from the experiments because they had someone to explain their hypotheses and justifications to, which the solos lacked (*ibid.*, pp. 127–129).

In a qualitative discussion, Okada & Simon (*ibid.*, pp. 130–133) look closer into what the relevant elements of these discussions might be. They identify five trigger conditions for explanation requests (such as *puzzling experimental outcome* or *disagreement with partner’s explanation*) and six activity patterns of reactions (such *generating an idea* or *reviewing the data*). Okada & Simon (*ibid.*, pp. 132–136) argue that such requests for explanations and the resulting reactions are how the pairs “*co-constructed new knowledge*”. They discuss one episode in detail, the pivotal points of which are one pair member first asking “*Have we answered the question?*” and then “*All we are doing is describing [...] what it will do every time [but] we don’t know why*” which led the partner to first note that they “*have two other cases we have to account for*” and then come to see already available information in new light: “*Oh, we have figured out that they are controlled chemically [...]*” (*ibid.*, p. 143). Summarizing and questioning the current state of understanding and then reaching a more stable state is similar to Miyake’s notions of *criticizing* and *confirming* (see page 95 above).

### Discussion

Yet again, both the setup and the researcher’s position are quite different from pair programming and studying it in industrial settings: The task was carefully designed by the researchers which allowed them to assess the quality of each individual step taken by the subjects; while the subjects themselves had only 120 different experiments to run and could see a complete record of all previous experiments and their outcome (*ibid.*, p. 118).

Nevertheless, their experiment still shows that the outcomes of pair work are not easily predicted by the individuals’ abilities and not even by the activities they perform on a technical level. Rather, a pair’s interaction and the mutual influence the individuals have on each other are what is relevant for understanding how the observed outcomes come about.

## 2.4.2 Small Groups and Knowledge Processing

As said above, I do not discuss the plethora of different group phenomena, but focus more on those that closely relate to the economic view of pair programming in which two software developers work on a technical task together expecting benefits for both their product and their own capabilities (see Section 2.3.1c on *Expected Effects and Mechanisms*). For simplicity, I use the terms “group” and “team” as synonyms in this section.

Traditionally, the effectiveness of group work was studied in relation to structural properties such as group size and composition, or nature of the task (Bossche et al., 2006, p. 492). Well-known examples of task classification systems were developed by Shaw (1981, pp. 363–365), who proposed six task dimensions such as *difficulty* or *solution multiplicity*, and Steiner (1972,

pp. 16–18), who introduced terms such as *disjunctive*, *conjunctive*, and *additive tasks*, where the group performance depends on best, worst, and average member’s performance, respectively. The problem with such task taxonomies in particular is that real-world tasks (unlike laboratory tasks) combine many different properties, while structural properties in general do not allow for specific predictions, e.g., about where difficulties of a particular group process will lie (Tschan, 2000, p. 143; Bossche et al., 2006, p. 492). As Bossche et al. (2006, p. 491) put it: “*fruitful collaboration is not merely a case of putting people with relevant knowledge together*”. (Such criticism of a ‘black-box’ perspective may ring familiar from my discussion of controlled experiments on pair programming, see page 67).

#### 2.4.2 a) Coordination and Shared Cognition

The consequence in group research was to consider the interaction processes themselves. The central notion of what allows group members to be productive together is that of **coordination** (Wittenbaum et al., 1998, p. 177). As Gabelica et al. (2016, p. 35) summarize, *coordination* has two different meanings for researchers: There is the *output approach* which considers the state of a group being coordinated and there is the *process approach* which focuses on the activities that lead to such a state. These two views are not completely disjoint but interact with each other: Group processes lead to some state which in turn shapes future processes, etc. Nevertheless, relevant concepts lean more towards one or the other perspective.

##### ***Output Approach: Coordination as a State***

Cannon-Bowers & Salas (2001) summarize many different related concepts from the context of team performance research under the term of **shared cognition**, which is, roughly speaking, the group members’ shared conception of the problem and how to approach it that allows them to work *together*. They introduce some terminology to capture the nuances. First, there is the question of *what* is shared (*ibid.*, pp. 196–198):

- **Task-specific knowledge**, e.g., a mental model of the task itself, allows coordinated action of team members without much discussion because all have compatible expectations. It can be generalized only to similar tasks.
- **Task-related knowledge** is more abstract and may be process-related, e.g., *how* to approach certain issue. It needs to be similar among the members for them to be effective.
- **Attitudes and beliefs** are more generic in nature. With compatible perceptions about the environment team members operate in, their cohesion and motivation may increase.
- **Team members’ knowledge of each other** including their knowledge levels regarding certain topics, preferences, strengths, weaknesses—also called a *team mental model*—may help with resource allocation and mutual compensation.

A related concept is that of *transactive memory systems* introduced by Wegner (1987, pp. 191–194): Individual group members learn about the area of expertise of others and then together form an information-processing system that has functions similar to a single memory system (see also Section 2.2.2). Even without explicitly assigned roles, groups can lessen the load of remembering certain things on any individual. Over time, group members learn what they can expect others to remember and for which type of information they themselves are the expert. New information is “*channeled*” to the individuals most likely to remember it; for retrieval then, the assumed expert gets asked. To relate these concepts back to software development: When developers talk about knowledge transfer in the context of pair programming as a work mode, they mostly refer to *task-related knowledge*, e.g., about how to approach a debugging problem, and to *task-specific knowledge*, e.g., a mental model of the software system. In contrast, pair programming as a practice on a

team level, i.e., how Beck (1999, p. 97) sees it when he refers to it as the central element that “ties the whole process together”, refers more to *shared beliefs* and a *team mental model* (see also Section 2.3.1a).

Secondly, Cannon-Bowers & Salas (2001, pp. 198–199) analyze what “shared” exactly means. Different notions reflect the requirements of certain situations:

- **Overlapping knowledge** in the form of a ‘knowledge base’ for some task, e.g., between a surgeon and a nurse.
- Almost **identical knowledge**, e.g., with regards to attitudes and beliefs in order to make common interpretations.
- **Complementary knowledge** embodied in specialized roles, e.g., different experts whose different perspectives are necessary for the task.
- **Distributed knowledge** across the team for situations that are too complex for a single team member to manage.

In the context of industrial pair programming, speaking of knowledge transfer may refer to building a *knowledge base* where long-term transfer between developers is intended, but also to situations of *complementary* or *distributed knowledge* where the knowledge of two developers is combined for immediate use. *Identical knowledge* in software development may come in the form of shared understanding of the requirements, for example. Sharedness in this sense may also be supported by pair programming, although other practices such as sprint plannings are dedicated to such purposes and also involve more than just two team members.

#### ***Process Approach: Coordination as a Group Activity***

Bossche et al. (2006, pp. 494–495) describe three processes—together called **team learning behavior**—which a team needs to employ if it is to arrive at a *shared cognition*:

1. **Construction**: One team member describes the situation and how to deal with it and thereby *constructs meaning*. The other team members actively try to understand it.
2. **Co-Construction** (Collaborative Construction): The team refines and combines individual meanings such that new meanings emerge.
3. **Constructive Conflict**: To go beyond mere mutual understanding of everyone’s positions, the team clarifies and negotiates different interpretations of the situation.

Gabelica et al. (2016, p. 37) add **team reflexivity** as another component that supports the team learning behavior at various points before, after, and during task execution: Goals and requirements are defined, a strategy is planned and adapted along the way to account for newly arisen problems and intermediate products that do not meet the expectations, and achievements and failures are reflected on to improve processes in the future.

How these processes actually pan out depends on many **beliefs** within the group, e.g., the *psychological safety* that speaking up in the group does not trigger negative reactions, the *interdependence* of individuals’ work and benefits from that of the others, the *social cohesion* that makes the members feel belong together, the *task cohesion* around shared commitment to a common goal, and the *group potency*, which is the belief that the group can be effective (Bossche et al., 2006, pp. 497–502).

#### **2.4.2 b) Effectiveness**

Hackman (1990, p. 4) acknowledges the differences between groups that are studied under laboratory conditions and organizational work groups who are *real*, intact social systems embedded in *organizational contexts*. He characterizes three dimensions of group effectiveness whose relative weights are different per context (*ibid.*, pp. 5–7):

1. Meeting output standards, e.g., quantity, quality, timeliness; or **performance** for short (Mathieu et al., 2000, p. 273).
2. Enhancing the capability of group members to work together in the future. Others called this “*team longevity*” (ibid., p. 273) or **viability** (Bossche et al., 2006, p. 497).
3. **Professional growth** and well-being of the group members, also called “*affective reactions*” (Mathieu et al., 2000, p. 273).

Relating back to my discussion of expected pair programming mechanisms and effects (see Section 2.3.1c), the COMBINE and UNDERSTAND mechanisms affect a pair’s *performance*, while *team viability* and *professional growth* are two ways of looking at LEARN effects.

#### 2.4.2 c) Exemplary Studies

Even the narrow area of group phenomena described above gets rather complicated to study in large teams. After all, each team member brings her own understanding to the table and needs to maintain her own mental model of the team. Therefore, the “*two-person team*” or *dyad* is often used as the “*smallest and simplest form of teamwork and knowledge distribution*” (Gabelica et al., 2016, p. 34).<sup>33</sup>

I now discuss three empirical studies that build on the above sketched model of how teams work to illustrate (a) the *variation* among such studies with respect to time frame, breadth of investigated aspects, and chosen research instruments, and (b) the *inappropriateness* of such approaches for understanding how knowledge transfer in pair programming actually works. Individual studies vary along different dimensions:

- **Level:** Concrete task for several minutes vs. group assignment over several weeks
- **Instruments to Measure Constructs:** Self-reported (questionnaire items) vs. inference from produced artifacts vs. characterization of behavior (rating form) vs. objective measurements
- **Relation of State and Process:** Shared cognition as a product of group behavior vs. process properties resulting from shared cognition (no study considered a circular or reciprocal relationship)

The characteristics of three studies by Mathieu et al. (2000), Bossche et al. (2006), and Gabelica et al. (2016), which all by-and-large support the general model of how teams coordinate their knowledge described above, are summarized in Table 2.14.

#### *Long-Running Group Task*

Bossche et al. (2006, pp. 502–503) studied 75 teams of 3 to 5 students working on a group assignment over a seven-week period whose task was to provide strategic advice for a company in form of a whitepaper and a presentation. Their statistically tested model had four stages (see also Table 2.14): (1) Team beliefs such as psychological safety and task cohesion positively influence (2) team learning behavior, which leads to a better (3) shared cognition among the team, which in turn leads to better (4) performance, viability, and learning/professional growth.

All constructs were measured with items in a questionnaire the subjects filled out in the last week of their project (ibid., pp. 504–507). All individual correlations between the stages were statistically significant as expected (ibid., pp. 509–510). From stages (1) to (3), the model’s fit was “*acceptable*”, but to account for all stages (1) to (4), additional paths needed to be included (ibid., pp. 510–513). In other words: The model is incomplete since the group beliefs of task cohesion and group potency appear to also positively affect the team effectiveness in ways that are not explained through team learning behavior and shared cognition alone.

---

<sup>33</sup>See also Flor & Hutchins (1991), discussed on page 78, who headlined their paper with “*distributed cognition in software teams*” but studied merely a single **pair** of programmers.

Study	Level	Staged Model with Constructs (and Instruments)
Mathieu et al. (2000)	two-person teams, six 10-minute missions in flight simulator	<ol style="list-style-type: none"> <li>1. State: team &amp; task mental models (matrix similarity)</li> <li>2. Behavior: team process (rated based on video)</li> <li>3. Outcome: performance (objective measurement)</li> </ol>
Bossche et al. (2006)	3–5 person teams, seven-week group assignment	<ol style="list-style-type: none"> <li>1. Beliefs: psychological safety, interdependence, social and task cohesion, group potency (questionnaire)</li> <li>2. Behavior: team learning behavior (questionnaire)</li> <li>3. State: common understanding of task and how to approach it (questionnaire)</li> <li>4. Outcome: performance, viability, professional growth (questionnaire)</li> </ol>
Gabelica et al. (2016)	two-person teams, four 15-minutes missions in flight simulator	<ol style="list-style-type: none"> <li>1. Beliefs: task cohesion, group potency (questionnaire)</li> <li>2. Behavior: team learning behavior, team reflexivity (questionnaire)</li> <li>3. State: common understanding of task and how to approach it (questionnaire)</li> <li>4. Outcome: performance (objective measurement)</li> </ol>

**Table 2.14:** Overview of knowledge coordination studies with pairs. Numbers in third column indicate stages in statistically tested models. All aspects from one stage positively correlate to all aspects of the next stage, with only few following exceptions in the studies of Bossche et al. (2006) and Gabelica et al. (2016).

**Discussion** While questionnaires may be suitable for capturing beliefs (stage 1) and for some aspects of team effectiveness (4), the teams' *actual* behavior (2) was not observed, but addressed with self-reporting items such as “*Team members are listening carefully to each other*” (ibid., p. 504). Furthermore, the state of shared cognition (3) was only measured once at the end of the seven-week assignment and only indirectly with items such as “*At this moment, this team has a common understanding of the task we have to handle*” (ibid., p. 506).

Although there were distinct questionnaire items for all three components of team learning behavior (i.e., construction, co-construction, constructive conflict) and also for team effectiveness (i.e., performance, viability, learning), they were all considered equal parts of the higher-level constructs and *not* analyzed separately (ibid., Tables 1 & 2).

### **Short Specialized Group Task**

Both Mathieu et al. (2000) and Gabelica et al. (2016) randomly assigned students to pairs (56 and 33 pairs, respectively) and trained them with a flight simulator: One pair member was the ‘pilot’ flying the plane, the other had ‘co-pilot’ responsibilities like setting the speed and gathering information. In both studies, the pairs flew multiple missions with predetermined objectives for a total duration of 60 minutes (six missions at 10 minutes and four missions at 15 minutes). Both studies had objective criteria for evaluating the pairs’ performances (time until completion, passed waypoints, etc.) and the statistical results of both overall support the above discussed central role of shared cognition.

The studies do, however, differ in the underlying models of how the performance is achieved and whether a state of shared cognition is cause or effect of team behavior: For Mathieu et al. (2000), (1) the pair’s shared cognition allows for better (2) process quality, which in turn increases the (3) performance. Gabelica et al. (2016) extended the model of Bossche et al. (2006, see page 100) and included team reflexivity behavior in stage (2) to account for the previously unexplained effects of task cohesion and group potency on shared cognition, thus

considering team beliefs and behavior as the *origin* of shared cognition (again, see Table 2.14 for an overview). The studies also differ in their operationalizations of relevant constructs:

**Shared Cognition** Gabelica et al. (2016) relied on a questionnaire with items such as “*Our team worked together in a well-coordinated fashion*” or “*We accomplished the mission smoothly and efficiently*” which came from a validated scale for testing the existence of a transactive memory system.

Mathieu et al. (2000), in contrast, addressed the pair members’ respective task and team mental models more directly: Prior to the study, the researchers identified  $n$  aspects of the specific task and of teamwork in general and let the team members fill out an  $n \times n$  matrix to rate the relation of all aspects against the other. In concrete terms: Multiple times between their missions, the subjects had to rate the relation of eight task-specific aspects (e.g., *intercept enemy* is closely related with *adjust airspeed*, but not with so with *escape enemy*) and of seven aspects of the pair (e.g., how strongly *quality of information* relates with *roles* or with *team spirit*). The ‘sharedness’ of the mental models was then defined as the similarity of the matrices.

Although this study attempts to measure the shared cognition directly, I doubt that a weighted adjacency matrix showing all possible relations of aspects of teamwork is an appropriate and rich enough representation of a “*team mental model*” which is about “*the knowledge, skills, attitudes, preferences, strengths, weaknesses, tendencies, and so forth*” of the teammates (*ibid.*, p. 274). I am also not convinced that a high correlation of two matrices indicates actual agreement between the subjects who filled them in.

**Process Quality** Mathieu et al. (*ibid.*) assessed the team process quality based on video recordings of the pairs’ missions with a 21-item rating form addressing the pair’s coordination, cooperation, and communication with items such as “*To what extent was information about important events and situations shared within the team?*” from “*0 – not at all*” to “*5 – to a very great extent*”.

Similar to the qualitative-quantitative analyses of pair programming (see page 75), such a scheme condenses each pair session to a single data point and disregards the actual interaction along the way.

**Team Beliefs and Behaviors** Similar to Bossche et al. (2006), Gabelica et al. (2016) used a questionnaire to assess task cohesion and group potency as well as team learning and team reflexivity behavior. Yet again, a questionnaire is reasonable for addressing beliefs, but less so for actual behavior.

**Results** The statistical results reported by Mathieu et al. (2000, p. 279) are similar to those of Bossche et al. (2006) discussed above: From one stage of the model to the next, all proposed connections had a significant positive correlation individually, while the model as a whole has “*substantial support*” but still appears to miss some indirect effect. In particular, the pairs’ shared cognition with regard to the task appears to only affect the team process quality, but not the overall performance.

Gabelica et al. (2016, pp. 43–44) also report an “*almost acceptable fit*”. In their data, group potency did not have a significant effect on either team process, but appears to affect shared cognition in some other way. Also, team reflexivity behavior was not significantly affected by task cohesion (the other tested team belief, next to group potency), leaving its origins in the dark. Finally, unlike the earlier study by Bossche et al. (2006), team learning behavior did *not* predict shared cognition. In their discussion, Gabelica et al. (2016, p. 47) thus interpret team learning behaviors as a relevant part of team reflexivity (which *did* affect shared cognition and thus performance), but not as a direct factor for team performance.

### 2.4.3 Summary of Psychological Research on Pair Work

To finish off this excursus into psychology, I summarize what can be learned about research in software engineering and about pair programming in particular. My overall impression from reading secondary literature is that the psychology research community's practice of building and refining a theory to explain different but related phenomena is far more advanced than in software engineering—at least for the narrow area of group phenomena I touched in this section. However, looking into (in part) well-cited empirical studies<sup>34</sup> was sobering.

First, individual studies as recent as 2016 still resort to “*two-person teams*” as the “*simplest form of teamwork*” to understand basic mechanisms of how performance is determined by beliefs and behaviors, and yet conclude that team reflexivity, which had a significant positive on team performance, “*still has a great deal of unknown antecedents to be identified*” (Gabelica et al., 2016, pp. 34 & 45).

Second, the analyses are oriented towards statistical models which require means of data collection that *quantify* social phenomena: Either questionnaires with rating scales are used to check for the perceived existence, but not the actual manifestation and specific properties of shared cognition and the team processes that lead to it (Bossche et al., 2006; Gabelica et al., 2016); or subjects are required to fit their mental models into a fixed mathematical form such as matrices (Mathieu et al., 2000). In fact, Bossche et al. (2006, p. 516) and Gabelica et al. (2016, p. 49) are fully aware of these limitations and note that qualitative analyses of actual behavior would help understanding the *how?* and *why?* better. So far, the actual team processes appear to be some kind of blind spot and psychologists direct their attention elsewhere.

Third, whether cognitive psychologists employ the pair as an extension of the individual to make otherwise internal processes observable (see Section 2.4.1) or social psychologists use it as the smallest possible group to simplify their analyses (see Section 2.4.2), the performed tasks are usually constricted and carefully chosen to fit the study's purpose, e.g.:

- the flight simulation had two distinct roles to enforce some difference in the individuals' cognition (Mathieu et al., 2000; Gabelica et al., 2016),
- the sewing machine is a complicated machine that can yet be completely understood in detail within one hour (Miyake, 1986),
- understanding the simulated genetic regulatory mechanism requires to perform a (small) set of predefined experiments and to entertain alternative hypotheses (Okada & Simon, 1997), and
- the visual perception task can be fully described in simple mathematical terms allowing to devise and compare multiple models (Bahrami et al., 2010).

It is unclear how the results translate to tasks which do not have such restrictions and where the group can find creative solutions.

Although the concepts of shared cognition and the processes leading to it are generally applicable to individuals' pre-existing troves of knowledge as well, all these tasks required less than an hour of instruction. There are more ways in which they differ drastically from pair programming in industrial software development: Commonly, software developers do neither need to react to the real-time behavior of their system nor is their work done once they understood something. Rather, they *construct* solutions for problems that (again, unlike the experimental tasks) have no clearly defined goal or even clearly defined conditions to decide whether a goal state has been reached. In industry, programming partners are *not*

---

<sup>34</sup>According to Google Scholar, Mathieu et al. (2000) and Bossche et al. (2006) were cited more than 2800 and 750 times, respectively (as of June 2020).

randomly assigned, briefly instructed how to work on some task together which they have little experience in and without the expectation to ever do it again.

Nevertheless, when studies from different researchers with many years in between, covering different aspects of teamwork with different operationalizations of the relevant constructs all work with and find support for the same underlying model summarized above, that model and its concepts may not be too bad. The following concrete research findings and more general observations are also applicable to PP and PP research:

- **No Black Box:** Considering group size, composition, task type, and other external factors alone is not enough to explain differences in group performances (see page 97).
- **Rich Interaction:** Pair performance is not simply the sum of the pair members' individual abilities and actions (see Section 2.4.1).
- **Not Only Quantitative Methods:** Statistical models do not explain how and why things happen (see page 103).
- **Central Concepts:** Although there are many nuances and details are not fully understood, the state of *shared cognition* and the coordination processes to get there appear to be key to understanding group performance (see Section 2.4.2a).
- **Real-World Effectiveness:** The mere technical performance is not all that matters for real-world groups in organizational settings; there are also the individuals' professional growth and the group ability to work together in the future (see Section 2.4.2b).

## 2.5 Summary of Related Work

Knowledge plays a central role in software development. In general, it takes new team members about three years to acquire the necessary knowledge to become *fluent* in a project (see Section 2.2.3).

Pair programming is a practice that software development teams may use to ramp up new team members and to dissolve and avoid knowledge silos in mature teams. Even when it is not employed strategically, software developers work in pairs on a per-task basis. Many of the *expected* benefits boil down to the central role of knowledge through the following assumed mechanisms: The pair members' existing knowledge is put to use to work on difficult problems and come up with better designs, lacking knowledge is required faster by a pair which is also helpful for debugging and catching defects on the fly, and the developers improve their abilities to work on future tasks (see Section 2.3.1c).

Whether, when, and how such effects come to be, appear to be difficult questions for software engineering researchers. Even in controlled settings with carefully chosen development tasks and under consideration of developer expertise as well as psychological traits, pair programming effects on cost and quality show mere tendencies with a lot of variation left unexplained (see Section 2.3.4a). Project-level case studies collect data in real software development projects, but also fall short of explaining what makes pair programming work (see Section 2.3.4b). The research situation is similar to that of group psychology, which also started with considering structural aspects of groups (e.g., size, composition, task type) and could thus explain general effects, but failed to make specific predictions and then focused more on the interaction processes (see Section 2.4.2).

Qualitative-quantitative studies on pair programming codify certain predefined aspects of pair programming processes (e.g., 'driving' times or communication properties), but all perform statistical analyses on aggregated data thus disregarding the process itself (see Section 2.3.4b). Again, the situation is similar in group psychology, where statistical models are used extensively, but often rely on group behavior getting reported after the fact by the subjects themselves with a number of rating items (see Section 2.4.2c).

Qualitative studies in general are oriented towards *developing* rather than *testing* a theory. Although more and more pair programming studies employ qualitative methods—even on the topic of knowledge transfer in particular—they are often concerned with isolated and narrow aspects without an eye on how an overall scientific understanding of pair programming may be developed and how practitioners may eventually benefit from such insights (see Section 2.3.4d). The few existing qualitative studies on knowledge transfer in pair programming either amount to enumerating two types of relevant knowledge or consider pair behavior in expert/novice constellations in general, but not knowledge transfer activities as such.

In summary, little is known about how software developers actually make use of their knowledge and deal with what they do not yet know in everyday industrial pair programming sessions. The goal of my thesis is to narrow this gap with a qualitative research approach. I discuss the basic ideas and properties of qualitative research in Chapter 3; I define my goal, discuss my method, and describe how I collected and analyzed which data in Chapter 4. The results of my work then constitute Part II with Chapters 5 to 11.



## Chapter 3 Qualitative Research Methods

---

*The origin of science is in the desire to know causes; and the origin of all false science and imposture is in the desire to accept false causes rather than none; or, which is the same thing, in the unwillingness to acknowledge our own ignorance.*

– William Hazlitt

<b>3.1 Purpose and Structure of this Chapter</b> . . . . .	108
<b>3.2 Research Methods of the Social Sciences</b> . . . . .	108
3.2.1 On Understanding and Reconstructing . . . . .	109
<i>Common Ground and Reciprocal Knowledge • Conversational Elements: Maxims, Turns, and Speech Acts</i>	
3.2.2 Common Characteristics of Qualitative Research Methods . . . . .	113
3.2.3 Variability of Qualitative Research Methods . . . . .	115
3.2.4 Quality Criteria for Qualitative Research . . . . .	115
<b>3.3 The Grounded Theory Methodology</b> . . . . .	117
3.3.1 Overview . . . . .	117
3.3.2 Collecting Data . . . . .	118
<i>Theoretical Sampling • Theoretical Saturation</i>	
3.3.3 Analyzing Data . . . . .	119
<i>Open Coding • Axial Coding • Conditional Matrix • Selective Coding • On Developing Concepts: Theoretical Sensitivity • Writing Memos</i>	
3.3.4 Different GTM Versions . . . . .	124
<i>Classic Grounded Theory • Constructivist Grounded Theory</i>	
3.3.5 Discussion of GTM as a Qualitative Research Approach . . . . .	126
<i>Meeting the Quality Criteria • Filling the Common Traits</i>	
<b>3.4 The Base Layer for Pair Programming Research</b> . . . . .	128
3.4.1 Auxiliary Practices for the Grounded Theory Methodology . . . . .	129
<i>Perspective on the Data • Concept Name Syntax Rules • Analysis Results Metamodel • Pair Coding</i>	
3.4.2 The Base Layer in a Nutshell . . . . .	130
<i>Layered Research Approach: Different Perspectives on the Data • Seven Key Decisions • The Base Concept Set</i>	
3.4.3 Example Application of the Base Concepts . . . . .	135
3.4.4 Notion of “Knowledge” in the Base Layer . . . . .	137

### 3.1 Purpose and Structure of this Chapter

This chapter lays the groundwork for my own qualitative research method to be discussed in Chapter 4. Qualitative research is neither a ‘fallback’ to a *qualitative analysis* after a statistical test failed to show significant effects, nor is just any analysis of somehow unstructured or *qualitative data*. It is a *paradigm* that affects how researchers analyze data, how to collect it, which questions to ask in the first place, and how they understand their own role.

I explain the basic ideas of **qualitative research methods**, their commonalities, most important differences, and quality criteria (Sections 3.2.2 to 3.2.4), after I introduced some key terms from **discourse analysis** (Section 3.2.1). Although these are not necessary to understand my research, they add clarity to the discussion later on because they make *tacit knowledge* which every speaker of a natural language possesses *explicit*.

I build on the **Grounded Theory Methodology** (GTM), a universal qualitative approach which is applicable to a variety of phenomena of interest (Section 3.3).

Although the GTM is open with regard to the research topic, its data analysis practices were originally developed for interviews rather than direct observation. For previous studies, colleagues from my research group started collecting recordings of industrial pair programming sessions, and developed auxiliary practices to analyze such data. I also build on the resulting research framework, the **Base Layer** for qualitative pair programming research (Section 3.4).

### 3.2 Research Methods of the Social Sciences

Software development in general and pair programming in particular are *social* phenomena (see, e.g., Weinberg, 1971, p. 45, cited on page 20). I therefore take a glimpse into the methodology of the social sciences, starting with the work of German sociologist Max Weber from the early 20th century. (Note that the following is by no means a proper introduction of the methodological debates of the 20th century.) Weber defined *social action* as:

“ [H]uman behaviour when and to the extent that the agent or agents see it as subjectively *meaningful*: the behaviour may be either internal or external, and may consist in the agent’s doing something, omitting to do something, or having something done to him. By ‘social’ action is meant an action in which the meaning intended by the agent or agents involves a relation to *another* person’s behaviour and in which that relation determines the way in which the action proceeds.

Weber (1922, p. 1), translated in Weber (1978, p. 7, emphasis in original)

In Weber’s sense, the meaning of an action is *the meaning as intended by the agent*, and is not “to be thought of as somehow objectively ‘correct’ or ‘true’” (*ibid.*, p. 7).

Austrian philosopher and sociologist Alfred Schütz furthered Weber’s thoughts in his monograph *Der sinnhafte Aufbau der sozialen Welt* (Schütz, 1932).<sup>1</sup> In the 1950s, he characterized the social sciences as follows:

“ The primary goal of the social sciences is to obtain organized knowledge of *social reality* [...] the sum total of objects and occurrences within the social cultural world *as experienced* by the common-sense thinking of men living their daily lives among their fellow-men, connected with them in manifold relations of interaction.

Schutz (1954, p. 261, emphasis added)

<sup>1</sup>The title literally reads “The meaningful construction of the social world”. Schütz emigrated to the United States and published in English under the names Schuetz and Schutz. His main work was translated into English and posthumously published as *Phenomenology of the Social World* (Schutz, 1967).

Like Weber, Schütz emphasizes the importance of the *subjective* meaning of human action: The *social reality* is already interpreted and given “*specific meaning and relevance structure*” by the human beings living in it—such as language, social conventions, and cultural institutions—*before* a researcher starts studying it (*ibid.*, p. 266). Any study on the social world needs to take these mental constructs into account. Schütz contrasts this situation with “*the natural science*” where it is up to the scientists to determine what is relevant for them as the observational fields have no inherent relevance: “*The world of nature [...] does not “mean” anything to the molecules, atoms, and electrons therein*” (*ibid.*, p. 266).

In the social reality, most interaction builds on understanding one another which involves *interpreting* each other’s behavior based on common-sense constructs, such as understanding the language, understanding an utterance as a question, understanding what the question is about, etc. (Schuetz, 1953, p. 17). This mutual understanding is not perfect, but it works sufficiently well if the conversation partners share a cultural background and common experiences (Schuetz, 1953, pp. 11–12; Przyborski & Slunecko, 2009, pp. 146–147).

In order to collect data, any empirical researcher concerned with social actions must either communicate directly with the subjects or observe their interaction (Przyborski & Slunecko, 2009, p. 147). Hence, the researcher faces the same task of understanding and interpreting the other’s behavior, for which Przyborski & Slunecko describe two options, each corresponding to a research paradigm. In Schütz’s terms, both approaches of the social sciences deal with *second-degree constructs* to organize their knowledge of the social reality. These are “*constructs of the constructs made by the actors on the social scene*” (Schutz, 1954, p. 267).

- **Quantitative approaches** try to *eliminate* any interpretation on the part of the researcher. This is achieved through standardized research instruments, which should produce objective, reliable, and valid data, e.g., by making sure all subjects understand survey questions in precisely the same way (Przyborski & Slunecko, 2009, p. 147). The (second-degree) constructs used by the researcher are manifest, e.g., in predetermined response categories of questionnaires. Their construction is generally carried out in advance (*ibid.*, p. 153). On the one hand, pre-structuring the observations this way allows for statistical analysis; on the other hand, such an approach may not appreciate the varying perspectives and experiences of the subjects (Patton, 2002, p. 14).
- **Qualitative approaches**, in contrast, emphasize reconstructing the subjectively intended meaning of the observed human beings through their actions and artifacts (Przyborski & Slunecko, 2009, p. 153). These methods are therefore sometimes called **reconstructive methods** instead (*ibid.*, p. 144). Hence, in qualitative research, “*the researcher is the instrument*” (Patton, 2002, p. 14), who, in order to allow for an appropriate reconstruction, is to collect as much of the actions or communication as possible (Przyborski & Slunecko, 2009, p. 148).

### 3.2.1 On Understanding and Reconstructing

Communication is a central aspect of qualitative methods in the social sciences, and it is in two ways. First, social action always involves communication, i.e., understanding and interpreting what the *other* (whose involvement makes the action *social*) means to say or do. Communication is therefore integral to the social *phenomenon being studied*, either directly, as communication *per se* (as linguists do), or as part of another social action (such as software development). Second, the researcher communicates with the subjects in order to *collect data*, either directly or indirectly, e.g., in interviews or through observation. The key insight here is that understanding

what the subjects meant to say or do from a researcher's perspective is *not* fundamentally different from common-sense understanding in everyday conversation.

The abilities and the underlying knowledge that allow speakers of a language to understand each other—and thus to communicate—is the concern of the linguistic branch of *pragmatics* (Ehrhardt & Heringer, 2011, p. 11). Much of what is *meant* in everyday communication is not explicitly said, as the following exchange illustrates:

A: “When did you two marry?”

B: “In Rome.”

(*ibid.*, p. 11)

On the surface, this exchange does not make sense: A asked for a point in time, but B answered with a location. If, however, B and their spouse did spend some time in Rome in the past, and B can rightfully assume that A knows when that was, B's answer is perfectly fine (*ibid.*, p. 11). Pragmatics studies *how* speakers employ *which* kind of knowledge to augment what was said to be able to understand what was meant (*ibid.*, p. 18).

There are a number of concepts and findings from the field of pragmatics that are useful for qualitative research in the social sciences in general and my research of knowledge transfer in pair programming in particular. Note the special nature of pragmatics concepts: They all capture *tacit* knowledge of the speakers of a language. Being aware of these concepts is not necessary for reconstructing the meaning of the subjects' interaction. Nevertheless, they provide a useful vocabulary later on for explaining my reconstructions in a more comprehensible way and for discussing my own concepts with respect to the existing literature.

### 3.2.1 a) Common Ground and Reciprocal Knowledge

Knowledge has an important role in communication. For example, in the exchange above, both A and B (presumably) *know* that B is married and that B spent some time in Rome. This type of shared knowledge is called **common ground**: “*the propositions whose truth [the speaker] takes for granted as part of the background of the conversation*” (Stalnaker, 2002, p. 81). However, as Ehrhardt & Heringer (2011, pp. 37, 40) point out, there is actually only *individual* knowledge while referring to shared or collective knowledge is merely a normative notion: It is this fiction which the involved speakers live in until proven otherwise that enables communication in the first place.

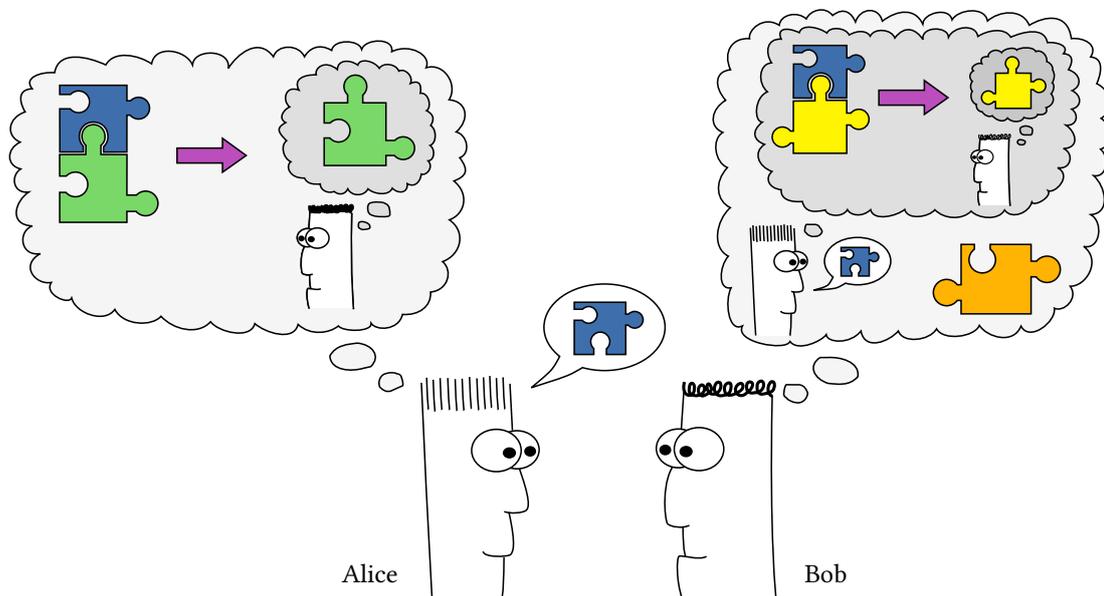
Garfinkel (1967, pp. 1–34) also speaks of the *indexicality* of everyday language: The meaning of an utterance cannot be understood from its words alone, as they only *refer* to the specific meaning intended by the speaker. In the exchange above, ‘Rome’ is indexical as it does not (only) refer to the capital of Italy, but also to B's biographical connection with it. A related notion is that of **deictic** expressions which refer to something from the speaker's point of view or **origo**, e.g., when she says something like ‘*this*’ or ‘*that*’ (Ehrhardt & Heringer, 2011, pp. 19–21). While communicating, the conversation partners constantly put themselves in the position of the other, assuming her *origo*, e.g., by saying something like ‘*I'm coming [to you]*’ instead of ‘*I'm going [towards you]*’ (*ibid.*, p. 38).

This changing of perspectives implies that conversation partners each maintain a mental model of their own and of their partner's knowledge, leading to **reciprocal knowledge** that is the basis of human sociality (*ibid.*, p. 38). Moreover, this reciprocal knowledge is *layered* in the sense that A thinks what B might think about A's knowledge (*ibid.*, p. 38).<sup>2</sup> Incorrect assumptions about what is common ground will inevitably happen and lead to

---

<sup>2</sup>Friedell (1969) introduces the notion of *common knowledge* between persons A and B, if A and B think it is true, A and B think that both think that it is true, A and B think that both think that both think that it is true, and so on.

misinterpretations of utterances and misunderstandings on one reciprocal knowledge level or another (*ibid.*, pp. 39–40). In a way, the purpose of communication is to extend, coordinate, and adapt the common ground (*ibid.*, p. 40). In Figure 3.1, I try to make these abstract ideas a bit easier to grasp.



**Figure 3.1:** Illustration of reciprocal and layered nature of communicative knowledge. Alice knows  and . She assumes that Bob already understood  and explains () the matching . Bob hears and understands Alice's words, but he does not actually know . Instead, he knows . He can infer that Alice assumed that he knows about some other piece () which fits to . He could start clearing up this discrepancy (e.g., by asking for , or by describing ).

The researcher's perspective adds yet another layer of complexity, as only  is observable (and the partners' actions before and afterwards).

### 3.2.1 b) Conversational Elements: Maxims, Turns, and Speech Acts

#### *Grice's Maxims*

A general principle of understanding one another is described by a number of **maxims** formulated by Grice (1975). I summarized them in Figure 3.2. Note that these maxims cannot be truly 'violated', but are metaphors for describing how communication works, i.e., the maxims are *descriptive*, not *normative* (Ehrhardt & Heringer, 2011, p. 73).

The maxims are expectations, and when one speaker in a conversation appears to violate one, her partner may start to think about alternative interpretations of her utterances that do not violate them. In the Rome-exchange on page 110, B's response may, at first, appear to A as a violation of the *maxim of relation* since it does not answer the posed question. A may then think about alternative explanations: Assuming B *does* consider 'In Rome' to be a relevant response to the question, B must have either misheard the question, or their wedding indeed took place in Rome at some point in time.

<p><b>Maxims of Quantity:</b></p> <ol style="list-style-type: none"> <li>1. Make your contribution as informative as is required (for the current purposes of the exchange).</li> <li>2. Do not make your contribution more informative than is required.</li> </ol> <p><b>Maxims of Quality:</b> Try to make your contribution one that is true.</p> <ol style="list-style-type: none"> <li>1. Do not say what you believe to be false.</li> <li>2. Do not say that for which you lack adequate evidence.</li> </ol>	<p><b>Maxim of Relation:</b> Be relevant.</p> <p><b>Maxims of Manner:</b></p> <ol style="list-style-type: none"> <li>1. Avoid obscurity of expression.</li> <li>2. Avoid ambiguity.</li> <li>3. Be brief. (Avoid unnecessary prolixity.)</li> <li>4. Be orderly.</li> </ol>
---	---

**Figure 3.2:** The maxims formulated by Grice (1975, pp. 45–46). He grouped them in four categories and formulated a “*supermaxim*” for the Quality category.

### Turn Taking

A conversation consists of **turns** during which the respective speaker has the proverbial “floor” and produces utterances (Yngve, 1970, p. 568, cited by Harrington, 2018, p. 135). Grice’s *maxim of relation* captures the observation that consecutive turns are not randomly strung together, but have *conditional relevance* (Schegloff, 1968, p. 1083): One turn imposes some constraints on what the partner’s next turn might be; together, they form a ‘whole’, and without the second, something would be ‘missing’. Recurring patterns of turn and matching reacting turn are called **adjacency pairs** (Schegloff & Sacks, 1973, p. 295), such as responding to a *greeting* with a *greeting*, or, as in the Rome-exchange above, to a *question* with an *answer*.

Not all utterances constitute complete turns, though. During the turn of one speaker (who occupies the “*front channel*”), the partner may signal her attention through **back channel** behavior, that is short utterances (‘yes’, ‘m-hm’, and similar) which do not interrupt the partner’s turn (Yngve, 1970, p. 568, cited by Harrington, 2018, p. 135).

### Speech Acts

Philosophy of language offers a different perspective on what individual turns actually are. To Austin (1962), *saying something* is *doing something* and he introduced the notion of **speech acts** to capture this idea. He splits spoken language into the following layers:

- The **locutionary act** is the act of *saying* something. It is realized by three sub-acts: With the *phonetic act*, the speaker produces a sequence of sounds, which, in the *phatic act*, produce a syntactical English sentence, which, through the *rhetic act*, refers to something. In the Rome-exchange, A’s phonetic act consists of producing the sounds /wen did ju tu: 'mæri/, which corresponds to the English sentence ‘*When did you two marry?*’ (phatic act), in which ‘*you two*’ refers to B and their spouse, ‘*marry*’ refers to a ceremony involving these two, etc. (rhetic act).
- The **illocutionary act** is performed *in saying* something, such as asking a question, or making a proposal. A simple notion would be that the illocution represents the *intention* of the speaker, or the *purpose* of the utterance. However, Searle (1969, p. 70) points out that there are also more subtle distinctions, such as relative position of speaker and hearer (marking the difference between *request* and *order*), degree of commitment (*expressing the intention* vs. *promising*), or expressed psychological state (*expressing belief in a statement* vs. *expressing intention in a promise*).

In the example above, A’s illocutionary act is successfully performed when B understood A’s intention: In saying ‘*When did you two marry?*’ A asked B a question.

- Finally, the **perlocutionary act** is performed *by saying* something, such as eliciting an answer, or informing the conversation partner. This entails all “*effects upon the feelings, thoughts, or actions of the audience, or of the speaker, or of other persons*”, whether they were intentional or not (Austin, 1962, p. 101).

In the example, A’s perlocutionary act consists in making B provide an answer.

Searle (1969, pp. 30–33) analyzed the concept of *illocutionary acts* in more detail and split them up into an **illocutionary force** (such as to ask, to promise, to warn, to threaten, etc.) and the **propositional content** (such as to come around, to quit smoking, etc.). Among other things, this distinction makes the difference clear between the two ways a phrase as ‘*I promise to come*’ can be negated: ‘*I do not promise to come*’ is a negation of the illocutionary force and ‘*I promise not to come*’ is a negation of the propositional content (*ibid.*, p. 32).

For the perspective of an *interpretative* researcher who is to reconstruct what happens in a social scene (see page 109), it is interesting to note how Searle legitimizes his linguistic characterizations: In essence, he argues that he is qualified to provide these because (a) the (English) language is governed by rules, and (b) he is a native speaker and has therefore “*mastered the rules*” (*ibid.*, pp. 12–15). In other words: He analytically tries to make his tacit knowledge explicit. To give but one example: In a conversation, illocutionary acts can be successful even if the illocutionary force is not explicit in the utterance but becomes clear from the context (*ibid.*, p. 68). Put differently, the illocutionary force of an utterance can always be *made explicit* (*ibid.*, p. 68), e.g., by a researcher. This falls under his *principle of expressibility* which states that whatever can be meant can be said (*ibid.*, pp. 19–21).

### 3.2.2 Common Characteristics of Qualitative Research Methods

There are many different qualitative methods. Patton (2002, pp. 81–135) compares no less than 16 “*perspectives*”—such as ethnography, ethnomethodology, hermeneutics, narrative analysis, and others—each of which gave rise to multiple qualitative research methods. Two reasons for this diversity are (a) roots in different disciplines (such as anthropology, philosophy, sociology, or psychology) and (b) the tendency to chose (or, historically, *develop*) a research method that suits the research topic (Flick et al., 2004, p. 8). Note that Patton’s structure is by no means the only one proposed to categorize the qualitative research methods of the social sciences (for a brief discussion, see Patton, 2002, pp. 79–80 & 131).

The methods share a number of core characteristics despite their variety. Patton (2002) and Flick et al. (2004) offer similar lists of 12 such characteristics in their textbooks. In Table 3.1, I provide a consolidated overview of what qualitative research is about in terms of research perspective and topics, overall design and sampling, as well as data collection and analysis.

Qualitative research builds on qualitative data, often collected through interviews or observations. **Interviews** provide the researcher with personal accounts of the subject’s experiences, feelings, and opinions, while direct **observation** can lead to detailed descriptions of the externally observable part of many different types of actions and interactions (Patton, 2002, p. 4). Some researchers see direct observation as the superior technique, as it is not limited to what the subjects are aware of and able to express (*ibid.*, p. 21). Others, however, do not regard interviews and direct observation as two separate techniques, because they are interwoven in actual field-work anyway (Lofland & Lofland, 1995, p. 19). Either way, the subjects should be able—especially during interviews—to express themselves freely in their terms, with the “*character and contour*” of their accounts being set by themselves rather than the researcher (*ibid.*, pp. 81, 85).

**1. NATURALISTIC INQUIRY IN EVERYDAY SITUATIONS** (PATTON-1, FLICK-3, FLICK-4)

Qualitative research is often concerned with everyday situations and/or everyday knowledge or *common sense*. Consequentially, qualitative methods study real-world situations as they unfold naturally. In contrast to laboratory settings, the researcher does not try to control the situation, there is no predetermined course of action, and people are observed in a familiar environment.

**2. OPENNESS & EMERGENT RESEARCH DESIGN** (PATTON-2, FLICK-2, FLICK-8)

A qualitative research method is selected based on the topic of interest, not the other way around. The methods do not prescribe rigid observation practices, but demand openness in data collection and analysis, and flexibility with regard to the research design when the understanding of the topic deepens.

**3. PURPOSEFUL SAMPLING** (PATTON-3, PATTON-8, FLICK-9)

Unlike for quantitative methods, statistical generalization from a sample to the population is not a goal. Instead, qualitative studies look for information-rich cases that allow to deepen the researcher's understanding. Early in the process, each case is treated as unique and studied in great detail; cross-case analyses follow later and are based on the well-understood individual cases.

**4. DETAILED, THICK, AND DIVERSE DATA** (PATTON-4, FLICK-5)

Qualitative methods aim at data that is rich in detail. Two typical and somewhat complementary ways of collecting data are in-depth, open-ended interviews and direct observation.

**5. IMPORTANCE OF THE RESEARCHER** (PATTON-5, PATTON-12, FLICK-6)

Qualitative researchers get close to the phenomena of interest to capture as much detail as possible. In general, the researcher's own actions and observations in the field, their own thoughts and insights, and their reflection on these are important for qualitative methods as opposed to an extraneous influence which needs to be eliminated. Consequentially, the qualitative researcher is not objective.

**6. EMPATHIC NEUTRALITY** (PATTON-6, FLICK-7, FLICK-10)

As the social reality is constructed by the human beings in it, it needs to be reconstructed by the researcher. For this, the subjects' perspectives need to be understood without judging them.

**7. CONTEXT MATTERS** (PATTON-11, FLICK-4)

Data should be collected in their 'natural environment', as the social, historical, and temporal context of the socially acting subjects is important for interpreting and analyzing their actions.

**8. HOLISTIC PERSPECTIVE** (PATTON-10, FLICK-7)

Social phenomena are complex in the sense that they cannot be fully understood by taking them apart, studying the pieces, and looking for simple cause-effect relationships. Qualitative methods aim at understanding phenomena holistically including their complex relationships.

**9. DISCOVERY OF THEORIES** (PATTON-9, FLICK-12)

Patton (2002, pp. 55–58) characterizes the overall qualitative research process as "*inductive analysis*": a bottom-up process which starts with concrete and detailed observations, from which overarching themes "*emerge*", which in turn serve as a form of hypothesis to be examined in the light of newly collected data. How this *emergence* precisely works depends on the particular research method. A procedure commonly used in qualitative methods is *abduction* (Flick et al., 2004, p. 8). This means that when the researcher notices a combination of features in her interpreted data for which there is not yet an explanation, she 'invents' one (Reichertz, 2004, p. 161).

---

**Table 3.1:** Overview of qualitative research traits based on Patton (2002, pp. 39–66) and Flick et al. (2004, pp. 8–9). The numbers, e.g., PATTON-2 or FLICK-8, refer to the items in the original lists of traits of qualitative research. I excluded items FLICK-1 (*there is not one single method*), FLICK-11 (*qualitative research is predominately text-based*), and PATTON-7 (*social systems are ever-changing*) here, because they are not defining criteria of qualitative research but only describe the status quo.

### 3.2.3 Variability of Qualitative Research Methods

As I mentioned above, there is no standard for how to organize the multitude of qualitative methods. Patton (2002, p. 134) therefore proposes a scheme comprising six core questions to characterize each particular research method. Any given qualitative method is opinionated with regard to only some of them, leaving the other aspects to be filled by the researcher. Unfortunately, Patton does not spell out any particular method along these lines. I illustrate his six questions with examples I consider appropriate based on his characterizations:

- 1) **Ontology:** *What do we believe about the nature of reality?*  
Logical positivism and other reality-oriented perspectives presume a single, verifiable reality; constructivism assumes multiple, individually and socially constructed realities (Patton, 2002, pp. 91–92, 96).
- 2) **Epistemology:** *How do we know what we know?*  
Reality-oriented perspectives aim for objective truth and are concerned with eliminating any influence of the researcher, and with validity, reliability, and objectivity (ibid., p. 93). Constructivism states that, ultimately, all understanding is subjective and “truth” can only be a consensus, but not an objective fact (ibid., p. 96).
- 3) **Method:** *How should we study the world?*
- 4) **Involvement:** *How do we personally engage in inquiry?*  
Reality-oriented perspectives may use triangulation for increased credibility, test theories and try to establish causal relationships (ibid., p. 93). Phenomenology stresses the importance of the researcher experiencing the phenomenon of interest as closely as possible (ibid., p. 106); heuristic inquiry even demands of the researcher to have personal experience and intense interest in the phenomenon (ibid., p. 107).
- 5) **Philosophy:** *What is worth knowing?*
- 6) **Discipline:** *What questions should we ask?*  
Ethnography assumes that any group of people will over time develop a “culture” with shared behavioral patterns and beliefs, and that these are worth understanding (ibid., p. 81). Orientational qualitative methods follow an ideology—such as Marxism, Freudianism, or feminism—that determines the perspective of all interpretation (ibid., p. 129).

### 3.2.4 Quality Criteria for Qualitative Research

While the set of traits I compiled in Table 3.1 above is *descriptive* in that it tries to capture what qualitative research is, Tracy (2010) sets out to formulate “a pedagogical model” of *normative* criteria of what qualitative research should be. She emphasizes that these “end goals” can be met by different means and that each project, researcher, context, etc. may warrant a different path (ibid., p. 837). For each goal, she gives an overview of possible means (ibid., pp. 840–848):

1. **Worthy Topic:** *Research topic is relevant, significant, and interesting.*  
Addressing topics set by one’s discipline, events in society, or personal life is better than choosing topics which are “convenient without larger significance or personal meaning”.
2. **Rich Rigor:** *Descriptions and explanations are complex and ‘rich’.*  
The researcher should spend enough time in the field to collect enough appropriate data and employ appropriate data collection and analysis procedures.
3. **Sincerity:** *Researcher’s role (biases, goals, mistakes, etc.) is transparent.*  
*Self-reflexivity* means for the researcher to honestly consider themselves as a research instrument with strengths and weaknesses, biases, values, and goals, and to include the first person perspective in writing. *Transparency* may be achieved through an audit trail

of all research activities and decisions, all interactions between researcher, subjects, and research contexts, as well as giving credit to others where credit is due.

4. **Credibility:** *Results are trustworthy enough for readers to act on them.*  
**Thick description** makes tacit aspects of a situation explicit with enough concrete detail to ‘show rather than tell’. **Triangulation and crystallization** both involve using multiple sources or types of data, multiple theoretical frameworks, or researchers, but differ in their outcomes depending on the researcher’s ontology (see page 115). Triangulation assumes a single reality from which “a more valid singular truth” is acquired, while crystallization is more compatible with multiple constructed realities as it produces “a more complex, in-depth [...] understanding of the issue”. **Multivocality** stresses the importance of addressing differences between subjects as well as between subjects and researcher. **Member reflection** covers a broad spectrum of activities to involve participants, e.g., by “providing opportunities for questions, critique, feedback, affirmation, and even collaboration”, going beyond mere “member checking” which presumes a single verifiable truth. This also helps to see whether the research is comprehended.
5. **Resonance:** *The research influences, affects, or moves the reader.*  
**Aesthetic merit** may come from a way of writing that goes hand in hand with the content, that is not boring and surprises the reader, and possibly includes personal story-telling. **Transferability** means that accessible writing and rich descriptions allow the reader to transfer the results to their own situation, despite them being bound to a specific and possibly different socio-cultural context.
6. **Significant Contribution:** *It extends knowledge, improves practice, generates research ideas.*  
**Theoretically significant** research applies existing concepts to new contexts, or extends concepts, or provides new concepts for further research; **heuristic significance** means to provide interesting suggestions for new research areas; **practically significant** research is useful and empowering, or helps framing some problem in another way; and finally **methodological significance** may come from new and creative ways of analyzing data or by qualitative analysis of phenomena that so far have been only studied quantitatively.
7. **Ethics:** *Effects of research actions on subjects and others are considered.*  
**Procedural ethics** is about the execution of the study and includes to “do no harm, avoid deception, negotiate informed consent, and ensure privacy and confidentiality”. **Situational ethics** mandates to go beyond the ground rules and adapt all considerations to the particular social scene(s). **Relational ethics** is about the relationship of researcher and participants, which should not only amount to using them as a data source, but also include returning to the scene to share findings. **Exiting ethics** then means to think about how a written report with, say, problematic findings may be perceived in a way that reflects negatively on the participants or a certain demographic.
8. **Meaningful Coherence:** *Theoretical framework, research method, and goals are aligned.*  
The study as a whole needs to make sense. Research question, findings, and conclusions need to relate to reviewed literature and open questions of the discipline. Basic assumptions of the theoretical framework need to be aligned with research activities (e.g., with the assumption of multiple individually constructed realities, member *checking* does not make sense). Any mismatches need to be explained.

### 3.3 The Grounded Theory Methodology

Most qualitative methods were developed for a specific area of interest, for instance: “*the narrative interview [...] was originally developed for the analysis of communal power processes, and objective hermeneutics [...] for studies of socializing interaction*” (Flick et al., 2004, p. 8). This is not just a historical fun fact, but has implications for researchers employing these methods today:

“ Ethnography focuses on culture, ethnomethodology on everyday life, symbolic interactionism on symbolic meanings in behavior, semiotics on signs, hermeneutics on interpretations, and phenomenology on lived experience. Their theoretical frameworks direct [the researcher] to particular aspects of human experience as especially deserving of attention in [their] attempt to make sense of the social world.

Patton (2002, p. 125)

The notable exception is the *Grounded Theory Methodology* (GTM): Although it, too, originates from a specific research interest, it is not limited to it.<sup>3</sup> It is a toolkit comprising procedures for developing a theory from empirical data. The GTM is not about testing a theory or preconceived concepts, but about developing concepts that capture relevant phenomena of social reality.

#### 3.3.1 Overview

There are different versions of the GTM, in particular the Glaserian, the Straussian, and the constructivist version. As Przyborski & Wohlrab-Sahr (2014, pp. 199–200) summarize, all variants share five essential properties:

1. **Theoretical Sampling:** Data collection is oriented towards furthering the theory development through information-rich cases, rather than aiming for statistical generalizability.
2. **Theory-Oriented Coding:** Data is *coded* not for the sake of coding but to develop *concepts* which are over time integrated into a *theory*.
3. **Constant Comparison:** Particular segments of the data are not analyzed and coded once and for all, but compared over and over again with new data. The same is done on the concept level, always looking for relevant differences and similarities.
4. **Memo Writing:** Writing is an integral part of the research from the very beginning. The researcher’s reflections, ideas, characterization of concepts, etc. are organized in written form and continuously amended.
5. **Process:** Data collection, coding, and memo writing are connected and the researcher switches between these activities in a non-linear fashion.

I see the three versions as follows: The original Glaserian GTM first defined these five properties, Straussian GTM then proposed three different perspectives of how *theory-oriented coding* should look like, and constructivist GTM challenged the positivist foundation of the other versions, which is more of a cross-cutting concern that affects how to approach both data collection and analysis.

In my analysis, I followed the Straussian coding procedures while assuming a constructivist perspective. I discuss the details of my particular research method together with the formulation of my research goal and data collection later in Chapter 4. Here in this section, I discuss the basic ideas and aspects of the GTM that are important to my work, but I do *not* explain how to perform a GT study in general.<sup>4</sup> I first discuss the general idea of theoretical sampling

<sup>3</sup>The original research was about *dying in hospitals*. See Charmaz (2006, pp. 4–9) for more historical context.

<sup>4</sup>See the books by Glaser & Strauss (1967), Strauss & Corbin (1990), and Charmaz (2006) or shorter summaries in English (Robson, 2002, pp. 492–497; Patton, 2002, pp. 124–129; Hildenbrand, 2004, pp. 17–23; Böhm, 2004, pp. 270–275) and German secondary literature (Przyborski & Wohlrab-Sahr, 2014, pp. 190–223).

in Section 3.3.2 and then the Straussian coding procedures in Section 3.3.3. In Section 3.3.4, I discuss the effects of a constructivist perspective using Patton's six qualifying questions (introduced in Section 3.2.3), before I take a step back in Section 3.3.5 to consider the GTM as a whole through the lens of the nine common characteristics of qualitative methods and the eight quality criteria (introduced in Sections 3.2.2 and 3.2.4).

### 3.3.2 Collecting Data

Fundamentally, in Grounded Theory Methodology "*all is data*" (Glaser, 2007, p. 57), e.g., notes from observations, existing literature, or even quantitative data. Although Glaser (*ibid.*, p. 53) emphasizes that "*[f]ield notes are preferable*", interviews are the predominant form in many studies. Much of the practical advice on how to *code* the data (see next section) pertains to the textual form of interviews and field observations, which should be transcribed on an as-needed basis (Strauss & Corbin, 1990, pp. 30–31).

#### 3.3.2 a) Theoretical Sampling

As rich concepts and dense relationships between them need to be developed *from* and *grounded in* data (hence the name of the methodology), the necessary data is not and cannot be collected in one single step before the analysis starts. Instead, these two concerns are interwoven: The analysis starts as soon as the first data is collected and new data is collected whenever the analysis goes into directions which the current data cannot satisfy. This mode of collecting data is called **theoretical sampling** (Strauss & Corbin, 1990, Ch. 11) with the goal of providing the researcher with information-rich cases. It contrasts with random sampling which is to allow statistical inferences to the population.

Early during the analysis, such sampling should be pragmatic to provide the best opportunities to collect relevant and interesting data, which might come unexpected and entail pivoting one's interest while being on site (*ibid.*, pp. 181–184). Later, the focus is more on capturing variation of already identified concepts as well as on how situations evolve over time (*ibid.*, pp. 185–186). Towards the end of a study, data collection is no longer explorative but becomes more selective as the researcher specifically looks for cases to either support the theory or to provide a negative case which cannot be explained yet (*ibid.*, pp. 187, 219).

#### 3.3.2 b) Theoretical Saturation

A notion related to that final phase and the question when to stop collecting data is that of **theoretical saturation**. To Glaser & Strauss (1967, p. 61), a concept is saturated when the researcher cannot develop additional properties, that is, specify it in more detail, by collecting additional data. As I see it, such a state can never be reached in a strict sense, as (a) there will always be *some* detail which sets apart two concrete incidents and which is not yet part of any concept, and (b) nobody can know whether the next incident would add a surprise aspect. Strauss & Corbin (1990, p. 188) put it slightly more pragmatic when they say that "*no new or relevant data seem to emerge*". Charmaz (2006, pp. 113–115) characterizes *saturation* as when "*fresh data no longer sparks new theoretical insights*", but ultimately questions its usefulness as a tool for the researcher who may settle too early for saturated but "*modest claims*". Rather, she should be "*open to what is happening in the field and be willing to grapple with it*". These three perspectives on *theoretical saturation* may be contrasted as follows: Glaser sees it as the researcher's ultimate goal, to Strauss, it marks the point when to stop collecting new data, and Charmaz submits that it may not be a sufficient condition to achieve *credibility*.

### 3.3.3 Analyzing Data

Strauss & Corbin (1990) propose an analysis process which involves three coding procedures: *Open coding* is about identifying relevant phenomena and conceptualizing their properties (Section 3.3.3a); in *axial coding*, the strategies which humans consider and employ to deal with certain phenomena as well as their consequences are identified (Sections 3.3.3b and 3.3.3c); in *selective coding* the core concept, the phenomenon that the study is ‘actually’ about, is identified (Section 3.3.3d). These three procedures are not sequential. Instead, the researcher switches between them, taking different perspectives: Focusing on phenomena in isolation, considering them in context, and considering them as a whole. In Section 3.3.3e, I discuss more deeply the matter of what *conceptualization* means and how it works in the GTM.

#### 3.3.3 a) Open Coding

On the level of *open coding* (Strauss & Corbin, 1990, Ch. 5), relevant phenomena are identified, labeled, and categorized. Although this might sound trivial, it is certainly not. For once, in social settings, there are no naturally separated phenomena ready to be labeled. For example, it is up to the researcher to decide on both the granularity of data segmentation (e.g., individual utterances/“transcript lines” or hour-long encounters of multiple people) and the focus of attention (e.g., individuals or groups).

While comparing different exemplars, the researcher over time develops more abstract concepts which are characterized by their *properties* (*ibid.*, p. 69). A concept has properties, for which a concrete instance has specific values giving it its “*dimensional profile*”. Strauss & Corbin (*ibid.*, p. 70) also call these “*general properties*” (of the concept) and “*specific properties*” (of the instance).

As new exemplars are studied in detail, some of them will be similar to already analyzed ones (and fall under the same concept) but at the same time be different in some way. To capture these variations, the concepts are continuously revised, e.g., by introducing new properties, which in turn requires the researcher to revisit already analyzed exemplars to see whether the amended concept still fits to it or what their specific value regarding a newly introduced property is. This process is called **constant comparison** (*ibid.*, pp. 62–63).

#### 3.3.3 b) Axial Coding

Straussian GTM builds on an “*action oriented model*” of social reality (Strauss & Corbin, 1990, p. 123), that is the idea that humans employ *strategies* to deal with a certain *phenomenon*. Or, in the words of the authors:

“ The purpose a grounded theory is to specify the conditions that give rise to specific sets of action/interaction pertaining to a phenomenon and the resulting consequences.

Strauss & Corbin (1990, p. 251)

While open coding considers phenomena in isolation, *axial coding* considers human behavior in response to such phenomena. To support an according analysis, Strauss & Corbin (*ibid.*, pp. 99–107) propose the **paradigm model** for thinking about social action:

- Some **causal conditions**, e.g., somebody saying or doing something, or simply a chance event, lead to a **phenomenon** that is experienced by the subject(s).
- In order to deal with such a phenomenon, humans employ **strategies**, which may be individuals’ or a collective’s actions or interactions within a group. These are purposeful in that the subjects want to deal with the phenomenon and they are processual, meaning

they may consist of sequences of actions and reactions. Although the term “strategy” may hint at deliberate behavior, Strauss & Corbin also include reflexive behavior.

- Any chosen strategy will have **consequences**, e.g., on the involved people or the broader context which in turn may cause some change in the phenomenon or a new one to arise.
- Through open coding, the phenomenon (and its causal conditions) may be characterized with arbitrarily many properties. Those relevant for the subjects dealing with the phenomenon, the “*specific set of perceived conditions*”, are called **context**.
- The broader context that reaches beyond the particular phenomenon is what is meant by **intervening conditions** which enable or preclude the subject from employing certain strategies. The conditions may be on different levels of the *conditional matrix*, involving e.g. the individuals’ biography or culture (see Section 3.3.3c).

### ***Theoretical Critique of the Paradigm Model***

While Glaser condemned the paradigm model as “*forcing*” the data to fit some pre-existing conception (see Kelle, 2007, for a discussion), others see its role more pragmatically: The model elements are not meant as placeholders that always need to be filled, but as a checklist to aid the researcher in interpreting the data and identify possibly missing concepts and/or data (Przyborski & Wohlrab-Sahr, 2014, p. 202). I follow the pragmatic line of reasoning, which is the only reasonable thing to do, given Strauss & Corbin’s imprecise and handwavy descriptions:

- They are not clear as to what kind of *causes* they refer to. On the one hand, the researcher should pay attention to participants using words such as ‘*because*’ or ‘*due to*’ which—without consulting other data sources—point to *perceived* causes; on the other hand, the researcher may also systematically look for events preceding some phenomenon which leans more towards *actual* causes (Strauss & Corbin, 1990, p. 101).
- With regards to the *strategies*, they propose to also look for missing or failed actions which “*should*” be “*ordinarily*” done in some situation (*ibid.*, p. 104). They are, however, not explicit as to the baseline of such a comparison, e.g., other analyzed situations, the subject’s own report on the matter, or even the researcher’s own experience (which would be similar to Searle’s argument legitimizing his linguistic characterizations, see page 113). The only given advice on how to identify strategies is to look for “*action oriented verbs or participles*” (*ibid.*, p. 105) which again pertains to interviews and not to observations.
- The same issue continues for *consequences*. Not only should the researcher look for the consequences of a “*failure to take action*”, but also for “*potential*” consequences and those happening “*in the future*” (*ibid.*, p. 106).

### ***Practical Application of the Paradigm Model***

Although a clear epistemological stance may not be Strauss & Corbin’s strong point (see also Section 3.3.4), their proposed procedure of how to use the paradigm model adds *structure* to the whole coding process which alternates between the exemplar level and the concept level:

1. The preparation for axial coding happens on the exemplar level, meaning that for concrete events, the researcher asks questions like *what* is the subject dealing with (phenomenon), *how* are they dealing with it (strategies), *why* is this happening (causal conditions), *why* is the subject behaving this way (intervening conditions), and *what* is the outcome of their actions (consequences). Strauss & Corbin (*ibid.*, p. 77) call this to “*open up the data*”.
2. Axial coding proper then begins on the concept level with considering one concept as the *phenomenon* and proposing a hypothetical connection to another concept, such as its *cause* or *strategy* (*ibid.*, pp. 107–108).

3. Such a hypothetical concept-level connection is then checked against all exemplars of the involved concepts. Some instances will probably confirm the hypothesis, others will point to not yet considered differences in the data (*ibid.*, pp. 108–109).
4. To account for such differences, the researcher may introduce new properties to the involved concepts. Not all properties that can be analytically distinguished during open coding will play a role in the paradigm model. It is up to the researcher to identify those properties that constitute the relevant *context* for the subjects' behavior (*ibid.*, p. 109).
5. With an established connection between a *phenomenon*-concept and another one, the researcher may look for differences on the level of properties to further qualify the relationship (*ibid.*, pp. 110–111).

Strauss & Corbin (*ibid.*, pp. 112–113) also emphasize that a particular strand of inquiry may start from something that strikes the researcher as a *strategy* or a *consequence* which may be tracked down through the data to fill in the other places of the paradigm model. The above mentioned 'checklist' property of the paradigm model can come into effect in any of these steps: The researcher may notice that concrete information on an event's *causes* or *consequences* is yet missing; or that a whole *phenomenon*-concept is yet lacking a *strategy*-concept; or that, given their similar *causal conditions*, two different concepts are better to be understood as variants of a more general one, etc.

A 'rich' grounded theory explicitly considers changes over time. The researcher basically has two options to capture these with the paradigm model: First, with a single paradigm instance that considers *processual* strategies that deal with a phenomenon over time (*ibid.*, p. 104); or second, with many paradigm instances connected end-to-end where the consequences of one are part of the causal conditions of the next (*ibid.*, p. 106). Either way, the overall purpose of axial coding is to develop *concepts* to *categories*, which means to systematically consider their causal and intervening conditions, employed strategies, and consequences through means of the paradigm model (Strauss & Corbin, 1990, pp. 78 & 97; Corbin & Strauss, 1990, pp. 7–8, see also Figure 3.3 for an excursus on common misconceptions).

### 3.3.3 c) Conditional Matrix

Strauss & Corbin (1990, pp. 161–171) propose the *conditional matrix* as an analytical tool to systematically consider different levels of social contexts in which the phenomenon of interest is embedded, such as the individual, a small group, a company, or a larger community.<sup>5</sup> In practical terms, they propose to subdivide the *causes*, *intervening conditions*, and *consequences* from the paradigm model and to trace actions and their causes and consequences through the different levels.

The example they give for illustrating this idea (*ibid.*, pp. 168–171) reads a bit like an elaborate joke that builds on ever more general explanations why something particular happened. It follows one incident from the "*action level*" over a total of seven levels such as "*organizational level*" and "*community level*" to the "*nation level*": A physician cannot proceed with an examination because there are no plastic gloves to be found in her size, because her unit and any other are in short supply, as are all other hospitals, because a perceived AIDS epidemic led to new national guidelines requiring plastic gloves for all contacts with patients.

To me, the point of the conditional matrix as a GTM-tool appears to be to make the researcher aware that all phenomena are embedded in different levels nested contexts, but

<sup>5</sup>Strauss & Corbin (1990, Fig. 10.1) depict these levels as concentric circles which might be confusing for the mathematically inclined who think of a *matrix* as something rectangular or table-like (I did). In general, however, a *matrix* is, according to Merriam Webster, "*something within or from which something else originates, develops, or takes form*".

Przyborski & Wohlrab-Sahr (2014, p. 191) criticize a common reduction of Grounded Theory ideas to mere classification and sorting of data. Over the last years, I spoke to researchers on multiple conferences and read articles of authors who genuinely thought that axial coding is about defining a taxonomy, with sub- and super-types for concepts. Although there is nothing wrong with a good taxonomy and such a thing might even be part of open coding, it is not what *axial coding* is about and I want to make my contribution in setting the record straight here.

Strauss & Corbin (1990) often use the term “*subcategory*”, which to a technical person may sound like a subclass in object-oriented programming, but is *not* what the authors had in mind. Upon introducing *categories* in the context of open coding, they say “*at this point any proposed relationships [between concepts] are still considered provisional*” and point to the chapter on axial coding (p. 65). The examples that follow before axial coding is introduced may indeed reinforce the ‘subclass’ notion in the reader: *types of work* is a subcategory of *food orchestrator* (pp. 67 & 71); *body strength, shaping, monitoring, training, and movement* are all subcategories of *building up the body* (p. 85). These are, however, mere “*potential subcategories*” (p. 87).

What a ‘proper’ subcategory might be is not made clear until the chapter on axial coding, where finally a definition is provided. It clarifies, that for Strauss & Corbin, “*subcategory*” refers to the elements of the paradigm model:

“ In axial coding our focus is on specifying a category (*phenomenon*) in terms of the conditions that give rise to it; the *context* (its specific set of properties) in which it is embedded; the action/interactional *strategies* by which it is handled, managed, carried out; and the *consequences* of those strategies. These specifying features of a category give it precision, thus we refer to them as *subcategories*. In essence, they too are categories, but because we relate them to a category in some form of relationships, we add the prefix “sub.” (p. 97)

I interpret the talk of ‘potential’ or ‘provisional’ subcategories as referring to those which have not yet undergone an examination in the light of the paradigm model, i.e., which are not yet labeled as context, action, consequence, etc. As a concrete example to illustrate a subcategory as defined above, they give the consequence of *pain relief* as a subcategory of *pain* after some action has been taken (pp. 98 & 106). Differentiating different types of *pain*, such as *high/low intensity pain* and *back/lower leg pain*, is achieved through properties and is the matter of open coding.

**Figure 3.3:** What is axial coding? Or: The mystery of the “subcategory”.

not all levels are relevant for a particular study. The conditional matrix basically amounts to asking *Why?* until the answers are no longer relevant for the studied phenomenon. The levels mentioned by Strauss & Corbin (1990, p. 162) are but one generic way of looking at things and it is up to the researcher to identify the relevant levels for her particular research.

### 3.3.3 d) Selective Coding

During selective coding, the researcher more and more assumes the role of an author (Böhm, 2004, p. 273). The idea is to identify what the research ‘is about’, the *core category*. This might be one of the categories that were developed during axial coding, or some previously unnamed aspect that is central to multiple categories (*ibid.*, pp. 273–274). This may even result in a shifted focus from what was originally thought of as central (*ibid.*, p. 273).

Strauss & Corbin (1990, pp. 119–125) propose to write a narrative that condenses the most important aspects into a “*story*”, first descriptively, then analytically by giving the story itself a conceptual name, identifying its properties, and arranging the other categories around it according to the paradigm model. In addition to coming closer to (a written form of) a theory, this process is meant to guide the researcher in systematically identifying gaps in the concepts and/or data. Böhm (2004, p. 274) cites a GT study in the wake of the Chernobyl disaster which

illustrates how this can work: The core category had the two properties of *subjective age* (*young* vs. *old*) and *perceived threat* (*no* vs. *severe*), which, after systematic consideration of all four combinations, led to the discovery of the connection between *feeling young* and *high perceived threat* because the combinations *young/no threat* and *old/severe threat* did not occur.

Robson (2002, p. 495) and Przyborski & Wohlrab-Sahr (2014, p. 211) note that focusing on a core category and filling the gaps around it also disregards and excludes other phenomena from further analysis—a point which Strauss & Corbin (1990, pp. 141–142) only make implicitly.

### 3.3.3 e) On Developing Concepts: Theoretical Sensitivity

Strauss & Corbin (1990, pp. 131–132) emphasize (and even rely on) insights to just happen along the research process: “After being immersed in the data for months one can’t help but note differences or the emerging patterns”. For all coding activities in a GTM study, **theoretical sensitivity** (*ibid.*, Ch. 3) is an important property on the part of the researcher: She needs to be able to ask good questions to “open up the data” to detect meaningful differences in the phenomena of interest, for which personal and professional experience can be a source as well as familiarity with relevant literature (*ibid.*, pp. 41–43, 77–79). But where does such a sensitivity have its place in a scientific process? Although Strauss & Corbin (*ibid.*, pp. 131, 148) exclusively speak of switching between *inductive* and *deductive* thinking, Hildenbrand (2004, p. 18) observes that the idea of logical *abduction* can be read “between the lines”. So what is abduction, now?

Considered from a logical perspective, GTM (like most qualitative research methods) uses all of *induction*, *deduction*, and *abduction* to reason about the relationship of concrete *instances* and abstract *concepts* (also called *types* or *rules* elsewhere, Reichertz, 2004). Induction and deduction are well-known logical notions: **Deduction** is the subordination of an instance (e.g., *someone robbed the medicine chest*) under a known type (*all medicine chest burglars are drug addicts*) from which properties of the instance follow (*this burglar is a drug addict*)—if the general rule is true, the result of its application is also valid (*ibid.*, pp. 160–161). **Induction** is the inference that an individual case (e.g., *there are particular clues on a crime scene*) belongs to a known type (*crime scenes of Mr. Jones have certain properties*) from which properties of the individual case follow (*Mr. Jones is responsible for this crime scene*)—which is only a probable but not an obligatory form of inference (*ibid.*, p. 161).

Both deduction and induction are based on associating observed instances to already *existing* types. In this regard, **abduction**—popularized in the early 20th century by philosopher Charles Sanders Peirce—is different in that it happens when the researcher encounters a particular instance for which no suitable type exists yet and she therefore invents (or discovers) a new one (*ibid.*, p. 161). Abduction is about looking for “*meaning-creating rules*” which remove “*what is surprising about the facts*”; it is arguably a first step of scientific discovery after which predictions can be derived *deductively* and facts can be searched to make the *induction* (*ibid.*, p. 163). To complete the example above (unfortunately, Reichertz does not provide one): For someone extremely naive, there might be no explanation for the surprising fact of *some items from the medicine chest went missing* until a (new) concept such as the following is introduced to them: *theft by a drug addict for whom the contents are desirable beyond their original purpose*. But to come back to the original point of how theoretical sensitivity comes into play: It enables the researcher to perform the logical abduction in a way that leads to relevant concepts which then can be used in the inductive and deductive reasoning which Strauss & Corbin describe, e.g., in the context of axial coding (see Section 3.3.3b).

The researcher introduces new explaining (or ‘theoretical’) concepts through abduction, and then refines them and ensures their consistency through the process of **constant comparison**

(see Section 3.3.3a) which goes back and forth between inductive and deductive thinking and takes place on the concrete phenomenon level as well as the concept level (Przyborski & Wohlrab-Sahr, 2014, p. 204). Glaser & Strauss (1967, pp. 55–58) characterize the underlying idea behind developing concepts with the GTM as simultaneously looking for relevant differences between minimally similar contexts (e.g., to get a better understanding of the variety of a phenomenon) and looking for similarities between maximally different contexts. Przyborski & Wohlrab-Sahr (2014, p. 205) argue that this way of making comparisons basically supplants theory verification/falsification done in theory-testing research approaches.

### 3.3.3 f) Writing Memos

Strauss & Corbin (1990, pp. 198, 203) characterize **memos** as “*the written forms of our abstract thinking about data*”; it is not about concrete incidents but about concepts. They distinguish three types of notes which may be intermingled in any particular memo: *Code notes* contain the ‘technical’ descriptions of concepts, their properties, and relationships to other concepts; *theoretical notes* are reflections on the current concepts, potential further properties, ideas from reading in literature, etc.; while in *operational notes*, the researcher considers on how to proceed in data collection and analysis, which particular questions to address next (*ibid.*, pp. 197–198, 205–208).

Memos are the immediate product of any analytic activity in a GTM study; writing memos is an integral part of whole research process from start to end (*ibid.*, p. 198). Whenever the researcher is hit by an idea, both Glaser & Strauss (1967, p. 107) and Strauss & Corbin (1990, p. 201) recommend to stop whatever she is doing and to take the time to write it down, e.g., when a new observation does not fit the existing categories (this is the moment a logical *abduction* happens, see previous section). Charmaz (2006, p. 72) calls memo-writing a *conversation with yourself*, which produces new ideas and insights, and helps getting questions clearer.

Many of the practical considerations of Strauss & Corbin (1990, pp. 199–203) appear a bit outdated and echo from an analog era of “*color coded cards*” and “*putting type-written pages into binders*”, such as including dates and document references, underline concept names, and keep multiple copies. Ultimately, the researcher should develop their own style, which may include “*computer programs*” (*ibid.*, p. 200). Nowadays, specialized software such as ATLAS.ti<sup>6</sup> supports direct annotation of a variety of different data types (text documents, images, audio, and video) and takes care of the house-keeping tasks of maintaining references between data and concepts, as well as between concepts.

### 3.3.4 Different GTM Versions

Considering the six core questions along the answers to which qualitative research approaches differ (see Section 3.2.3), the three GTM versions—“classic” Glaserian and Straussian GTM on the one hand, Charmaz’s constructivist GTM on the other—all follow a general approach of developing concepts from and grounded in data and thus more or less agree on the **Method** (*How should we study the world?*). All are relatively open regarding the researcher’s personal involvement and the questions of what is worth knowing and which questions one should ask (**Involvement**, **Philosophy**, and **Discipline**), meaning that these are areas to be filled for any concrete study by the respective researchers.

Most notably, however, classic and constructivist GTM strongly *disagree* on the matters of **Ontology** (*What do we believe about the nature of reality?*) and **Epistemology** (*How do*

---

<sup>6</sup>Introduced by Muhr (1994), it was originally meant for *text interpretation* (hence, “ti”), and has been extended since then. Product homepage: <https://atlasti.com/>.

*we know what we know?*), the effects of which I discuss in this section. Note this is not an extensive review of the different schools of thought; Stol et al. (2016) discuss the differences and implications for software engineering research.

### 3.3.4 a) Classic Grounded Theory

Although Glaser and Strauss do not discuss their epistemology explicitly in their books, arguably, their stance is that of *absolute ontology* (Charmaz, 2006, pp. 7–10), i.e., the assumption that there is one single, verifiable reality governed by natural laws. Not by accident do they speak of the “*discovery of theory*” (Glaser & Strauss, 1967, p. 1) where concepts “*emerge*” (Strauss & Corbin, 1990, pp. 131–132) through the application of the coding procedures as if it is already ‘out there’, waiting to be found by an objective researcher who should maintain an “*analytical distance*”, recognize and avoid bias, and obtain valid and reliable data (*ibid.*, p. 18)—all of which are hallmarks of *epistemological positivism* (see page 115). Time and time again it appears as if Strauss & Corbin treat observations by the researcher and interviewee reports (whether on actual events, or simply as what “*someone [...] ordinarily would do*”, *ibid.*, p. 104) as equivalent when it comes to studying social reality (see my list of critique on page 120).

A related issue is how the results of a GTM study should be evaluated. Strauss & Corbin (*ibid.*, Ch. 14) mention three areas: The validity, reliability, and credibility of the *data*, the adequacy of the research *process*, and the *empirical grounding* of the findings.<sup>7</sup> They do not elaborate on the first area, but their criteria for the other two boil down to asking the researcher to provide information on how the GTM coding procedures were applied: “*The criteria are meant as guidelines*”, i.e., departing from them is allowable if it is warranted and explained; and “*indicate what your procedural operations were*” (*ibid.*, pp. 257–258).

### 3.3.4 b) Constructivist Grounded Theory

This contrasts with Charmaz’ Constructivist Grounded Theory. She argues that basic GTM elements “*such as coding, memo-writing, and sampling for theory development, and comparative methods are, in many ways, neutral*” (Charmaz, 2006, p. 9) in that they do not conflict with the two central ontological and epistemological points she stresses.

#### ***Multiple Realities and Solipsism***

First, all humans (that is, subjects *and* researchers) are not objective, but “*make assumptions about what is real, possess stocks of knowledge, occupy social statuses, and pursue purposes that influence their respective views and actions*” (*ibid.*, p. 15). Consequentially, the researcher needs to be aware that all data is constructed by people: Just as documents made by the subjects are influenced by their social, cultural, and organizational backgrounds, all records created by the researcher (e.g., interview or fieldnotes) are not simply ‘facts’, but are also a product of directed attention and framing (*ibid.*, pp. 16, 67). Charmaz (*ibid.*, pp. 18–19) proposes to collect enough background information on the involved settings, processes, and persons, and to make sure to capture their multiple views.

Her second point is that the researcher cannot look into the subjects’ minds, but can only try to enter their social settings and attempt careful interpretations, meaning she should not assume that the subjects share the same tacit assumptions as herself (*ibid.*, pp. 14, 19). Two

<sup>7</sup>Early in their book, Strauss & Corbin (1990, p. 23) also propose that a good grounded theory should “*fit*” the particular area of everyday reality, it should be comprehensible and make sense to the studied subjects, should be abstract enough and include variation to be applicable to related contexts, and provide control. However, they do not explain how their GTM variant addresses these concerns.

concrete data collection guidelines that go beyond what Strauss & Corbin proposed are (a) to address what the *subjects* consider interesting or problematic, and (b) to pay attention to their particular *use of language* (Charmaz, 2006, p. 22).

### **Coding Procedures**

Charmaz (*ibid.*, pp. 42–60) proposes two original types of coding. During **initial coding**, the researcher should stay close to the data and move quickly to find simple and precise codes. This is similar to early-stage Straussian open coding, but Charmaz adds constructivistic rationales for two coding techniques: Word-by-word coding may be helpful as it forces the researcher to dig into the *meaning* of the subjects' chosen words (*ibid.*, p. 50), whereas line-by-line coding may help uncovering tacit assumptions as it “*frees you from becoming so immersed in your respondents' worldviews that you accept them without question*” (*ibid.*, p. 51).<sup>8</sup> **Focused coding** then covers larger amounts of data more selectively and addresses larger segments of data at once (*ibid.*, p. 57). It combines the aspect of more abstract concepts (or categories) of later-stage Straussian open coding and the excluding property of selective coding.

Charmaz (*ibid.*, pp. 62–63, 66) also critically comments on Straussian *axial coding* (involving the paradigm model, see 3.3.3b) and Glaserian *theoretical coding* (who proposed 18 “coding families”, see Glaser (1978), all of which serve a similar purpose to the paradigm model): Both may be helpful in the analysis by allowing the researcher to code for the causes, contexts, and consequences of subjects' actions, but it may also limit the researcher and “*lend an aura of objectivity to an analysis*” it does not deserve.

### **3.3.5 Discussion of GTM as a Qualitative Research Approach**

To round off my discussion of the Grounded Theory Methodology, I reconsider the eight general quality criteria (Section 3.3.5a) and the nine common traits of qualitative research (Section 3.3.5b) to determine which aspects are covered by the method itself and which fall onto the researcher to take extra steps.

#### **3.3.5 a) Meeting the Quality Criteria**

Charmaz (2006, pp. 181–183) discusses a number of criteria that go beyond Strauss & Corbin's concerns of data quality and accurate coding process description. She emphasizes the importance of the particular research discipline and the purpose of the particular study, and lists four areas as “*ideas*” for what to look out for, each neatly summarized as a single question by Stol et al. (2016, Table 1). All of Charmaz's concerns can be found among the quality criteria discussed in Section 3.2.4:

- **Credibility:** *Is there sufficient data to merit claims?*  
(→ **Rich Rigor**, **Credibility** through *thick description*, **Meaningful Coherence**)
- **Originality:** *Do the categories offer new insights?*  
(→ **Worthy Topic**, **Significant Contribution** in the sense of *theoretical significance*)
- **Resonance:** *Does the theory make sense to participants?*  
(→ **Significant Contribution** in the sense of *practical significance*, **Credibility** through *multivocality* and *member reflection*, **Resonance** in terms of *transferability*)
- **Usefulness:** *Does the theory offer useful interpretations?*  
(→ **Significant Contribution** in terms of *practical, theoretical, and heuristical significance*)

<sup>8</sup>I imagine the last point to be an issue especially when the researcher identifies herself with the subjects and/or shares a common background. As Przyborski & Wohlrab-Sahr (2014, p. 16) point out, making one's understanding explicit gets more difficult the closer researcher and subject are biographically and culturally.

Overall, Charmaz’s criteria address six out of eight criteria proposed by Tracy (2010), which leaves two areas to the researcher: **Sincerity** and **Ethics**. I will later discuss my own research in the light of Tracy’s quality criteria, which are more general than Charmaz’s.

### 3.3.5 b) Filling the Common Traits

Not all traits of qualitative research are explicitly addressed by elements of the Grounded Theory Methodology. In Table 3.2, I summarized what Straussian and Charmaz’s GTM have to offer with regards to the nine common traits of qualitative research I introduced in Section 3.2.2.

Trait	Strauss & Corbin (1990)	Charmaz (2006)
1. NATURALISTIC INQUIRY IN EVERYDAY SITUATIONS	concerned with concrete experiences from everyday reality [p. 23]	about events and experiences in the participants’ lives [pp. 2–3]
2. OPENNESS & EMERGENT RESEARCH DESIGN	must not follow initial ideas too rigidly, need to adjust [pp. 180–183]; <i>open coding</i> (3.3.3a)	flexible data collection and analysis guidelines, follow interesting data [pp. 2–3]; <i>initial coding</i> (page 126)
3. PURPOSEFUL SAMPLING	<i>theoretical sampling</i> (3.3.2)	similar to Straussian GTM [pp. 99–108]
4. DETAILED, THICK, AND DIVERSE DATA	“ <i>diverse data</i> ” [pp. 23 & 180], collected through means of <i>theoretical sampling</i> (3.3.2)	capture details and multiple perspectives (page 125)
5. IMPORTANCE OF THE RESEARCHER	<i>memo writing</i> (3.3.3f); <i>theoretical sensitivity</i> (3.3.3e)	respect subjects, establish rapport [p. 19]; <i>memo writing</i> (3.3.3f); awareness: research data is constructed, too (page 125);
6. EMPATHIC NEUTRALITY	implicitly: considering <i>intervening conditions</i> during <i>axial coding</i> (3.3.3b)	consider tacit assumptions, “ <i>careful interpretation</i> ” (page 125)
7. CONTEXT MATTERS	<i>conditional matrix</i> (3.3.3c)	capture background information on people, settings, situations (page 125)
8. HOLISTIC PERSPECTIVE	<i>conditional matrix</i> (3.3.3c); <i>selective coding</i> (3.3.3d)	<i>focused coding</i> (page 126)
9. DISCOVERY OF THEORIES	switching between coding procedures & between abduction, induction, and deduction (3.3.3e)	similar to Straussian GTM [pp. 103–104]

**Table 3.2:** Mapping of common qualitative research traits (Table 3.1) on elements of GTM versions with references to my discussion or to the respective books (in square brackets).

Aspects like OPENNESS & EMERGENT RESEARCH DESIGN and PURPOSEFUL SAMPLING are clearly emphasized in their GTM versions. Others, like DETAILED, THICK, AND DIVERSE DATA, are, at least for Straussian GTM, more of an assumption the rest of the methodology builds upon, but not directly addressed. Rather, pieces of advice regarding data collection—such as to adjust one’s focus while interviewing—seem primarily motivated by the prospect to “*save time later*” since one “*won’t need to reinterview or observe again in order to retrieve important missing data*” (Strauss & Corbin, 1990, p. 183).

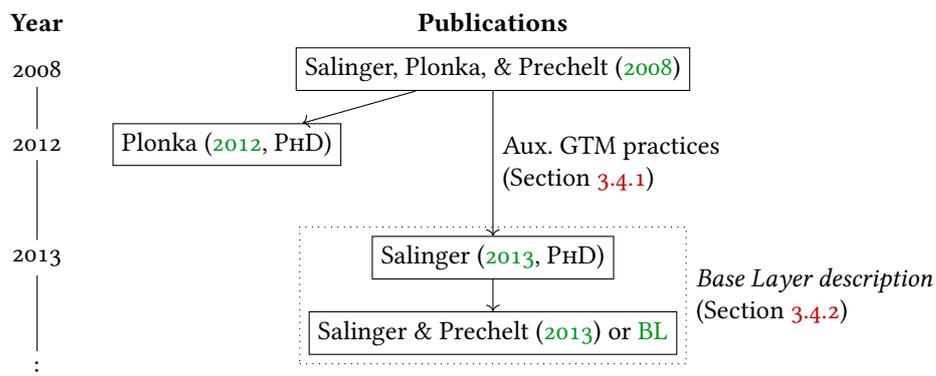
Furthermore, although all GTM forms mention interviews as well as observations as possible data sources, many *concrete* coding guidelines pertain to *text* or a “*document*”. For example, Strauss & Corbin (1990, pp. 72–73, 81–84) list the following “*variations on ways of doing open coding*”: (1) line-by-line, phrase-by-phrase, or single words, (2) sentence or paragraph, or (3) entire document. Of course, Strauss & Corbin repeatedly mention that such a document may be a “*fieldnote*”. But as Charmaz observed, all data is constructed by people (see page 125). So, what am I learning about the subjects’ social reality from coding individual words or phrases of my own fieldnotes? Charmaz (2006, p. 53) acknowledges the difference between transcribed interviews with the subjects’ accounts, fieldnotes written by the researcher, and general observations which “*may not be amendable to line-by-line coding*”. I discuss constructive solutions from my research group for the resulting difficulties in Section 3.4.

Two related aspects are how to actually engage in NATURALISTIC INQUIRY IN EVERYDAY SITUATIONS and what the IMPORTANCE OF THE RESEARCHER entails. I consider both aspects to be highly specific for each particular study, so I understand why neither Strauss & Corbin (1990) nor Charmaz (2006) discuss these points beyond the bits I mentioned in Table 3.2. I therefore consider it as my responsibility to deal with the aspects myself and add my own considerations concerning how to collect data and my role as a researcher in Chapter 4.

### 3.4 The Base Layer for Pair Programming Research

My research group AGSE<sup>9</sup> started studying pair programming in 2004 with the overall research goal to identify and categorize behavioral patterns to inform practitioners. They started with recordings of controlled setups with recruited students to compare pair and solo work as well as different types of tasks. In early 2007 followed the first recording of an industrial PP session.

AGSE members Salinger, Plonka, & Prechelt (2008) describe a qualitative research approach that extends the Grounded Theory Methodology by four auxiliary practices to deal with the peculiarities of the data and the phenomenon under investigation. Plonka (2012, Sec. 4.8) decided to use both quantitative and different qualitative data analysis approaches. I discussed her work in Section 2.3.4 on pages 73, 81, and 84. Salinger (2013), in contrast, continued with the GTM-based approach and developed the *base layer*, a framework for qualitative research on pair programming. Salinger & Prechelt (2013) then published a translated and shortened description of the base layer as a book. For brevity, I refer to that book as “BL”.



**Figure 3.4:** Timeline of our early PP research. Solid arrows indicate reuse of ideas. Figure A.1 continues this chart to the present day.

<sup>9</sup>German acronym for “Arbeitsgruppe Software Engineering” (Software Engineering Work Group).

### 3.4.1 Auxiliary Practices for the Grounded Theory Methodology

Salinger et al. (2008, pp. 13–14) describe their experience with applying Straussian GTM practices on video recordings of pair programming sessions. Open coding (see Section 3.3.3a) led to almost 200 different concepts within a few days, so they reflected on what has happened and identified a number of problems:

- (1) It appears too late to **filter** out some (aspects of) phenomena only in selective coding (i.e., after substantial open and axial coding, see Section 3.3.3d). Coding all verbal and facial expressions, gestures, computer interactions, and so on led to an unmanageable amount of codings. Too many concepts make it difficult to see the forest for the trees; Salinger et al. feared to miss on relevant phenomena because of this.
- (2) Straussian GTM is not explicit about its **epistemology** and the role of the researcher (see Sections 3.3.4a and 3.3.5b). Neither were Salinger et al. during open coding and thus mixed descriptive codes and interpretative ones, such as *uses documentation* vs. *gains knowledge of detail* for the same PP situation.

The nature of the material (high-detail recordings of two software developers working on a non-trivial task) added further complications which Straussian practices alone could not solve:

- (3) **Data segmentation:** While (transcripts of) interviews can be relatively structured (e.g., by lines or by question), an hour-long pair programming session is not. Codes with different granularity such as coarse *handle problem* and fine *test defect fix* were a consequence.
- (4) **Housekeeping:** There are many different aspects to a PP session. The resulting multitude of concepts needed to be grouped to keep it manageable.

To cope with these problems, Salinger et al. (*ibid.*, pp. 14–18) propose four practices: Define a *Perspective on the Data*, establish *Concept Name Syntax Rules*, define an *Analysis Results Metamodel*, and *Code in Pairs*.

#### 3.4.1 a) Perspective on the Data

To avoid “*drowning*” in concepts and potentially miss relevant phenomena in the data (problem 1) as well as to establish the role and position of the researcher (problem 2), Salinger et al. (2008) propose to formulate a perspective by answering these three questions (see also Salinger, 2013, pp. 109–110):

1. “*In which respects do you expect the data to provide insight?*”  
The answer to this is supposed to act as a filter telling the researcher when to stop.
2. “*What kinds of phenomena do the researchers allow themselves to identify in the data?*”  
This determines the researcher’s epistemological standpoint.
3. “*What type of result do you want the analysis to bring forth?*”  
Aiming for a coding scheme is cheaper than a full theory.

As Salinger (*ibid.*, p. 110) emphasizes, such a perspective is not meant to restrict the researcher, but to direct her attention: It should be regularly examined and, if need be, modified. Defining such a perspective has also been adopted outside our research group, e.g., by Jones & Fleming (2013, Sec. 1, see also discussion on page 83).

#### 3.4.1 b) Concept Name Syntax Rules

In contrast to GTM concept names which should be “*logically related to the data*” they represent and “*graphic enough*” to quickly remind the researcher (Strauss & Corbin, 1990, p. 67), but have no fixed format, Salinger et al. propose to introduce a syntax for concept names. This helps with problems (3) and (4).

### 3.4.1 c) Analysis Results Metamodel

To clarify what the *things* the GT researcher creates along their analysis are (and how they map to the terms their respective analysis software uses), Salinger et al. propose to formalize the structure of the analysis elements as a meta-model. As such, thinking about one's meta-model does not address any of the problems (1) to (4) directly, but helps to clarify thinking along the way.

### 3.4.1 d) Pair Coding

To make sure the other practices have their intended effect, Salinger et al. propose to perform GTM coding in pairs with another researcher. By doing so, concept definitions also become more precise. Overall, this practice helps addressing all four problems (1) to (4).

Like the *Perspective on the Data*, the *Pair Coding* practice as proposed by Salinger et al. has been explicitly adopted by researchers outside our group and even outside the software engineering community, e.g., by Sobo et al. (2011).

## 3.4.2 The Base Layer in a Nutshell

The *base layer* is not meant to be used as a mere coding scheme for annotating PP sessions. Rather, it is a *research framework*, a first step that aids researchers in identifying relevant phenomena and setting directions for a long qualitative research process to eventually understand pair programming as a whole. Below, I explain its origin, rationale, and most important elements.

### 3.4.2 a) Layered Research Approach: Different Perspectives on the Data

Salinger (2013) applied practices from Straussian GTM (mostly open coding) and the four auxiliary practices discussed above. Throughout Salinger's PhD research, the impression that pair programming is a complex phenomenon did not vanish. While Plonka (2012, Ch. 5–9) dealt with this by addressing PP from a number of different angles, Salinger (2013) decided to lay the groundwork for a long-term research instead, thereby addressing the problem of qualitative PP research being not open for further work (see my discussion on page 87). Over time, driven by empirical observations and reflection on the research process, he arrived at the following *Perspective on the Data* (*ibid.*, pp. 123–125):

1. **Area of Interest:** Understand what activities make up the pair programming process. Conceptualize the basic activities of the individual pair members towards an extensional definition of *activity* in the context of PP, i.e., to answer the question *What is it that developers do during PP?* by enumerating all types. These concepts should serve as a starting point for more specialized studies that focus on theory-building.
2. **Epistemological Stance:** The *base concepts* take a predominantly behavioristic perspective that is based on directly observable behavior. In particular, they are not concerned with cognitive processes. Furthermore, there are no evaluations with regards to technical progress in the PP sessions.
3. **Result Type:** The technical outcome is not a final coding scheme, but a set of concepts that is to be understood as a framework to help researchers find further research focuses. The individual concepts have only few *properties* (in the GTM sense, see Section 3.3.3a); identifying relevant ones is left for further work.

Salinger describes the relationship between his *base concept set* and further research with a *layer* metaphor: The *base layer* provides concepts to address all PP activities in a fundamental

sense. Further research may then layer on top and focus on some type of *base activity* and develop more properties for addressing more subtle differences without having to reinvent the wheel. Alternatively, specialized studies may shift their attention to a coarser granularity, e.g., addressing multiple base activities at once, possibly even consider the pair as a whole, again adding another layer of concepts. At the time of his writing, of course, no such additional layer existed, so nobody could know how exactly this layering would work in practice.

### 3.4.2 b) Seven Key Decisions

Salinger (2013) made a number of observations that led to seven key decisions shaping the set of base concepts (see also BL, Sec. 2.3). I explain them below with instances from my related work discussion for illustrating the difference they make.

**(1) Verbal Communication over Computer Interaction** There are fundamental differences between a pair’s verbal communication and their human-computer interaction, especially with respect to their complexity and variety. The base layer therefore distinguishes between human-human interaction or HHI activities which are directed at the partner (*social action*, see Section 3.2), and HCI/HEI activities (human-computer and human-environment interaction). There are about 60 different HHI activity concepts and only 8 HCI/HEI concepts.

This contrasts with Flor (1998) who also studied pair programmers but generalized both the computer screen and the pair’s communication to “*medium*” (see discussion on page 80).

**(2) Capture Intentions** The base concepts capture the *illocutionary act* of the pair programmers’ activities, that is, utterances are characterized as to what the speaker intends to do with them rather than their surface structure (see also page 112).

This is unlike Bryant (2004) who considered utterances to be questions when they looked like questions (see discussion on page 74).

**(3) Emerging Segmentation of Data** There is no natural segmentation of a pair’s discourse into “sentences”, as e.g. Bryant et al. (2008) suggest (see discussion on page 72). The base concepts have no strict built-in granularity; rather, the granularity of coded base activities relies on (tacit) communication knowledge (see Section 3.2.1) and an understanding of software development on part of the researcher. The same sequence of words, for example, when uttered in different contexts, may be considered *one explanation* activity or as multiple *partial explanation* activities with interspersed instructional activities what to do.

**(4) Behavioristic Ideal** Since understanding any social action, be it as a pair member or as a researcher, requires some interpretation (again, see Section 3.2.1), a purely behavioristic approach cannot work. However, there should be *some* directly observable evidence that makes one’s interpretation reasonable.

An example by Plonka (2012, pp. 185–186) illustrates the different perspectives. A less experienced pair member says “*Ok, so this is done now, so we can move on to the next bit*” to which the more senior partner replies “*We could also test that first*” (translation by Plonka). She characterizes this reaction as “*nudging*”, as a “*strategy [...] to provide a subtle learning opportunity*”. It is entirely possible that the experienced pair member did exactly that. Salinger, however, would characterize the situation closer to what is directly observable, e.g., as a rejected proposal (coded as *challenge\_step*, see below) or a rejected assessment (*disagree\_completion*), depending on the context. This also requires *some* interpretation, of course, but is less of a stretch. To make all of this more palpable, I searched for the section in the recordings Plonka referred to (she mentions neither session nor timestamp) and discuss the whole 100-second episode in Example 3.1 on page 136.

(5) **Model Discourse World, not Activity World** There is a difference between talking about some X (such as a design proposal) and X itself (e.g., putting the proposal into action by changing source code). The base layer is mostly concerned with discourse objects, that is the things the developers' conversation is about. A *step*, for example, is something the developers refer to in their conversation as an atomic unit of work, which is independent of whether they actually perform it *en bloc* or even whether they perform it at all.

Xu et al. (2005), who counted the number of different concepts experienced and novice programmers maintain in their PP session (see discussion on page 77), did not make such a distinction: Creating and deleting classes or members in the source code was in a sense equivalent to discussing such changes.

(6) **Model Dialog Episodes** Salinger noted a small number of types of recurring discourse objects and typical lifecycle of being introduced, evaluated, and/or modified (and potentially put into action). The concept name syntax rule (see Section 3.4.1b) of the base layer reflects this and allows for sequences of codings that make episodes in the dialog visible. The general format for all base concept names is <verb>\_<object>, leading to codes such as *propose\_design* and *amend\_design*. Overall, there are 16 discourse objects that mostly fall into two large categories, each with 1 to 5 verbs, and a total of 13 different verbs.

(7) **Reflect Relevant Phenomena** The base concepts all address phenomena that are potentially relevant for practitioners. Some of the discourse objects even reflect software development terminology, such as *design* or *requirement*. Individual base concepts do not necessarily capture pair programmer behavior that is very common (e.g., *disagree\_strategy* was seen only once). Rather, they capture relevant differences such as between the somewhat similar concepts *standard of knowledge* and *gap in knowledge* (see discussion below). This, too, contrasts with Bryant (2004) whose coding scheme is *exhaustive* in that all utterances can be coded, but not necessarily *rich* in that it would capture relevant differences (see page 74).

### 3.4.2 c) The Base Concept Set

The perspective on the data and the seven key decisions led to a fine-grained analysis covering all pair programmer activity on its 'atomic' level. The terminology is as follows:

- A PP session can be broken down to a series of **base activities** which the developers individually perform. These are typically individual utterances (HHI, or human-human interaction) or coherent streams of editing or navigation operations (HCI, or human-computer interaction).
- Each of these is characterized by sometimes two or three, but usually only one **base concept**. The whole set contains 58 HHI base concepts<sup>10</sup> and 8 HCI base concepts. Each base concept represents the primary intention of the developer who performs the base activity. More precisely: All base concept names consist of an object and a verb, where the verb represents the *illocutionary force* and the object part characterizes the *propositional content* (see page 112).
- The **base layer** is the base concept set *plus* the rules for (a) when they apply to a concrete instance and (b) how to segment the continuous stream of reality into *activities*. Additionally, there are (c) guidelines for when and how to include new concepts and properties in the base set and (d) some ideas on how to create additional concept layers (BL, Ch. 22–23).

---

<sup>10</sup>That is one concept less in the BL book than Salinger (2013) originally had: The rare *remember\_source of information* has been subsumed under the more general *explain\_knowledge* (BL, pp. 209–210).

Throughout this thesis, I use the base concepts as a vocabulary to talk about PP processes. However, the full description of all base concepts, their properties, and application rules span over 150 pages in the [BL](#) book, so I will only discuss the parts here that are relevant for my work and occur in my examples multiple times. In this sense, the remainder of this section is like a dictionary and thus rather dull. The next Section [3.4.3](#) illustrates a practical application of the base concepts.

### Discourse Objects

The **object** part of the HHI concepts represents discourse object types. I discuss the most relevant 10 of them, which fall into two large categories: *product- & process-oriented concepts* and *universal concepts*. There are also HCI/HEI concepts, whose object is always *sth* (meaning ‘something’) and all variation is expressed through the accompanying verb as in *examine\_sth*, *write\_sth*, etc.

Roughly speaking, these three categories can be characterized as (1) *decision-making* where the developers discuss what they want to do on a technical, product-oriented side and on their process-level, (2) *knowledge transfer* where the partners talk about their insights, about what they know and do not know, things they suspect or are unsure about, and (3) *execution* where pair programmers primarily interact with their computer.

HHI: Product- and Process-Oriented Objects (“Decision-Making”)	HHI: Universal Objects (“Knowledge Transfer”)	HCI/HEI (“Execution”)
<i>design, requirement, step, strategy, completion, state, todo</i>	<i>finding, hypothesis, standard of knowledge, gap in knowledge, knowledge</i>	<i>sth</i>

**Table 3.3:** Object classification in the base layer

The **product- and process-oriented concepts** are also abbreviated to **P&P concepts**:

- **design**: A property of a worked-on artifact the developers can decide on; could be as local as a variable name or as global as an architectural constraint ([BL](#), Ch. 4).
- **requirement**: A decision affecting the developers from outside their session ([BL](#), Ch. 5).
- **step & strategy**: Consideration on how to proceed, on a tactical and a strategical level, respectively ([BL](#), Ch. 6 & 9). The difference between the two lies in how the developers refer to it (model the discourse world, see key decision #5 on page [132](#)): The same development activity can be treated as something atomic (*step*) or as something consisting of multiple parts (*strategy*) in their discourse.
- **completion & state**: Assessment on how far a *step* or *strategy* has been worked through ([BL](#), Ch. 7 & 10).
- **todo**: Consideration on what to do at some point in future ([BL](#), Ch. 8).

The second category is called **universal concepts** because the developers may refer to both product and process aspects or none of them, including various types of knowledge and meta-knowledge:

- **finding**: A developer's insight that the pair programmers talk about (BL, Ch. 12). Can be a perceived event **P** (such as a finished full-text search), a discovered issue **D** (such as an identified defect), or a thought **T** (such as an idea). All of these can either be catalyzed **c** or uncatalyzed **u** depending on whether or not they are triggered by what the partner just did or said, or what is visible on-screen. Note that the base concepts only refer to the pair's discourse: Some insights may remain completely private in the developers' thoughts.
- **hypothesis**: Expressed uncertainty (BL, Ch. 13). This is unlike all other universal concepts where the pair programmers present the information content as "true". This can be a hard-to-verify hypothesis **HTV** that goes beyond what the pair expects to be able to check, a can-check hypothesis **CC** which the pair thinks could be assessed with little effort, or doubted knowledge **DK**.
- **standard of knowledge**: Talk about meta-knowledge such as having or not having some knowledge (BL, Ch. 14). Base activities pertaining to such a discourse object serve three purposes: To prepare knowledge transfer **PT**, e.g., by stating how little one knows, to acknowledge knowledge transfer **AT**, e.g., after the partner explained something, or to refuse knowledge transfer **RT**, e.g., to justify not giving an answer to a question.
- **gap in knowledge**: A special type of meta-knowledge: a lack of knowledge shared by both developers and that is relevant for the task (BL, Ch. 15).
- **knowledge**: Mostly pre-existing knowledge (see discussion below in Section 3.4.4), but also a fallback category, e.g., for utterances for which it is not sufficiently clear whether the speaker treats it as an insight (a *finding*) or as something she already knew before (BL, Ch. 16).

Discourse objects in the base layer are *plain*. On the one hand, a previous HHI activity itself does not become an object.<sup>11</sup> Thus, there are no higher-level references, e.g., disagreeing with a *challenge\_knowledge* is no *disagree\_(challenge\_knowledge)* but still a *disagree\_knowledge*: The verb indicates the illocutionary force (here: *not affirmative, not constructive*), the object indicates the episode it relates to.

On the other hand, discourse objects are anonymous and indistinguishable. There is no index to distinguish two, say, *step* objects in a discourse. The base layer itself offers no means of tracking the topics of discussions with 'competing' alternative proposal or ideas.

### Verbs

There are two somewhat independent classifications of the **verbs**: First, individual utterances can be *initiative* (such as a proposal) or *reactive* (such as an evaluation of a proposal). As a manner of speaking, the according verb parts of the base concepts are also called *initiative* (e.g., *propose*) or *reactive* (e.g., *agree*), indicating their utterances are either predominantly one thing or the other. Second, utterances may or may not introduce new aspects into the discourse. Simple acceptance or rejection of ideas are *unconstructive*, rejecting with a counter-proposal, for example, is *constructive*.

The base layer distinguishes four different ways to start a new discourse episode with **initiative verbs**, each with a slightly different *illocutionary force*:

- **propose** is to make a proposal which consists of some informational content (such as a design aspect for a *propose\_design*), and optionally also carries a positive evaluation of

---

<sup>11</sup>HCI/HEI activities, however, can become objects of the *activity* concepts, e.g., in a *stop\_activity* (BL, Ch. 17).

	unconstructive	constructive
initiative	<i>ask</i>	<i>propose, explain, remember</i>
reactive	<i>agree, disagree, decide</i>	<i>amend, challenge</i>

**Table 3.4:** Verb classification in the base layer

the speaker and also optionally a request to the partner to evaluate it. The combinations of the optional properties lead to three types of *propose* activities (BL, Sec. 4.2.1):

**OE** to obtain evaluation (own positive evaluation and request for partner evaluation)

**PI** to provide information (own positive evaluation and no request to partner)

**LO** to look for orientation (no own evaluation, request to partner only)

All of *design, requirement, step, todo, strategy*, and *hypothesis* have been seen to be *proposed*, which are all *constructive*, can ask for an evaluation, and carry one's own evaluation.

- *explain* always includes a positive evaluation of the speaker and does not ask for evaluation of the partner. It has been observed in combination with *completion* and *state* as well as *finding, standard of knowledge, gap in knowledge*, and *knowledge*.
- *remember* is similar to *explain* in this regard but also carries the expectation that the partner recognizes the content. In the latest version of the base layer (BL), there is only the *requirement* object associated with this verb.
- *ask* has no information content (it is *unconstructive*) but requests the partner to produce one. It has been observed in combination with *design, step, strategy, standard of knowledge*, and *knowledge*. It appears sensible for *requirement, completion, state*, and *todo*, but less so for *finding, hypothesis*, and *gap in knowledge*, because their characterizing properties are not usually *requested* in conversations: One does not commonly ask someone to present something as a new idea, with some uncertainty, or as a shared lack of relevant knowledge.

Once a pair member introduced a discourse object into the dialog, the partner (or the original speaker) may react to it with a **reactive verb**:

- *agree* and *disagree* are both *unconstructive* in that they do not add new aspects to the discourse object but simply accept or reject it as it is,
- *decide* is also an unconstructive agreement of one discussed option if there was more than one,
- *amend* is an extension of the discourse object which implies agreement, and
- *challenge* is a counter-proposal which implies disagreement.

### 3.4.3 Example Application of the Base Concepts

Building on the base layer does not mean to just use the base concepts as a coding scheme. They are a means to improve the researcher's *theoretical sensitivity* as they enumerate illocutionary forces (verbs) and types of propositional contents (objects) that are relevant in pair programming. Using the base concepts is not about attaching a label to a line of transcribed pair conversation, but about 'opening up the data' (Strauss & Corbin, 1990, p. 77, see discussion in Sections 3.3.3b and 3.3.3e). This includes thinking about the developers' intentions as well as appropriate data segmentation (key decisions #2 and #3, see page 131).

To illustrate what such an application looks like, I searched for the original material that Plonka (2012, pp. 185–186) excerpted to illustrate her teaching strategy of *nudging* (see also page 131). The short form can be found here in Example 3.1; the line numbers are the same as in the pair's full exchange in the appendix (see Example C.5).

**Example 3.1: Coding with Base Concepts** (DA5, 22:50–24:30)

The two developers are in the process of writing a test case. They already introduced about 20 lines of test setup which produces an array of objects and are now about to write the first assertion.

Coded Transcript	Commentary
(1) D8: “(##for##) <*selects <i>foreach</i> template from autocompletion*> Well, I go through and if I find one, what do I do with it?” <i>propose_design</i> <sub>OE</sub>	D8 starts to write a for-loop and speaks along while the IDE completes for her. This is clearly about the test case’s <i>design</i> . Although D8 ends with a syntactical question, this is not an <i>ask_design</i> because she presents a partial design proposal (iterate over all objects until some match is found), which makes this a <i>propose_design</i> . She is positive about this part and asks her partner to complete it: Type OE, obtain evaluation.
(2) D2: “Actually, it should have found exactly one. Normally, it should be only one activity.” <i>disagree_design</i> + <i>explain_knowledge</i>	D2 makes clear he disagrees with D8’s <i>design</i> proposal without making one of his own ( <i>disagree</i> rather than <i>challenge</i> ). His rationale can be coded as <i>explain_knowledge</i> (there is no concept that is more specific).
(6) D2: “Actually, yes, only that (!...!) actually, it would be enough that there is exactly one activity.” <i>amend_design</i>	D2 proposes to insert an assertion that does not loop over all objects but just checks the array size. The context makes clear that he now refers more concretely to the source code ( <i>design</i> ) and that he affirms and refines what D8 said before ( <i>amend</i> rather than <i>challenge</i> ).
(21) D2: “I hope, there is only one. ‘There can be only one.’ That’s the Highlander principle.” <i>explain_knowledge</i>	D2 states he has some reservations (again, there is no concept more specific than <i>knowledge</i> ). Since his joke (a 1980s movie reference) is closely related to this, I segmented his utterances as one <i>explain_knowledge</i> . An alternative coding would have been a separate <i>say_off_topic</i> .
(23) D8: “OK. This is done now. And now <*inserts two empty lines*> we go on.” <i>explain_completion</i>	After inserting the assertion statement, D8 evaluates their current progress ( <i>completion</i> ): The matter of checking the object array (either in a for-loop or as a single assertion) is done, and the next thing can be addressed. However, this ‘next thing’ is not mentioned; she does not make a concrete proposal, so this is neither <i>design</i> nor a <i>step</i> .
(24) D2: “We can also test this first?” <i>propose_step</i> <sub>OE</sub>	D2, in contrast, does make a concrete proposal on the level of a tactical work <i>step</i> . Again, the speaker himself evaluates it positively and asks the partner to evaluate it, too: Type OE, obtain evaluation.
(25) D8: “Yes? OK. <*deletes empty lines*>” <i>agree_step</i>	D8 seems surprised by D2’s proposal, as if she expected the session to take a different route (which she does not verbalize), but agrees to it.
(26) D2: “You know, I have to admit, I’m a bit nervous.” <i>explain_knowledge</i>	D2 appears to justify his proposal to run the test case before adding new logic. If his disagreement with D8’s assessment of the situation was clearer, one could annotate <i>disagree_completion</i> instead.

My interpretation of D2’s overall behavior would be as follows: D8 wanted to implement the test case in a simple, but explicit way (a for-loop with assertions on each list item). D2 made an additional assumption about the system’s behavior and proposed a shorter, but implicit test implementation (checking the list length to be exactly one). He is, however, unsure whether his hypothesis is true and therefore wants to run the test case before they add any more assertions.

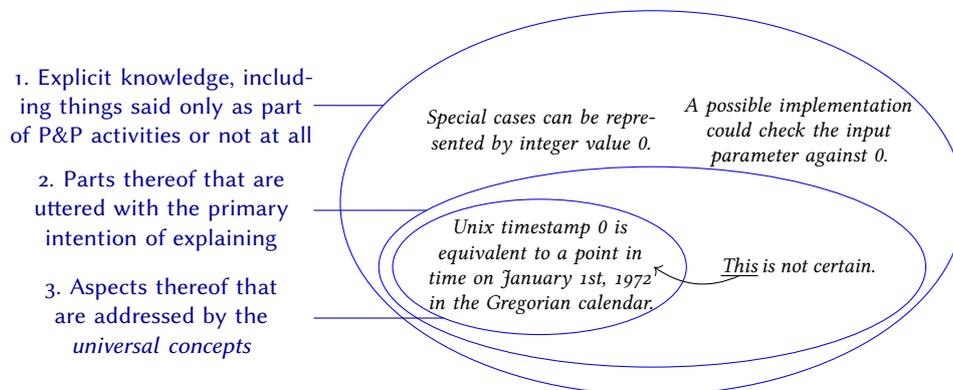
### 3.4.4 Notion of “Knowledge” in the Base Layer

*Knowledge* is an important concept in the Base Layer. A simple notion of “knowledge” would be the awareness and understanding of facts, truths, and pieces of information (Salinger, 2013, pp. 147–151). In this sense, however, *all* base activities could be considered as knowledge transfer and be coded with *explain\_knowledge* by default (BL, p. 110): An utterance such as D2: “*We can also test this first?*” (see line 24 in Example 3.1 above) makes his partner D8 aware of the *fact* that D2 wants to execute the test case. However, such an overly broad view does not correspond well with what practitioners consider knowledge transfer (see Section 2.3.1c). In order to be relevant (key decision #7 on page 132), the base layer’s *knowledge* concept is more narrow.

The development of the base layer followed a behavioristic ideal, and a discourse-oriented and intention-focused approach (see key decisions #2, #4, and #5 on pages 131 and 132). The first consequence is that the base layer is limited to the part of developer knowledge that can be communicated verbally or through sketches and gestures (Salinger, 2013, p. 147). Thus, it considers only *explicit knowledge*. Salinger (*ibid.*, pp. 148–150, 240–248) distinguishes three sets of explicit knowledge that a developer possesses *at a given point in time* during a PP session.

1. The whole explicit knowledge a developer possesses, regardless of whether she ever verbalizes it or not.
2. The subset thereof which the developer already uttered with the primary intention to explain said piece. This excludes two things: (a) things that are never said and (b) things that are part of P&P utterances (e.g., *design* or *strategy*, see page 133).
3. The subset thereof with all the pieces of knowledge that are explicitly addressed by one of the universal concept classes (e.g., *knowledge*, *finding*, *hypothesis*, see page 134).

Figure 3.5 illustrates these three sets for a particular moment in a pair programming session.



**Figure 3.5:** Three classes of explicit knowledge considered by Salinger (2013) during base layer development, illustrated through pieces of knowledge in the (hypothesized) state of mind of developer B1 when he says “*Null is January, 1st of 1972, or something*”. Such an utterance is coded with the universal concept *explain\_knowledge*.

The second consequence is that the base layer does not make a difference between *actually true* statements and statements that are merely *considered to be true* by the involved developers and are therefore possibly false (*ibid.*, pp. 147–148). Such a differentiation is usually not practical for a researcher anyway.



# Chapter 4 Research Goal, Method, and Data

---

*The observer, when he seems to himself to be observing a stone, is really, if physics is to be believed, observing the effects of the stone upon himself.*

– Bertrand Russell

<b>4.1 Purpose and Structure of this Chapter</b>	140
<b>4.2 Goal Definition.</b>	141
4.2.1 Topic of Interest: Knowledge Transfer	141
4.2.2 Type of Results: Vocabulary and Behavioral Patterns	142
4.2.3 Scope of Analysis: The Industrial PP Session	143
<i>Naturalistic Industrial Setting • Pair Programming: Two Developers Working on Shared Task • Limits of Data Collection: Pair Programming Sessions</i>	
<b>4.3 Data Collection</b>	144
4.3.1 Different Contexts and Headlines	145
4.3.2 Generic Data Collection Protocol	145
<i>Overview • Recording Sessions • Questionnaires • Quick Analysis • Reflective Interview • Field Observation &amp; Ad Hoc Interviews • Team Workshops • Session Repository</i>	
4.3.3 Supporting Practices	152
<i>Motivation: Difficulties of Applying the Grounded Theory Methodology • Consider Sessions as Cases • Long-Term Engagement with Companies • Evaluation with Practitioners</i>	
4.3.4 Discussion of Data Collection	154
<i>Limitation of Scope • Effects of Recording Infrastructure • Effects of Pre-Existing Notions • Summary of Data Quality</i>	
4.3.5 Selecting Data for Analysis	157
<i>Excluding Data • Theoretical Sampling</i>	
<b>4.4 Case Descriptions</b>	160
4.4.1 AA1: Complementary Frontend and Backend Knowledge	160
4.4.2 CA2: Undiscussed Design Rationale	162
4.4.3 DA2: A New-Hire’s Successful First Session	163
4.4.4 JA1: Pair Review with Domain Expert and Programming Expert	163
4.4.5 OA1: The Impossible Task	164
<b>4.5 Analysis Method.</b>	165
4.5.1 Perspective on the Data	165
<i>Area of Interest, Focus • Epistemological Stance</i>	
4.5.2 Coding	167
<i>Translating and Transcribing • Base Coding: Reconstructing Intended Meaning • Reconstructing Technical Information and Subjective Understanding • Conceptualizing Knowledge • Working with the QDA Software ATLAS.ti</i>	
<b>4.6 Discussion of Overall Research Method</b>	178

## 4.1 Purpose and Structure of this Chapter

In Chapter 3, I discussed the basic ideas underlying various qualitative research approaches, their common traits and the ways in which they differ, as well as high-level quality criteria of qualitative studies. Chapter 4 now is shaped by the common trait of OPENNESS & EMERGENT RESEARCH DESIGN and by the quality criterion of *Sincerity*.

Unlike *fixed method research* that relies on preplanned procedures, such as controlled experiments (Anastas, 2000, pp. 23–29; Wohlin et al., 2012, p. 9), there is no canonical order of steps in qualitative research. Rather, the **research design is emergent** (see Table 3.1 on page 114). Different activities affect each other: Reflecting on analysis results made me think about realistic goals, understanding the motivations of companies and developers led to better ways to collect data and to evaluate findings. I therefore discuss my research goal and method, as well as the data I collected and analyzed data together in one chapter.

**Sincerity** means that the researcher’s role in the process is openly discussed, e.g., through *self-reflexivity* of the researcher and through *transparency* of all research activities, interactions with settings and subjects, as well as crediting others for their work (see Section 3.2.4 on page 115), all of which I do in this chapter. I did not start my research in a vacuum. My research group AGSE began investigating pair programming in 2004, first with students and later with professional software developers. From these efforts, I inherited four things:

1. The overall *research goal* to inform practitioners. It is based on the assumption that for a non-trivial software development practice (such as pair programming) there are both beneficial and problematic ‘implementations’ in industrial contexts. By studying practitioners one may understand what the differences are such that, ultimately, software developers can be advised on how to employ the practice more efficiently.
2. *Methodological insights* on how to conduct qualitative research on pair programming which first resulted in an extension of Straussian Grounded Theory Methodology by four practices (perspective on the data, concept name syntax rules, analysis results meta-model, pair coding) and then a framework called the *base layer* which is tailored for analyzing recordings of PP sessions (see Section 3.4).
3. Existing *data* in the form of recorded industrial PP sessions. Between 2007 and 2008, Salinger and Plonka recorded 28 sessions of developers working on their everyday tasks in six different companies.
4. A *data collection protocol* (procedures and technical know-how) as well as audio- and video equipment for doing such recordings.

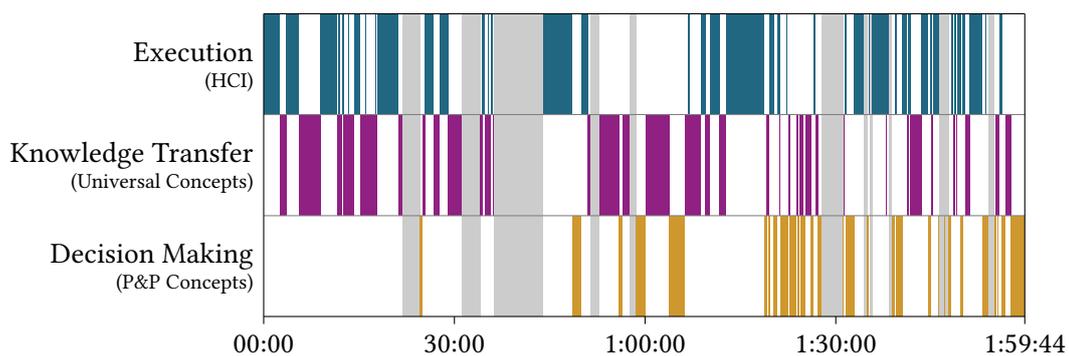
Setting my own **research goal, scope, and type of results** was affected by practical considerations, feedback from practitioners, and reflections on what can be validated (Section 4.2). I amended the **data collection** protocol by three supporting practices (considering sessions as cases, long-term engagement with companies, and evaluation with practitioners) and collected data in additional industrial contexts, four of which are relevant for this work. I explain all of these, my phases of theoretical sampling, along with a critical discussion of the data collection as such in Section 4.3. In Section 4.4, I characterize the five PP sessions or ‘cases’ that turned out most suitable for illustrating my findings (see Appendix C for information on all 27 analyzed sessions). I applied and adapted the coding procedures from Straussian GTM. My epistemological stance as well as my means of reconstruction and qualitative analysis I describe in Section 4.5. In Section 4.6, I summarize my overall research approach in terms of the common traits of qualitative research and the differentiating aspects, as well as the general quality criteria (introduced in Sections 3.2.2 to 3.2.4).

## 4.2 Goal Definition

In terms of the quality criteria formulated by Tracy (2010, see discussion in Section 3.2.4), my goal definition is influenced by identifying a *worthy topic* that is relevant and interesting (Section 4.2.1) while aiming for a *practically significant contribution* that is useful or helps re-framing some practical problem (Section 4.2.2). One aspect of *rich rigor* is to collect appropriate data, which in my case are recordings of industrial PP sessions. I discuss the pros and cons of this choice in Section 4.2.3.

### 4.2.1 Topic of Interest: Knowledge Transfer

After Salinger (2013) laid the groundwork for qualitatively analyzing pair programming by developing the base concepts (see Section 3.4.2), two large areas for further studies emerged: The first of which is *decision making* in pairs, which involves how pairs formulate and discuss proposals regarding the technicalities of their work as well their process on a tactical and a strategical level (addressed by the *product- and process-related* or *P&P concepts*, see page 133). The second area is that of *knowledge transfer* in the sense of the *universal concepts* (see page 134). Empirically speaking, neither area is more fundamental than the other: From what I had already seen in industrial PP sessions, developers transfer knowledge in the context of decision making and make decisions in order to transfer knowledge. Figure 4.1 illustrates how execution, decision-making and knowledge transfer are interwoven throughout a PP session.



**Figure 4.1:** Plot of session KA1, with stretches of time colored according to the pair’s type of activities. Grey areas are pauses in the pair programming process, waiting times, and interrupts.

As discussed in Section 2.3.1c, many of the benefits practitioners hope to get as an effect of working in pairs involve *knowledge* in some way. On the one hand, software developers bring relevant knowledge to the table when they start working on a technical task allowing them to complete it; on the other hand, developers may improve their knowledge while working on some task by learning together or from each other. Motivated by its practical relevance and little in-depth research so far (see Section 2.3.5), I want to understand how knowledge transfer in this sense actually works.

🔗 **Goal 1:** Understand how knowledge transfer works in pair programming in industrial settings, in particular how developers deal with what they individually and collectively know and do not know, i.e., what the underlying mechanisms of the exchange of existing knowledge and the acquisition of new knowledge are.

### 4.2.2 Type of Results: Vocabulary and Behavioral Patterns

In conversations with practitioners, simple models and dichotomies such as *driver/navigator* (Williams & Kessler, 2002, pp. 3–4) or *expert/novice* (*ibid.*, pp. 93–121) were not met with much resonance. One engineering manager said to me that he found it “*too simplistic*” and “*even offensive*” as it does not reflect his daily experience. I want to provide developers with terms to speak about pair programming, to enable them to reflect on their own practice and understand how others behave in similar situations. I characterize the variation of behavior instead of telling developers how to pair program ‘right’.

🔗 **Goal 2:** Characterize the underlying mechanisms in terms of their process properties. Formulate results in a way that is comprehensible and relevant for software developers, allowing them to reflect on their own process and identify which mechanisms work well and which are problematic.

I started out with the idea of identifying behavioral *patterns* and *anti-patterns*. However, based on the data which can be reasonably collected and analyzed (see discussion below), only few consequences of the respective behaviors can be observed and attributed to the developers’ decisions, thus making an assessment difficult. Using software engineering common sense, I attempted to evaluate individual actions of the pair programmers, but in most cases, I was able to come up with plausible explanations for why a seemingly problematic action can still be beneficial, as the following example illustrates:

#### Example 4.1: Good or Bad Behavior? (AA1)

Developers **A1** and **A2** work on a bugfix in a system consisting of a Java frontend, in which **A1** is more proficient, and an Objective-C backend, which is more familiar to **A2**. Throughout their session, **A2** would often look up implementation details in the Java code or try things out in the running application—all of which the Java-part expert **A1** already knew about or explicitly said were not relevant for their current task. Now, is **A2**’s look-up behavior ‘good’ or ‘bad’?

One could argue that **A2**’s excursions are unnecessary and therefore a waste of time or even that his partner **A1** would be better off working alone. However, there are plausible positive effects of such behavior: Even if the Java-part expert **A1** thinks that a certain piece of code is irrelevant for their task, he could be wrong and actually determining the reason why it is irrelevant (or not) could help both developers. Also, even if the Java-part expert **A1** thinks he knows how the system behaves without trying it out, **A2**’s attempt to recreate a system failure (and having a procedure to recreate it later at will) can be helpful for setting a baseline for the bugfix.

The developers do not make their reasoning explicit in the session. However, **A1** does not appear to be bothered by **A2**’s behavior. Rather, after brief protest, **A1** would closely follow **A2**’s reading and intersperse bits of information to make sure **A2** understands the whole picture.

Ultimately, each pair programming situation is unique in that no pair of developers ever works on the exact same task again with the same levels of understanding, prior knowledge, etc. There is also no alternative reality to compare against in which the pair behaved differently. Grounded Theory practices (in particular *open coding* and *axial coding*, see Sections 3.3.3a and 3.3.3b) aid the researcher in identifying similarities and relevant differences across situations, and in developing *concepts* to characterize purposeful human behavior. I focus on situations where pair programmers explicitly deal with what they know and do not know. In terms of the Straussian paradigm model (see Section 3.3.3b), identifying *causal* and *intervening conditions*, as well as relevant *context* properties is doable for individual cases. And even though developers do not usually explicitly explain their *strategies* to narrow their knowledge gaps, their behavior

can be interpreted and recurring patterns can be conceptualized. The *consequences* of their behavior, however, may only be traced within a PP session, but not to the relevant levels of how PP affects the quality of the companies product or the developers' individual capabilities for future tasks (see discussion of expected PP effects in Section 2.3.1c).

I therefore characterize pair behavior not in terms of outcome, but by pointing out relevant properties of their processes, so the practitioners reading or hearing my results can make up their own mind. In general, it appears reasonable to give experienced software developers the means to reflect on their process and let themselves decide what 'good' and 'bad' pair programming behavior is.

### 4.2.3 Scope of Analysis: The Industrial PP Session

Salinger & Plonka each had different approaches in detail, but agreed on two basic decisions which I follow as well: To base their analyses on audio and video recordings of PP sessions (Plonka, 2012, Sec. 4.3.2; Salinger, 2013, Sec. 4.1) and to prefer data from industrial settings (Plonka, 2012, Sec. 4.2; Salinger, 2013, Sec. 4.2.3 & p. 433).

#### 4.2.3 a) Naturalistic Industrial Setting

Many prior PP studies are limited due to their contrived research contexts which differ in relevant ways from industrial settings. There are, for example, unknown systems (developers cannot bring in existing knowledge), small systems (there is not much to know about), lack of social context (there is nobody to ask), little variability in developer experience (asking may not help much), unfamiliar programming partners (not knowing how and what to ask), and little prior exposure to pair programming (see also discussion on pages 67 to 68).

To overcome these limitations, PP research should take place in industrial settings with *professional software developers* working on their *everyday tasks*. This also entails that developers work in their normal development environment, with partners they choose to work with, at times and to an extent they decide.

I primarily rely on *observation* of developers working in pairs, as opposed to interviews. To enable a thorough analysis, pair programming needs to be *recorded*. In particular, the pair members' interactions with one another and their computer(s) as well as the contents of their screen(s) need to be captured in audio and video. The necessary recording infrastructure somewhat reduces the naturalism of the observed session; I discuss the effects in Section 4.3.4b.

#### 4.2.3 b) Pair Programming: Two Developers Working on Shared Task

Real-world software development activities are complex and sometimes evade simple definitions. This is my pragmatic definition of *pair programming*:

- **Number of Developers:** There are two developers involved with the task. A pair member may also temporarily leave, and additional developers may temporarily join to help out with some aspect. I exclude groups of three or more developers who work together throughout on a task ("mob programming").
- **Alignment Towards Goal:** The efforts of the developers are directed towards a joint task. I exclude any planned division of labor (as in *I implement function A, and you do B.*), or other circumstances where the developers sit physically close, but work on individual tasks at their own speed.
- **Type of Goal:** The developers have a productive goal in that they work towards some concrete improvement of a relevant artifact (such as discussing requirements and design,

implementing, testing, reviewing, debugging, refactoring, and reading documentation). Diversity in terms of different programming pairs working on different types of tasks in different technical domains is desirable, but I exclude activities like going through a tutorial to improve one's skills detached from any concrete task.

The developers may work on one machine exclusively, or on more than one machine as long they are tightly coupled (as in distributed pair programming) or if one machine remains the focus of attention most of the time whereas other laptops and such are used only briefly, e.g., to quickly look something up.

#### 4.2.3 c) Limits of Data Collection: Pair Programming Sessions

Although the effects of developers learning from another through PP may go beyond a single pair and beyond a single programming session (e.g., knowledge may spread across the development team), I restrict the scope of my analysis to what happens **during PP sessions**.

In Section 2.3.1a, I introduced the distinction of pair programming *as a work mode* and pair programming *as a practice*. My research is concerned with the *work mode*, i.e., every instance of software developers working together as a pair on a technical task in the sense defined above. I exclude the questions *When do developers pair-program?* and *When should developers pair-program?*. My investigation starts at a point where the **developers already made the decision to work as a pair**. Their decision, just as the project they work in, the task(s) they work on, their software system, and their team structure all may 'echo' in their session and are therefore helpful for understanding their activities, but are not prime concerns of my analysis.

The limitation of focusing on in-session behavior only, i.e., the time when developers actually pair-program, is in part motivated by two pragmatic reasons: First, recordings of industrial PP sessions were already available in my research group. This data was collected for research interests that considered the *PP session* as the unit of analysis—as opposed to other possibilities such as a single developer, a particular programming pair, a technical task, or a software development team. Only late in the research process I realized the consequences of this, which I discuss in Section 4.3.4. The second reason is that a thorough analysis needs recordings, not just reports, and the technical side of recording just a single pair at a time is already challenging and requires a lot of attention. Nowadays, the technicalities of recording all activity within a software development team have become more feasible and some studies have indeed been conducted. Socha et al. (2015, 2016) recorded 11 days of software development with multiple video cameras and audio recorders resulting in six terabytes of data consisting of thousands of photographs, 292 hours of screen captures, and 380 hours of video of developer interactions. Nevertheless, analyzing such data with *qualitative* methods remains a monstrous effort—and even then covers only one software development team from one company for possibly less than a sprint.

Overall, I deem the limitation on in-session behavior to be not too strict, as any effects that the PP work mode may have on the developers *beyond* the session need to have a manifestation *during* the session, e.g., if they actually are to learn something through a session, something in this regard needs to happen in the session. I am limited, however, in assessing the practical importance of the acquired and exchanged knowledge.

### 4.3 Data Collection

Some of the data I analyze was collected by other researchers, some I collected myself. In this section, I describe why and how the data I considered analyzing was collected. I also discuss

how I selected which PP sessions to analyze in depth. The sessions themselves I describe in Section 4.4; my analysis method follows in Section 4.5.

Between 2007 and 2008, Salinger & Plonka collected pair programming data in six companies (called A to F).<sup>1</sup> After I joined the research group, I led data collections in four additional companies (K, M, O, and P) and was involved in three more (J, L, and N). There were slightly different headlines for all these industry cooperations. I discuss the most important differences in Section 4.3.1.

Salinger & Plonka devised a protocol that served as the basis for collecting data in all companies. The protocol is not to be understood as a rigid order of steps, but as a template which needs to be filled with concrete tactical actions once the researcher is on site. I describe the basic form (Section 4.3.2) and three practices that support the data collection (Section 4.3.3). I critically examine the overall data collection approach (Section 4.3.4), before I explain how I selected the particular data for my analysis (Section 4.3.5).

### 4.3.1 Different Contexts and Headlines

With each industry contact, there were slightly different sets of mutual expectations which resulted from prior discussions with the partners and from evolved research interests in our group. I discriminate three types of research interests here, because they shaped the researcher's behavior and likely the subjects' behavior (because they knew *why* the researcher is around). Table 4.1 then gives an overview of the *individual* contexts (and involved researchers) for each such research "headline".

- **PP:** Most companies were specifically approached with the intention to understand pair programming. While for the first contacts there was no particular focus yet, Plonka (2012) was interested in the *driver and navigator* roles (see discussion on page 73) and her data collection at companies C to F was influenced by that (e.g., by the choice of the camera angle, which I discuss later on page 156). Additionally, the data collection period was branded to these companies as a "*workshop*" to help developers reflect on their PP process (Plonka, 2009). Schenk (2018) was particularly interested in distributed pair programming and chose her contacts J and L accordingly; I am interested in knowledge transfer and all participants from companies K, M, O, and P knew that.
- **Agile:** Companies K and O were approached for a larger research effort to understand agile software development in general. According data from these contexts was collected and analyzed (see Zieris & Salinger, 2013, which is about company K). All particular PP sessions, however, were recorded exclusively for the purpose of understanding knowledge transfer in pair programming.
- **Onboarding:** Company N was approached for understanding their onboarding process, i.e., how new hires are integrated into the company. Pair programming was not a designated part of that process, but a number of developers agreed to be recorded while working in pairs. The recordings were therefore a window into the onboarding process and were not made to understand pair programming.

### 4.3.2 Generic Data Collection Protocol

The data collection protocol is *generic* in two ways. First, it may be adapted in each particular installment at a company on-site to deal with constraints, to seize opportunities, and to fit the particular research focus of the researcher (see previous section). I discuss my own amendments

<sup>1</sup>See their respective PhD theses for details: Plonka (2012, pp. 59–68) and Salinger (2013, pp. 95–103).

Company	Year	Researchers	Headline	Focus
A	2007	Salinger	PP	–
B	2007	Salinger, Plonka	PP	–
C	2008	Plonka	PP	workshop, roles
D	2008	Plonka	PP	workshop, roles
E	2008	Plonka	PP	workshop, roles
F	2008	Plonka	PP	workshop, roles
J	2013	Schenk	PP	distributed PP
K	2013	Salinger, Zieris	Agile & PP	knowledge transfer
L	2014	Schenk	PP	distributed PP
M	2014	Zieris	PP	knowledge transfer
N	2016	Salinger, Schmeisky, Zieris	Onboarding	–
O	2016	Zieris	Agile & PP	knowledge transfer
P	2018	Zieris	PP	knowledge transfer, evaluation

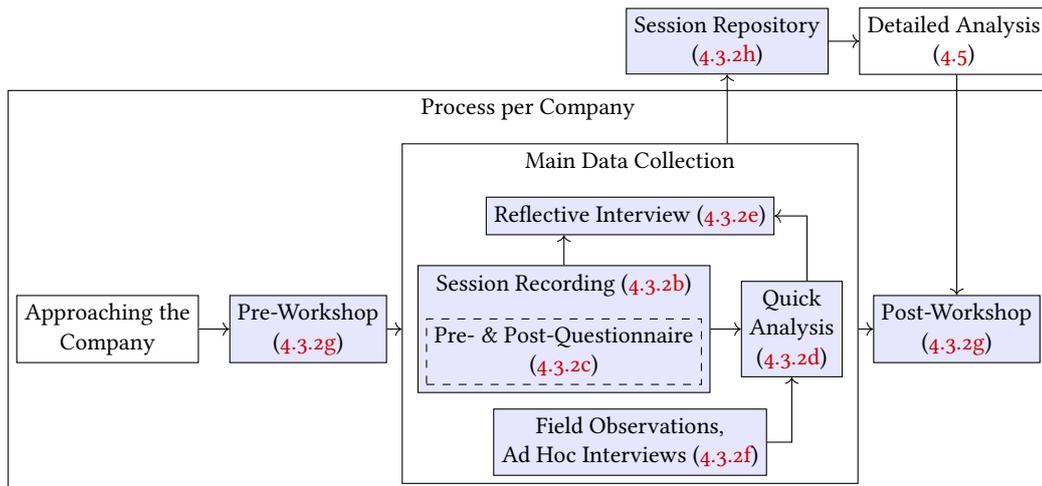
**Table 4.1:** Contexts for PP sessions recorded by AGSE researchers. The research direction (“headline” and “focus”) was set by the named researchers; I myself was involved in the technical part of recording the sessions in all contexts from J to P. Sessions in three additional industrial contexts (G, H, and I) were recorded with very low technical quality by students with little oversight, so important context information is missing.

in Section 4.3.3. Second, the protocol is still more or less independent from any particular research question regarding pair programming, as the resulting data can be reused for different purposes (some conditions apply, which I discuss in Section 4.3.4).

#### 4.3.2 a) Protocol Overview

After a company has been approached and probed whether the company would be open to have some of their programming sessions recorded, the overall research goal, the procedure, extent, and purpose of the main data collection are explained in a presentation for the development teams. It is made clear that all participation is voluntary and that their individual agreement to be recorded can be revoked at any point during a session. These are the steps for each session recording (see also Figure 4.2):

- After a pair announces that it is willing to have their next pair session recorded, the recording infrastructure is set up. The **session recording** is started once the developers are ready (see Section 4.3.2b for details).
- Optionally, both developers fill out **questionnaires** before and/or after their session in which the developers state their names, development and pair programming experience, characterize the nature of their task, and whether it went as they intended (Section 4.3.2c).
- Afterwards, the researcher does a **quick analysis** of the material during which she looks for peculiarities that catch her attention (Section 4.3.2d). The main purpose of this step is to inform the next activity.
- The researcher then conducts a **reflective interview** with the developers on the day after the recording. This activity serves different purposes, including collecting background information and providing developers with feedback in return for them agreeing to have their PP session recorded and scrutinized (Section 4.3.2e). These interviews are audio-recorded.



**Figure 4.2:** Protocol overview. Highlighted activities involve data collection.

Parallel to the session recordings, the researcher makes other **field observations**, e.g., of the general team activities such as the daily stand-up meetings or other ceremonies (see Section 4.3.2f).

As a milestone—often, but not exclusively, at the end of the relationship with the company—comes a presentation or **workshop** during which the company-specific findings are summarized to a group of developers (possibly beyond the original development team) or to process-inclined roles such as Scrum Masters or agile coaches (see Section 4.3.2g). This serves the two purposes of passing on and validating the findings (*member reflection*, see page 116).

The **detailed analysis** then commences off-site (which I discuss later in Section 4.5). To this end, all session recordings, notes from the quick analyses, interview recordings, and field observation notes are organized in a **repository**, which serves as the basis for *theoretical sampling* across different companies/contexts (see Section 4.3.2h).

#### 4.3.2 b) Recording Sessions

The developers themselves decide when and for how long they want to be recorded. They work on their own machines, in their normal environment, on their everyday tasks, and with the partner they chose. Some pairs take breaks while a recording is still running, which I treat as separate PP sessions if their break is 15 minutes or longer.<sup>2</sup> Overall, the lengths range from 0:25 hours (session MA1) to 5:27 hours (session JA9), with a median length of 1:30 hours and the vast majority running between 0:45 and 2:30 hours.

The session recordings as technical artifacts consist of a screencast of the pair’s monitor(s), the pair’s conversation as audio, and a webcam video showing the pair members’ interaction. These three sources are combined to a self-contained video file as illustrated in Figure 4.3. Both webcam feed and screencast are captured at between 5 and 15 frames per second (depending on hardware capabilities), which is enough to distinguish individual keystrokes, follow mouse movements, and see the developers’ gestures. The final video resolution depends on the developers’ display(s) and recording setup and ranges from 1024×768 to 2560×1440 pixels.

The recording process relies on one of three generations of hardware and software components, the first of which was developed by Plonka & Salinger, the other two by me. The general

<sup>2</sup>One recording sitting in company D, for example, lasted 3.5 hours, but had a two-hour break after 31 minutes, leading to the two PP sessions DA5 and DA6.

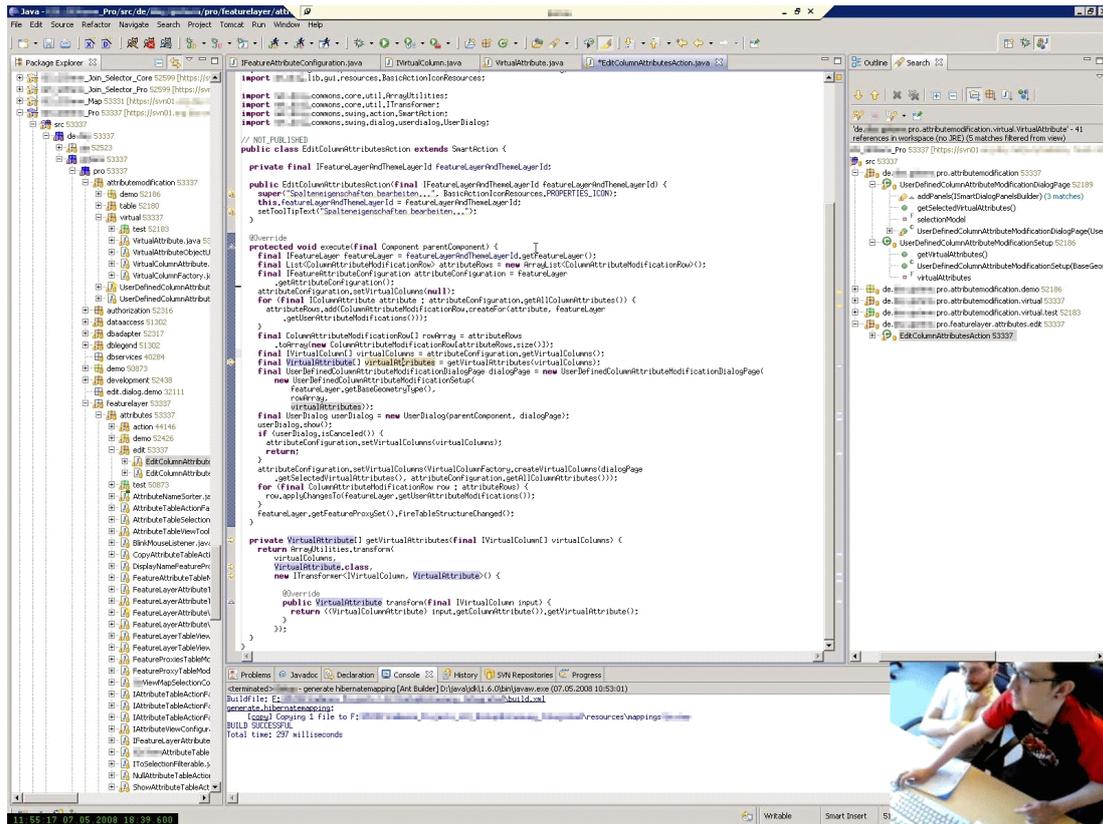


Figure 4.3: Still frame of a session recording (here: session CA2 at 18:39). The screencast is in the background, the webcam video is layered on top.

setup works like this: The developers work on one machine, and screencast and webcam feed are transmitted to another machine where they are recorded using TechSmith Camtasia; I explain the details in a separate technical report (Zieris & Prechelt, 2020b, Appendix A). The most relevant difference is that Generation 1 is an unattended recording which the researcher only gets to see once the pair is done, while Generations 2 and 3 are an online recording which allows the researcher to also watch the session live while it is still being recorded.

The way how the recorded PP sessions in companies C, D, E, and F came to be is less naturalistic than described above. To the companies, the whole operation was branded as a “workshop” to provide developers with reflections on their PP process for which they otherwise would be too immersed in their ongoing work (Plonka, 2009, p. 3). Here, one work station was set up for all pairs to use (Plonka, 2012, pp. 60–61) and the developers would put their names on a list to choose either a morning or an afternoon slot to be recorded for a planned maximum duration of 1.5 to 2 hours (Plonka, 2009, p. 4). Nevertheless, once a recording was started, the developers (just as in all other companies) were left to work on tasks of their choosing for as long as they wanted.

#### 4.3.2 c) Questionnaires

In companies A to F, the developers were asked to individually fill out a questionnaire prior to their session and another one afterwards. There were three versions of these questionnaires. The topics of the respective items can be seen in Figure 4.4; the questionnaires themselves are available online as part of a technical report (Zieris & Prechelt, 2020b).

Task & Pair Items (“Pre-Session”)	Process Items (“Post-Session”)
<ol style="list-style-type: none"> <li>1. Task classification (new functionality, extend functionality, test cases, debugging, refactoring, or other)</li> <li>2. Short description of the task</li> <li>3. Characterization of (expected) difficulties</li> <li>4. Estimated time to completion [added in version 3]</li> <li>5. Why is task worth to be worked on by a pair</li> <li>6. Professional software development experience</li> <li>7. Pair programming experience [added in version 2]</li> <li>8. How well attuned to respective partner</li> <li>9. Expectations towards the reflective interview later on [added in version 2]</li> </ol>	<ol style="list-style-type: none"> <li>1. School grade for recent session</li> <li>2. Compare progress with expectations</li> <li>3. Divide session into phases</li> <li>4. Name most important phases</li> <li>5. Assess session-specific importance of each: knowledge transfer, developing a strategy, bug fixing, developing a design/an architecture, developing an algorithm, knowing an API, having the right idea [removed in version 2]</li> <li>6. Points where pair constellation should have been given up</li> <li>7. Points where pair constellation was especially beneficial [added in version 2]</li> </ol>

**Figure 4.4:** Shortened questionnaire items. There were three different versions: Version 1 was used for companies A and B, version 2 for sessions CA1, CB1, CA4, and CA5; and version 3 for sessions CA2 and CA3 as well as for companies D, E, and F. Developers from companies J and K received and answered the *task & pair items* (except 4, 6, and 9) via e-mail *after* their respective sessions.

I did not use any pre-session questionnaires for my recordings because they contradicted with my naturalistic approach to data collection (see discussion following in Section 4.3.4c). In companies J and K, the respective researchers (that is, Schenk and myself) e-mailed questions to the developers *after* the session recordings to get demographic information and a characterization of their task. In the other installments (N by Schmeisky; M, O, P by me), no questionnaires were used. Instead, we asked the developers directly in the reflective interviews.

#### 4.3.2 d) Quick Analysis

Shortly after a session was recorded, the researcher reviews the material for the first time. (With Generation 2 and 3 of the recording infrastructure used in companies J to M, O, and P, the PP session could be watched live by the researcher, thus allowing for the quick analysis to start right away.) This step is not structured in a formal sense, but is open for any phenomena that catch the researcher’s attention. It relies primarily on software engineering common sense and a preliminary understanding of pair programming (a form of *theoretical sensitivity*, see Section 3.3.3e). Possible aspects include (1) understanding what the pair did from a software engineering perspective, (2) what went well and where they encountered problems, and (3) any peculiarities in the pair’s process independent from the technical task.

Either way, the two purposes of this first glimpse at the rich data are (a) to have a grasp of what happened in the session to have some common ground with the developers for the reflective interview the next day (see next section), and (b) to collect open questions on the context that may easily be answered while still being on site, but less so later in the analysis.

#### 4.3.2 e) Reflective Interview

After the researcher had the time to look through the material at least once, she sits down with the two involved developers for a reflective interview. Here, both developers get the chance to speak about their experience in the session, possibly guided by their assessment in the

post-session questionnaire. The developers were assured that all contents of these interviews would remain anonymous to their colleagues and superiors.

The original intention of these interviews was to provide the developers with a learning opportunity (Plonka, 2012, p. 63) and thereby compensate them (and their employers) for participating in the study (Salinger, 2013, p. 99). Plonka (2012, p. 63) also noted that these interviews may help identifying phenomena that the developers themselves perceive as relevant.

From my own data collection, I came across two additional uses for these interviews. First, the developers can be specifically asked for context details to make later sense-making easier. Second, to the developers, these interviews are the most tangible outcome of this research. Developers K2 and K3 remembered the contents of their first reflective interview even six months later (see also Section 4.3.5b on my concrete data collection activities below); developers O3 and O4 wanted to record another session to hear whether they ‘improved’ and also motivated colleagues to be recorded too. In the terminology of Tracy (2010), this is a form of *member reflection* (see page 116). It should be noted that I carefully avoided bringing negative and positive connotations into the discussion, and emphasized that I do not know what a ‘good’ or ‘bad’ PP session actually is.

Later still, it occurred to me to emphasize more the *member reflection* property of these interviews to validate results from earlier analyses. The most recently recorded session could provide concrete exemplars of general phenomena to be discussed directly with the involved developers (details on how I approached this evaluation follow in Section 4.3.3d). Overall, the reflective interviews should not be understood as data collection only, but also as validation of results and preparation of further data collection (through motivating developers).

#### 4.3.2 f) Field Observation & Ad Hoc Interviews

Although my research is focused on the developers’ in-session behavior, the pair’s socio-technical context still needs to be considered for an appropriate interpretation of the events, but at the same time cannot be expected to become observable from an isolated session recording alone. Therefore, additional information on the context of each PP session needs to be collected, e.g., on the organizational structure the developers work in, the domain, its fundamental requirements and constraints, the nature of the task they work on, their motivation to work together on said task at this point in time.

In companies C to F, some of such information was collected through pre-session questionnaires, in other cases the reflective interviews posed an opportunity (see Sections 4.3.2c and 4.3.2e above). In companies O and P, I followed a more holistic approach and took structured fieldnotes of the teams’ everyday software development and of discussions with developers at the proverbial water cooler. For these, I brought in my own software development expertise for small talk on technical issues which made it easier to not stand out as a researcher and get ‘close’ to the action.

#### 4.3.2 g) Team Workshops

Presentations and discussions with a whole development team at once may take place before and/or after the main data collection. These workshops are similar to the reflective interviews (Section 4.3.2e) in that they also do not have a rigid structure and combine elements of data collection (e.g., learning more about the particular context), preparation of data collection (by motivating developers with initial insights), and validation (by running by concepts and findings to check for resonance; details follow in Section 4.3.3d).

### 4.3.2 h) Overview of Data in Session Repository

I organized all PP session recordings and additional data collected by my research group in a **session repository**. Since the companies did not explicitly agree to being referenced by name, I introduced unique identifiers to be used in publications (e.g., in Salinger & Prechelt, 2013; Schenk et al., 2014; Zieris & Prechelt, 2014, 2016, 2020a):

- Companies are represented by single letters: A, B, C, and so on.
- Sessions are grouped by their technical context ('project', each with a different set of requirements and/or different technology stack) and then counted up with Arabic numbers. **CB1** is the first recording in the second project in the third company.
- Developers are identified through their company and Arabic numbers, such as **C3**.

I provide a mapping of how prior PhD theses by Plonka (2012), Salinger (2013), and Schenk (2018) referred to the companies, sessions, and developers in Appendix C.23.

I summarize the contents of the repository in a technical report (Zieris & Prechelt, 2020b), which also includes a technical description of the three generations of session recording setups and a reprint of the handout that was given to company C in their post-workshop. Table 4.2 provides an overview of all available data.

Company	Session Recordings		Questionnaire		Reflection Interviews	Field Notes	Workshop	
	<i>n</i>	IDs	pre	post			pre	post
A	1	AA1	✓	✓	-	-	-	-
B	4	BA1, BB1-3	✓	✓	-	-	✓	-
C	6	CA1-5, CB1	✓	✓	(✓)	(✓)	-	✓
D	6	DA1-6	(✓)	✓	(✓)	-	-	-
E	7	EA1-7	✓	✓	(✓)	(✓)	-	-
F	4	FA1-4	✓	✓	(✓)	(✓)	-	(✓)
J	9	JA1-9	-	(✓)	-	-	-	(✓)
K	8	KA1-2, KB1-2, KC1-4	-	(✓)	(✓)	-	-	(✓)
L	2	LA1, LB1	-	-	-	-	-	-
M	1	MA1	-	-	-	-	-	-
N	5	NA1-5	-	-	(✓)	-	-	-
O	10	OA1-10	-	-	(✓)	✓	-	✓
P	4	PA1-4	-	-	✓	✓	✓	-

**Table 4.2:** Overview of available data from all 13 companies per data collection activity. Data may be ✓-complete, (✓)-partial, or non-existing. The standards for *complete* are as follows: Questionnaires for all sessions were handed out and then filled out by both partners individually; reflection interviews are all audio-recorded; fieldnotes contain context information and detail; for pre- or post-workshops, there are extensive notes or a written handout for the company. Much of the data from interviews, fieldnotes, and workshops I classify as *partial* because they are handwritten notes with no indication what are *quotes/observations* and what are *researcher ideas/hypotheses*.

### 4.3.3 Supporting Practices

In addition to amending the purpose and adapting the implementation details of the data collection steps discussed above, I also introduced three new ‘practices’. These are not isolated activities, but address more crosscutting concerns. They are motivated by difficulties of applying the Grounded Theory Methodology to the available data and by differences between my predecessors’ and my own research interest and goal.

#### 4.3.3 a) Motivation: Difficulties of Applying the Grounded Theory Methodology

Establishing a relationship to a company which results in being able to collect data in the fashion outlined above takes time. **Theoretical sampling** (see Section 3.3.2) in the sense of *collecting* additional data the moment a research need arises might be feasible for an interview-based study with subjects who can be easily accessed and asked regarding some topic deemed irrelevant before. Finding an observation context with my desired properties, however, is more difficult. At each company, my colleagues and I collected more data than would have been strictly necessary to serve the research interest at hand to allow for later analysis once a particular need emerges. The data is organized in a session repository (see Section 4.3.2h above) which serves as one source for theoretical sampling. Strauss & Corbin (1990, p. 186), too, recognize the problem of not having unlimited access and conclude that sampling more by chance (i.e., availability) than choice can still be successful although it may take more time.

I already summarized the problems my research group encountered while first applying **open coding** to video recordings of pair programming sessions (drowning in data, unclear epistemology, inconsistent concept granularity, and too many aspects to consider) along with four practices to counter them (see Section 3.4.1). The most important extension is the *perspective on the data*, in which the researcher (1) defines the area of interest, (2) makes explicit her epistemological standpoint, (3) and sets the desired result type. I come back to this later in Section 4.5 (Analysis Method).

During **axial** and **selective coding**, phenomena are no longer considered in isolation, but analyzed with regards to their context, cause, intervening conditions, interaction strategies, and respective consequences. Videos of pair programming sessions, however, are limited in that not all consequences of the developers’ actions are on record, and even with reflective interviews conducted afterwards, there are few personal accounts of causes and considered alternative strategies. Follow-up questions in the interview are limited to what caught the researcher’s attention in the quick analysis. The best I can do *years* after a PP session was recorded is to compile all available information and try to reconstruct the setting. In other words, I consider each session as a distinct *case* (first practice, see Section 4.3.3b). For collecting new data, I tried to avoid a strategy of “*smash and grab*” in favor of “*slow*” and “*careful*” observation (Dey, 1999, p. 119, who formulated this as a critique of Glaser & Strauss, 1967) as part of a long-term engagement with the companies (second practice, see Section 4.3.3c).

For Strauss & Corbin (1990), the **evaluation** of a GT study boils down to assessing data quality and process quality (see Section 3.3.4a). I discuss them in Section 4.3.4 and Chapter 13, respectively. As Charmaz (2006) and Tracy (2010) argue, there are other relevant quality criteria as well, e.g., what they call *resonance* and *practical significance*, respectively (see Section 3.3.5a): Do the results make sense to software developers who pair program and does it offer them deeper insights about their work? While Charmaz (2006) does not add any thoughts on the practical form of such an evaluation, Tracy (2010, p. 844) speaks of providing the participants with “*opportunities for questions, critique, feedback, affirmation, and even collaboration*”. I involved practitioners in different ways (third practice, see Section 4.3.3d).

#### 4.3.3 b) Practice: Consider Sessions as Cases

Plonka (2012) analyzed all sessions from companies C to F (except for DA1 which lacks the webcam feed). However, none of the excerpts provided in her publications contain information from which session, developer pair, or company it comes from. It appears that she treated all the pair programming sessions as more or less interchangeable representatives of the phenomenon of “*Pair Programming in Industrial Settings*”, which may be appropriate for her research interest and chosen analysis approaches (see my discussion in Section 2.3.4 on pages 73, 81, and 84).

Similarly, Schenk (2018) analyzed eight consecutive sessions of one professional pair and was concerned with phenomena relating to the pair’s *distributed setting*. All excerpts are presented as timeless representatives of these phenomena regardless of their role in the overall development process (see my discussion on page 84).

This contrasts with Salinger (2013). Despite employing mostly *open coding* of local process phenomena, he provides detailed characterizations of the sessions as a whole (*ibid.*, pp. 86–102), and discusses multiple excerpts with session identifier and timestamp.

In my research, I also employ *axial* and *selective coding* which require to consider the particular contexts in which the pair programmers find themselves. I therefore follow and extend Salinger’s approach and consider each PP session as a unique *case* with the pair members, their task, and project forming the background in front of which all their actions need to be interpreted. An overview and the descriptions of five ‘main’ cases follow in Section 4.4, the other 22 sessions I characterize in Appendix C.

#### 4.3.3 c) Practice: Long-Term Engagement with Companies

Each data collection phase in the companies C to F took place within five workdays. On the last day of the “*data collection week*”, the researcher presented initial insights to the developers (Plonka, 2012, p. 67).<sup>3</sup> This is a tight schedule which, together with organizing up to three recording sessions per day, leaves little room for field observations and data analysis. Recording all data per company within one week (and virtually no pair constellation twice, see discussion on page 156 below) also means that one-off behaviors cannot easily be detected as such. Plonka (2009, p. 12) herself remarked regarding the data collection in company C that one session per pair may not represent the pair well and a two-week time frame would be better.

In my own data collection, I wanted to cover longer time frames, spend more time on-site without the pressure to get as many recordings done as possible. I also wanted to get back to the developers after I had more time to perform more than just a quick analysis of their material. In the end, I managed to establish two long-term connections (see Section 4.3.5b).

#### 4.3.3 d) Practice: Evaluation with Practitioners

As defined as part of my research goal, the results need to be comprehensible for software developers (Section 4.2.2 above). The research results pertain to different levels of abstraction: individual activities or utterances, the pursuit of individual topics, and the overall setting and trajectory of a pair programming session. Not all of these are equally accessible to developers. In particular, the largest granularity should be relatable for most practitioners, but the fine details do not need to. Consequentially, the research process needs to include activities to perform this *member reflection* (see page 116).

I perform evaluations on two levels. First, I explain the high-level concepts to practitioners with some personal PP experience during the aforementioned workshops as well as one-on-one

<sup>3</sup>There is no written record of these insights for companies D and E (see Table 4.2).

interviews and ask whether such things sound familiar to them. The downside of this approach is similar to that of a survey (see Section 2.3.2b): The respondents may say they experienced something, but there is no way of telling whether something like this actually happened to them or whether it simply seems plausible to them. The second approach is more concrete as I use the reflective interviews to talk with the developers about their recently recorded PP session with my terminology. This way, I already know which concrete events the developers and I refer to.

In both cases, I check which ideas ring a bell (as in ‘*Ah, I know that one*’) or sound interesting to the other party (as in ‘*I should try this*’). The overall response was positive: My findings resonate with practitioners in the abstract and as a means to reflect on their own process. These discussions appear to be memorable, too: In company K, when I spoke to a pair of developers in October 2013, they could still repeat the key ideas of our first discussion in May. Only late in the analysis, the name of one central concept turned out to be misleading, and I changed it. I report the details in Chapter 13, after I presented my results.

#### 4.3.4 Discussion of Data Collection

The data collection procedure sketched above has a number of properties which affect the data quality in terms of the kind of data which can be collected this way and the developers’ behavior.

##### 4.3.4 a) Limitation of Scope

Due to the method’s design, the recordings will not reflect all kinds of relevant PP situations that occur in practice. In particular, I expect the following gaps:

- **Not all companies:** Due to the naturalistic approach, I did not *request* developers to pair-program and I did not target companies with little or no pair programming usage.<sup>4</sup>
- **Not all developers:** All recordings are voluntary and some developers may not want to be recorded. In company P, for example, one team member was generally inclined to working in pairs, but did not want to be part of the data collection, despite an emphatic recommendation by a colleague who just had his first reflective interview.
- **No short sessions:** The majority of the sessions in the repository is one hour or longer, with the shortest one being about 25 minutes long. In the team workshops (see Section 4.3.2g), however, some developers reported of common session lengths of 10 or 15 minutes. Since the recording setup poses an overhead to the normal work flow, *ad hoc* pairings are difficult to record. Furthermore, pairs that had already gone through organizing and setting up a recording then possibly work longer than they otherwise would have.
- **No tense situations:** The mere presence of a researcher on site may be regarded as a distraction. In companies O and P, there were multiple months between the first team workshop and the start of the main data collection, and in both cases it was due to the Scrum Masters wanting to postpone the research activity until a turbulent phase in their respective project was over. A second data collection phase in company O did not happen because of immense time pressure for the software developers—even though both developers and Scrum Masters were very happy with the insights from the first round. It is not clear whether any pair programming was done in these stressful phases.

---

<sup>4</sup>I briefly considered giving up this naturalistic stance by asking developers in non-pairing companies to pair in order to collect session recordings of—presumably, due to a lack of practice—bad pair programming. I discuss such tactical issues in Section 4.3.5b on my theoretical sampling process.

There are other limitations of the data which are not strictly due to design, but due to practicalities of getting in contact with a company and traveling:

- **Western Cultural Background:** All companies are based in Germany with the exception of company M which is in Oslo, Norway.
- **Language Limitations:** Most developers are native German speakers. The L-developers are the only native English speakers, and the M- and O-developers use English as their work language.

#### 4.3.4 b) Effects of Recording Infrastructure

In companies A to D, my colleagues also equipped the developers' IDEs with a plugin to collect technical information on their current activities, focus, etc. (see Salinger, 2013, pp. 85 & 461–479). This led to some artifacts in the programming sessions. For instance, in CA2 where the developers spend 1:20 minutes trying (and failing) to look up an ID from the issue tracker on the remote development computer with the running IDE logger before tabbing out of the remote desktop session to find the necessary information within five seconds on the local machine which they first avoided to ensure continuous data collection. In session CA3, the IDE is repeatedly unresponsive for 80 seconds totaling about one third (!) of the whole session, which may have been due to a defect in the data collection plugin.

There are several instances of developers not working on their own machine and this affecting their work: In sessions CA2 and BA1, the developers are irritated multiple times because some IDE setting is not as they are used to; session DA2 starts with several minutes of the pair waiting for an SVN update to complete since the workspace had not been used for a while.

The recording infrastructure was not always fully compatible with the local circumstances and the developers' habits: In session EA1, the keyboard shortcut for stepwise debugging in the IDE also paused the screencast recording. Not only did this lead to some gaps in the screencast, but appears to have also confused the developers (as the continuous audio recording reveals). All pairs in companies F appear to have used two monitors, but recording infrastructure generation 1 was not able to capture both, so the screencast is incomplete.

Although the wireless microphones and webcam were supposed to not bother the developers, they occasionally fiddled with them, e.g., before leaving the desk for a minute and again upon their return. One pair knocked over the webcam from its tripod; another pair took a break together to get some candy and wandered beyond the wireless transmitter radius while still talking about their task. Reports on how the subjects felt regarding the data collection are available from the C developers only, some of which say they felt being watched while others claim to have forgotten the camera after five minutes (Plonka, 2009, p. 12).

#### 4.3.4 c) Effects of Pre-Existing Notions

There were pre-existing notions of what the social reality of industrial software development looks like which were deeply embedded in the data collection process and affect its outcomes.

##### *Not Recording All Aspects*

The first such notion is *pair programming* itself. My research (and that of Plonka and Salinger, for that matter) started with the text-book definition of '*two developers jointly working on one computer*'. Neither of these quantities is fixed in everyday industrial development, but screencast software, microphones, and camera angle are all set up for *two* developers on *one* computer. Developers may suddenly open their own laptop or interact with other developers who are

out of reach of the microphones and/or beyond the camera angle (as is noted by developer D4 in session DA2 after a third developer joins the pair: “*We need more microphones!*”).

Another impact of the research interest on the way data was collected can be seen in the sessions recorded by Plonka, who was initially interested in the *driver/navigator* metaphor (Plonka et al., 2011; Plonka, 2012, Ch. 6). In order to easily see who is in control of keyboard and mouse, the camera angle of sessions in companies C, D, E, and F centers on the developers’ hands and occasionally cuts off their faces. Sessions from companies A and B (recorded under the lead of Salinger, 2013), company J (by Schenk, 2018), and K, M, O, and P (recorded by me) focus on the developers’ faces instead.

In both cases, possibly relevant aspects of the PP process are not recorded making the reconstruction more difficult. Without all screen contents, technical information is missing and sense-making gets more difficult; without seeing the developers faces and viewing direction, interpreting their behavior is more difficult.

### ***Affecting Developer Behavior***

The second, related notion is that a *PP session* pertaining to a *task* is a meaningful unit a software developer’s workday. However, some companies (such as Q, with which I had contact for evaluation interviews only and did not record any PP sessions) form pairs independent of concrete tasks for multiple days on end during which the pair members indeed behave as *one*, taking coffee breaks together without really starting or ending a “session”. In contrast, the data collection procedure described above is session-centric. Questionnaires before and/or after the recording frame the session in two senses. First, they introduce a ceremonial start and end: In the beginning of sessions CA2 and EA1, the developers filling out the questionnaires was accidentally recorded: It takes them more than nine minutes, which might be an unnatural interrupt. Second, the pre-session questionnaire asked the developers to think about the work time ahead. In particular, the questionnaire asked for task classification and description, a characterization of the expected difficulties, and an estimated time to completion (see Figure 4.4). Although this may yield valuable context information for the researcher, it may impose an unnatural focus on the developers by making them think about aspects they would not have thought about had it not been for the researcher. In newly collected data, I therefore refrained from using pre-session questionnaires.

Another effect of session-centrism can be observed in multiple session recordings in various companies: Pair members are occasionally interrupted by their colleagues with technical or organizational concerns (as is also reported in the ethnographic study by Chong & Siino, 2006, discussed on page 69). A common and unnatural reaction of the pairs in the recordings is to send away the interrupter unsatisfied (as seen in session AA1 at 1:47:05) or to point out that they just walked into a recording (as seen in session KA1 at 04:30), as if the pair wanted to protect the integrity of the data collection and possibly their colleagues privacy.<sup>5</sup> The recordings in company E are peculiar in another way, which also possibly indicates an intention of the developers to protect the data collection: The work station for the session recordings was set up in a meeting room, such that the pairs worked completely secluded from the rest of their team.

Such effects appear more pronounced in the recordings that were done under the “Understand PP” headline (see Table 4.1 on page 146). For instance, the pairings in companies C to F do not appear to be holding up to the naturalistic ideal as only *one* of the overall 21

---

<sup>5</sup>Colleagues could only ‘walk in’ on a session recording because (a) in company K the webcam was not mounted on a highly visible tripod but on the monitor (unlike in companies C to F, see discussion of camera angles on page 156), and (b) the recording was done remotely with no researcher on-site to notice. This inadvertent ‘covert recording’ of uninitiated colleagues is an ethical concern that only occurred to me while writing this document.

pairs was recorded twice. Although Plonka (2012, p. 67), who performed the recordings, states that “*forming of the pairs was done during their daily meeting or spontaneously throughout the day*”, I suspect that this statement describes the *usual* pair forming, but not how it was done for the recordings, which looks more like carefully arranged pairings, even if possibly done by the developers themselves to help the researcher. Indeed, in the final report handed out to company C, Plonka (2009) mentions a list with a morning and an afternoon slot each day, into which pairs could write their names if they wanted to be recorded (see also discussion on page 148). In my own data collection in company P, the team at one point discussed how the next pair should be formed based on what I at the time understood as organizational constraints. However, when I asked about this in the next reflective interview, the developers revealed to me that they intended to give another, previously not-recorded colleague the chance to also benefit from the feedback I could provide.

In a sense, the employed data collection method may have interfered with a completely ‘natural’ formation of pairs. However, none of the pairs was asked by the respective researchers to work in a certain constellation or on a particular task, so the data collection can be characterized as reasonably naturalistic.

#### 4.3.4 d) Summary of Data Quality

Notwithstanding the above limitations of the data collection the session repository still comprises diverse, realistic, detailed data. At the time of writing, it contains 67 recordings from 13 different companies featuring 57 different professional software developers who worked together (mostly) co-located in 41 different constellations of (mostly) two members (see Zieris & Prechelt, 2020b, for more details). In these sessions, the developers worked on their actual industrial tasks for as long as they wanted, and in most cases also freely chose who to work with and when to start. The exception here are the 23 sessions from companies C to E, where the developers had to sign up for either a morning or an afternoon slot, and were possibly inclined to work with partners they would normally not pair with, expecting to learn something in the reflective interview. For the same reason, it can also not be ruled out that the developers in all companies worked in pairs more often for the recordings than normal.

All of the above concerns may affect the *frequency* of phenomena (such as more or fewer conflict situations, more or less easy tasks, or more or less fatigue due to longer sessions), but none of these appear likely to produce entirely artificial behavior. My qualitative research is not concerned with the frequency of phenomena, but the above considerations would need to be kept in mind by others for drawing conclusions that go beyond my own.

### 4.3.5 Selecting Data for Analysis

My analysis does not cover all available data; I discuss the reasons to exclude some sessions in Section 4.3.5a. I also collected additional data according to the principle of *theoretical sampling* once my analysis led me to not yet covered aspects, the phases of which I summarize in Section 4.3.5b.

#### 4.3.5 a) Excluding Data

I disregarded a number of sessions since they were beyond the scope of what I consider to be “pair programming” (see Section 4.2.3b), and others because relevant data was missing.

- **Sessions dealing with pet projects.** This excludes session CB1 of developers C4 and C8 which has nothing to do with company C’s domain or product. Similarly, I disqualified

both sessions from the L context in which the respective pairs go through tutorials but do not have a productive goal.

- **Programming groups with more than two developers.** This excludes sessions OA3 and OA4 (four developers) as well as OA9 and OA10 (three developers).<sup>6</sup>
- **Incoherent, mostly solitary activities.** In the sessions from company N, the developers were in the middle of an unstructured onboarding process and tried to configure their development machines to fit the corporate standard. Although they were sitting next to each other, they did not have a shared productive goal.
- **Sessions lacking the webcam feed or one of two screens in a dual-monitor setup.** This excludes sessions DA1, OA6, and OA7 (no webcam) as well as all four sessions from company F (missing second screen in dual-monitor setup).

#### 4.3.5 b) Theoretical Sampling

My process of theoretical sampling from the repository of available sessions and collecting new data from industrial partners can be divided into the following eight phases.

##### 1. Readily Available Data

I started my analysis pragmatically with the sessions that were readily analyzable as video files. When I began my work, these were sessions from three different companies, in particular, sessions BA1, CA1–CA5, and DA2. (Session CB1 was also available, but I excluded it because it is about a pet project only.)

##### 2. All Technical Contexts

For more diversity of backgrounds, I went on to make sure to have at least one session from each technical context. For a lack of a better criterion, I started with the earliest available session: In case later sessions pick up on the technical work of earlier ones, I could advance chronologically. This is how I selected sessions AA1, EA1, and JA1.

I also included JA2 (from the same context as JA1) because it was the first instance of a developer pair being recorded twice with several days in between. The second such instance were sessions BB1–BB3 featuring the same pair as session BA1. The according video files were originally considered corrupt by Salinger, but I was able to mostly recover them and included them in my sample.

##### 3. First Long-Term Engagement

In 2013, company K posed the first opportunity to record my own data. Here, most interaction with the subjects was done remotely (recording the PP sessions, administering the post-questionnaires, and conducting the reflective interviews). This was also my first attempt of a *long-term engagement* (see Section 4.3.3c): Overall, there were four recording dates with weeks and months in between. Developer K2 took part in all of them, first together with K1 in March, and then with K3 in May and again in October. I also presented some of my findings to a larger group of developers from the company in a post-workshop in November, which motivated another developer, K4, to record two sessions with K2 a week later.

For my analysis, I selected the first one to two sessions from each technical context: KA1, KB1, and KC1 & KC2.

---

<sup>6</sup>Some *included* sessions (such as DA2, OA2, OA8) temporarily involve more than two developers for a few minutes, but not throughout the session.

#### 4. *Non-German Context*

Up to this point, all analyzed sessions involved native German speakers in German companies. I spent three months in Oslo, Norway, getting in touch with the local software development community in order to collect more diverse data. I contacted 56 companies, and eventually established a single industry contact M where I recorded one session: MA1.

#### 5. *Reorientation?*

The sessions which I analyzed up to this point covered what I felt was a broad spectrum:

- There were different types of software development tasks (discussing requirements and design alternatives, planning work, implementing new functionality or tests, performing tests, exploring and demonstrating features, reviewing code, debugging, refactoring, changing configuration, and reading documentation),
- with software developers being at their company for many years or in their first week,
- with those who never paired before and others who do it regularly,
- with knowledge-wise homogeneous as well as heterogeneous pairs.

However, all sessions had one thing in common: The pairs *all* made steady progress, possibly with the occasional detour, but nothing that would throw them off the rails.

With my original goal in mind to identify beneficial and problematic behavioral patterns (see page 142), I briefly considered giving up my naturalistic stance of only studying pair programming as it actually happens. I considered looking for developers with either no or only bad pairing experiences and specifically ask them to work in pairs hoping to get to see what ‘bad pair programming’ looks like.

#### 6. *Second Long-Term Engagement*

Before I could actually get into planning such a data collection with industrial partners, the opportunity arose to observe pair programming at company O that claimed to do *all* software development in pairs. Expecting to collect data from real pair programming experts, I went to stay at the company on-site for four weeks in May and June 2016. I took my time getting to know the company and the developers; the first of four recording dates only happened after more than one week. To my surprise, the first two sessions OA1 and OA2 turned out to be very frustrating for the pair as they made virtually no progress—and not only for technical difficulty, but also because they apparently had trouble working as a pair under these conditions.

This was my second installment of a *long-term engagement*: I presented findings to the Scrum Master at the end of June 2016, and revisited the company in August 2017 to talk to developers, Scrum Masters, and the Product Owner about my consolidated findings (*evaluation with practitioners*, Section 4.3.3d). I provided the Scrum Master with a manuscript of my findings and he made it a required reading for at least two developers before they left the company.

In my analysis, I included all sessions from company O that were not excluded for any of the above criteria, i.e., OA1, OA2, OA5, and OA8.

#### 7. *Evaluation (1): Similar Context*

Since it was not possible to kick off another round of data collection for evaluating my findings in company O, I looked for further opportunities and found them in company P. After getting in contact with the Scrum Master in October 2017, I started my interaction with developers in February 2018 with a presentation of my findings to check them for resonance (*evaluation with practitioners*, see Section 13.2.2). Received with some enthusiasm, I later stayed one week in June 2018 to record four sessions PA1–PA4 with developers who had been to my presentation and with whom I discussed their upcoming and their previous sessions in terms of my grounded concepts (*evaluation with practitioners* again, see Section 13.2.2).

### 8. Evaluation (2): Consulting Sector

All companies up to this point developed their own software, meaning that the technology stack and aspects of the application domain are relatively stable for the developers—with individual developers forming an exception, such as D4 who was in his first week when he was recorded, or hired consultant O4. I therefore got in contact with consulting companies Q and R where I did not record any pair programming sessions, but discussed my findings and probed for relevant differences between in-house development and the consulting business. I had a one-hour interview with a technical manager from company Q in January 2019; in company R, I performed a workshop in March 2019 similar to the one I did at company P. I discuss the outcome of these as part of my evaluation in Section 13.2.2.

## 4.4 Case Descriptions

Through the data selection process described in the previous section, I ended up analyzing 27 PP sessions or *cases*. Brief characterizations can be found in Table 4.3. While the results presented in the main part were derived from the whole set and I present in-depth excerpts from most of the sessions to illustrate the nuances of my concepts, the main concepts can be explained using only a handful of sessions. Since I recur to these five ‘main cases’ throughout my thesis, I discuss them here in one place. The other 22, I characterize in less detail in Appendix C.

### 4.4.1 AA1: Complementary Frontend and Backend Knowledge

#### *Company, Pair, and Software System*

Company A develops a web-based Content Management System (CMS). The system has two major components: There is an Objective-C backend (also called “CM” or “Kernel”) which deals with business rules and SQL database interaction; and there is a Java frontend (also called “GUI”) which interacts with the backend through an XML API and renders HTML output to be displayed in a web browser.

Both developers A1 and A2 know their domain well and are generally experienced developers. Developer A1 has been a software developer for 10 years, the last 4 years at company A; he knows the structure and individual classes of the Java frontend well and is familiar with the Eclipse IDE and the Java programming language. His colleague A2 started professional software development 7 years ago at company A; he is more familiar with the backend and the SQL database, the VIM editor, the UNIX shell, and the Objective-C programming language. However, each of them would also be able to work in the other part of the system. Both developers have been pair-programming regularly for two years.

#### *Session*

The pair wants to resolve inconsistencies between four list views. Each list displays CMS entities of a certain type (such as *task* or *unreachable link*) which can (among other things) be “active” or “inactive”. In the list views, an entity’s icon and label should reflect its activeness, but in some cases the icons did not match, sometimes the labels. The pair’s session can be divided in five phases (2:22 hours in total):

- They understand and fix the implementations of three lists in the frontend (42 minutes).
- They understand and fix the implementation of the forth list which also involves changing the backend and the XML API (40 minutes).

ID	Start Time	Length	Pair	Session Content
<i>Company A: Content Management System</i> (Java, Objective-C, SQL)				
AA1	2007-01-26 13:43	02:22	A1 A2	Fix five similar bugs touching both frontend & backend
<i>Company B: Social Media</i> (PHP, JavaScript, SQL, HTML, CSS)				
BA1	2007-09-14 13:38	01:46	B1 B2	Read foreign code, implement cache, discuss specification
BB1	2007-04-27 13:25	01:21	B1 B2	New feature from scratch (template); discuss requirements
BB2	2007-04-27 16:51	01:51	B1 B2	↳ impl. model, controller, template; discuss requirements
BB3	2007-04-27 18:58	01:32	B1 B2	↳ implement template, controller; discuss requirements
<i>Company C: Graphical Geo Information System</i> (Java)				
CA1	2008-05-05 13:27	01:18	C1 C2	Implement new form in GUI (C1 already started)
CA2	2008-05-07 11:36	01:24	C2 C5	Architecture discussion (C5 already started), refactoring
CA3	2008-05-07 15:34	02:10	C6 C7	Implement context menu entry, incl. test case & refactoring
CA4	2008-05-08 10:25	01:34	C4 C7	Implement selection feature w/ special key-binding
CA5	2008-05-09 10:32	01:23	C3 C4	Implement feature to split graphical elements
<i>Company D: Estate Customer Relationship Management</i> (Java, XML)				
DA2	2008-10-08 10:12	02:24	D3 D4	Planned feature impl., turned to widespread refactoring
<i>Company E: Logistics and Routing</i> (C++, XML)				
EA1	2008-10-27 11:29	01:17	E1 E2	Step-by-step debugging of display error in the GUI
<i>Company J: Data Management for Public Radio Broadcast</i> (Java)				
JA1	2013-01-31 14:05	01:07	J1 J2	Walkthrough of J2's code, discuss possible refactorings
JA2	2013-02-13 10:51	01:15	J1 J2	Review of J2's new API, define requirements
<i>Company K: Real Estate Platform</i> (Java, SQL, CoffeeScript)				
KA1	2013-03-14 10:37	01:59	K1 K2	Dev. env. setup, discuss inter-system API design, 1st impl.
KB1	2013-05-02 13:45	00:53	K2 K3	Add new class to model, write and debug database migration
KC1	2013-10-29 11:24	00:59	K2 K3	Test env. setup, discuss test approaches for GUI feature
KC2	2013-10-29 12:59	02:01	K2 K3	↳ trying diff. test approaches, struggling w/ debugger
<i>Company M: Data Analysis in Energy and Transportation Sector</i> (SQL)				
MA1	2014-10-16 11:42	00:25	M1 M2	Explanation of table model
<i>Company O: Online Project Planning</i> (CoffeeScript)				
OA1	2016-06-01 10:51	01:24	O3 O4	Understand foreign component, try to read state for testing
OA2	2016-06-01 13:27	01:32	O3 O4	↳ try to set up (parts of) component for testing
OA5	2016-06-08 17:11	01:09	O1 O3	Bug fix: amend test cases, refactor prod. code, fix the bug
OA8	2016-06-15 13:47	01:16	O3 O4	Failing test: Investigate prod. and test code, correct mocks
<i>Company P: Online Car Part Resale</i> (PHP, SQL)				
PA1	2018-06-05 11:24	00:58	P1 P2	Walkthrough of DB migration (written by P1), discuss req.
PA2	2018-06-05 13:35	01:30	P1 P2	↳ test of migration, debugging, refactor test cases
PA3	2018-06-06 12:23	01:31	P1 P3	Implement new API endpoint w/ tests (P3 already started)
PA4	2018-06-07 11:09	01:42	P1 P3	↳ implement DB access with OR-mapper

**Table 4.3:** Context and characterization of analyzed PP sessions. Some sessions continue earlier ones (↳). The *start time* is the reference point for the timestamps given in the examples in this document; the *length* column states how many hours and minutes later the pairs concluded their session. Sessions MA1, OA1, OA2, OA5, and OA8 are in English, all others are in German. Developers C4, C6, and O3 are female, all other are male. The respective pairs are temporarily joined by additional developers in sessions DA2 (first 5 minutes by D7, then 12 minutes by D6), KA1 (8 minutes by K4), OA2 (14 minutes by O6), and OA8 (17 minutes by O1).

- The pair discusses a second inconsistently rendered property (whether entities are “mirrored”), but decide to not address it now. They write down instructions for a manual “click test” for their changes (19 minutes).
- They discuss when to address which open issues and discover a fifth inconsistent list type. They fix it completely in the frontend and partially in the backend (18 minutes).
- Finally, they discuss multiple design options to address the fundamental problem which precludes the fifth type from being fully fixed in the current architecture (23 minutes).

Along the way, they discuss the graphical interface with their product manager, help out a colleague with her problem, and directly address smaller design issues with a number of refactorings. The session as a whole is characterized by the two developers combining their complementary frontend and backend knowledge to build up a common understanding of the bugs.

#### 4.4.2 CA2: Undiscussed Design Rationale

##### *Company, Pair, and Software System*

Company C develops a graphical geo-information system. The software is written in Java and makes heavy use of object-oriented design, resulting in class names such as *FeatureLayerAttributeTableCellRendererFactory*. Developer C2 has been a software developer for 9 years with more than 8 years at company C; his colleague C5 has 20 years of experience and joined the company 2.5 years ago.

##### *Session*

The software has two variants (“Basis” and “Pro”) which involve three components (*basis* and *pro*, as well as a common *core*). The relevant classes deal with a table-based dialog that allows to configure attributes of graphical map elements. There are both standard and calculated (“virtual”) attributes. The task is to add some table-related logic to the Basis version which is generic enough to also work with Pro elements.

The developers discuss two approaches: C2 favors a simple solution, C5 already began with a more indirect design. C5 moved an interface from *pro* to *basis* prior to the session (where it can refer to other *basis* and the *core* components), and left an implementing class in the *pro* component, which necessarily makes his design more complex. C2 mistakenly considered the implementation to be in the *basis* module already and made his simpler design proposal on these grounds. They work on the task for 1:14 hours:

- C5 presents his recent changes, the pair discusses their design options and they decide to move the class to the *basis* module to follow C2’s simpler design (13 minutes).
- They perform the move operation and adapt involved interfaces and classes (21 minutes).
- They perform some manual tests, correct a small number of newly introduced defects, and commit their changes (27 minutes).
- In the end, they discuss how to proceed with the overall task. During this discussion, C5’s original (complicated) design resurfaces for which he appears to have some other rationale which he is not able to explain to his partner (13 minutes).

Throughout the session, the developers have a similar understanding of the software system but do not fully discuss all the discrepancies that come up.

### 4.4.3 DA2: A New-Hire's Successful First Session

#### *Company, Pair, and Software System*

Company D develops a large customer-relationship management system that is based on Eclipse and comprises about 50 top-level modules written in Java. D3 has been with the company for three months and has been pair programming since then. It is his first programming job for which he started learning Java. D4 is in his very first week, started professional software development one year ago, but has never pair-programmed before.

#### *Session*

The originally planned implementation of a new feature (a toolbar in the calendar module of the application) turned into a small but widespread refactoring during which D4 (despite being a new-hire) explains many things to D3. Their session of 2:23 hours has six phases.

- D3 explains the rough structure of the system and D4 asks many questions about various system and company aspects, some related, others unrelated to their task (22 minutes).
- The pair has trouble finding the right place to start, they bring more experienced colleagues into the session, first D7 and afterwards D6. New-hire D4 and senior D6 together identify a design flaw that precludes straightforward design and agree on extracting an abstract superclass from a number of implementation classes using the Template Method design pattern (Gamma et al., 1995, pp. 325–330) after which the senior D6 happily leaves the session (17 minutes).
- The pair introduces an abstract class to be subclassed by a number of existing implementations (32 minutes, including a 13-minute stand-up meeting with the team).
- After some changes the pair notices that the existing implementations are spread across different modules and they struggle to find a place where their new class can be accessed by all of them (18 minutes).
- For the main part of the session the pair goes through almost 30 classes and execute the same refactoring steps again and again. Along the way, new-hire D4 provides explanations on various programming techniques and technologies (47 minutes).
- They conclude with a successful manual test of the application, commit their changes, and discuss future design ideas (7 minutes).

After some turbulent setup period, the session is characterized by repetitive work during which the new-hire learns enough about the system to keep going while teaching his colleague about software development in general.

### 4.4.4 JA1: Pair Review with Domain Expert and Programming Expert

#### *Company, Pair, and Software System*

Company J develops and runs their software system to aggregate recordings of news segments from a number of radio stations. Developer J2 started professional software development at company J 2.5 years earlier; he is the main author of the software. Developer J1 works at a consulting firm that regularly supports company J; he has been a software developer for a little over six years and has been working in this particular context for eight months. Before that, the two used to work together in the same office (but on different topics) for two to three months until J2 moved to a different city. They have been using distributed pair programming<sup>7</sup> for about half a year.

<sup>7</sup>They use Saros (see <https://www.saros-project.org/>) which allows them to each work on their individual machine with full control over their input devices. Source code changes are automatically synchronized in real-time; the optional “follow mode” mirrors one developer’s scrolling and file switching to the partner’s IDE.

**Session**

Experienced consultant **J1** is invited by **J2** to help with refactoring a software system which **J2** wrote several months earlier and which is unknown to **J1**. The session lasts 1:07 hours:

- **J2** explains the overall structure of the software system: Each radio station provides a file server access to where the live recordings of the news segments are continually written. The processing per radio station then consists of determining the moment when a news segment ends, fetching the finished recording, and eventually converting it to a different format (8 minutes).
- The pair goes through the monitoring implementation for one such radio station: It consists of a single method of 100 lines and 11 if-statements which are nested up to five levels deep. They realize that the class basically implements a state automaton, and **J1** learns that each radio station needs a slightly different one (22 minutes).
- They attempt to extract parts of the long function into subroutines, but understand that their plan does not work because of the control flow (35 minutes).
- With nine additional station-specific situations and a planned functional extension they conclude to rewrite the whole module from scratch in the future (2 minutes).

The developers combine their respective knowledge in the application domain and regarding software development in general.

**4.4.5 OA1: The Impossible Task****Company, Pair, and Software System**

Company O is a self-proclaimed “*all-PP company*” which develops a web-based project planning tool. The developers are a mix of directly employed personnel and a few hired consultants and freelancers. They work with a newly introduced technology (React and Redux) which all of them need to get familiar with. **O3** joined company O a month ago and has been a software developer for one year. **O4** is a hired consultant with more development experience; he joined the team five months earlier.

**Session**

The session revolves around a dynamic browser-based form: Interaction with one of its parts may reveal new form parts or alter existing ones. A new feature pre-selects a specific value in a drop-down menu. Developers **O3** and **O4** are tasked with implementing a test case to check whether the pre-selection is effective. Their session spans 1:24 hours.

- **O3** and **O4** try to understand the structure of the fixture that is created in an existing test case in order to find the property that contains the initial form field value so they can write an assertion for it. Their means of inspecting the fixture is to insert `console.log` statements into the test case and execute it. Each test run takes between 2.5 and 4 minutes, so in the meantime they (a) prepare the next iteration of `console.log` statements in the test logic, (b) work on two machines (**O3**'s computer and **O4**'s laptop) to look for ways to make the output more informative, and (c) inspect the rendered form in the web browser. They complete seven iterations of `<log statement>` → `<test run>` → `<output>` without much insight, before they decide to take a break (48 minutes + break of 11 minutes).
- They concentrate more on reading and understanding the existing test and production code and formulate ideas about where the initial value might be stored (19 minutes).
- They perform an eighth `console.log`-iteration to test their hypothesis. The test run fails due to a compilation error and they decide to take a lunch break (6 minutes).

The session is frustrating for both developers as they have little applicable knowledge and effectively make no progress.

## 4.5 Analysis Method

My research is about what actually happens during pair programming rather than what developers say and think about pair programming, so I primarily analyze the recorded PP sessions; all other data sources (reflective interviews, questionnaires, ad hoc interviews) are auxiliary. The building blocks of my analysis method are the following:

- From Straussian Grounded Theory Methodology (as described in Section 3.3):
  - *Theoretical sampling*: Collect relevant data to inform theory development.
  - *Open coding*: Understand individual concrete events and find conceptualizations.
  - *Axial coding*: Identify strategies applied under certain conditions based on an action-oriented model involving causes and consequences.
  - *Selective coding*: Integrate theory and systematically consider context properties.
  - *Constant comparison*: Achieve consistent and densely connected concepts through coding data not once, but revisiting it when concepts change.
  - *Memo writing*: Capturing observations, ideas, and insights in written form early on and throughout the analysis process.
- Salinger et al.’s practice of defining a *perspective on the data* (see Section 3.4.1a).
- Salinger’s *base layer* (see Section 3.4.2) for making sense of the raw data.

In this section, I describe how I applied all of these (except for theoretical sampling, which I already described in Section 4.3.5b).

### 4.5.1 Perspective on the Data

The *perspective on the data* is a practice that amends the Grounded Theory Methodology. Salinger et al. (2008) proposed to define and adapt such a perspective over time to deal with the difficulties of applying Straussian *open coding* to video recordings of pair programming sessions. It consists of answering three questions:

1. “*In which respects do you expect the data to provide insight?*”  
This defines the research interest and helps setting a filter to not go through all available data with the finest granularity possible. I discuss my perspective in Section 4.5.1a.
2. “*What kinds of phenomena do the researchers allow themselves to identify in the data?*”  
This determines the researcher’s epistemological standpoint. I discuss mine in Section 4.5.1b.
3. “*What type of result do you want the analysis to bring forth?*”  
I already discussed this in Section 4.2.2: I want to develop a vocabulary and formulate behavioral patterns to enable practitioners to reflect on their pair programming processes.

#### 4.5.1 a) Area of Interest, Focus

The general area of interest for which I expect to gain insights from studying recorded sessions is how pair programmers deal with what they do and do not know (see Section 4.2.1). Taking a perspective on the data and thus limiting one’s attention to certain phases and aspects of a PP session (as opposed to a ‘complete’ analysis) is similar to choosing where to look at and what to look for during a field observation. The difference is that the PP session is already recorded and setting such a focus is therefore not final. As proposed by Salinger (2013, p. 110, see also Section 3.4.1a), I reexamined and adapted my perspective along the way.

For my initial perspective, I followed the proposal of Salinger & Prechelt (2013, p. 30) which is to start a study on knowledge transfer with phases where the two developers explicitly deal

with pre-existing knowledge, e.g., in dialogs consisting of pairs of *ask\_knowledge* / *explain\_knowledge*. As illustrated in Figure 4.1, knowledge transfer activities are not generally limited to the beginning or the end of a session, but occur throughout.

I amended my perspective over time, as I described in Section 4.3.5b on my theoretical sampling process. Each of the result chapters 6 to 11 takes a different perspective: The general quality of the pair process, the practical meaning of *knowledge* in a PP session, and the general activities pairs engage in to transfer knowledge on four levels of granularity. (Note that the order of the chapters does not reflect the sequence in which I changed my focus; see Chapter 12 for the actual timeline.) In my analysis, I mostly focus on the individual developer in a pair situation, but also on the pair as a whole dealing with a software development task. Although it was not originally planned, part of my research was also concerned with the conditions under which either notion of *a pair* (two individuals vs. one pair) makes more sense (see Chapter 6).

#### 4.5.1 b) Epistemological Stance

I base my analysis on the developers' observable behavior that is recorded in the video material which I interpret based on:

- my own understanding of human communication in general and the German and English language in particular,
- personal interaction with the developers (as recorded in the field notes and the reflective interviews),
- my own theoretical and practical knowledge of software development, and
- reconstructed technical properties of the technical environment of the developers.

I assume the existence of an 'objective reality' when it comes to the technical aspects of the software system the developers work in. It has certain verifiable properties, which can be tested by the developers (which they sometimes do), but not by me.

I also acknowledge, however, the existence of 'subjective realities' which each of the developers constructs individually. For example, to a developer, 'My partner just asked me a question' may be true, but this is not an objectively verifiable truth like 'This file has 102 lines'. There is, however, a *shared reality* of the pair members which they construct and maintain together, and its truths matter for their communication (see also Section 3.2.1a on the notion—and fiction—of *common ground*). Of course, as an interpreting observer of a pair's actions, I construct my own reality as well, but I do not take part in their shared reality.

In concrete terms this means that the developers may *think* they understand each other where they actually do not, and I may *think* I understand the developers but actually do not, as well as the opposites and any combination of these. For me, there is no way around this but to state that these types of misinterpretation may happen, but that I deem them unproblematic for two reasons. First, I combine the above 'data sources' to resolve ambiguities, e.g., in making sense of a cryptic developer utterance by reconstructing some technical detail based on the video material or my own developer experience. I discuss such reconstruction work in Section 4.5.2c below. Second, I usually do not rely on single occurrences for developing a concept such that the effect of a few of them being 'wrong' is small.

Through constant comparison, instances in the data are not coded once and for all, but are revisited whenever a concept is amended in some way. Occasionally, revisiting an already coded segment led to a better understanding of the particular situation, e.g., in terms of intervening conditions that could better explain why the developers behaved in one way or another (see also Section 3.3.3b on axial coding). During my analysis, I wrote an operational memo for one rare instance where my understanding changed in a meaningful way (see Appendix C.1.2).

## 4.5.2 Coding

In a qualitative analysis, *coding* means to make sense of the continuous stream of reality, to think of a segmentation, to assign labels to capture meaning, and to consolidate and rework the labels according to some procedure. Here, I describe how I practically applied the coding procedures proposed by Strauss & Corbin (1990) and the details of my sense-making approach.

My overall analysis process exhibited two trends: The general trend of data analysis starting out as exploratory and getting more and more focused on the one hand (*ibid.*, p. 219, see also Section 3.3.2a), and my change in perspective from knowledge transfer activities on a small and on a larger scale on the other (see Section 4.5.1a). The first analysis steps and new observations later in the process were more exploratory in nature and based on building understanding for concrete situations in a bottom-up fashion, starting from individual utterances. Later phases were more confirmatory and aimed at integrating my theory on all granularity levels. My general approach to coding a PP session was as follows:

1. I watched the session as a whole to get an overall impression. For fresh data, this happened during the quick analysis as part of data collection protocol described in Section 4.3.2d. Here, I did not yet focus on knowledge transfer phenomena.  
I performed the analysis directly on the video material, and not on a transcript. I discuss the issues of transcribing and translating in Section 4.5.2a below.
2. For interesting, puzzling, or just ‘different’ phases, I performed *base coding*, i.e., I applied the base concepts to reconstruct the developers’ intentions and get a better understanding of what the pair is doing. I describe this process in Section 4.5.2b below.
3. With a basic understanding of what the pairs do, I focused on the parts of the session during which knowledge or the apparent lack thereof played a role and switched to *open coding*, which I did on different granularity levels: On the utterance level of the *base activities* for which I developed new properties, and also in the form of new concepts covering larger segments up to PP sessions as a whole.
4. Reconstructing the PP sessions’ technical background was sometimes necessary for open coding in order to make sense of the developers’ behavior, but more often for *axial coding* for which I encountered the following limitations due to the nature of my primary data (see also Sections 3.3.3b and 3.3.3c on the *paradigm model* and *conditional matrix*):
  - The *causes* of the phenomena the developers deal with may be traced back up to the session start, but only seldom to larger contexts such as the team level.
  - *Strategies* can only be seen in the form of actual behavior and not in the form of merely considered-but-not-executed alternatives. In cases where pairs deal with a phenomenon over time, multiple failed attempts might hint at the variation. On rare occasions, when the recording researcher noted interesting behavior already during the quick analysis, the developers possibly offer some introspection during the reflective interview.
  - *Intervening conditions*, such as biographical aspects that enable or prevent the developers from choosing some strategies, can be sometimes distilled from considering behavior of the same developers in other sessions with a different task and possibly a different partner. This allows to get a better understanding of what ‘normal’ behavior for each developer is.
  - *Consequences* are, like the causal conditions, limited to the scope of the session recording. Again, the reflective interview may contain additional information—if it occurred to the recording researcher to ask according questions.

Despite these limitations, the paradigm model was helpful as its underlying action-oriented perspective provides a clarifying lens (see also Charmaz’s pragmatic view on this matter,

discussed on page 126). In Section 4.5.2c, I explain how I approached the sometimes necessary reconstruction of (objective) technical knowledge and the developers' (subjective) understanding.

5. With *selective coding*, tracking the individual developers' knowledge levels throughout a session became relevant and I also reflected more on what means to *know* something, for the subjects (see Section 4.5.2d below) as well as me as a researcher (see Section 4.5.1b on my epistemological stance above).

Throughout my coding process, I often switched between the perspectives described above. During this process, I only kept concepts meeting the following criteria:

- **There is a satisfactory operationalization.** This excludes concepts which cannot be pinned down to the recorded developers' in-session behavior.

The pair's *session goal*, for instance, may appear like a potentially relevant concept, but my available data does not allow for a rich concept. Many pairs simply do not discuss their goals, some may not even have anything goal-like, and others may have already discussed their strategy prior to the recording. Without the necessary grounding, such a concept would remain vague.

- **The phenomenon is relevant for the PP session.** Not everything that happens during pair programming carries the same weight.

One example of a discarded concept-candidate is the *scope qualification* which marks the difference between a given explanation referring to, say, 'some' or 'most' of the things. Such a difference matters in a strict logical sense, but, as it turned out, not so much in the verbal communication in a pair programming situation.

- **It is relevant for my research interest.** This excludes observations for which I did not see a potential to get a deeper understanding of knowledge transfer in pair programming. I do not have any concrete examples from my research for this criterion, because my general research interest did not change much and I did not consider concept-candidates outside of this interest.

I discuss the resulting concepts and my findings in Chapters 5 to 11.

#### 4.5.2 a) Translating and Transcribing

In my research, two natural languages are involved: German as the mother tongue of many of the recorded developers and myself, and the language in which most of the recorded sessions were held; and English as a language that some developers chose for their sessions, and the language in which I present my research results. I am generally confident to have a good enough understanding of the languages spoken by the developers.

In this document, any originally English utterances are presented verbatim. I translated all German utterances into English while attempting to reproduce the information density, order and length of the information pieces contained in the utterances, as well as grammatical (in)accuracy. In general, the two languages are grammatically similar. There are, however, a few small, but relevant differences which I discuss in the appropriate place as part of my results (see Section 8.2.5). Overall, I feel confident about my German-to-English translations; the original German transcripts for all examples can be found in Appendix C.

All analysis steps were performed directly on the video material. Based on the excerpts I *did* transcribe, I estimate the full transcripts of a PP session would fill between 20 and 30 pages per recorded hour and would then only include the verbal channel, but neither the developers' facial expression, gesture, and posture, nor the screen contents. Just as Plonka (2012, pp. 72–73) and Salinger (2013, p. 106) before me, I decided to not transcribe whole sessions. Strauss & Corbin (1990, pp. 30–31), too, propose to transcribe only as much as needed. For me, there

were three reasons to transcribe a passage: (1) to save the effort of reunderstanding individual utterances with unclear pronunciation or high background noise when revisiting the same material later; (2) about a dozen times to get a better understanding of the precise order of overlapping turns, i.e., who refers to what at times when both developers talk at the same time; and (3) to prepare the examples contained in written reports such as this thesis.

Speech is not written language, as both have features that the other lacks (punctuation marks, pitch, tempo, etc.). Consequentially, there is not *one* single way to transcribe spontaneous verbal speech. I roughly follow the transcription scheme which Salinger (2013, pp. 451–459) developed. Its full notation as I use it can be found in Appendix B. In particular, my transcriptions have the following properties:

- The transcript mostly captures the developers' speech as a string of words that is grouped to form sentences with standard punctuation, such as comma, full stop, and question mark. Occasional additions are relevant human-computer interactions such as `<*clicks button*>` and para-verbal expressions such as `<laughing>`.
- The transcript has a logical turn-by-turn structure even if there was some cross-talk in the original dialog (see also Section 3.2.1b on *turns*). `Developers` and their `utterances` are color-coded consistently throughout this document.
- Identifiers in the source code that are clearly referenced in an utterance are set in monospaced font. Similarly, I highlight the `emphasis` of an utterance when the same sequence of words could have a different plausible meaning without it.
- Stuttering, minor repairs, prosody, tempo, and non-standard pronunciation are not transcribed as long as they appear to carry no special meaning to the speaker or the partner.
- Meaningful pauses are set in parentheses with one dot per second as in this two-second pause (`(. .)`). Human-computer interaction without concurrent speaking is marked by commas as in these five seconds (`(, , , , ,)`).

#### 4.5.2 b) Base Coding: Reconstructing Intended Meaning

In interview-based qualitative studies, subjects speak to the interviewer about things they are asked about and/or that are relevant to them. If need be, the interviewer may ask follow-up questions or ask the subject to elaborate on some issue. A recorded pair programming session, in contrast, proceeds without such outsider-oriented communication (just as any other direct observation, for that matter). The developers themselves set the level of detail with which they go into things. They only need to understand one another, and do not care about any researcher for the most part.

During *open coding*, the GT researcher is to label a phenomenon as to '*what it is*' or '*what it represents*' (see Section 3.3.3a), which may be clear for an involved party such as an interviewer or programming partner, but not necessarily for an observing researcher. Whenever it was not clear to me what happened at a point in a PP session, I applied what I call *base coding* as a step before open coding.

When Salinger & Prechelt (2013) formulated the idea of a *base layer* for qualitative PP research, they could not know how other research would build upon that foundation. For me, the base layer, i.e., the base concept set and its rules, turned out especially helpful for open coding in that it supported my *theoretical sensitivity* (see Section 3.3.3e). Applying the base concepts helped me reconstruct what the developers were actually doing in that a segmentation of the data emerges through the application of the intention-revealing base concepts. This process is not unlike what Charmaz (2006) describes as a benefit of word-by-word and line-by-line coding: It helps working out the *meaning* of words the subjects chose and it helps

uncovering tacit assumptions (see discussion on page 126). I will now explain what I mean by that.

Recall that each base concept represents a type of intention behind a software developer's action in a pair programming situation. The base concept set is a 'list' of things a developer *could* mean. With a selection of a few plausible annotations in mind, the base layer provides rules for making many common distinctions. An utterance like "*We could simply convert that thing to a normal id*" (developer B2 in session BA1 at 15:30) could be both a *propose\_step* or a *propose\_design* since it refers to a technical detail (*design*) but also implies a unit of work (*step*). The base layer rules say that the central aspect should be coded (BL, Sec. 6.3.2). As a researcher, I now have a specific question to answer which brings me closer to understanding what is happening in the scene: Is the pair in the middle of planning their process (what to do, in what order), or are they defining the design (what it should be like)?

I did not use the base layer as a coding scheme to get to a somewhat complete annotation of a PP session, but to find helpful angles to look at the data. Refer back to Section 3.4.3 where I illustrated how coding with the base concepts (that is, coding-as-a-process, not coding-as-an-outcome) helped reconstructing that the 'expert' developer D2's actions were very likely *not* meant to teach his 'novice' partner D8 something about writing tests, but to express uncertainty regarding his own proposal.

#### 4.5.2 c) Reconstructing Technical Information and Subjective Understanding

In some instances, a developer's utterances made little sense to me (and also appeared to have confused their partner) such that I could not figure out their intentions. According to Grice's maxims, such obscurity is usually not intentional, but points to misunderstanding or some completely different communicative purpose (see Section 3.2.1b). When I suspected a misunderstanding (either on my part or between the pair members), I started reconstructing pieces of task-relevant knowledge as well as whether each of the pair members knows them at every point in time during the PP session.

In Example 4.2 below, I give a concrete case of such a technical reconstruction which took me more than one day of concentrated work. It was necessary to make sense of the developers' utterances. In the beginning, I noticed some confusion between the two developers in session CA2, e.g., with developer C2 asking "*Is that so? Do we need this at all?*" and C5 stammering "*I did (!...!) we have, we do need*" and admitting "*I don't know what you are talking about right now*", but I could not pinpoint their actual problem. Since I wanted to provide practitioners with relevant insights into pair programming, I figured that dissecting the pair's confusion might be a learning opportunity for me.

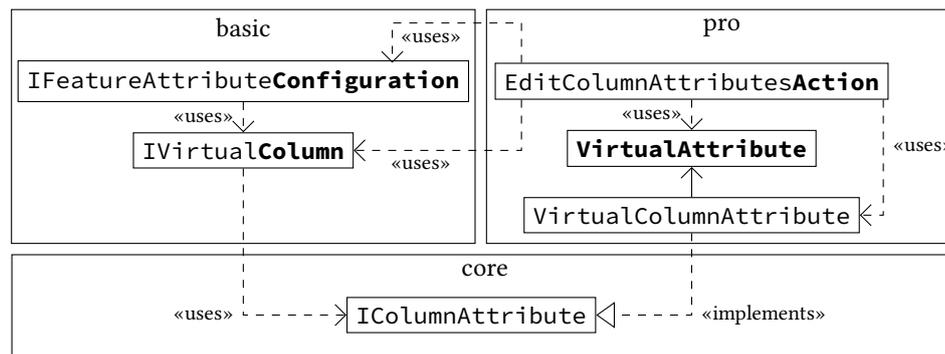
This particular reconstruction helped me, first, to understand parts of the software architecture and, second, that although it first appeared as if C5 knows less about the system ("*I don't know what you are talking about right now*"), it was in fact his partner C2 who was mistaken about parts of it. Eventually, the analysis of such episodes led me to the characterization of different Modes of knowledge transfer (see Chapter 9) and to the distinction between a latent Knowledge Need and an acute Knowledge Want (see Sections 7.2 and 11.2).

Note that the contents of the example box may be difficult to understand. After all, it is the essence of a conversation between two experts who need ten minutes to detect and clear up a subtle misunderstanding about a complex software system.

**Example 4.2: Knowledge Reconstructed From Interactions (CA2, 10:07–20:21)**

This reconstruction comes from the beginning of session CA2. Concrete evidence from the session is set in parentheses.

- The software has three subsystems, which I here call `basic`, `pro`, and `core`.  
(The pair refers to all three identifiers and these correspond to package names of the Java classes they review and edit throughout the session. This appears to be common knowledge for the pair.)
  - The relevant classes deal with a table-based dialog to configure attributes of graphical map elements. There are standard and calculated (“virtual”) attributes.  
(They manually test this dialog. Both refer to the “virtual attributes” like to common ground.)
- The following UML diagram represents the involved classes and their relationships:



The goal is to have the Configuration use the VirtualAttributes. This cannot be done directly as the `basic` module cannot access the `pro` module.

(C5 opens the four source code files with bold names between 10:08 and 13:50 allowing the above UML reconstruction even though neither developer talks about the relationships explicitly.)

- C5 already worked on this task. He added the Column interface and changed the Configuration and the Action, all of which is unknown to C2.  
(At 10:08, C5 says, “I better show you what I did already,” and C2 asks to see the new interface; from 14:26 to 18:12, C2 reads the source code of the changed Action class.)
- C2 understands the usage of the commonly accessible IColumnAttribute, but thinks it is too complicated. He proposes to have the Configuration depend on VirtualAttribute directly.  
(At 10:58, he asks, “Do we actually have a ColumnAttribute here? [...] Do we need this at all for the visualization in the attribute table?”; at 12:30, he says “I would have done a setVirtualAttribute in the FeatureAttributeConfiguration”; at 18:15, he continues, “I’d go the easy way and get the VirtualAttributes directly from the AttributeConfiguration.”)
- C5, however, tries several times to explain that the current complicated design is due to *not* wanting to move the VirtualAttribute from the `pro`-part of the system.  
(He refers to some agreements with another colleague at 10:16 and 11:29, and again at 1:14:17; he attempts to explain his design five times between 10:48 and 19:35.)
- C2 mistakenly deemed VirtualAttribute to be in `basic` and then realizes that it actually lies in `pro`. In his alternative reality, C5’s design would indeed have been too complicated and C2’s proposal of a direct dependency would be reasonable.  
(At 20:20, the pair clears up the misconception: “The VirtualAttribute is here in pro.”—“In pro is the VirtualAttribute?”—“Yes.”—“Ah, yes. I left it there, because we did not need it anywhere yet.”)

The reasons *why* C5 did not want to move the VirtualAttribute probably lie somewhere in the discussion he had with the other colleague. However, that part of C5’s explicit knowledge cannot be reconstructed from the pair’s in-session activities.

#### 4.5.2 d) Conceptualizing Knowledge

To analyze knowledge transfer in pair programming, it is necessary to have some notion of what it means for the developers to *know* something. I do not administer tests or ask developers to self-assess what they know, but rely on the session recordings as my primary source for determining what the developers do and do not know. More precisely, for the individual topics that come up over the course of a PP session, I infer what each of the pair members already knew in the beginning of the session, what she learns or understands along the way, and what remains non-understood until the end. There are a number of ways to do this:

- With some base activities, in particular the universal concepts (see page 134), developers directly address what they know and do not know:
  - An *ask\_knowledge* indicates the developer lacks some knowledge (and expects her partner to possess it), while an *explain\_knowledge* indicates the speaker possesses knowledge (and expects her partner to lack it).
  - The reaction to an explanation can also give clues: The developer who receives an *explain\_knowledge* may make clear she already knew (an *agree\_knowledge* of type *known*). Alternatively, acknowledging the *explain\_knowledge* (e.g., through an *explain\_standard of knowledge<sub>AT</sub>*) or simply *not* rejecting it indicates the developer lacked knowledge before.
  - A known lack of knowledge (which is *meta-knowledge*) may also become visible: Occasionally, pair programmers express uncertainty in a *propose\_hypothesis*, or surprise by new discoveries with an *explain\_finding*, or talk about their knowledge level explicitly with *standard of knowledge* and *gap in knowledge* activities, e.g., in a reaction to question they cannot answer.
- In addition to more or less concrete base activities, a developer’s lack of knowledge in a relevant area may become visible through confusion, difficulty in assessing the partner’s proposals, etc. (details follow as part of my results in Chapter 6).
- Topics that resurface multiple times indicate that the earlier exchanges may not have been sufficient for the developer to acquire and retain the respective knowledge.

With the epistemological considerations of Section 4.5.1b in mind, whenever I say something like ‘*developer A knows X*’, what I really mean is that ‘*I have seen enough evidence in her actions and interaction with her partner to be convinced that she would agree to: I know X.*’<sup>8</sup> This interaction-based approach has a number of consequences:

- I only get to see knowledge that can be verbalized, i.e., *explicit* knowledge as opposed to *implicit* knowledge (see Section 2.2.2). Although non-verbal knowledge transfer may be possible, e.g., learning how to debug by watching an experienced engineer, it is not of my concern.
- I only get to see knowledge-in-transfer that is actually verbalized, but not the larger body of knowledge-in-use. Developers do not iterate through everything they know, so I cannot reconstruct the full bodies of knowledge they possess. However, I can still characterize the pair members’ knowledge levels with respect to the topics the developers talk about in their session.

---

<sup>8</sup>This is similar to a *knowledge* definition provided by German linguists Ehrhardt & Heringer (2011, p. 37), which they call “*belief-knowledge*” (German: “*Glaubwissen*”).

- *Relative* differences between the two developers' knowledge levels are easier to see than *absolute* knowledge gaps which are shared by both developers: Knowledge which one developer puts to use but which the partner does not possess possibly leads to confusion and then observable actions. However, to observe both developers not knowing something requires one of them to address the *unknown* unknown to make it a *known* unknown.
- I am not concerned with an 'objective truth' that goes beyond what the developers consider *true* or *false*. I take source code and other screen contents into account and can sometimes detect misunderstandings, but there is no practical way to validate everything the developers say. For my research, there is little to be gained from going after things that both developers treat as true to identify the instances where they both are, in fact, mistaken.<sup>9</sup> I therefore assume the developers know everything there is to know unless there are reasons for me to believe otherwise. Programming partners working out conflicting views, however, are an aspect of my research, as is illustrated in Example 4.2.
- I also do not categorically exclude opinions which developers occasionally express. As Example 4.3 illustrates, a clear line between hard facts and personal opinions is difficult to draw anyway. (I eventually did exclude opinions from my analysis for a lack of frequency: The developers in my data rarely expressed opinions without also providing or pointing out some factual information, see Section 6.4.4d.)

**Example 4.3: Expressing Opinions** (CA4, 34:25–34:41)

The pair is setting up a test case for which they need to provide a dummy event with a `DisplayPoint` instance. C4 appears to dislike that class, but does not explain why. Her partner C7 appears to agree anyway.

C4: "The `DisplayPoint` really sucks, you know."

C7: "In what sense?"

C4: "Don't like it. That's such a crappy (..) object < \*opens class `DisplayPoint` \* >"

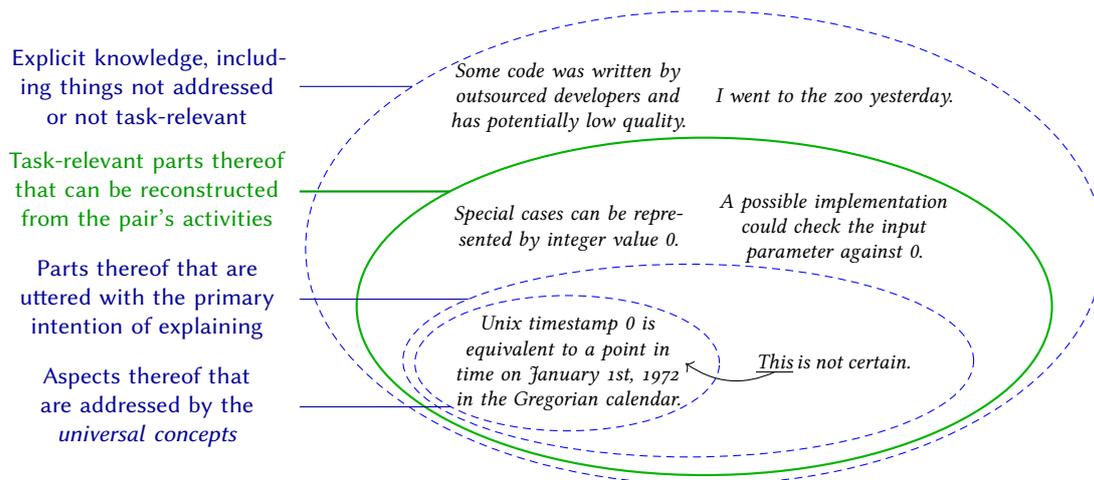
C7: "< \*reads code (,,) \* > Urgh."

C4: "< \*closes class `DisplayPoint` \* > Anyway, doesn't matter."

Neither from their exchange nor from the class itself (a data class with three attributes and no logic other than three getter methods) could I reconstruct where the pair's aversion comes from. It appears to be more a matter of taste: First, a `DisplayPoint` object can be easily created (which they do in the next minute in a single line of code); second, C4 explicitly says that it "*doesn't matter*".

In summary, the type of knowledge I address is the subset of a developer's explicit knowledge that can be reconstructed from the pair's activities. This goes beyond the notion of *knowledge* that Salinger (2013) settled on when developing the base layer (see Section 3.4.4) in that I also address parts of a developer's body of knowledge that she never verbalizes with the primary intention of explaining it. Nevertheless, there will be task-relevant parts of a developer's body of explicit knowledge which *cannot* be reconstructed from the pair's activities. See Figure 4.5 for a schematic overview.

<sup>9</sup>Uncovering mistakenly believed objective falsehoods may be necessary for other research interests, e.g., those focusing on pair debugging which may involve false assumptions on part of both developers.



**Figure 4.5:** Further subdivision (solid green ellipse) of the classes of explicit knowledge discussed in Figure 3.5 (dashed blue ellipses), illustrated through the (hypothesized) pieces of developer B1's knowledge when he says "Null is January, 1st of 1972, or something". (Parts of the source code were indeed written by outsourced developers, as the team lead told in a separate interview. I do not know whether B1 actually went to the zoo.)

#### 4.5.2 e) Working with the QDA Software ATLAS.ti

##### *Terminology and Features*

I primarily worked with ATLAS.ti,<sup>10</sup> a software for qualitative data analysis (QDA). It supports audio and video files as well as text documents, which are all called *primary documents*. Segments of primary documents are called *quotations*. Quotations from all primary documents can be arbitrarily played back which makes comparisons of different parts of one PP session or across PP sessions convenient.

*Codes* can be assigned to quotations (e.g., for *open coding*). ATLAS.ti supports connections between quotations and other quotations (e.g., for *axial coding* on the exemplar level) as well as between codes and other codes (e.g., for *axial coding* on the concept level). ATLAS.ti comes with a number of tools to help with *constant comparison*, e.g., by listing all quotations that are labeled with some code with the option to play them one by one.

Exemplars of most of these data types (in particular primary documents, quotations, codes, code-code connections, and quotation-quotation connections) can be commented in a free text field. Additionally, there are also stand-alone *memos*, which may be connected to other memos, codes, and quotations.

##### *Basic Usage*

For each of the 27 analyzed PP sessions, the video files and auxiliary data (such as scanned questionnaires or recorded reflection interviews) are imported as *primary documents*. After the first review of a session (see page 167), I put a general summary of the session (like in Section 4.4) in the comment field of the according primary document.

For relevant phenomena, I create a *quotation* and assign one or more *codes* to capture its meaning with a concept, possibly characterized in more detail with specific properties. I use a number of descriptive codes, e.g., one for each developer such as P.D3 for developer D3, which allows to later select all coded segments of the same developer even across sessions. I added comments to a quotation for one or more of these purposes:

<sup>10</sup>Product homepage: <https://atlasti.com/>

- Summary of reconstructed technical information and the pair’s action (especially for segments of a minute or more).
- Notes for axial coding on exemplar level, e.g., discussion of particular causes and consequences of the pair’s concrete behavior.
- Transcripts of difficult to understand utterances.

Most of the codes’ comments contain a short memo summarizing its purpose and *theoretical memos* (see Section 3.3.3f on memo types). I coded links between quotations as part of axial coding (e.g., two related phenomena following each other). Code-code links I used to capture conceptual category-property relationships (see below) and for axial coding on the concept level (causes, consequences, etc.), which effectively represent *code memos*. I used ATLAS.ti “memo” feature for my *operational memos*.

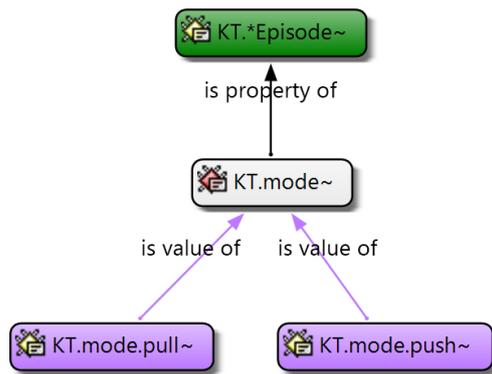
With all coding activities considered, I reviewed each of the 27 selected PP sessions at least two times in full. Sessions sampled early in the process and those with, at the time, new phenomena have more detailed codings (such as CA2, JA1, and OA8, each with several hundred segments), while sessions I analyzed during selective coding are notably less densely coded (between a dozen and 50 coded segments). Overall, I created almost 2800 data segments and coded 7700 labels, amounting to roughly one concept label and, on average, 2 to 3 qualifying properties per segment. I summarize these uses in Table 4.4.

ATLAS.ti Element	Count	Use in Analysis
primary document	70	PP sessions (video files) fieldnotes & questionnaires (PDFs) reflection interviews (audio files)
primary document comment	27	session characterization
quotation	2 764	phenomenon, data segment
quotation comment	206	reconstruction notes axial coding (exemplar level)
quotation-quotation link	789	axial coding (exemplar level)
code	416	concept (generic) property of concept descriptive markers (e.g., developer)
code comment	288	theoretical memos
code-code link	236	concept-property connection axial coding (concept level) code memos
coding (= code-quotation link)	7 685	concept indication (specific) property of exemplar
memo	35	operational memos

**Table 4.4:** Mapping of ATLAS.ti elements to uses in my analysis process, including the respective count of elements I created.

### *Limitations and Solutions*

My first problem with ATLAS.ti was the lack of support for **concept properties or property values**, which characterize concepts and phenomena in more detail, respectively (see Section 3.3.3a). Fortunately, it is possible to define different types of code-code relationships. I ended up using three different types of codes (representing concepts, properties, and property values—all indistinguishable for ATLAS.ti). See Figure 4.6 for a concrete example.



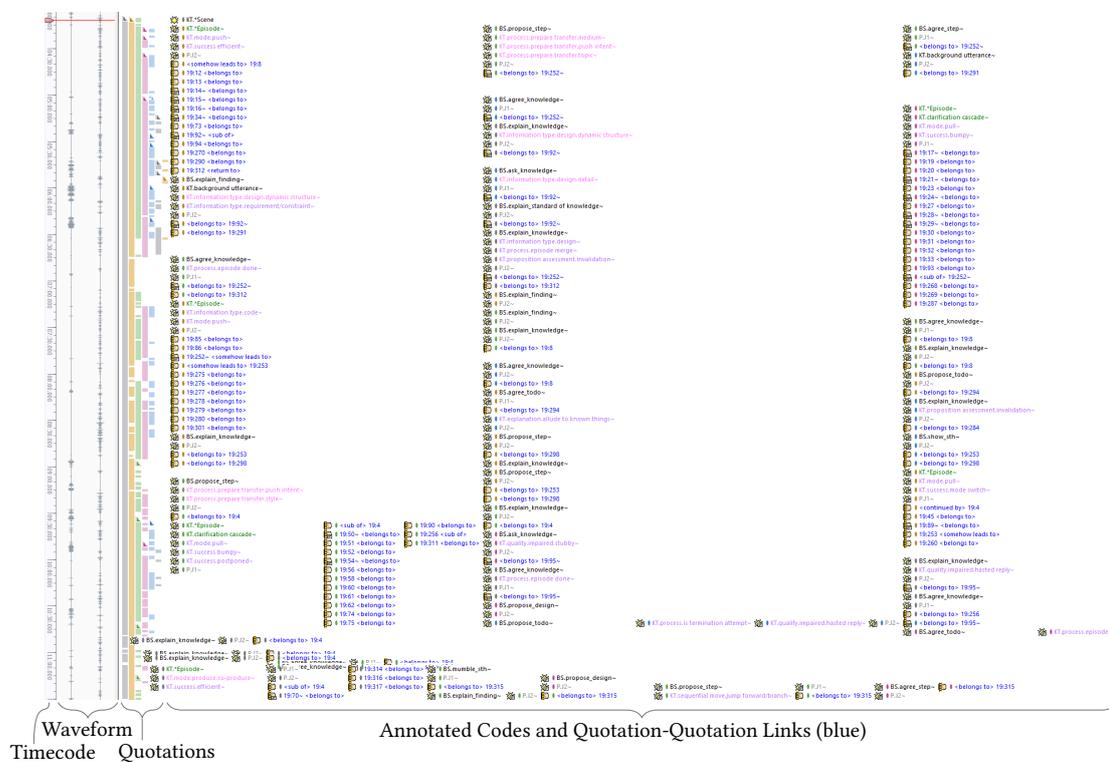
**Figure 4.6:** Example of four ATLAS.ti *codes* representing GT concepts, properties, and property values. From top to bottom: *Episode* is a concept of which there are occurrences in the real world; *Mode* is a property of that concept; *Push* and *Pull* are two of the values for this property. In the data, there are no segments coded with property codes, meaning there are only instances of *Episode*, *Push* & *Pull*, but no *Mode*-instances.

The second problem relates to my need for overlapping quotations: I coded the two developers' activities individually (which sometimes overlap) and I had concepts on different granularity levels (where larger ones span multiple smaller ones).<sup>11</sup> Each of these segments is not only coded with a concept label, but also potentially with one or more qualifying specific properties. ATLAS.ti provides a **visual timeline** displaying the quotations and assigned codes, but the user interface appears to be designed for only a handful of codes and little overlap between quotations. Since there is **no filter option**, all quotations and their assigned codes are stacked on top of each other. The resulting visual mess is not informative (see Figure 4.7a), and also makes the user interface increasingly unresponsive with more codings. Luckily, ATLAS.ti data can be exported to an XML file containing all segmentation and coding information. To get a better overview of my coded data, I wrote a visualizer tool that converts the XML export to an SVG file, which allowed me to use *x*- and *y*-dimension, color, scale, text, and icons to encode different meanings (see Figure 4.7b).<sup>12</sup>

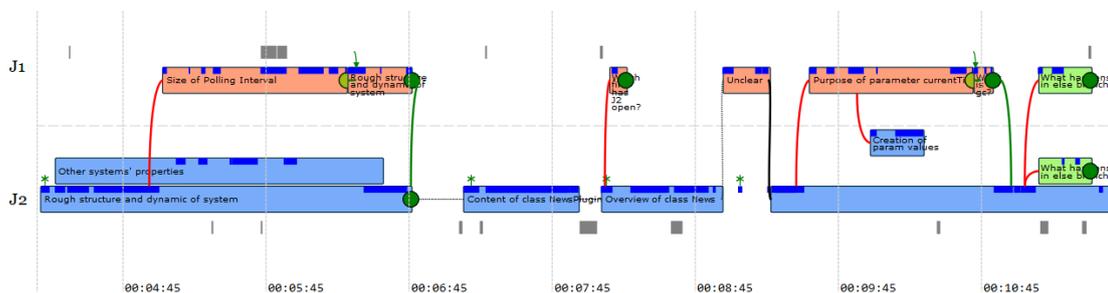
<sup>11</sup>Although QDA software tends to obscure this, a phenomenon is *not* the same as a segment of raw data; it is not simply all that happens during a time span, but pertains only to certain aspects of the subjects' actions (see also Salinger, 2013, p. 115). Consequentially, not all small segments happening during a long-running segment automatically *belong* to it (e.g., one pair member may write down an idea while the other spends a long time reading source code). I therefore explicitly coded connections between quotations to distinguish parallel from embedded phenomena.

<sup>12</sup>Salinger (*ibid.*, p. 114) encountered a similar problem. He, too, wrote a piece of software to create visual representations of what he called *tracks*, a collection of related annotations on a timeline.

**Figure 4.7:** Comparison of ATLAS.ti and my own visualization for the first seven annotated minutes of session JA1



(a) View in QDA software ATLAS.ti. Time is going from top to bottom; elements from left to right: first grey stereo waveform, then all quotations stacked and color-coded, then annotations for the quotations. The user interface is clearly not designed for more than just a few overlapping quotations and more than a handful of annotations. Is it not possible to filter this view (e.g., disregarding specific properties to focus on the concepts, or focusing on small or large granularity only). Color and horizontal position of the quotations are set by the software.



(b) Plotted by my visualization tool. Time is going from left to right, developers are separated vertically. This view shows high-level concepts as large blocks, smaller granularity as small blue and grey boxes; they are grouped by developer and colored based on specific properties of the coded instance. I use similar visualizations in Chapter 10.

## 4.6 Discussion of Overall Research Method

I will now provide an overview of my research method in terms of the nine common traits of qualitative research approaches, the six differentiating questions, and the eight quality criteria (all introduced and discussed in Sections 3.2.2 to 3.2.4).

### Where the Common Traits of Qualitative Research are in My Method

I already discussed the extent to which Straussian and Charmazian GTM directly address the common traits of qualitative research in Section 3.3.5b. Two of the traits—how to engage in NATURALISTIC INQUIRY and the IMPORTANCE OF THE RESEARCHER—are hardly addressed by the methodologies themselves, so I introduced various according elements of my data collection method described in Section 4.3. Table 4.5 maps all nine traits to the corresponding elements of my research process.

### My Answers to the Six Differentiating Questions

In Section 3.3.4, I discussed the differences of the classic and constructivist GTM versions regarding the six core questions. While the **Method** (*how to study the world?*) is generally agreed upon, **Ontology** (*what is the nature of reality?*) and **Epistemology** (*how do we know?*) mark the biggest difference. I discussed my own position, which involves positivism for technical aspects and constructivism for any social action, in Section 4.5.1b. For my research, the answers to the related aspects of **Philosophy** (*what is worth knowing?*) and **Discipline** (*what questions should we ask?*) come my from field of software engineering: I want to understand how knowledge transfer in pair programming works, such that practitioners can reap the benefits of this work mode (see also Section 4.2.1). This also shapes my **Involvement** (*how to personally engage?*): To the study participants, I am a visiting researcher, but also a *developer*. I learned that talking to practitioners about software development helped establishing trust.

### Meeting the Quality Criteria

As discussed in Section 3.3.5a, the eight quality criteria proposed by Tracy (2010) are a superset of the criteria proposed by Strauss & Corbin (1990) and Charmaz (2006). I consider each of them below:

1. **Worthy Topic:** *Research topic is relevant, significant, and interesting.*  
Pair programming is a relevant topic in software engineering, and knowledge transfer as a central expected benefit is not well understood yet (see Section 4.2.1).
2. **Rich Rigor:** *Descriptions and explanations are complex and ‘rich’.*  
I used both existing and newly collected data that is rich in detail and comes from diverse contexts (Section 4.3.4d and Table 4.3). I performed in-depth qualitative analyses (Section 4.5).
3. **Sincerity:** *Researcher’s role is transparent.*  
Meeting this criterion is the main purpose of this chapter. To accurately reflect the interactions between subjects, settings, and involved researchers (not only me), I ...
  - ... discuss the generic data collection protocol that all involved researchers followed in Section 4.3.2;
  - ... emphasize important differences, such as the different research “headlines” for industry contacts in Section 4.3.1, the non-naturalistic workshop character in com-

panies C to F on page 148, and the different versions of the pre- and post-session questionnaires in Section 4.3.2c;

- ... and describe my own data collection efforts in Section 4.3.5b. (The concrete timeline of all my research activities follows in Chapter 12.)

To reiterate and appreciate the work of my colleagues:

- Laura Plonka, Stephan Salinger, Julia Schenk, and Holger Schmeisky established industry contacts and collected data (see Section 4.3.1 and Table 4.1).
- Laura Plonka and Stephan Salinger developed the original data collection protocol (see Section 4.3.2).
- Laura Plonka and Stephan Salinger came up with Generation 1 of the recording infrastructure (see page 147).

4. **Credibility:** *Results are trustworthy enough for readers to act on them.*

This thesis contains many examples from the analyzed sessions, at least one for each aspect of my overall theory. I employed *member reflection* at various points in the research process (see Section 4.3.3d).

5. **Resonance:** *The research influences, affects, or moves the reader.*

During my research, I often spoke with practitioners, trying to understand which issues are relevant to them and which means of (visual) representation resonate with them. I then spent much time on compiling this document, tried out different ways to embed and display examples, employ visual aids, etc., with the goal to make it an interesting read. I hope to achieve naturalistic transferability through my in-depth examples.

6. **Significant Contribution:** *It extends knowledge, improves practice, generates research ideas.*

I deem my work to be *theoretically significant* as it both extends existing concepts (of the base layer, see Section 6.2) and introduces many new ones which show the variation in how pair programming processes unfold (see Chapters 6 to 11). Such variation was often ignored by prior research which considered PP as something canonical (see discussion on page 67 in Section 2.3.4a on PP experiments). However, *practical significance* is my main goal: I come back to this in Chapter 13.

7. **Ethics:** *Effects of research actions on subjects and others are considered.*

**Procedural ethics:** All participation was based on informed consent (see page 146); one developer explicitly opted-out of visual data collection, which, as far as I can tell, did not reflect negative on him in the team (see page 154). No company or developer is identified by name in any publication (see Section 4.3.2h). In addition, personal information from particular PP sessions or reflective interviews was not transferred to colleagues or superiors (see Section 4.3.2e).

**Relational ethics:** Both reflective interviews and workshops were meant to give something back to the developers quickly (see Sections 4.3.2e and 4.3.2g).

8. **Meaningful Coherence:** *Theoretical framework, research method, and goals are aligned.*

I followed the coding procedures from Strauss & Corbin (1990) as they provide structure for the analysis process. The action-oriented model that underlies the paradigm model fits my topic well. Just as Charmaz (2006) described, the paradigm model provided clarity during my analysis (see Section 4.5.2).

The positivist epistemological stance of “classic GTM” is not embedded in my pragmatic application of Straussian coding procedures (see also Sections 3.3.3b and 3.3.4b). I therefore see no contradiction in following these procedures while still assuming a (mostly) constructivist stance myself.

**1. NATURALISTIC INQUIRY IN EVERYDAY SITUATIONS**

- Minimally invasive *in vivo* recordings of industrial PP sessions (Sections 4.2.3a and 4.3.2b)
- Field observations (Section 4.3.2f)

**Limitations:**

- Some effects of recording infrastructure (Section 4.3.4b)
- Some effects of pre-existing notions (Section 4.3.4c)

**2. OPENNESS & EMERGENT RESEARCH DESIGN**

- Generic protocol is not rigid, but is adapted on site (Section 4.3.2)
- Recorded sessions allow for different analysis perspectives (Section 4.3.2b)
- Open-minded quick analysis to inform reflective interviews (Sections 4.3.2d and 4.3.2e)
- Adjustment of purpose of reflective interviews (Section 4.3.2e)
- Invention of long-term engagement and practitioner evaluation along the way (Sections 4.3.3c and 4.3.3d)
- Adjustment of result type over time (Section 4.2.2)

**3. PURPOSEFUL SAMPLING**

- Theoretical sampling process in eight phases (Section 4.3.5b)

**4. DETAILED, THICK, AND DIVERSE DATA**

- Video recordings of PP sessions with abundant detail (Section 4.3.2b), supported by questionnaires, interviews, field observations, workshops (Sections 4.3.2c and 4.3.2e to 4.3.2g)
- Repository of sessions from different companies/application domains, technology stacks, developer experiences, and task types (Section 4.3.4d and Table 4.3)

**5. IMPORTANCE OF THE RESEARCHER**

- Focus of researcher during quick analysis (Section 4.3.2d)
- Conducting the reflective interviews (Section 4.3.2e)
- Reflection on effect of researcher headline and pre-existing notions on data collection (Sections 4.3.1 and 4.3.4c)
- Reflection on epistemology and effects of different languages (Sections 4.5.1b and 4.5.2a)
- Reflection on applicability of GTM (Section 4.3.3a)

**6. EMPATHIC NEUTRALITY**

- Reflective interviews to appreciate developer's perspective (Section 4.3.2e)
- Base coding to actively consider developers' intentions (Section 4.5.2b)

**7. CONTEXT MATTERS**

- Questionnaires, reflective interviews, field observation, and team workshops as different data sources (Sections 4.3.2c and 4.3.2e to 4.3.2g)
- Consider sessions as cases rather than excerpts in isolation (Section 4.3.3b)

**8. HOLISTIC PERSPECTIVE**

- Consider sessions as cases rather than excerpts in isolation (Section 4.3.3b)

**Limitation:**

- Scope of pair programming session (Section 4.2.3c)

**9. DISCOVERY OF THEORIES**

- Grounded theory methodology with amendments (Section 4.5)
- 

**Table 4.5:** Common traits of qualitative approaches (see Section 3.2.2) and where they are implemented in my research process. See referenced sections for details.

— Part II —

# Results



## Chapter 5 Results Overview

---

*I can't recall who first pointed out that the word  
'explain' means literally to 'flatten out.'*

– Philip Slater

5.1	Purpose and Structure of this Chapter . . . . .	183
5.2	Pair Programming Process . . . . .	183
5.3	Knowledge Transfer Episodes . . . . .	184
5.4	Pair Programming Session Dynamics . . . . .	185
5.5	A Recurring Example . . . . .	186

### 5.1 Purpose and Structure of this Chapter

Working in pairs makes software development easier and harder at the same time. On the one hand, it gets *easier* because the programming partner might already know certain things, which therefore do not need to be found out. Additionally, anything that the pair learns along the way—names of classes, order of parameters, special values—is less likely to be forgotten. On the other hand, it gets *harder* because the programming partner's mind is opaque and there is no direct way of exchanging information. No act of communication can be totally explicit, so some common ground—which needs to be established and maintained (see Section 3.2.1a)—is required between the participants.

In the following chapters, I present several observations of how professional software developers transfer knowledge during pair programming sessions. This chapter is an overview of the most important concepts that are the result of my analyses which together provide a basic 'vocabulary'. These concepts address a wide range of different phenomena: from aspects of individual utterances to whole pair programming sessions; some are concrete and directly observable, others are more abstract; some pertain to the individual programmer, others to the pair as a whole. Most of the concepts are highly interconnected, such that linear writing has to rely heavily on forward and backward references. In the next sections, I will therefore paint the big picture without going too much into the details, hoping this will ease the reader's overall understanding.

### 5.2 Pair Programming Process

The concept of **Fluency** captures the process quality of a pair programming situation: Roughly speaking, it is 'how well' the developers work *as a pair*. This is not necessarily the rate of technical progress: A pair might be relatively slow from a technical perspective, but still have a fruitful pair programming session. I distinguish three qualitative levels of **Fluency**: **normal PP**, **Focus Phases**, and **Breakdowns**.

**Normal PP** is characterized by a steady process: Apart from occasional waiting times (e.g., compiler or test runs) or brief moments of silence, there is an ongoing stream of communication

between the two developers. Proposals are made, insights are shared, and—most importantly—they are *evaluated*, i.e., agreed or disagreed to, discussed and amended. The pair recognizes their inevitable misunderstandings (Section 3.2.1a) and tries to clear them up. Some pairs, in some situations can go beyond this and reach a **Focus Phase**. During these, developers are “*exceptionally in sync with one another*” (as also observed by Chong & Hurlbutt, 2007): There are little to no pauses; developers sometimes even speak simultaneously; proposals do not need to be fully verbalized because the partner understands them anyway; there are no discussions, just agreement. Misunderstandings do not happen. At the other end of the spectrum are **Breakdowns**: On the surface, such a process is really slow-moving, there are prolonged periods of silence with no apparent activity. Looking deeper, it is no longer a *pair* process: If a pair member makes a proposal at all, it is *not* evaluated or discussed by the partner, making it unlikely to yield any of the expected benefits of working together with a partner (see Section 2.3.1b). Sometimes *both* partners do actually produce ideas, but *neither* evaluates them, and long stretches of time pass where nothing really happens. The developers also do not attempt clearing up any misunderstandings.

The basis for a pair’s **Fluency** is its **Togetherness**, the degree to which a pair is working “*with one mind*” (Williams et al., 2000). **Togetherness** is not directly observable, but its effects and the pair’s activities to **Maintain Togetherness** are. With high **Togetherness**, it requires only little effort of the pair’s members to make each other understood, which is one aspect of what makes **Focus Phases** so fast. High **Togetherness** is not simply thinking the same all the time, though. More to the point is a notion of compatible mental states such that misunderstandings are rare. In a way, **Togetherness** is the ongoing mutual lack of being surprised by what the partner says and does.

**Fluency** and **Togetherness** are general pair programming phenomena, which is why I discuss them first in Chapter 6. They are, however, related to knowledge and knowledge transfer. On the one hand, one way to **Maintain Togetherness** is to transfer knowledge: One developer already worked on some module, and before her newly joined partner can get productive as well, she needs to explain any important changes she already made; or two developers dig through some stack trace together, alternating in who has the next idea how to proceed (a “*tag team*”, see Bryant et al., 2008, Sec. 6.3), which requires some ‘resynchronization’ every now and then, such that the partners “*maintain a firm grasp of what is happening during the session at a number of levels of abstraction*” (*ibid.*). On the other hand, possessing shared knowledge of the software system and software development in general enables their **Togetherness** and thus **Fluency**. Knowing very little about the current system can lead to a **Breakdown**, if the system is complex enough and/or involves unfamiliar technology, and the pair lacks the ability to deal with such a profound knowledge gap. On the flipside, knowing very much about the current system can lead to **Focus Phases**, if the developers clearly understand their task, have a strategy to adhere to, and manage to follow a straight process while deferring side issues.

In summary, what the pair members know and do not know affects their **Togetherness** and transferring knowledge is one way of **Maintaining Togetherness**.

### 5.3 Knowledge Transfer Episodes

Software development involves dealing with knowledge gaps, and so does pair programming. Every now and then, a pair member comes to perceive an urge to do something about a particular knowledge gap, a **Knowledge Want**. From the researcher’s point of view, neither the knowledge gap nor the **Knowledge Want** are directly observable. A **Knowledge Want** pertains to a *perceived* knowledge gap, not necessarily an actual gap: After all, a developer

may be mistaken and provide explanations and pursue the clarification of some **Topic** where her partner actually has no knowledge gap.

A **Knowledge Want** can take one of three forms. First, there are **internal Knowledge Wants**: The developer feels she does not understand something, the knowledge gap needing attention is her own (*I want to know this*). To close the gap, she might start to ask questions, read in the source code, or play around with the running program. Second, there are **external Knowledge Wants**: The developer feels her *partner* has a misunderstanding or knowledge gap that should be closed (*I want you to know this*). In this case, she may start to offer explanations—without being asked to do so. Finally, there are **collective Knowledge Wants**, where both developers share a knowledge gap and both agree that it should be addressed (*we want to know this*). **Collective Knowledge Wants** can be closed by reading source code and thinking as well as exchanging ideas and hypotheses.

The information that fills the underlying knowledge gap I call **Target Content**. There are different types of **Topics** and **Target Contents** in pair programming sessions: Most of them pertain to the specific software system the pair works in, which I call **S knowledge**, others belong to the category of generic software development knowledge, or **G knowledge**. I introduce the three basic knowledge concepts of **Knowledge Want**, **Topic**, and **Target Content** in Chapter 7.

Developers actively pursue the clarification of a **Topic** through activities such as asking questions, proving explanations, or interacting with the software and its source code. Both asking and explaining are complicated activities which come in many forms. In Chapter 8, I call them by their abstract function **Explanation Elicitors** and **Explanations**, of which I distinguish five and three types, respectively.

In pair programming, there is typically only *one* pair member invested in each particular **Topic** and her partner may or may not follow her lead. The whole of the activities she and her partner engage in to clarify a particular **Topic** I call an **Episode**. The goal of such an **Episode** is to satisfy the underlying **Knowledge Want** by transferring or acquiring the **Target Content**. In other cases, the **Propellor** may end the **Episode** after she decided the **Knowledge Want** is no longer worth pursuing. Depending on the number of interactions it takes to handle the **Knowledge Want**, an **Episode** can be a few seconds or several minutes long.

There are different strategies how pair programmers approach **internal**, **external**, and **collective Knowledge Wants**. An **internal Knowledge Want**, for example, may be pursued by asking the partner for the **Target Content**. In other scenarios, the **Propellor** may dive right into the source code without asking her partner. The way a particular **Episode** is carried out is its **Mode** of which I distinguish four main ones: In a **Pull Episode** the partner is asked about the **Topic** until the **Target Content** is transferred; in a **Push Episode** explanations are offered until the partner appears to have understood the relevant **Target Content**; in **Pioneering Episodes** one developer acquires the **Target Content** by herself; and in **Co-Producing Episodes** both developers do this hand-in-hand. I discuss **Episodes**, the different ways they can start and end, as well as their **Modes** in Chapter 9.

## 5.4 Pair Programming Session Dynamics

Each individual knowledge transfer **Episode** only deals with a single **Knowledge Want**. In the course of solving a technical task together, typically many of these arise over time and lead to dozens of **Episodes**. In session AA1, for example, I analyzed almost 90 of them. Some of these **Knowledge Wants** arise from pending technical decisions or simply from opportunities to address some general **Topic** just *now*. Quite often, however, dealing with one **Knowledge Want**

leads to another. I distinguish **Sub-Episodes** and **Catalyzed Episodes**. **Sub-Episodes** branch off of a main **Episode** and should be, in the eyes of the developers, addressed before the original **Topic** can commence. A **Catalyzed Episode**, in contrast, pertains to a **Knowledge Want** that arises in the midst of another **Episode** and which the developer address right away even though they do not deem it be strictly necessary at the moment. Depending on how the pairs handle such new **Knowledge Wants** they may end up **Branching Wildly** with many open **Topics** or they proceed in a more linear fashion when they **Return Explicitly** and make use of **Scope Limiting**. I discuss these patterns in Chapter 10.

Zooming out further brings PP sessions as a whole into focus. I integrate all the above mentioned parts in Chapter 11. Pairs do not start a session as ‘experts’ or ‘novices’ but with session-specific **Initial Constellation** of **Knowledge Needs** which depends on their respective pre-existing knowledge and what their tasks demand of them. Furthermore, not all sessions have a purely technical goal, but may also involve explicit transfer of general software development knowledge (**G knowledge**). The **Knowledge Needs** a pair wants to address in their session marks their **Target Constellation**. With each knowledge transfer **Episode** they proceed a little on their trajectory. Over all analyzed sessions, I identified six recurring **Initial Constellations** and a shared overall session dynamic consisting of three steps. First, pair address their **Primary Gap** which is their difference in system understanding (**S knowledge**). Then, they go about closing their **Secondary Gap**, which is their shared lack of system understanding that is required to productively work on their task. Finally, with the **S Knowledge Needs** out of the way, pairs may actively start seizing their **G Opportunity** by transferring **G knowledge** from one partner to the other.

## 5.5 A Recurring Example

Throughout all result Chapters 6 to 11, I will use the same excerpt from the beginning of session JA1 as a recurring example to illustrate the respective chapters’ concepts. Each chapter stands for a different analysis angle and I think coming back to the same data again and again to see what else it has to offer not only resembles my research process but also allows the reader to get somewhat familiar with at least a small portion of my complex data.

Unlike other Examples boxes in this thesis, the box below is a mere description of the events in a pairs’ process without any discussion. As described in Section 4.4.4, the pair members in question work in different cities and have an audio connection but cannot see each other. The session recording, however, contains both developers’ screens and webcams, allowing to analyze their facial expressions and gestures.

### **Example 5.1: The “Raw Data” of the Recurring Example (JA1, 02:29–06:15)**

J1 is an experienced software developer who was invited by J2 to help with refactoring a software system which J2 wrote several months prior. The software system (*“the plugin”*) aggregates recordings of news segments from a number of radio stations (*“waves”*). Each radio station provides a file server access (*“shares”*) to where the live recordings of the news segments are continually written. The processing per radio station then consists of determining the moment when a news segment ends, fetching the finished recording, and finally converting it to a different format (*“transcoding”*).

In the excerpt below, J2 explains the static and dynamic architecture of the system, which involves polling the file servers. J1 wants to know the size of the polling interval (which is 30 seconds), but J2 misunderstands the question. J2 thought he was asked *How long are you polling?* (which is theoretically up to an hour until the next news segment starts, and practically about seven minutes tops until the current news segment ends). The excerpt ends with the pair having resolved this misunderstanding.

## Example 5.1 (continued)

Since this is a distributed session and the developers do not see each other's webcam feeds, non-verbal actions such as nodding cannot be seen by the partner (but are part of the session recording available to the researcher).

(1) J2: "Do you know the NewsPlugin, or don't you know it?"

(2) J1: "<exhales audibly> Just show to it me again."

(3) J2: "That with the news segments. I guess we should share it."

The pair starts a Saros session (see footnote 7 on page 163) and J2 shares the source code. After 1:30 minutes and while the source code is still transferring to J1's machine, J2 continues his explanation:

(4) J2: "OK, I can give you the big picture of what this plugin does, overall."

(5) J1: "Yep."

(6) J2: "In the end, the news recording of every hour pops out."

(7) J1: "M-hm."

(8) J2: "The way this works is that there are multiple processors, so there is the central plugin and multiple processors which each handle one wave."

(9) J1: "<\*nods\*>"

(10) J2: "For most of them, right after the full hour there is a check, if there is a new file on the remote share."

(11) "If so, the most recent file is selected and it starts checking how the file changes size-wise."

(12) J1: "<\*stops nodding, looks to his upper right\*>"

(13) J2: "I mean, it looks until the file does not get bigger anymore, then it is apparently ready."

(14) J1: "M-hm"

(15) J2: "And then it is fetched and handed over to transcoding."

(16) J1: "In what time window are you looking?"

(17) J2: "I start looking two minutes after the full hour, because then it's guaranteed that news files exist if any exist."

(18) J1: "OK"

(19) J2: "And then monitor this file as long as needed until it's ready. That can take up to seven minutes, depending on the wave."

(20) J1: "Hm ya but mh the time window for the change?"

(21) J2: "Yes, right, that is, er, time window for the change is variable, depends on how the news go."

(22) J2: "I can't know that, right? You know, they always start a new file. When the news are over, again a file is created. I mean, I basically never have more than the news."

(23) J1: "Yes, no, I mean 'cause you said you look for so long until the size stops changing, right?"

(24) J2: "M-hm"

(25) J1: "Then you need to plan for a time window in which a change could happen, right?"

(26) J2: "Yeah, well, until up to five before the hour. I really take my time."

(27) J1: "<laughs> No, I really mean the size now, the size of the time window, I mean (!...!) You wait for 10 seconds, then after 10 seconds you decide: In those 10 seconds nothing has changed, so the file appears to be ready."

(28) J2: "Ahhh, that's what you mean. No, 30 seconds."

(29) J1: "30 seconds, that's what I wanted."

(30) J2: "That's 30 seconds long the time window. Now I got you."



# Chapter 6 Process Fluency and Pair Togetherness

---

6.1	Purpose and Structure of this Chapter . . . . .	189
6.2	Dialog Structure in Pair Programming . . . . .	190
6.2.1	Dialog in the Base Layer . . . . .	190
6.2.2	Five Types of Base Activities . . . . .	191
	★ Initiative Activity • ▲ Pair-Referential Activity • ◀ Self-Referential Activity •	
	● Corrective Activity • ◆ Conversational Defect	
6.2.3	Discussion of Recurring Example . . . . .	199
6.3	<b>Fluency</b> . . . . .	200
6.3.1	Foreword to the <b>Fluency</b> Examples . . . . .	200
6.3.2	<b>Normal</b> Pair Programming . . . . .	201
6.3.3	<b>Focus Phases</b> . . . . .	204
6.3.4	<b>Breakdowns</b> in Pair Programming. . . . .	210
	<i>No Progress as a Pair • No Progress at All</i>	
6.4	<b>Togetherness</b> . . . . .	222
6.4.1	Degrees of <b>Togetherness</b> : Understanding Intentions . . . . .	223
6.4.2	Facilitators and Inhibitors of <b>Togetherness</b> . . . . .	224
	<i>Factor: Shared Understanding of the System • Factor: Shared Understanding of Software Development • Factor: One Shared Plan • Factor: Workspace Awareness • Factor: Language Barrier • Factors' Interplay</i>	
6.4.3	Not <b>Maintaining Togetherness</b> . . . . .	228
	<i>By Choice • By Accident</i>	
6.4.4	<b>Maintaining Togetherness</b> . . . . .	230
	<i>Excluded Factor: Language • Excluded Factor: Workspace Awareness • Excluded Factor: One Shared Plan • Excluded Type: Opinions</i>	
6.5	Discussion of Related Work and Summary . . . . .	234

## 6.1 Purpose and Structure of this Chapter

Unlike the results that will be presented in the following chapters, this chapter is not yet specific to the *knowledge transfer* aspect of pair programming, but covers more fundamental PP mechanisms. The observation which triggered this particular line of inquiry was the recording of session **OA1** in which the pair struggled for hours and made virtually no headway. This session contrasted with others I had seen before, e.g., session **CA5** in which the pair downright *raced* through their task, producing one valuable code change after the other.

I introduce the concept of **Fluency** to describe the qualitative difference between these ways of how pair programming sessions are carried out. A PP session's **Fluency** is not constant, e.g., pairs may recover from a **Breakdown** and work **normal** just a few minutes later; highly productive **Focus Phases** are also limited to a few minutes at most, whereas the majority of the session progresses **normal Fluency**. This variation can be explained with the programming pair's momentary **Togetherness**: It conceptualizes how well the developers understand each other, or more precisely, the extent to which they are able to understand each other's activities. Unlike **Fluency**, a pair's **Togetherness** is not directly observable, but is it put to test with every base activity of a pair member: Resulting confusion or the lack thereof allow for an operationalization. Furthermore, many of the things pair programmers do, including some forms of *knowledge transfer*, can be interpreted as **Maintaining Togetherness**, basically making sure both pair members are on the same page. A pair's **Togetherness** influences its **Fluency**: **Low Togetherness** may lead to undesirable **Breakdowns**, while **high Togetherness** may lead to highly productive **Focus Phases**.

This chapter is structured as follows: In Section 6.2, I lay the foundation for operationalizing **Fluency** based on properties of a pair's base activities. A few observations and conceptualizations concerning dialog structures in pair programming processes are already manifest in the Base Layer (**BL**). Most importantly, there is a distinction between *initiative* and *reactive* utterances. I analyze different properties of reactive utterances (such as *timeliness*, *evaluativeness*, *appropriateness*), and characterize base activities according to their role in the pair's dialog as either ★ initiative, ▲ pair-referential, ◀ self-referential, ● corrective, or ◆ defective.

I then use patterns of base activities to operationalize three levels of pair programming **Fluency** in Section 6.3: **normal**, **Focus Phases**, and **Breakdowns**. Note that a **fluent** development process is not *necessarily* the same as a good development process: Subjectively successful sessions might be non-**fluent** for longer stretches of time, and a **fluent** process does not guarantee technical progress. However, from a software engineering stance, **normal** and **Focus Phases** are arguably desirable, while **Breakdowns** are problematic.

In Section 6.4, I explain the concept of **Togetherness**, a quality of a pair at a point in time. It is not directly observable, but positive and negative effects on a pair's **Fluency** are. Together with different strategies employed by pair programmers to **Maintain Togetherness**, I characterize a number of facilitators and inhibitors that promote or hinder **Togetherness**: (1) *shared understanding of the system*, (2) *shared understanding of software development in general*, (3) *one shared plan*, (4) *workspace awareness*, and (5) *language barrier*. In a broad sense, **knowledge transfer** in pair programming can be characterized as **Maintaining Togetherness** concerning factors (1) to (4). The following Chapters 7 to 11, however, mostly discuss the details of knowledge transfer in the narrower sense of factors (1) and (2), i.e., establishing and maintaining a shared understanding of the system and software development in general.

In Section 6.5, I discuss other research related the phenomena discussed here.

## 6.2 Dialog Structure in Pair Programming

### 6.2.1 Dialog in the Base Layer

Pair programming, as Beck (1999, p. 100) remarked, “*is a dialog*”. And indeed, as Salinger & Prechelt (2013) observed, pair programmer activities are not isolated, but may refer to the partner's (or one's own) previous activities. Consequentially, one of the key decisions of the base layer is to *model dialog episodes*: If a developer intends to make a connection to a previous activity or its topic, this relationship should be encoded with a base concept (see also page 132).

As a consequence, the base layer distinguishes *initiative* base activities (which bring up some idea, proposal, or piece of knowledge through *propose* or *explain*) from *reactive* ones (which refer to something brought up before with *agree*, *amend*, *challenge*, *decide*, or *disagree*)—see page 134. However, this distinction is not actually employed in the base layer (BL, p. 49).

The way a reactive activity refers to a previous utterance may be explicit or implicit (BL, p. 49). These references are on a *pragmatic* level, i.e., they do not necessarily exist in syntactical form (e.g., by using identifiers), but are embedded in the meaning of the uttered words and need to be understood by the developers. Normal communication always involves *construction* by the speaker and *reconstruction* by the listener (see Section 3.2.1). Amazingly, in everyday communication as well as in pair programming, this process works more often than not. And when it does not work, mistakes can be detected and corrected, e.g., by asking for clarification.

The researcher, however, needs to reconstruct the meaning of the utterances without being able to ask for clarification and therefore has to find other ways to get to a convincing interpretation. Salinger & Prechelt (*ibid.*, pp. 206–207) propose a number of method hints: *step back* and consider more context, *paraphrase*, and *peek into the future*. Either way, such reconstructions are not perfect and there is no guarantee that (a) all references intended (and possibly understood) by the developers are also understood by the researcher, and (b) all that was ‘understood’ by the researcher was actually intended this way by the developers. I discussed this issue in Section 4.5 (pages 166 and 170 to 174).

## 6.2.2 Five Types of Base Activities

I refine the *initiative/reactive* distinction made in the base layer into five types of activities. They are represented by mnemonic symbols which I use as nouns in the text and as markers in the verbatim excerpts to follow. There are four **conversational roles**: An ★ *initial activity* starts a new thread of conversation, a ▲ *pair-referential activity* refers back to running topic, a ◀ *self-referential activity* refers back to one’s topic without the partner being involved, and a ● *corrective activity* is about clearing up a misunderstanding. The fifth type is the ◆ *conversational defect* which marks a missing reaction in the conversation, though not all ◆ are necessarily problematic. Each base activity, regardless of its **conversational role**, and even the absence of an activity where one would be expected, can be a ◆. I discuss each type in detail on the following pages; refer to Table 6.1 for a summary.

Symbol	Short Description	Symbol Mnemonic
<i>Property: conversational role</i>		
★	Initiatives such as proposals, questions, or explanations; may be <b>expecting</b> a reaction or <b>non-expecting</b> (Section 6.2.2a)	Start of something new
▲	A reaction to a ★ that both partners are involved in; may be <b>prompt</b> or <b>delayed</b> , <b>evaluative</b> or <b>non-evaluative</b> , <b>appropriate</b> or <b>misled</b> (Section 6.2.2b)	Arrowhead pointing to partner’s previous activity
◀	Referring to a ★ in which the partner is not yet or not anymore involved (Section 6.2.2c)	Arrowhead pointing to one’s own previous activity
●	Attempting to clear up a misunderstanding (Section 6.2.2d)	A point to make connection
<i>Event: conversational defect</i>		
◆	A missed connection in the dialog, e.g., not referring to an <b>expecting</b> ★; can be a non-action (see Section 6.2.2e).	A communicative thread coming to a standstill

**Table 6.1:** Further differentiation of the concepts of the Base Layer (BL). The **conversational role** extends the *initiative/reactive* distinction (see page 134); ◆ is a newly introduced aspect.

In the examples and descriptions to follow, I use the symbols extensively. In examples with multiple conversation threads, I use *indices* to make the (reconstructed) references visible, as in this sequence with two interleaved topics A and B: ★<sub>A</sub>–▲<sub>A</sub>–★<sub>B</sub>–◀<sub>A</sub>.

Note that the examples given to illustrate the five types all come from session CA2 and are (almost) in chronological order. I chose examples this way to avoid context switches for the reader; the concepts were developed late in the process after I had analyzed all of the pair programming sessions described in Section 4.4.

### 6.2.2 a) ★ – Initiative Activity

Initiative activities, or ★ for short, introduce some new aspect into the discourse. Most ★s carry an implicit call to the partner to react to it in a certain way, such as *proposals* which request an evaluation, *questions* which should be answered, and *explanations* which are directed at the partner and should be acknowledged. (Such pairs of actions and expected reactions are called *adjacency pairs*, see Section 3.2.1b).

This is a simplification, of course. Not all proposals are uttered with the expectation that the partner evaluates them (see *propose* activities in mode P1 *provide information*, see page 135). The point is that the speaker of a ★ usually appears to have an expectation of the dialog’s next turn when she makes a proposal, asks a question, or gives an explanation, as can be seen in Examples 6.1 and 6.3 below. I call such ★s **expecting**. Sometimes, however, a ★ is indifferent or **non-expecting**, as in Example 6.11 on page 197, where both developers utter some fact or observation with no clear indication as to what they expect their partner to do with it.

Regardless of the nature of the implicit call, however, there is no guarantee that the partner complies to it, as she might not interpret the ★ as **expecting**, or might not be able or willing to respond. Likewise, *not* requesting an evaluation with a **non-expecting** ★ does not always hinder the partner from providing one. This is a general property of speech acts (Austin, 1962, p. 105, see also my discussion on page 112).

### 6.2.2 b) ▲ – Pair-Referential Activity

As soon as a ★ has been started by one pair member, the partner may direct a reaction to the original speaker. Such reactions and any further turn by either of the two referring to the same ★ are *pair-referential activities*, or ▲. A ▲ shows that what was said before was understood and referred to, and possibly also evaluated. This is basically what the base layer’s notion of *reactive* utterances is about: verbs such as *agree*, *amend*, *challenge*, *decide*, and *disagree*. Here are a few examples of how ▲s can look like:

- Putting a proposal-★ directly into action indicates the developer understood and appreciates it to some degree.

#### Example 6.1: Agreeing to Proposal Through Action (CA2, 10:14–10:47)

Early in their session, C5 starts to explain the changes he already performed alone. C2 guides this explanation through a proposal (*propose\_step* in turn 3), which C5 immediately puts into action without verbally commenting on it (turn 4).

- |  |                                      |
|--|--------------------------------------|
| (1) ★ <sub>A</sub> C5: “What I did [...], is to extend the IFeatureAttributeConfiguration with an IVirtualColumn, where I can get an object, where you can get the IColumns, where you might get to a Provider, or something.” |                                      |
|  | <i>explain_knowledge</i>             |
| (2) ▲ <sub>A</sub> C2: “M-hm. Sure thing.”   | <i>explain_standard of knowledge</i> |
| (3) ★ <sub>B</sub> C2: “Show them please, what they look like.”  | <i>propose_step</i>                  |
| (4) ▲ <sub>B</sub> C5: < *opens IVirtualColumn.java* >   | <i>do_sth</i>                        |

- Answering a question-★ also indicates the speaker understood it and refers to it.

**Example 6.2: Reacting to Question With Answer** (CA2, 19:54–20:00)

C2 is in the process of understanding C5’s recent changes. C2 asks about the current package location of a class and C5 explains that he moved it already.

★ C2: “And the `VirtualAttribute`, where is it?” *ask\_knowledge*

▲ C5: “That one, I moved.” *explain\_knowledge*

▲ C2: “Ah, so we’re clear.” *explain\_state*

C2’s last utterance refers to the same ★ which both partners are involved in, so it is a ▲, too.

- Some questions to answers are ‘improper’ in that they do not directly refer to the question as such, e.g., with an explanation why the question is not worth answering (BL, pp. 154–155). These are also ▲s, since the reacting partner understood and evaluates the ★.

**Example 6.3: Reacting to Question With ‘Improper’ Answer** (CA2, 28:16–28:33)

C5 wants to use Eclipse’s “Change Method Signature” refactoring to ease the pair’s next implementation step, but cannot remember the keyboard shortcut, and asks his partner about it (turn 1). C2 understands the underlying idea, but doubts its benefits and reacts with an *improper* answer (turn 2).

(1) ★ C5: “Erm, do you know the (!...!) how I access the feature to change a method?”  
*ask\_knowledge (propose\_step)*

(2) ▲ C2: “That doesn’t get you anywhere.” *explain\_knowledge (disagree\_step)*

(3) ▲ C5: “It does. In the method, once opened, I can turn the `IVirtualColumn` into `I (!...!)`”  
*challenge\_knowledge + propose\_design*

(4) ▲ C2: “That doesn’t buy you much. But it’s ALT SHIFT C, do it with ALT SHIFT C.”  
*challenge\_knowledge + explain\_knowledge*

Note that *pair-referential* does not imply *affirmative*: C2 clearly rejects C5’s idea and C5 objects C2’s assessment (*disagree\_step* and *challenge\_knowledge*, respectively), but both reactions refer to C5’s original ★ that both developers are involved in.

There is a continuum of developer activities between ★ and ▲: Both are directed at the partner to some degree, while the novelty of what they introduce into the discourse varies. I distinguish these two types mainly for readability, not for distinguishing different levels of *Fluency*. Occasionally, I annotate both ★ and ▲ on one activity, e.g., when the developer starts a new conversational strand based on a previous one.

### 6.2.2 c) ◀ – Self-Referential Activity

Sometimes a developer corrects, amends, or otherwise evaluates her own ★ before the partner reacts to it. I call these activities *self-referential* or ◀. If the partner did already react but the speaker did not yet acknowledge her partner’s involvement, an amendment of one’s own ★ is also a ◀ (as opposed to a ▲). Relevant for self-referential activities ◀ is that—from what the researcher can tell—the speaker is in a sense ‘alone’ in her ★, as is shown in the next example:

**Example 6.4: Self-Referential Activity without Partner Involvement** (CA2, 31:17–31:57)

C5 has performed the “Change Method Signature” refactoring he proposed in Example 6.3, which resulted in a number of compilation errors, which the pair started to go through one by one. After fixing the first error, the pair gets interrupted for a few seconds, after which C2 appears to zone out and C5 focuses on the second error alone:

- (1) ★ C5: < \*opens the file with the next compilation error from their list, sets text cursor next to the error\* > do\_sth

The pair gets interrupted for 10 seconds, both look back at the screen afterwards:

- (2) ◀ C5: “< \*moves cursor along the line with the error\* > OK” examine\_sth  
 (3) ◆ C2: < \*starts to look around the office\* >  
 (4) ◀ C5: “Oh, I expected he would change this one, too.” explain\_finding  
 (5) ◆ C2: < \*looks around the office\* >  
 (6) ◀ C5: < \*manually renames a method\* > “Mhm” < \*undoes the renaming, lets the IDE generate an empty method body instead\* > write\_sth + mumble\_sth  
 (7) ◆ C2: < \*still looks around the office, eventually looks back at screen\* >

After the interruption, C5’s picks up on his own ★, while C2 did not get involved in any way. C2 does not react to C5’s *explain\_finding* in turn 4, and C5’s two attempts to fix the compilation error in turn 6 remain unseen by C2. In fact, C5 does not even seem to be aware that C2 is not paying attention for about 20 seconds.

If, on the other hand, the partner is already involved with the respective ★ and the original speaker acknowledged this involvement, she is no longer ‘alone’ and I assume all further activities of the two developers to be pair-referential ▲, as is illustrated in the next example:

**Example 6.5: Self-Referential Activity With Partner Involvement** (CA2, 32:41–33:07)

The pair worked through all but one compilation error. C5 opens the class with the last one and proposes a code change shortly after.

- (1) ★ C5: < \*opens the last compilation error instance\* > do\_sth  
 (2) ◀ “Erm (.) ah, ok, we need to change them. (##VirtualAttributes##)” propose\_design  
 (3) ▲ C2: “And rename them” amend\_design  
 (4) ▲ C5: < \*changes the type of a class variable to VirtualAttributes and renames it to virtualAttributes.\* > write\_sth

The pair had planned to go through all compilation errors. This is the last one on their list, so in this context the act of opening the file (turn 1) is effectively a ★ like ‘*Let’s find out what needs to be done here*’, even though neither of the two said a word. I characterize C5’s *propose\_design* in turn 2 as a ◀ because his partner is not yet involved in the ★.

C2’s involvement only becomes evident with his *amend\_design* in turn 3. This is why I consider C5’s next turn 4—where he acknowledges C2’s involvement by putting C2’s proposal into action—a pair-referential ▲ instead of a self-referential ◀.

**6.2.2 d) ● – Corrective Activity**

If a partner recognizes some sort of misunderstanding between herself and her partner, she may attempt to repair this with a ● *corrective activity*. Such misunderstandings might be due to misread or simply not understood intentions, or due to one partner not possessing knowledge that was considered *common ground* by the other.

Neither one developer’s perception of a misunderstanding nor the misunderstanding itself can be directly observed, only the actions of the pair members dealing with it. For the involved

developers, the detection can happen in two places: Internally (a developer notices something in herself) and externally (a developer notices something in her partner).

### **Internal Detection of Misunderstanding**

Upon trying to understand her partner’s base activity, the developer may notice that she lacks information to react properly and then ask for it explicitly.

#### **Example 6.6: Corrective Activity (CA2, 35:21–35:39)**

C5 proposes to move a class to a different package, but his partner C2 is not able to interpret the meaning of “*this*”.

- |   |                                      |
|---|--------------------------------------|
| ★ C5: “We need to move this as well.”   | <i>propose_design</i>                |
| ● C2: “Where are you? In what class? Or in which module?”   | <i>ask_knowledge</i>                 |
| ● C5: “What I did is, there is (!!...!!)”   | <i>explain_knowledge</i>             |
| ● C2: “In which module are you right now?”  | <i>ask_knowledge</i>                 |
| ● C5: “I moved the Factory, with which I create these VirtualColumn, or these IVirtualColumn, I also moved to basis.” | <i>explain_knowledge</i>             |
| ● C2: “Ah, I see.”  | <i>explain_standard of knowledge</i> |

Note that C5 does not directly answer C2’s question, but explains the broader context of what they did in the previous minutes, presumably to make sure to clear up any misunderstanding.

The developer may also have trouble understanding the partner’s base activity altogether. A repair can then be as simple as a ‘*pardon?*’—a request to repeat what was said before.

#### **Example 6.7: Corrective Activity (CA2, 37:52–38:18)**

Both developers read in the source code and C2 seems a bit lost. In the excerpt below, C5 therefore starts explaining what the current class is about. C5 does not use the full class name and C2 does not understand the explanation, which leads to series of corrective activities.

- |  |                                     |
|--|-------------------------------------|
| ★ C5: “This is another implementation, which does not use the Abstract.” | <i>explain_knowledge</i>            |
| ● C2: “(.....) Another implementation of FeatureAttributeConfiguration?” | <i>ask_knowledge</i>                |
| ● C5: “Yes, yes.”  | <i>agree_knowledge</i>              |
| ● C2: “Which does not use <u>what?</u> ”                                 | <i>ask_knowledge</i>                |
| ● C5: “Which does not use the abstract class we adjusted.”               | <i>explain_knowledge</i>            |
| ● C2: “(...) Ok, so we need to adjust it here?”                          | <i>propose_design</i> <sub>OE</sub> |
| ● C5: “<with relief> Yep, exactly.”                                      | <i>agree_design</i>                 |

### **External Detection of Misunderstanding**

The original speaker might catch on cues in her partner’s ▲ that make her start a ●:

1. **Timeliness:** A ▲ may be **prompt**, coming immediately after the ★, indicating it was easy for the partner to understand and respond to it; or it may be **delayed** for some time, indicating to the original speaker that it took her partner a bit longer to understand the ★ and/or to formulate a response, possibly because of a wrong or lack of understanding.
2. **Evaluativeness:** The base layer discriminates between *constructive* and *unconstructive* activities (BL, p. 49), i.e., those that add new aspects into the discourse (such as *amend* or *challenge*) and those that merely make a judgment (such as *agree* or *disagree*). Both imply that the reacting speaker (1) understood the ★ and (2) evaluates it. I call such ▲s **evaluative**.

There are, however, ▲s which do *not* evaluate the ★, but merely acknowledge that it was made. From the researcher perspective, it is usually hard to distinguish whether the reacting partner did not understand the ★ or whether she *did* understand it, but does not evaluate it. I therefore do not distinguish these cases and call all such ▲s *non-evaluative*.

3. **Appropriateness:** A ▲ may be *appropriate*, or it may be *misled* when the partner did not understand the original ★. Note that this is not *objective* appropriateness: I characterize a ▲ as *appropriate* when the partner (for the moment) behaves as if it was appropriate.

A developer who receives a ▲ which is *delayed*, *non-evaluative*, or *misled*—or no reaction at all (in a sense maximally *delayed* and *non-evaluative*)—may start a ●, e.g., by rephrasing her ★ to make the partner understand her intention or by providing additional information to make sure that what she falsely assumed to be *common ground* will be understood by both pair members. The next example shows a ● after a *prompt* and *appropriate*, but *non-evaluative* ▲.

**Example 6.8: Expected Common Ground** (CA2, 37:15–37:30)

C5 is faced with the choice between two methods—`getColumnAttributes` and `getALLColumnAttributes`—and asks his partner about it, presumably expecting C2 to possess the necessary background knowledge to make the choice. C2 appears to understand the design proposal, but is not able to make a judgment, so his reaction remains *non-evaluative*. C5 makes his choice, and explains what the difference between the options was.

- ★ C5: “OK, that one needs all, right? Think so.” *propose\_design*<sub>OE</sub>
- ▲ C2: “No idea (!!Yes!!) what it does. Ok.” *explain\_standard of knowledge*<sub>RT</sub>
- C5: “< \*chooses `getALLColumnAttributes` \* > (..) That’s for getting a unique name for the attribute.” *explain\_knowledge*
- C2: “M-hm. I see. Ok.” *explain\_standard of knowledge*<sub>AT</sub>

The base concept indices are explained in Section 3.4.2c (pages 134 and 135): C5’s proposal is of mode *obtain evaluation* (OE), i.e., requesting the partner to make a judgment while expressing a positive evaluation himself; C2 first *refuses the knowledge transfer* (RT); after C5’s explanation, C2 *acknowledges the knowledge transfer* (AT).

Additional references while developers are in the midst of correction (as in Example 6.7) are not coded as ▲ or ◀: The actions are just part of an attempt to repair each other’s understanding of the first ‘level’ of reciprocal communicative knowledge (see Section 3.2.1a); I do not discriminate higher levels.

In the base layer, neither misunderstandings nor corrections are explicitly addressed. In fact, Salinger & Prechelt propose the researcher should consider the partner’s reaction in case the original speaker’s primary intention is difficult to reconstruct—which only works if the partner correctly understood the intention (BL, p. 206).

### 6.2.2 e) ◆ – Conversational Defect

Sometimes developers do *not* refer to the current ★ at all. There are several forms such a missing link, a ◆ *defect*, can take in a conversation. None of these phenomena are addressed by existing concepts or properties in the base layer.<sup>1</sup>

- One partner may start a new ★ without making clear its relation to the previous ★ (e.g., whether it is meant as a counter-proposal), as in the next example:

<sup>1</sup>In their pair study, Okada & Simon (1997, Table 7) identified *disregard of the request* as one of six reaction patterns following a request for explanation. See also discussion on page 96.

**Example 6.9: Unrelated Proposal** (CA2, 43:02–43:26)

After the refactoring is finished and the IDE does not list any more compilation errors, C5 proposes to start the application to check whether they broke anything. C2 has a different idea regarding an API extension, which he outlines without acknowledging C5's proposal.

- (1) ★<sub>A</sub> C5: "Now, we're error-free again." *explain\_state*
- (2) ▲<sub>A</sub> C2: "OK." *agree\_state*
- (3) ★<sub>B</sub> C5: "Now (,,,) what I'd be interested in now, if we check the GUI, (!!Wait a second!!) if the GUI still works." *propose\_step*
- (4) ◆<sub>B</sub>★<sub>C</sub> C2: "I'd like to build a method to get them as raw as they are (!!M-hm!!) and then I'd like to take a look at the thing with the dialog and so on." *propose\_strategy*
- (5) ▲<sub>C</sub> C5: "OK." *agree\_strategy*

C2 does not take a clear stance on C5's proposal to check whether the application is still working. The "thing with the dialog" might be a reference to the GUI as well, but judging from C2's own words it is not clear whether he understood C5's proposal. Quite possibly, he was not even listening to C5 ("Wait a second"). If C2 really had testing the GUI in mind at this point (the following events in the session do not make this clear), it was an idea that was independent from C5's, so a connection to C5's ★<sub>B</sub> is missing.

Note that—just like in software—a *defect* does not necessarily lead to a *failure*: C5 does not object C2's interjection and this particular ◆ in their conversation does not appear to be problematic for their process.

- There might be no reaction whatsoever, as in the next example, where the ★ was clearly expecting an evaluation:

**Example 6.10: Non-Actions** (CA2, 55:41–55:49)

C2 has moved the text cursor to the position of an unnecessary `if`-block which the pair previously talked about. He asks C5 about this, does not get any reaction, and then deletes the statements.

- (1) ★ C2: "This we wanted to delete, with the equals null?" *propose\_design*<sub>OE</sub>
- (2) ◆ C5: "(.)"
- (3) ● C2: "Am I right?"
- (4) ◆ C5: "(...)"
- (5) ◀ C2: < \*deletes the lines in question\* > *write\_sth*

C5 remains silent both times and stares at the screen, possibly thinking about the implications. The following events do not allow for a conclusive interpretation of his behavior.

- The developer might continue some older ★ of her own, without relating to the most recent actions, as in the following example:

**Example 6.11: Following One's Own Initiative** (CA2, 1:14:07–1:14:42)

C2 tries to explain the problem he sees with the next steps towards the implementing of the currently planned design, while C5 (again) tries to formulate an argument in favor of the design he started prior to the session, but which the pair undid during the last hour. Neither of the two actually reacts to what the partner just said, they merely fill each other's pauses.<sup>a</sup>

- (1) ★<sub>A</sub> C2: "Well, our problem (.) is (.) that we don't have FeatureProxies per se < \*looks at C5\* > (.) for our case." *explain\_knowledge*
- (2) ◆★<sub>B</sub> C5: "Well, you know I talked to < \*\*lead developer\*\* > yesterday (!!or do we?!!) about this." *explain\_knowledge*
- (3) ◆◀<sub>A</sub> C2: "< \*looks at screen\* > (.) although (!...!)" *examine\_sth*

## Example 6.11 (continued)

- (4) ◆<sub>B</sub> C5: “<\*pulls a sheet of paper towards him\*> Which explains what I did with the Column earlier.” *explain\_knowledge*
- (5) ◆<sub>A</sub> C2: “Although (,,,) well, I mean (!...!) (...).” *examine\_sth*

Note that both ★<sub>A</sub> and ★<sub>B</sub> are **non-expecting**: There is no question to be answered and no proposal to be evaluated, but just a fact being stated without a clear indication what the partner is supposed to do with it. On a technical level, C5’s first utterance might actually have a connection to C2’s assessment (e.g., something like ‘*we wouldn’t be in this mess if we just followed my original design*’), but neither does C5 make it explicit, nor does C2 seem to understand the utterance in this sense.

<sup>a</sup>Jones & Gerard (1967, cited by Argyle et al., 1981, p. 223) call such an interaction *pseudo-contingent*.

The ◆-property is independent from the types previously discussed: Any action of the other four types (and even non-actions) can be a ◆ as well.

The relevant criterion for a ◆ is that the developer had time to understand the previous turn but does not react to it as would be expected under “normal” communication conditions (i.e., both speaking the same language, being conscious about what they do, not engaging in any other non-literal use of language, etc., Searle, 1969, p. 57). Accordingly, there is another type of ◆, in which the developer *does* recognize one of the markers mentioned above (e.g., taking longer than normal and react *delayed* or only react in a **non-evaluative** fashion, see Section 6.2.2d) but does *not* engage in a ●. In the majority of the analyzed sessions, there were very few instances of this type, such as the one in the next example. Sessions OA1 and OA8, however, are exceptional in that they contain several such non-correcting ◆s. I discuss the resulting **Breakdowns** in Section 6.3.4.

**Example 6.12: Not Clearing Up** (CA2, 16:45–17:05)

Early in the session, the pair looks at a method which C5 changed prior to the session. C2 did not know the source code and started reading. He discovered that a variable was initialized several statements before it was used the first time, and moved the initialization statement to the last possible point. In the excerpt below, C5 disagrees with C2’s change as it does not fit with his design idea, and he explains that currently the change is not technically wrong, but it will be wrong in the near future.

- (1) ★ C5: “Well, as it is now, it has no effects, because the other things are not working yet. But if we keep that order, the moment this <\*points to statement\*> gets active, if we implement it the way I thought, this would run into an error.” *explain\_knowledge*
- (2) ▲ C2: “<indifferent> It is right then.” *agree\_knowledge*
- (3) C5: <\*turns around sharply, looks at C2, gasps (!...!)\*>
- (4) ★ C2: “OK, but now let’s have a look <\*takes mouse again, and continues reading in code\*>” *propose\_step*
- (5) ◆ C5: “M-hm” *agree\_step*

C2’s indifferent reaction comes as a complete surprise for C5, as if C2 has no idea what C5 just said but just agreed without thinking to get going again. C5 appears as if he is about to say something (turn 3) but is cut off by C2. The ◆ lies in C5 not insisting on a clarification (turn 5) *despite* the apparent differences between his and C2’s understanding. This matter remains unresolved.

Note that even though the color and the name “*defective activities*” may suggest that ◆s are ‘bad’, they are basically a marker of violated expectations. They may occur in **normal PP** and even during **Focus Phases** without indicating much harm—just like not every defect in a software causes it to fail.

### 6.2.3 Discussion of Recurring Example

The recurring example from session JA1 does not show all five base activity types, but it illustrates the breadth of ▲ and ● phenomena. (See Example 5.1 for technical background.)

#### Example 6.13: Five Types of Base Activities (JA1, 04:09–06:15)

J2 starts an explanation (★) and J1 listens. All following turns are pair-referential activities (▲).

- (4) ★ J2: "OK, I can give you the big picture of what this plugin does, overall."
- (5) ▲ J1: "Yep."
- (6) ▲ J2: "In the end, the news recording of every hour pops out."
- (7) ▲ J1: "M-hm."
- (8) ▲ J2: "The way this works is that there are multiple processors, so there is the central plugin and multiple processors which each handle one wave."
- (9) J1: < \*nods\* >
- (10) ▲ J2: "For most of them, right after the full hour there is a check, if there is a new file on the remote share."
- (11) ▲ "If so, the most recent file is selected and it starts checking how the file changes size-wise."
- (12) J1: < \*stops nodding, looks to his upper right\* >
- (13) ▲ J2: "I mean, it looks until the file does not get bigger anymore, then it is apparently ready."
- (14) ▲ J1: "M-hm"
- (15) ▲ J2: "And then it is fetched and handed over to transcoding."

Subsequently, after J1's question (★), J2 over and over provides answers (▲) without seeing any misunderstanding. J1, however, rephrases and reframes his question (●) until J2 also acknowledges the misunderstanding (●) and provides the required information.

- (16) ★ J1: "In what time window are you looking?"
- (17) ▲ J2: "I start looking two minutes after the full hour, because then it's guaranteed that news files exist if any exist."
- (18) ▲ J1: "OK"
- (19) ▲ J2: "And then monitor this file as long as needed until it's ready. That can take up to seven minutes, depending on the wave."
- (20) ● J1: "Hm ya but mh the time window for the change?"
- (21) ▲ J2: "Yes, right, that is, er, time window for the change is variable, depends on how the news go."
- (22) ▲ "I can't know that, right? You know, they always start a new file. When the news are over, again a file is created. I mean, I basically never have more than the news."
- (23) ● J1: "Yes, no, I mean 'cause you said you look for so long until the size stops changing, right?"
- (24) ● J2: "M-hm"
- (25) ● J1: "Then you need to plan for a time window in which a change could happen, right?"
- (26) ▲ J2: "Yeah, well, until up to five before the hour. I really take my time."
- (27) ● J1: "<laughs> No, I really mean the size now, the size of the time window, I mean (!...!) You wait for 10 seconds, then after 10 seconds you decide: In those 10 seconds nothing has changed, so the file appears to be ready."
- (28) ● J2: "Ahhh, that's what you mean. No, 30 seconds."
- (29) ● J1: "30 seconds, that's what I wanted."
- (30) ● J2: "That's 30 seconds long the time window. Now I got you."

Equipped with the vocabulary introduced in this section, different levels of pair programming process quality, or **Fluency**, can be characterized.

## 6.3 Fluency

**Fluency** is an observable property of the pair programming process. I distinguish three levels:

- **Normal PP** is *normal* in the sense that most of the analyzed pair programming sessions are this **fluent** most of the time: There are no long pauses between base activities and developers evaluate each other's initiatives. Misunderstandings occur and are cleared up.
- **Focus Phases** are episodes that are *faster* than a **normal** process. There are virtually no pauses between the base activities and both developers are able to focus entirely on the current task without the need to correct misunderstandings.
- A **Breakdown** is an episode during which there is no real *pair* process. The developers neither request nor provide an evaluation of their respective ideas. Misunderstandings occur and they are not repaired. There are long intervals of silence between the base activities.

Each of these levels is characterized by a pattern of base activities (see Table 6.2).

**Fluency** is not the same as the rate of technical progress: The pair may struggle with a difficult problem, which requires them to take multiple attempts, but still have a **fluent** pair programming process in which they produce, evaluate, and decide on ideas together. The beginning of session JA1, for example, has a lengthy exchange between developers J1 and J2 during which very little technically relevant information is transferred (see Example 6.13), but it is **fluent** and an example of **normal** PP.

### 6.3.1 Foreword to the Fluency Examples

A word of warning: The examples in the following sections are special in a number of ways. First, they each cover several minutes of session time, which is considerably longer than most examples in this document. This is due to the process nature of the **Fluency** concept which cannot be reasonably illustrated in short exchanges. Second, I provide much technical background information from the respective PP session before I actually present the transcript. This information is necessary because the pair's dialog does not only contain conversational references (marked by the icons ★, ▲, etc.) but also technical references, which I mark with boxed numbers 1, 2, etc. My intention is to enable the reader to come close to understanding the developers' utterances and actions the way their respective partner could. Third, the passages are transcribed in great detail, including gestures and computer interactions (HCI activities). The transcripts are coded with base concepts to make the speakers' (reconstructed) primary intentions clear. Each of the examples spans multiple pages, so you better get comfortable.

The three **Fluency** levels also vary along another dimension: the ease with which an outsider can understand the developers' activities and utterances. During a **Focus Phase**, the developers do not need many words to understand each other—many things remain implicit which leaves fewer clues for the reconstruction. In **normal** PP, the developers usually understand each other, and when they do not, misconceptions are cleared up, i.e., things are made explicit that otherwise would remain implicit, which means more clues for me as the researcher. During a **Breakdown**, however, miscommunication is *not* cleared up, so the pair members' confusion becomes observable, but more difficult to fully reconstruct.

The relative scarcity of verbal cues in **Focus Phases** and **Breakdowns** makes reconstructing the developers' behavior more ambiguous, sometimes leaving more than one plausible inter-

Concept	Description/Characterization
Fluency	Property of the pair process, characterizes both the rate at which the pair performs base activities and the degree to which these activities refer to each other. I distinguish three levels:
– normal PP Section 6.3.2	The ★s request an evaluation or are otherwise directed at the partner, who reacts with <b>evaluative</b> and <b>prompt</b> ▲s. There are only few ◀s and rarely any ◆s. Occasional ●s are <b>normal</b> , they clear up misunderstandings.
– Focus Phase Section 6.3.3	The ★s are less directed at the partner but more at the pair as a whole; ▲s are <b>prompt</b> (sometimes they start even before the ★ is finished, because the partner understood the ★ already) and <b>evaluative</b> , usually affirmative, e.g., in the form of proposals put directly into action. Misunderstandings and ◆s are very rare, making ●s mostly unnecessary. Both partners are usually involved in the ★s, so ◀ rarely happen.
– Breakdown Section 6.3.4	The ★s do often <i>not request</i> an evaluation or lack a clear indication how the partner is supposed to react; there are many ◀s and ◆s. If the partner gets involved in an ★, the ▲s tend to be <b>delayed</b> and <b>non-evaluative</b> . Most prominently, ●s are scarce, misunderstandings that underlie <b>misled</b> or missing reactions are <i>not</i> addressed and cleared up.

Table 6.2: Fluency and its three levels

pretation. In the respective examples, I will generally provide only *one* interpretation, instead of iterating all the ways the pair members might have (mis)understood each other. My goal is not to provide a bit-perfect linguistic coding of the respective episodes, but a characterization of the general process properties, which lies in the pair being able to understand each other easily without many words or the pair failing to clear up misconceptions, respectively.

In first discuss **normal PP** in Section 6.3.2 as a baseline, then characterize how well pairs can work during **Focus Phases** in Section 6.3.3, before I discuss **Breakdowns** as the other extreme in Section 6.3.4.

### 6.3.2 Normal Pair Programming

In **normal PP**, both developers produce and evaluate ideas, and the overall process is **fluent** in the sense that there are no long pauses between the base activities and the pair as a whole gets a better understanding over time. This does not mean that the pair members agree with each other all time: *Negatively* evaluated proposals, for example, do not automatically lower a pair's **Fluency** if the partner is able to articulate her disagreement and the original speaker understands it. Misunderstandings concerning the partner's intentions also happen, but this happens in everyday communication as well. In **normal PP**, the developers detect and repair those misunderstandings. The beginning of session JA1 (see Example 6.13 on page 199) is one example; the following excerpt from session OA8 is another:

#### Example 6.14: Normal Pair Programming (OA8, 49:10–51:19)

##### Session Background

The software has a calendar with draggable events represented by horizontal bars. When such a bar is dragged, a 'ghost' is displayed under the user's cursor. Multi-day events are defined by a start and an end date; they may extend over weekends but are measured in workdays. The ghost of such an event therefore has a variable length depending on how many weekends it covers. For a smooth animation, the rendering logic depends on where the user grabs the bar.

## Example 6.14 (continued)

This logic is currently broken and is to be fixed by developers O3 and O4. At this point in their session, they are in the process of writing a test case to automatically reproduce the faulty behavior. The pair spoke English in their session, which neither of them can speak effortlessly, but which is their strongest common language. The transcript below is verbatim.

**Technical Background**

The source code (see Figure 6.1) shows the relevant properties of the event bar’s data structure: [1] start and end of the event determine its duration, [2] `offsetFraction` is the relative grabbing position (ranging from 0 to 1), [3] `offsetDays` is the zero-based index of the event day that corresponds to the value of `offsetFraction`. The pair knows these identifiers, but is not fully aware of the meaning of all of them. The consideration of weekends [4] is done elsewhere.

There was an existing test case ([5], code lines 94–100 in Figure 6.1) for a three-day event being grabbed right in the middle (`offsetFraction: 0.5`) for which the correct day index would be `offsetDays: 1` (day 0 is in range 0 to 0.33, day 2 is between 0.67 to 1.00). The pair already copied that test case, set the event’s start and end nearly a month apart, and changed the `offsetFraction` to 0.95 to simulate grabbing the bar near its end.

```

94 dataModel.getStart.returns moment '2000-03-04'
95 dataModel.getEnd.returns moment '2000-03-07'
96 rowView.emit 'dragstart', offsetFraction: 0.5
97
98 expectedDragProperties = sandbox.match
99   offsetDays: 1
100 expect(timeline.setDragProperties).to.have.been.calledWith expectedDragProperties

103 dataModel.getStart.returns moment '2016-06-02'
104 dataModel.getEnd.returns moment '2016-06-29'
105 rowView.emit 'dragstart', offsetFraction: 0.95
106
107 expectedDragProperties = sandbox.match
108   offsetDays: 1
109 expect(timeline.setDragProperties).to.have.been.calledWith expectedDragProperties

```

**Figure 6.1:** Relevant excerpts of the test code written in CoffeeScript. The original test case is in lines 94–100, the copied (and not yet completely adapted) test case starts in line 103.

**What happens**

The pair understands that they need to set a specific expectation value in code line 108 (see Figure 6.1), realize they do not yet know the meaning of the field “`offsetDays`” (aspect [3]), and come up with an idea how to figure it out. In their dialog, they refer to all aspects [1]–[5].

- (1) ★<sub>A</sub> O4: “< \*moves cursor to line 108\* > And the offset is?” O4 asks for a specific value of [3] to expect in the test case. *ask\_design*
- (2) ▲<sub>A</sub> O3: “Erm, offsetDays would be (.) a lot. Because, like the difference between these two guys < \*points to start and end dates\* >, right?” O3 only gives a characterization. *propose\_hypothesis*
- (3) ★<sub>B</sub> O4: “What was the meaning of offset Days?” O4 takes a step back and ponders the meaning of [3]. *ask\_knowledge*
- (4) ▲<sub>B</sub> O3: “It’s the distance < \*holds up two index fingers\* >. (..) So, how big is the bar < \*points at screen with two fingers\* >.” O3 believes that “`offsetDays`” refers to the duration of the event (which would be aspect [1]). *explain\_knowledge*

## Example 6.14 (continued)

- (5) ▲<sub>B</sub> O4: “No, it’s an offset, not a duration or width.” *challenge\_knowledge* O4 finds O3’s explanation implausible.
- (6) ▲<sub>B</sub> O3: “‘Offset’ is distance, so it’s like distance < \*mimics growing and shrinking distance between her two index fingers\* > (..) so the distance that goes from the beginning of the bar to the end of the (.) bar, I think.” *propose\_hypothesis* O3 argues in favor of her explanation, but she now sounds less convinced.
- (7) ▲<sub>B</sub> O4: “< \*hovers lines 94 and 95 in the previous test case’s setup\* > No, it must be three or five days here in this. But it’s 1! < \*selects the assertion in line 99\* >” *challenge\_hypothesis* O4 found a counterexample in [5] (the original test case). He probably meant to say ‘No, then it would have been three or five days in this test.’
- (8) ★<sub>C</sub> O3: “(.....) (Erm, has it to do something with the weekend?)” *propose\_hypothesis* O3 half-heartedly hypothesizes that [4] may explain the low number of 1.
- (9) ◀<sub>B</sub> ♦<sub>C</sub> O4: “< \*hovers line 96\* > (#zero point five#) (.) < \*hovers lines 94 and 95\* > three days” *think\_aloud\_activity* O4 ignores O3’s hypothesis. Instead, he re-reads the values for [2] (offset Fraction, “zero point five”) and [1] (event duration, “three days”) from [5] (original test case).
- (10) ★<sub>D</sub> O3: “But we can check that if we console-log these. < \*looks at O4\* >” *propose\_step* O3 wants to instrument the production code and play around. It is not clear whether “that” refers to her weekend-hypothesis ([4]) or the field’s meaning ([3]).
- (11) ● O4: “Hm?” Either way, O4 does not comprehend her idea.
- (12) ▲<sub>D</sub> O3: “We can check that, if you console-log, erm, so this, the real variable value, here in the < \*production code file\* >, we can like console-log the (..)” *propose\_step* O3 points O4 to the production code to make her idea easier to grasp.
- (13) ▲<sub>D</sub> O4: < \*switches to production code\* > *do\_sth* O4 follows along.
- (14) ▲<sub>D</sub> O3: “that object that has the offsetDays (..)” *amend\_activity + propose\_step* O3 directs her partner to right location.
- (15) ▲<sub>D</sub> O4: < \*scrolls down\* > *do\_sth* O4 navigates to the calculation.
- (16) ▲<sub>D</sub> O3: “yeah, we can console-log this. < \*looks at O4\* >” *agree\_activity + agree\_step* O3 repeats her idea to print out the field’s value ([3]).
- (17) ▲<sub>D</sub> ★<sub>E</sub> O4: “(..) In the test, we have to think about what the right value is before we start the test. < \*looks at O3\* >” *disagree\_step + explain\_knowledge* O4 argues against O3’s proposal. He seems to think that she wants to make the test green by recording the current (faulty) behavior and simply declaring it ‘correct’ in the test case.
- (18) ● O3: “Right, but like, what’s the meaning in the real life of offsetDays? < \*looks at O4\* >” *agree\_knowledge + propose\_step* O3 starts another attempt of her explanation, as O4 did not understand her idea.
- (19) ● O4: “This is what I want, I’m (thinking about).” *explain\_standard\_of\_knowledge* O4 recognizes his original question regarding [3].

## Example 6.14 (continued)

- (20) ● O3: “<nods> So, in order to do that I would say, let’s check these values. Let’s console-log this.” Now that they are on the same page, O3 repeats her proposal.  
*propose\_step*
- (21) ● O4: “Ah, you mean on the <\*hovers the calendar view with cursor\*>, when I make a manual test, we log it out.” O4 rephrases the idea to make sure he got it this time.  
*explain\_standard of knowledge*
- (22) ● O3: “Yeah, exactly. <\*looks at O4\*>” O3 acknowledges O4 understanding her idea.  
*agree\_standard of knowledge<sup>a</sup>*
- (23) ● O4: “Ok (..) to have feeling what it, what the meaning of this is.” O4 drives the point home by circling back to his original question of the field’s meaning.  
*explain\_standard of knowledge*
- (24) ● O3: “Exactly.” *agree\_standard of knowledge* Which is what O3 had in mind.

<sup>a</sup>The base layer does not contain the concept *agree\_standard of knowledge* as according phenomena were rare and the more general *agree\_knowledge* was sufficient (BL, p. 142). I introduce this concept here because of key decision #6 *Model dialog episodes* (BL, pp. 40–41, discussed on page 132), as the pair clearly has a dialog episode around O4’s *standard of knowledge*.

To summarize the key properties of **normal** pair programming:

- Both developers produce ideas and the ★s are **expecting**. In Example 6.14, see the *ask* activities in turns (1) and (3), and the *propose* and *explain* activities where the respective speaker looks at their partner in turns (10) and (17). O3’s half-hearted *propose\_hypothesis* in turn (8) is the only exception.
- Both developers evaluate ideas. The ▲s following the ★s are **prompt** and **evaluative**; there are no long pauses between the subsequent ▲s either, and few ◀s and ◆s (again, the *think aloud* period of turns (8)–(9) being the only exception).
- Misunderstandings are detected and dealt with by ●s; see the two times O4 did not understand O3’s proposal (turns 11 & 17).

This level of pair programming **Fluency** sets the baseline for characterizing episodes where the pair comes close to “*working with one mind*”, the **Focus Phases**, as well as the other extreme of **Breakdowns** where *pair* is no longer an appropriate characterization.

### 6.3.3 Focus Phases in Pair Programming

**Focus Phases** are mentioned in multiple publications on pair programming by both practitioners and researchers. Here are a few examples:

- Williams et al. (2000, p. 20) characterize pair programming in general as “*two programmers [who] are like a unified, intelligent organism working with one mind*”.
- Belshee (2005, Sec. 1.2) calls it “*Pair Flow*” where “*the solution and problem spaces are shared between the minds of the participants*” who then work “*significantly better*”.
- Chong & Hurlbutt (2007, Sec. 5.1.2) note that pairs “*sometimes slipped into a mode of behavior where they were exceptionally in sync with one another*”. They add that it “*was always recognizable from the incomplete verbal utterances between the two participants*”.
- Salinger et al. (2008, p. 20) have observed *pair phases* which are “*characterized by a high density of communication acts referring to just one narrow issue*”.

In contrast to Williams et al. (2000) and Belshee (2005), both Chong & Hurlbutt (2007, Sec. 5.1.2) and Salinger et al. (2008, p. 20) note that such a mode only lasts for a short duration. These

sources neither provide a detailed characterization or give concrete examples nor do they address how such a mode may arise.

I have identified the **Focus Phase** as a phenomenon which fits these characteristics. During a **Focus Phase** the pair members appear to not need much evaluation of ideas: proposals are made, immediately approved, and put into action (e.g., ★-▲ or ★-▲-▲). There are hardly any misunderstandings, so ● are not necessary.

Session & Time			Notes
#1	CA5	19:12–20:11	Stripping down of copy-pasted code, see Example 6.15
#2		20:32–21:42	Continuation of <b>Focus Phase</b> #1: Pair works through remaining 20 lines of copy-pasted code in the same manner (about 35 base activities, not discussed here). The pause between <b>Focus Phases</b> #1 and #2 is due to C3 typing only with his index fingers and that is how long it takes him to complete a TODO comment.
#3		26:32–28:06	Again five minutes later during which the pair integrates the (hull for the) new feature in existing structures in <i>normal</i> PP. Now, the pair inspects three different classes, decides on one abstract class to extend, adds about ten lines of remaining integration code such that they only need to add the actual geometry logic, and removes two blocks of now unused code (about 40 base activities, not discussed here).
#4	AA1	1:53:20–1:54:56	The pair adds two new parameters to a constructor and amends the calls (see Example C.1)
#5		1:55:45–1:57:04	The pair adds two new properties to backend response message (see Example C.2)
#6		1:57:38–2:00:55	The pair discusses design options; they perform no code changes and do not even read code during this period (see Example C.3)

**Table 6.3:** Overview of the **Focus Phases** I identified in my data. Note that I did not systematically search for exemplars.

Since my research topic was not the quality of PP processes but the mechanisms of knowledge transfer in pair programming, I did not systematically search for **Focus Phases**, but only report on six exemplars that caught my attention along the way (see Table 6.3): Three in session CA5 and three more in session AA1. Here, I discuss one concrete **Focus Phase** of one minute length where the pair in session CA5 quickly strips down existing code (Example 6.15). For two further **Focus Phases**, I provide the full transcripts and notes in the appendix: One where the pair from session AA1 adds logic to existing code for two minutes (Example C.1), and the other to illustrate that changing source code is not necessary for a pair to have a **Focus Phase**: Late in session AA1, the pair develops and discusses multiple design ideas for about three minutes (Example C.3).

**Example 6.15: A Focus Phase** (CA5, 19:12–20:11)

**Session Background**

Company C develops a graphical geo-information system. A new feature should allow users to cut existing *geometries* (such as points, lines, polygons) into parts by drawing arbitrary shapes across them. Developers C3 and C4 already found the right place in the class hierarchy of existing geometry-altering actions and copy-pasted the method body of an existing action into a new class.

**Technical Background**

Figure 6.2 shows the relevant parts of the source code after the pair has copy-pasted it.

## Example 6.15 (continued)

```

31 protected void execute(final Component parentComponent) {
32     final Message message = (Message) validator.validate();
33     if (message != null) {
34         SwingMessageIndicator.showMessage(parentComponent, message);
35         return false;
36     }
37
38     final EditOptions editOptions = new EditOptions(new MapModelSelection(snapThemeModel));
39     final IScaleRange scaleRange = new ScaleRange(0, getFeatureLayer().getBaseScale());
40     editOptions.setScaleRange(scaleRange);
41     try {
42         final EditGeometryType editGeometryType = getEditStrategy().getEditGeometryType(
43             getFeatureLayer());
44         editOptions.setGeometryType(editGeometryType);
45     }
46
47     return true;
48 }

```

**Figure 6.2:** Relevant excerpts of the Java code before the *Focus Phase*. The code has just been pasted and does not yet compile.

There are five relevant technical aspects:

- 1 The original signature declared a boolean return value, whereas the new signature has none (Java void), so code does not yet compile due to the return values in lines 68 and 35.
- 2 The *Validator* code (lines 32–36) is not necessary for the new functionality.
- 3 The *SnapThemeModel* code (line 38) is necessary.
- 4 The *FeatureLayer* code (line 39) is also necessary.
- 5 The *EditGeometry* code (lines 41–45) needs to be changed: Instead of determining the edit GeometryType programmatically, it can be hard-coded to POLYGON or MULTI\_POLYGON.

### What happens?

In less than a minute, the pair understands all five technical aspects and performs the necessary changes. Code line numbers in the transcript below refer to the original state as in Figure 6.2; see Figure 6.3 for the final state of the code.

#### 1 Clean-Up of Return Values (19:12–19:20)

- (1) ★<sub>A</sub> C3: “OK, there are a lot of things we don’t need, or don’t have.” After briefly looking at the pasted code, C3 concludes that they are not done yet for their feature.  
*examine\_sth + explain\_completion*
- (2) ◀<sub>A</sub> C3: “<cursor to line 68, deletes return statement, cursor to line 19>” C3 deletes the last return and brings the cursor back up.  
*write\_sth*
- (3) ★<sub>B</sub> C4: “You can just return here <points to line 35>” C4 suggests to reduce the first return statement to a value-less return.  
*propose\_design<sub>P1</sub>*
- (4) ▲<sub>B</sub> C3: “<cursor to line 35, deletes value false>” C3 silently navigates to line 35 and puts the proposal into action.  
*write\_sth (agree\_design)*

#### 2 Validator Code (19:20–19:29)

- (5) ★<sub>C</sub> C4: “Though, if you think about it, we don’t actually need it.” C4 broadens the scope to the whole *Validator* code (2nd “it”), deeming it unnecessary.  
*amend\_design*
- (6) ▲<sub>C</sub> C3: “<cursor along line 32> Probably, this again turns out to be (. ) it depends (!...!) (#validator#)” C3 understands the reference and possibly thinks about how validation could look in their case.  
*agree\_design + examine\_sth*

## Example 6.15 (continued)

- (7) ▲<sub>C</sub> C4: “Could be we need it. Then we can get it back anyway.” *propose\_strategy*<sub>PI</sub> C4 admits that the *Validator* code might eventually turn out to be necessary, but proposes to restore the code if need be, implicitly suggesting to remove the code now.
- (8) ▲<sub>C</sub> C3: “Yes, would get rid of it. <\*deletes lines 32–37\*>” *agree\_strategy* + *amend\_design* C3 agrees with her reasoning and deletes the respective statements.

## 3 SnapThemeModel Code (19:29–19:32)

- (9) ★<sub>D</sub> C3: “OK, *snapThemeModel* <\*cursor along line 38\*> we need that.” *examine\_sth* + *propose\_design*<sub>PI</sub> C3 moves the cursor to the *SnapThemeModel* code and reaches the conclusion to keep the code in place.
- (10) ▲<sub>D</sub> C4: “We need that.” *propose\_design*<sub>PI</sub> C4 proposes the same in unison with C3.

Note that neither developer makes their reasoning explicit, but both behave as if the conclusion was obvious.

## 4 FeatureLayer Code (19:32–19:46)

The program statements in code line 39 create a *ScaleRange* object which determines the spatial resolution used in calculations, which in turn affects the precision with which user inputs are captured. The planned feature (splitting geometries) needs all participating objects (existing geometries from a *featureLayer* and new shapes to split them) to operate with the same precision, so the code in question must be kept.

- (11) ★<sub>E</sub> C3: “(#featureLayer baseScale#) <\*cursor along line 39\*>” *examine\_sth* It takes C3 slightly longer to reach a verdict this time.
- (12) ◀<sub>E</sub> C3: “Guess we need, too, ’cause what we split <\*makes a cutting gesture, looks to C4\*> we put a new line automatically into a polygon and that should have exactly its input precision.” *propose\_design*<sub>PI</sub> + *explain\_knowledge* C3 explains the *featureLayer* must be kept because it provides the precision. His *propose\_design* is not *expecting* an evaluation (type P1 “provide information”, as all *propose* so far); he only looks at C4 in the course of the following explanation.
- (13) ▲<sub>E</sub> C4: “<nods> M-hm.” *agree\_design* C4 is now involved in the ★, too.
- (14) ▲<sub>E</sub> C4: “We do need the *featureLayer* we’re editing, yeah.” *explain\_standard\_of\_knowledge* C4 rephrases C3 explanation.
- (15) ▲<sub>E</sub> C3: “Exactly.” *agree\_knowledge* C3 in turn agrees to C4’s explanation.

## 5 EditGeometryType (19:46–20:01)

As the pair reaches the *EditGeometry* code, they start completing each other’s thoughts and sentences.

- (16) ★<sub>F</sub> C3: “The (#editStrategy getGeometryType#), that’s wrong.” *explain\_finding* C3 quickly finds the existing code it is not suitable for their new feature.
- (17) ▲<sub>F</sub> C4: “That is in either case (!...!)” *propose\_design*<sub>PI</sub> } Both speak at the same time and both propose to use a fixed value instead of calling a method.
- (18) ▲<sub>F</sub> C3: “Here we always use (!...!)” *propose\_design*<sub>PI</sub> }
- (19) ▲<sub>F</sub> C4: “POLYGON, right” *amend\_design* C4 thinks this value should be POLYGON.

Example 6.15 (continued)

- (20) ▲<sub>F</sub> C3: <\*types ‘EditGeometryType’ in line 42, a list of enum values opens\*> C3 uses the auto-completion to enter the name of the enumeration type `EditGeometryType`.  
*write\_sth* `EditGeometryType`.

```

EditGeometryType.( () .getEditGeometryType (
  ◦ LINE EditGeometryType - EditGeometryType
  ◦ MULTI_LINE EditGeometryType - EditGeometryType
  ◦ MULTI_POINT EditGeometryType - EditGeometryType
  ◦ MULTI_POLYGON EditGeometryType - EditGeometryType
  ◦ POINT EditGeometryType - EditGeometryType
  ◦ POLYGON EditGeometryType - EditGeometryType
  ◦ RECTANGLE EditGeometryType - EditGeometryType

```

- (21) ▲<sub>F</sub> C3: “(##dot##), er, that’s the question” Seeing the possible values, C3 has second thoughts regarding the POLYGON proposal.  
*disagree\_design*
- (22) ▲<sub>F</sub>★<sub>G</sub> C4: “MULTI?” *challenge\_design* C4 proposes the more general MULTI\_POLYGON as an alternative.
- (23) ◀<sub>G</sub> C4: “Nope.” *disagree\_design* C4 immediately retracts her idea.
- (24) ▲<sub>G</sub> C3: “That remains to be decided whether we (!...!)” C3 remarks that the choice might not be trivial and alludes to postponing it.  
*propose\_todo*
- (25) ▲<sub>F</sub> C3: “Well, we can simply use POLYGON for now. <\*selects “POLYGON” from list of enum values\*>” *decide\_design* } C3 proposes POLYGON as a temporary solution. C4 comes to the same conclusion at the same time.
- (26) ▲<sub>F</sub> C4: “Let’s start with POLYGON.” *decide\_design* }
- (27) ▲<sub>F</sub> C3: <\*deletes lines 42–43\*> *write\_sth* C3 removes remaining `EditGeometry` code.
- (28) ▲<sub>F</sub> C4: “Right.” *agree\_design* It is not clear whether C4 acknowledges C3 reaching the same design conclusion (↑ 25), or C3 putting this decision into action (↑ 27).

Note that it appears clear to both developers that “MULTI” (↑ 22) refers to MULTI\_POLYGON although there are three MULTI\_ options available. Right after this excerpt, at 20:22, the pair explicitly refers to MULTI\_POLYGON for the first time, which makes clear what they were talking about earlier.

Leaving a TODO comment (20:01–20:11)

- (29) ★<sub>H</sub> C4: “There’s still a bracket” C3’s recent edit (↑ 27) left an unpaired parenthesis in the code.  
*explain\_finding<sub>Dc</sub>*
- (30) ◀<sub>G</sub>★<sub>J</sub> C3: “Maybe leave TODO\_NOW here <\*calls “TODO\_NOW” macro between line 41 and 42\*>” *write\_sth + propose\_design<sub>P1</sub>* C3 picks up on his *propose\_todo* (↑ 24), announces to put a TODO\_NOW comment in the code to revisit the topic later, and calls the according macro. Note that this is not a ♦ since C3 had no time yet to react to ★<sub>H</sub>.
- (31) ●<sub>H</sub> C4: “still a bracket (.) too much, at the end” C4 figured that C3 might have not got her finding and repeats it with more detail.  
*amend\_finding*
- (32) ●<sub>H</sub> C3: “<\*cursor to line 43, deletes left-over parenthesis\*> Too much, yes <\*cursor back to TODO line\*>” C3, however, had already started moving the cursor to the position of the left-over parenthesis and then deleted it. I interpret his word repetition as a ●<sub>H</sub> reparation.  
*write\_sth + agree\_finding*

## Example 6.15 (continued)

- (33) ●<sub>H</sub> C4: “M-hm” *agree\_finding*
- (34) ◀<sub>J</sub>★<sub>K</sub> C3: “Er, you’ve always added the CAD?” The TODO\_NOW macro call (↑ 30) inserted *propose\_design*<sub>OE</sub> C4’s user name (they work on her machine) and from 15 minutes earlier, C3 remembers C4’s commenting style of prefixing comments with a ticket ID (which all start with “CAD”). His proposal is in mode OE (*obtain evaluation*), so he is *expecting* feedback.
- (35) ◀<sub>K</sub> C3: <\*types ‘CAD-660’\*> *write\_sth*
- (36) ▲<sub>K</sub> C4: “M-hm.” *agree\_design*
- (37) ▲<sub>K</sub> C3: “Six-sixty was it?” *ask\_knowledge*
- (38) ▲<sub>K</sub> C4: “M-hm.” *agree\_knowledge*
- C3 apparently remembered the ticket ID but nevertheless validated both style and number with C4.

C4 does not explicitly refer to C3’s adding of the TODO comment (★<sub>J</sub>), but she looks at the screen the whole time, watching C3 type. Based on her behavior concerning ★<sub>K</sub> and beyond this excerpt, I assume that she fully understands and appreciates what C3 is doing and what the TODO comment he is about to write will be about.

Remember: *All* of the above happened in less than 60 seconds. Although their utterances and actions were presented in a linear format, C3’s and C4’s turns often overlapped, i.e., they spoke simultaneously several times, but without any real misunderstandings along the way. Additionally, C3 used the keyboard constantly, navigating and editing while he spoke and listened to his partner. Figure C.6 in Appendix C condenses the whole Focus Phase with all its concurrency into one detailed image.

```
protected void execute(final Component parentComponent) {
    final EditOptions editOptions = new EditOptions(new MapModelSelection(snapThemeModel));
    final IScaleRange scaleRange = new ScaleRange(0, getFeatureLayer().getBaseScale());
    editOptions.setScaleRange(scaleRange);
    try {
        // TODO_NOW (<*>C4’s account name*>) 09.05.2008: CAD-660
        final EditGeometryType editGeometryType = EditGeometryType.POLYGON;
        editOptions.setGeometryType(editGeometryType);
    }
}
```

Figure 6.3: Source code after the Focus Phase, less than 60 seconds after Figure 6.2

The two developers in Example 6.15 have a perfect understanding of both what the original code they copy-pasted did and what the new class ought to do, so deciding which parts they actually need to keep, delete, or amend takes very little time. Their Focus Phase illustrates the following general properties:

1. Following each ★, there are no long sequences; all ▲s are *prompt*. The *EditGeometryType* sequence is an exception (turns 16–28), but its ▲s are constructive, i.e., they add new ideas to the discourse.
2. Proposal-★s are mostly in mode PI (*provide information*), i.e., they are *non-expecting*—and are positively evaluated with *agree* or *amend* activities anyway.
3. The partner gets involved quickly in any ★, i.e., there are only few ◀s, which could even count as mid-sentence self-corrections.
4. There are only few ●s, and the only instance (turns 31–33) was not even necessary but more of a precaution, as C3 already understood his partner’s first attempt.

The six Focus Phases are the only instances I identified in my data. This is not to say that this Fluency level is unique for the particular pairs C3/C4 and A1/A2. Since understanding pair

Fluency was not at the core of my thesis, analyses of further sessions are left as further work. Judging from the instances I analyzed, I can still provide some further characterization:

- **Focus Phases** do not simply *occur*, but it appears that some conditions must be met and that the pair needs to undertake some preparatory work to enable this level of **Fluency**.
- Pairs only enter **Focus Phases** for short time spans of a few minutes maximum (see Table 6.3); this pace cannot be kept up for whole sessions. In the reflective interview C3 and C4 both said that the overall tempo of their session CA5 was high, possibly too high, and therefore exhausting.
- Nevertheless, developers seem to enjoy this high **Fluency**: At the end of the 90-second **Focus Phase** #4 (see Example C.1), when A2 removes the FIXME comment, A1 utters a pleased “*Not bad*” as if he is actually surprised how far they got in little time. At the end of **Focus Phase** #3 (not discussed here), C4 celebrates “*There, now you can kick the IOException!*” and mimics a dance move; C3 is happy, too: “*Poof, it’s gone. Nice how this shrivels.*”

### 6.3.4 Breakdowns in Pair Programming

In **normal** PP (Section 6.3.2), developers verbally refer to each other’s proposals and make evaluations. During a **Focus Phase** (Section 6.3.3), developers seem to understand each other despite much decreased verbosity. In some situation, however, developers do not evaluate each other’s initiatives and there are long periods of silence. The **Fluency** is low and the *pair* programming process is **broken down**.

#### 6.3.4 a) No Progress as a Pair

Both **Breakdown** specimen discussed here come from the pair O3/O4 but originate from two different sessions. The **Breakdown** in Example 6.16 is from session OA8 and happened just a few minutes before to the **normal** PP episode already discussed in Example 6.14. Here, both pair members produce hypotheses for why an existing test case started failing after recent changes in the production code. However, they do not manage to mutually understand and discuss their ideas. Even though there is little to no disagreement on which tactical steps to take next, the conclusions drawn from their observations differ and are not reconciled. Ironically, O4 has the correct idea within seconds, but since O3 inadvertently keeps distracting him with non-relevant proposals, it takes the pair 28 minutes to add two lines of mocking code in the tests. In addition, O4 appears to understand that some of O3’s proposals and interpretations are wrong but does not explain what he thinks of them.

#### Example 6.16: Breakdown with Seemingly Unhelpful Partner (OA8, 13:42–41:42)

##### Session Background

Prior to the session, the developers O3 and O4 changed the implementation of a method to work with different primitive operations but did not adapt the mock object used in the tests. At first, the pair is accompanied by developer O1, before he leaves the group for a scheduled meeting. The developers speak English in this session, which is neither developer’s native language but their strongest common language. The transcript below is verbatim. This excerpt ends eight minutes before Example 6.14 where O3 and O4’s pair programming was **normal**.

##### Technical Background

The old logic relied on the `duration()` method, whereas the current logic subtracts `getStart()` from `getEnd()` instead (see Figure 6.4). In the test logic (see Figure 6.5), `duration()` was mocked,

## Example 6.16 (continued)

but not `getStart()` and `getEnd()`, so the calling logic receives no valid value to calculate the output and the assertion in the test case fails.

```
# Old Logic
offsetDays = Math.floor dataModel.duration().asDays() * offsetFraction

# Current Logic
durationInDays = dataModel.getEnd()?.diff dataModel.getStart(), 'days'
offsetDays = Math.floor durationInDays * offsetFraction
```

**Figure 6.4:** CoffeeScript code for the old and the current production logic, both return integer values for `offsetDays`. The current logic uses an existence operator (“?”) to check whether `getEnd()` is callable before it calls the `diff` method. (In the actual source code, `durationInDays` is calculated in a separate method, which I inlined here for readability.)

```
94 dataModel.duration.returns
95   asDays: sandbox.stub().returns 3
96 rowView.emit 'dragstart', offsetFraction: 0.5
97
98 expectedDragProperties = sandbox.match
99   offsetDays: 1
100 expect(timeline.setDragProperties).to.have.been.calledWith expectedDragProperties
```

**Figure 6.5:** Relevant excerpts of the CoffeeScript test code before the **Breakdown**. Lines 94–95 are the test setup (where the `duration` method is mocked), line 96 is the execution (which no longer uses the `duration` method), and 98–100 is the assertion (which now fails). See Figure 6.6 for the state of the code 28 minutes later.

For *problem analysis*, the developers need to understand the following four facts:

- 1 The assertion (in line 100) does not directly compare two integers, but uses a *matcher* mechanism to test whether an object’s field `offsetDays` has an expected integer value (lines 98–99). Understanding this is necessary for comprehending the failing test’s error message which includes a dump of the whole object.
- 2 On the current production code, the assertion fails because the actual value of `offsetDays` is NaN (Not a Number) instead of the expected 1.
- 3 The test fails because `getEnd()` is not defined in the mock object (lines 94–95): The defect is in the test code, not the production code.
- 4 CoffeeScript’s existence operator (“?”) returns `undefined` for uncallable functions (here: the undefined `getEnd()`). Performing calculations on such a value in JavaScript (here: multiplication) yields the value NaN, not-a-number.

The three facts 1 2 3 are system-specific knowledge (S knowledge), 4 is system-independent general software development knowledge (G knowledge).

Once these four facts are understood, the *problem solving* part is easy: Provide mock implementations for `getStart()` and `getEnd()` such that their return values are three days apart.

### What happens

The excerpt below has multiple parts. At first, all three developers (O1, O3, O4) develop individual hypotheses for the test failure. O1 and O3 both suspect a defect in the production code while O4 thinks the problem is in the test code. O4 is overruled, and the group looks into the production code. Although there is quite some confusion and little technical progress, the process can still be characterized as *normal*, since there are ●s, i.e., attempts to clear up misconceptions.

The **Breakdown** happens in the second part after O1 left the group for a scheduled meeting. The **Breakdown** needs to be seen from the two remaining developers’ perspectives: On the one hand, O4 now gets to pursue his hypothesis and search for the problem in the test code, but because

## Example 6.16 (continued)

he is effectively unable to explain it to O3, she keeps distracting him from checking the hypothesis and implementing the fix. On the other hand, O3's hypothesis is neither invalidated by the checks O4 performs, nor appreciated by O4, as he mostly ignores her. Eventually, after 28 minutes that lie between the states shown in Figures 6.5 and 6.6, O4's hypothesis turns out to be technically correct (and, from what I can reconstruct, O3's ideas are mostly wrong)—but neither developer could have known this from the onset.

**Part 1: Deciding on the First Expedition (13:42–17:53)**

The excerpt starts with the three developers O1, O3, and O4 reading the error message of the failed assertion.

- (1) ★<sub>A</sub> O4: “(#setDragProperties#) (, ,) the offsetDays has <\*> selects part of the console output\*> (, ) not-a-number. (..) field offsetDays has the special value Ok?” *explain\_finding* O4 reads the error message of the failed assertion and understands [2], i.e., that the field offsetDays has the special value NaN instead of 1.
- (2) ▲★<sub>B</sub> O3: “Because, because it doesn't have any arguments anymore. We changed the function. So, it says that it should be, it was expecting some arguments there and didn't receive any arguments. I would, can you go to the code, to what we changed?” *amend\_finding + propose\_step* O3 provides a hypothesis for how the failure comes to be and suspects the defect to be in the production code. She possibly thinks that NaN is some sort of default value for a function's parameters if it gets called without arguments.
- (3) ▲<sub>B</sub> O4: “<puzzled> We just changed the calculation.” *challenge\_finding* O4 does not think they removed the arguments of any function call.
- (4) ▲<sub>B</sub> O3: <\*> *agree\_finding*
- (5) ▲★<sub>C</sub> O4: “Maybe it's not defined, the result is not defined, maybe. (, , , , ,) <\*> moves cursor to test code\*>” *propose\_hypothesis* O4 starts to formulate his hypothesis suspecting [3]. Unlike O3, he appears to be suspicious about the test code.
- (6) ●<sub>C</sub>★<sub>D</sub> O1: “Not-a-number is the result here of offsetDays, so it's not-a-number, is the result (!!...!!)” *challenge\_hypothesis* O1 has been reading the test output until now. He now corrects O4's imprecise formulation that “the result is not defined”.
- (7) ◀<sub>C</sub>♦<sub>D</sub> O4: “<\*> hovers definition of duration mock\*> Maybe, maybe, we have to give the start and end point for the data Model.” *amend\_hypothesis* O4 does not react to O1, but instead hypothesizes [3], i.e., that the mocked data model needs more logic.
- (8) ♦<sub>C</sub>●<sub>D</sub> O1: “<annoyed> Can you look here? <\*> points to screen\*> offsetDay is NaN, not-a-number <\*> looks at O4\*> (!!Yeah!!) A number is expected and we don't return a number.” *explain\_finding* O1 wants to make sure that O4 understands [2], the test failure, correctly—not knowing that he already does—and repeats his correction.
- (9) ★<sub>E</sub> O1: “<\*> looks at O4\*> It might just be like we missed the brackets and pass a function or something. That might already be (!!...!!)” *propose\_hypothesis* O1 goes on to formulate another hypothesis, which is similar to what O3 appears to suspect. Both expect the production code to contain a defect.
- (10) ◀<sub>C</sub> O4: “My idea is that (!!...!!)” O4 attempts to reiterate his hypothesis, but gets cut off.
- (11) ♦<sub>C</sub>▲<sub>E</sub> O3: “Yeah, I don't know which arguments we are passing right now.” *agree\_hypothesis* O3 sees a similarity between her hypothesis and O1's.

## Example 6.16 (continued)

- (12) ▲<sub>E</sub> O1: “So let’s have a look at the function `setDragProperties`.” *propose\_step* O1 proposes to take a look at the production code.

They follow O1’s proposal, and end up searching for and then looking at the code under test for the next three minutes. Then O1 leaves the group for a scheduled meeting.

## Part 2: First Experiment (17:53–25:53)

- (13) ◀<sub>C</sub>★<sub>F</sub> O4: “<\*>hovers changed production code\*> O4 sees his hypothesis supported that the `dataModel.getEnd()` is not defined in the test, I guess.” *propose\_hypothesis* test execution triggers a method call that is not mocked (3).
- (14) ▲<sub>F</sub> O3: “But if it calls this function (!..!)” *disagree\_hypothesis* O3 possibly thinks that calling an undefined function would produce a different error message (and is unaware of the existence operator “?”).
- (15) ◆◀<sub>F</sub> O4: “<\*(.....) switches to test code, hovers mock definition (,,,,,,),>” *examine\_sth* O4 does not react to O3’s disagreement, but instead silently switches to the test code, reads it for 40 seconds.
- (16) ◀<sub>F</sub>★<sub>G</sub> O4: “(I have an idea.) <\*>switches to production code, inserts `console.log` statement (,,,,,,),>” *explain\_standard\_of\_knowledge* O4 introduces a debugging statement in the production code. Right before the calculation is performed, O4 prints out the `dataModel` object.
- (17) ▲<sub>G</sub> O3: “You can also use, like, the debug (~) for the test.” *propose\_step + amend\_activity* Upon seeing the logging statement, O3 convinces O4 to use a browser-based debugger.

O4 has some trouble getting this setup running, and it took the pair nearly six minutes to see the output of the `console.log` statement (19:40 to 25:26). Neither O4 nor O3 appear to find valuable information in there, so while deciding where to put a breakpoint they look at the test code and the assertion again. The transcript picks up at this point:

- (18) ★<sub>H</sub> O3: “So the way I see it now, this is getting an object, but it was expecting a number.” *explain\_finding* O3 appears certain (not an *hypothesis*). Technically, she is wrong; she does not understand the matcher assert logic (1).
- (19) ▲<sub>H</sub> O4: “<\*>hovers assertion\*> The expectation is that `offsetDays` is 1. And `offsetDays` is not-a-number. This is the value.” *challenge\_finding* O4 reiterates the failure (2), but does not address O3’s misunderstanding of the matcher logic (1). To O4, (1) is transparent, so “*expectation is that offsetDays is 1*” is a shorthand for ‘*the matcher will look for 1 in the attribute offsetDays of an object*’.

## Part 3: Confusion (25:53–28:06)

- (20) ★<sub>J</sub> O4: “<\*>hovers test code (,,,)\*> And (,,, (#emit#) (,,,,) (#to.have.been.calledWith expectedDragProperties#) (,,,))” *examine\_sth* O4 does not look at O3 to see whether she understands him, but instead reads in the test code without making explicit what he is after.
- (21) ◀<sub>H</sub> O3: “<\*>points at screen\*> But it’s out-putting an object. So, an object is not a number.” *challenge\_finding* O3 does not understand O4’s explanation and insists on her observation.
- (22) ◆<sub>H</sub> O4: “(~) <\*>looks puzzled at O3\*>” *mumble\_sth* O4 is confused: O3 talks about “*not a number*”, he talks about “*not-a-number*”.

Example 6.16 (continued)

- (23) ◀<sub>H</sub> O3: “That’s what I’m understanding of this. (...) Because it was expecting this guy (#to.have.been.calledWith#) with a number. But expectedDragProperties is not a number, it’s an object! It’s a key-value pair. <\*looks at O4\*>”  
*challenge\_finding* O3 repeats her understanding, oblivious to the fact that there is no assert statement expecting a number. Both the *expected* and *actual* part of the assertion refer to objects (see line 100 in Figure 6.5). She does not understand [1] and appears to see the assert statement as the faulty one.
- (24) ◆<sub>H</sub> ◀<sub>J</sub> O4: “Erm (...) <\*hovers call of setDrag Properties in test code\*> this is the function which (!...!) <\*reads in test code\*> (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)”  
*mumble\_sth + examine\_sth* O4 reads in the test code silently for more than 30 seconds without making clear what he is looking for (★<sub>J</sub>). It is not clear whether he tries to understand what O3 meant to say.
- (25) ◀<sub>H</sub> O3: “<\*points to screen\*> <slightly help-less> The expectedDragProperties, the value, it’s an object, so (!...!)”  
*challenge\_finding* O3 has been looking at the screen the whole time until she repeats her understanding once again.
- (26) ◆<sub>H</sub> ◀<sub>J</sub> O4: <\*moves cursor around the screen (,,, ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)\*>  
*do\_sth* O4 continues reading and ignoring O3. Both keep looking at the test code for about half a minute.

Part 4: A Silver Lining (28:06–31:12)

- (27) ●★<sub>K</sub> O3: “What are you thinking?”  
*ask\_standard\_of\_knowledge* O3 breaks the silence and asks for an explanation of O4’s doing.
- (28) ▲<sub>K</sub> O4: “What I first wanted to do was, how I started was this console output <\*hovers console output\*> (#dataModel#), which is like that <\*points at the output\*>. And I made this code at the <”\*production class”\*> in this new function. (#console.log#). <\*points at screen\*>”  
*explain\_knowledge* O4 starts tracing his steps so far, careful to not leave anything out he did so far. He seems to be aware of the confusion between him and O3.
- (29) ▲<sub>K</sub> O3: “The thing is, in the test we don’t use the real data model, we stub, we fake one.”  
*explain\_knowledge* O3 appears to retrospectively disagree with the idea to insert a console.log statement in the production code († 16).
- (30) ▲<sub>K</sub> O4: “Yes, and the problem is, maybe in the fake one, this end and start is not defined. <\*looks at O3\*> So we maybe have to define them, so this function can be used.”  
*agree\_knowledge + propose\_hypothesis* O4 skips ahead in his explanation and jumps right to the (technically correct) conclusion of [3], but addressing neither of O3’s hypotheses nor any of the misunderstandings he did notice before.
- (31) ▲<sub>K</sub> O3: “Yeah, maybe we have to add it to the stub, yeah.”  
*agree\_hypothesis* O3 appears to understand O4’s hypothesis of [3].
- (32) ●▲<sub>K</sub> O4: “This was my idea.”  
*agree\_hypothesis* O4 sees his hypothesis understood.
- (33) ▲<sub>K</sub> O3: “M-hm.”  
*agree\_hypothesis* O3 seems content.

While ★<sub>K</sub> appears to end the Breakdown, the underlying problems were not addressed. O3 still does not get the full picture: She is missing [1] and consequently [2]. During the next two minutes (29:18–31:11), O3 watches O4 add a debugging statement to see the value of `durationInDays` (see Figure 6.4). After running the test case once again, the pair has the next Breakdown.

## Example 6.16 (continued)

**Part 5: Second Experiment (31:12–32:09)**

The output from the next (still failing) test run is displayed on screen along with the debugging output and the pair reads it:

- (34) ★<sub>L</sub> O4: “<\*reads the output\*> (#undefined#) ok, but not (...) not not-a-number.” *explain\_finding* The value of `durationInDays` does not match O4’s expectations: Instead of `NaN`, the logic he suspected to contain the defect produced `undefined`.
- (35) ▲<sub>L</sub>●<sub>H</sub> O3: “(...) Yeah, because what is not a number is the argument, not the function call.” *amend\_finding* Together with O3’s next utterance (see below), this can be interpreted as her picking up her previous initiative ★<sub>H</sub> (and as a symptom of her not understanding [1]).
- (36) ◆<sub>H</sub> O4: “(...) Erm.” *mumble\_sth* It is unclear whether O4 thinks about his own finding or is too baffled by O3.
- (37) ◀<sub>H</sub> O3: “Because the test is, it should have been called with these arguments, and it’s the arguments that is not a number.” *amend\_finding* The recent debugging output appears to not have altered O3’s understanding of the problem as she basically repeats her previous hypothesis.
- (38) ◆<sub>H</sub>◀<sub>L</sub> O4: “<\*reads in production code\*> Here we make the output (...) (#durationInDays#) (...) ok, it makes this calculation afterwards. <\*hovers line containing call of `Math.floor`\*>” *examine\_sth* In contrast, O4 probably wants to find out why the output read `undefined` and not the expected `NaN`. He identifies another step in the calculation.
- (39) ◀<sub>L</sub> O4: “(...) So, I would like to know what (!...!) let me try please” *propose\_step* O4 is not explicit about what he wants to try out.
- (40) ▲<sub>L</sub> O3: “Sure!” *agree\_step* O3 instantly agrees anyway.

It is possible that O3 sees her own hypothesis supported by the outputs so far, so she does not feel an urgent need to learn more about the system right now and therefore lets O4 work on his mental model.

**Part 6: Third Experiment (32:09–33:49)**

O4 introduces a debugging statement to see whether the newly found calculation transforms the `undefined` into a `NaN`. He then runs the test case again and the output gets displayed.

- (41) ★<sub>M</sub> O3: “(#Not-a-number#)” *read\_sth* O3 is the first to read the output.
- (42) ▲<sub>M</sub> O4: “OK, this is what I thought.” *explain\_finding* O4 sees his hypothesis confirmed. He now understands the whole problem including [1] [2] [3] and [4].
- (43) ★<sub>N</sub> O3: “Which is weird, because the `Math` function should be (!...!) <\*turns to her own machine\*>” *explain\_finding* For the first time, O3 now has evidence not in line with her understanding *and* is aware of it. O3 does not understand how the outcome comes to be (lack of [4]) and wants to reproduce it on her machine.
- (44) ◆<sub>N</sub>◀<sub>M</sub> O4: “In the test, maybe we can (!...!) <\*opens test code\*>” *propose\_step* O4 now switches to solving the problem, ignoring O3’s actions.



## Example 6.16 (continued)

- (57) ★<sub>R</sub> O3: “Yeah, but *<keeps looking at the source code\*>* I’m pretty sure the problem is here. Can we debug this?”  
*propose\_hypothesis + propose\_step* Consequentially, O3 keeps looking for ways to understand the failure.
- (58) ▲<sub>R</sub>◆ O4: “<puzzled> We already did.”  
*disagree\_step* O4 is again puzzled that O3 does not recognize the console.log statement from their recent debugging efforts. Yet, he still only reacts to the immediate proposal.
- (59) ▲<sub>R</sub> O3: “But (.) check *durationInDays*, what is the value of *durationInDays*?”  
*ask\_knowledge* Luckily, O3’s next question allows O4 to answer with a summary of his analysis result.
- (60) ▲<sub>R</sub> O4: “The value is the result of this function. And the result is, erm, *<reads logging output\*>* undefined, yes. And the result of that calculation is not-a-number.”  
*explain\_knowledge* O4 explains one part of how the test failure comes to be: An intermediate result of a calculation (*durationInDays*, see Figure 6.4) is undefined and that results in an NaN (the concrete form of [4]).
- (61) ●<sub>O</sub> O4: “Have a look how it was before *<opens Git client and reads diff\*>* It was (*#Math.floor dataModel.duration.asDays#*) and this *duration* was set in the test. And I think, now, this *duration* is calculated differently, so we have to set other variables, so that the calculation is possible, the start and the end.”  
*explain\_knowledge + propose\_hypothesis* O4 then picks up his earlier explanation based on the production code diff. He explains that he thinks the undefined value occurs due to not having mocked the necessary methods ([3]), and proposes to amend the mock object accordingly. Note that his explanations do *not* touch the topics of what the test failure actually was ([2]) and how the assertion works ([1]).
- (62) ▲<sub>O</sub> O3: “Ok, ok, I see.”  
*explain\_standard\_of\_knowledge* Nevertheless, O3 seems content with O4 understanding the situation, although she arguably could not have learned [1], [2], and [4] (all of which puzzled her in the last 25 minutes) from O4’s explanations.

The above exchange ends at 37:07. It takes another four and a half minutes to implement the solution (the remainder of the transcript is on page 422 in the Appendix), the last two of which O4 works alone. At 41:42, the implementation of the test case is complete as shown in Figure 6.6 and the test execution is successful.

```

94 #dataModel.duration.returns
95 #   asDays: sandbox.stub().returns 3
96 dataModel.getStart.returns moment '2000-03-04'
97 dataModel.getEnd.returns moment '2000-03-07'
98 rowView.emit 'dragstart', offsetFraction: 0.5
99
100 expectedDragProperties = sandbox.match
101   offsetDays: 1
102 expect(timeline.setDragProperties).to.have.been.calledWith expectedDragProperties

```

**Figure 6.6:** Relevant excerpts of the CoffeeScript code after the Breakdown, 28 minutes after the development episode began. Compared with the original state (see Figure 6.5), the pair commented out the original mocking code in lines 94–95 and added new mocking code in lines 96–97.

The above excerpt illustrates a **broken** pair process: Although both developers tactically agree on which ‘experiments’ to conduct, they do not reconcile their different interpretations of the outcomes. This happens four times:

- O4’s hypothesis is rejected by O3, but no clarification follows (*propose\_hypothesis* and *disagree\_hypothesis* in turns 13 and 14).
- O3 agrees to the first experiment and shares her confident interpretation which O4 rejects, but no clarification follows (*amend\_activity*, then *explain\_finding* and *challenge\_finding* in turns 17–19).
- O3 tacitly agrees to the second experiment and shares her confident interpretation which O4 is puzzled by, but, again, no clarification follows (implicit agreement between turns 33 and 34, then *explain\_finding*, *amend\_finding*, and *mumble\_sth* in turns 34–36).
- O3 explicitly agrees to the third experiment, but now O4 is confident in his interpretation and O3 is confused, and yet again, no clarification follows (*agree\_step*, 2× *explain\_finding* in turns 40–42).

The length of the excerpt illustrates the **cost** of a **broken** pair process: O4 comes up with a hypothesis and a way to test it in *under a minute* (turns 5 & 7) and tries to put it into action. O3 seems to not understand O4’s plan (neither through his explanations nor his actions) and (unwillingly) keeps distracting him from completing it. Instead of just a few minutes, it takes *over 20 minutes* until O4 could start with the problem solving part (which followed after the excerpt above). Arguably, O3 was not only of little help during these 20 minutes. She also slowed O4 down to a critical extent, diminishing any benefit pair programming could have in this situation. The following points are important to note, though.

1. Even though it took the pair a long time to accomplish seemingly little, the time expense is not the defining criterion of a **Breakdown**. Rather it is the lack of evaluation of the partner’s ideas and not clearing up misunderstandings along the way (see the many ◀s and ♦s, and few ●s).
2. That O4’s hypothesis turned out to be correct and that O3 kept distracting him is ironic, but not the point of this example: O4 apparently did not understand what O3 was talking about, so he could not make a judgment about which idea was best. If anything, his idea being correct made the whole episode easier to reconstruct for me as a researcher.
3. It is worth taking the individual perspectives of the pair members. Both developers attempt to start a new communication strand aimed at clearing things up, see O3’s ●★<sub>K</sub> and O4’s ●★<sub>O</sub> (starting in turns 27 and 50). However, both developers did so from a state of relatively stable understanding: O3 does it *before* she is confused by an output for the first time in turn (43), and O4 does it *after* his problem analysis is complete in turn (42). In *normal* PP, clearing up misunderstandings works even in the midst of an understanding process as can be seen in Example 6.14 discussed earlier, which is from the same session a couple of minutes later.
4. The pair probably could have accomplished the same outcome in less time had they agreed on *one* idea to pursue, e.g., either O4’s strand (i.e., his ★s: A, C, F, G, J, L, and M) or O3’s strand (her ★s: B, H, N, P, and R). Even if they had followed O3’s objectively wrong idea first, this would have resulted in fewer context switches (e.g., visible in O4 being puzzled in turns 22, 24, 36, and 48) and reparation attempts (see ●★<sub>K</sub> and ●★<sub>O</sub>, taking more than one and three minutes, respectively).

#### 6.3.4 b) No Progress at All

While in the previous example from session OA8, the pair members at least had an idea on how to proceed, the same two developers had a hard time in session OA1. They need to write a

test case for some production code they do not know which is implemented with a technology they are not familiar with. The whole session is characterized by a glacially slow progress and the two developers having difficulties in both understanding the class they are supposed to test and with expressing their difficulties in a way that they could work together on a shared mental model. Unlike most of their session, the excerpt transcribed below can be understood with reasonable effort (see the transcript on page 418 in the Appendix for an impression of just how incoherent large parts of the rest of the session OA1 are).

#### Example 6.17: Breakdown Due to Huge Knowledge Gaps (OA1, 59:36–1:08:53)

##### Session Background

Developers O3 and O4 should write a test case to check a newly implemented default selection in a form. The form is complicated (interaction with one of its parts may reveal new form parts or alter existing ones) and was implemented by a colleague who is currently not available. Not only does the pair not know the code they are supposed to test, they are also not familiar with the underlying technology.

The pair tried to understand an existing test case for the form during the first hour. They inserted `console.log` statements in the test code that prints runtime objects as JSON strings. In their setup, every run of the test case takes about between 2.5 and 4 minutes to complete, so they only managed to run seven iterations of *changing log statements—starting test run—reading output*, and basically only learned that the objects they were looking at did *not* contain any information on the initial values—which was the part they were supposed to test.

During the session, O4 started looking up things on his own laptop. His screen and webcam were not recorded, so it is not clear at times what he is looking at.

##### Technical Background

The form itself is built in layers which are completely contained in a single source code file:

- 1 TeamSelect (**exported**): Most of the form dynamics is encapsulated in this layer. It is *exported*, i.e., accessible by other modules.
- 2 TeamSelectMapped: Layer 1 enhanced by a virtual, read-only field by means of a function called `filterTeamSelected` (which itself is also **exported**).
- 3 TeamSelectForm: Layer 2 connected to a data store and also with the new initial value definition—the functionality to be tested.
- 4 TranslatedTeamSelect: Layer 3 enhanced by internationalization information.
- 5 TeamSelectContainer (**exported**): Layer 4 attached to a so-called *state container* (here: Redux) by means of a function called `mapsToProps`.

The layerings are done in a functional way (using higher-order functions), making it difficult to relate the form's runtime structure to the compile-time structure of its parts. There is also an existing test class in which the state of TeamSelect (layer 1) is set directly, so that test case is not useful for testing the default-setting logic setting happening on layer 3. (Note that there is much more to know about their system and the involved technology which the pair neither figured out nor came across in their session, so reconstructing the full extent of their knowledge gap is not feasible.)

##### What happens?

First, the pair reiterates some of the basic facts they learned during the first hour (e.g., *TeamSelect comes from the file named TeamSelect.coffee*, or *TeamSelect does not contain information about initial values*), without producing any new ideas. Due to many pauses and interrupting one another this takes over three minutes (59:36–1:02:51, see page 418 for the transcript).

In the following minutes, O4 makes a test design proposal (basically, to test 5 instead of 1), to which both developers eventually agree by and large. But along the way, each of them has some reservations, follows an idea of their own in which the respective partner is not involved, and fail to evaluate each other's arguments and hypotheses that the design proposal relies on.

Example 6.17 (continued)

- (1) ★<sub>A</sub> O3: “If we test against the real React component, I think, we can (!...!) so we just have to render the React component, the real one instead of the mocked one and check if the prop is there or not.” *propose\_design* O3 proposes to not use the test fixture which sets the state on [1] directly (“the mocked one”), but “the real React component” without any externally forced state. It is not clear whether O3 understands any of the five layers yet.
  - (2) ♦★<sub>B</sub> O4: “(.....) This initial value, it is here. TeamSelectForm is used Translated TeamSelect. TranslatedTeamSelect is in the TeamSelectContainer, which is also exported, but which is not a part of TeamSelect.” *explain\_finding* O4 does not react to O3’s proposal. Instead, he reads in the production code and understands that the initial values are defined in [3], that [3] is wrapped in [4] and then in [5], and that [1] (which is the subject of the existing test cases) does not contain any of them.
  - (3) ▲<sub>B</sub> O3: “(....) Hm. <\*selects definition of TeamSelectContainer\*> Yeah? (.....) Yeah, but this is the (!!...!!)” *disagree\_finding* O3 takes some time to think about TeamSelectContainer and appears to disagree with O4, but is cut off by O4 before she could formulate her thoughts.
  - (4) ♦◀<sub>B</sub> O4: “So maybe we have to test TeamSelectContainer.” *propose\_design* O4 proposes to test [5] instead of [1]. (This makes sense: [5] is the only exported component that includes the logic to-be-tested.)
  - (5) ▲<sub>B</sub> O3: “(.....) I don’t know.” *explain\_standard\_of\_knowledge* O3 does not evaluate the proposal and says she cannot evaluate it.
  - (6) ♦★<sub>C</sub> O4: “(....) Or is TeamSelectContainer used somewhere else? <\*turns to his machine\*>” *propose\_step<sub>PI</sub> + search\_sth* O4 does not react to O3. Instead, O4 wonders whether there are already existing test cases for [5] and starts a search on his own laptop.
- From this point, the pair no longer maintains a pair process: O4 is engaged in ★<sub>C</sub> and O3 in ★<sub>B</sub> without their respective partner following along.
- (7) ◀<sub>B</sub> O3: “So, TeamSelectContainer is the wrapper to make it a stateful component, so it connects to Redux. But I don’t know if we need to call the (.....) if we need Redux here anyway.” *explain\_knowledge + disagree\_design* O3 reiterates what they know about [5], and appears to raise some concerns about O4’s proposal to test [5]—possibly because she wants to keep the test subject small and the test cases simple—though she does not make her argument and her intention explicit.
  - (8) ♦◀<sub>C</sub> O4: <\*continues searching (,,,,,,,) \*> *search\_sth*
  - (9) ♦◀<sub>B</sub> O3: <\*looks at source code (,,,,,,,) \*> *examine\_sth*
- } O4 does not react to O3’s objection, but instead silently continues his search. Neither developer says a word for 33 seconds.
- (10) ★<sub>D</sub> O3: “Ah, what about this filter SelectedTeam? <\*selects definition of said method\*>” *explain\_finding* O3 develops a new idea for a test subject and notices the function filterSelectedTeam (from layer [2]).
  - (11) ● O4: “<\*looks to O3\*> Sorry?” O3 gets O4’s attention.
  - (12) ▲<sub>D</sub> O3: “What about this filterSelected Team? So it gets the formValue <\*hovers code\*>, it’s exactly what we want. (#props. fields.teamSelect.value#) <\*open debugger in web browser (,,,,,) \*>” *propose\_design + examine\_sth* O3 proposes filterSelectedTeam as the test subject, and finds clues that it is involved in the fields selection. Technically speaking however, the function is used in [2] and not involved in setting the initial values, so O3’s finding is wrong.



## Example 6.17 (continued)

- (27) ▲<sub>B</sub> O4: “It’s defined in a component which is covering, which wraps this TeamSelect component. (. . . . .) This is my impression, I’m not sure.” O4 goes on to reiterate his understanding of the layers, which he explained more than five minutes ago († 2).  
*propose\_hypothesis*
- (28) ▲<sub>B</sub> O3: “OK, so let’s try to render instead of the TeamSelect, to render the TeamSelect Container.” O3 agrees to the O4’s proposal again—three more minutes after she first agreed to it († 15).  
*decide\_design*

After this excerpt, the pair started implementing the proposal.

In the above excerpt, the pair process *broke down* at the point where the developers each started following their own ★s. On the one hand, O4 went searching for other test cases (★<sub>C</sub>) and it is not clear whether O3 knew and understood what he was doing (she *did* notice that O4 did *something* as she apologized for interrupting him). O4 reported that his search yielded no results, but the fact that he searches for almost three and half minutes (and continues for another minute and a half after the above excerpt) indicates that the search process is not trivial and could benefit from a partner’s input. On the other hand, O3 raised concerns about O4’s initial proposal (turns 3, 5, & 7) which were not addressed, and O3 identified two other test subject candidates (results of ★<sub>D</sub> and ★<sub>E</sub>), which O4 did not discuss.

One could argue that the above excerpt was not a *broken* pair process but two developers each working solo in parallel. The situation, however, would still be problematic: O4 gets interrupted in his search by O3 twice, and O3 does not learn much in the time she works alone: Her decision to agree to O4’s proposal has not changed, although her concerns about it were not alleviated, and she still does not know more about the code than that “*at some point, we have the initial value that is coming from somewhere*”.

O3 later described the frustration of this *broken* pair process with the difficulty of producing clear ideas of her own *and* following her partner:

“ I was lost [. . .] I was finding it difficult to follow my thinking and [. . .] to understand also for myself what was the problem and also follow O4’s way of thinking. [. . .] I was in the middle of a thought and then O4 had another idea, and also the other way around. O4 was having an idea, and I would interrupt him.

O3 in reflection interview after sessions OA1/OA2 (see Section 4.3.2e)

However, understanding the problem as well as dealing with one’s own and the partner’s thoughts sounds like what happens in *all* pair programming. Indeed, the pair O3/O4 is perfectly capable of *normal* PP (see Example 6.14). So what was different here? What are the reasons for *breaking down*?

## 6.4 Togetherness

In the previous section, I characterized three level of pair process *Fluency*: *normal* PP, *Focus Phases*, and *Breakdowns*. A pair’s *Fluency* is exhibited behavior. I propose *Togetherness* as a not directly observable property of the pair to explain the differences in *Fluency*: It is the degree to which the pair members are able to *fully* understand each other’s activities, including all intentions and meanings associated by the respective speaker or actor.

*Togetherness* is a potential: A pair with high *Togetherness* can work *normal* or reach *Focus Phases*; a pair with low *Togetherness* is susceptible to a *Breakdown*. High *Togetherness* is not the same as being ‘*mental clones*’. It is more about being compatible in a certain sense.

**Togetherness** allows correctly understanding an utterance as a question, proposal, or evaluation, what it entails on a technical level (as meant by the speaker), why the speaker made it, etc.—but it is *not* about knowing that the partner would ask that question in advance, knowing the answer to it, or being able to come up with the same proposal, being able to evaluate it, or agreeing to it, etc.

### 6.4.1 Degrees of **Togetherness**: Understanding Intentions

**Togetherness** as a potential is not directly observable. After all, ‘*thinking with one mind*’ is only a figure of speech and any common understanding is just an illusion: There is only individual knowledge (see Section 3.2.1a). In a sense, a pair’s **Togetherness** as the ability of the pair members to understand each other is put to test with every base activity. As in any form of communication, developers during a PP session need to interpret their partner’s utterances and behavior in order to make sense of them. How well this works characterizes a pair’s momentary **Togetherness** (see also Table 6.4): Pair members with medium **Togetherness** are able to correctly understand their partner’s *primary intentions*. In other words: They could assign base concepts to utterances and actions. Pairs with high **Togetherness** also understand partial utterances and implicit meanings, they ‘sense’ what their partner is about to say or do (as in Example 6.3: C5: “*Do you know how I access the feature to change a method?*”—C2: “*That doesn’t get you anywhere.*”). Pairs with low **Togetherness** at times have more difficulties understanding even the primary intentions and/or the propositional content of their partner’s utterance.

Concept	Description
<b>Togetherness</b>	The degree to which the pair members are able to <i>fully</i> understand each other’s activities, including all intentions and meanings associated by the respective speaker or actor. This is what allows two developers to work <i>as a pair</i> . A pair’s <b>Togetherness</b> is not directly observable and it may change over the course of a PP session.
– medium	Pair members can correctly understand their partner’s <i>primary intentions</i> (what the base concepts capture), e.g., understand an <i>ask_knowledge</i> as a question and a <i>propose_design</i> as a design proposal. Such a pair works with <b>normal Fluency</b> , both <b>Focus Phases</b> and <b>Breakdowns</b> are unlikely.
– high	Pair members can also correctly understand their partner’s <i>underlying intentions</i> , e.g., a developer might ask a technical question (primary intention) because she has a certain implementation procedure in mind (underlying intention), which itself remains completely implicit. Such pairs work with <b>normal Fluency</b> , possibly reach a <b>Focus Phase</b> where they correctly understand (or guess) partial utterances and complete each other’s thoughts and sentences.
– low	Pair members misinterpret their partner’s intentions more easily (i.e., they only <i>think</i> they understand their partner) or are not able to come up with an interpretation at all. <b>Normal PP</b> is possible but the pair process is in danger of <b>breaking down</b> .

**Table 6.4:** **Togetherness** and its three degrees

Usually, the pair member’s intentions and subjective meanings of their actions remain implicit and need to be reconstructed by the researcher. Sometimes, however, intentions becomes an explicit topic of the programmers’ dialog, as the next two examples show.

**Example 6.18: Asking for Intention** (DA2, 1:14:25–1:14:44)

In the middle of a manual refactoring D4 wonders what to do with a method parameter. Without commenting it, D4 temporarily renames the parameter in the method signature to trigger the IDE to show compiler errors everywhere the parameter is used. His partner D3 follows these actions but does not understand the maneuver and asks for D4’s intention.

D4: “<renames method parameter “list” to “lista”, IDE shows compiler errors where “list” is referenced\*> OK, so here, it gets the (!...!) objects then, probably <undoes renaming\*>”

D3: “Why did you just rename it to list?”

D4: “No, only to see whether (!...!) what it’s doing with it.”

D3: “I see.”

**Example 6.19: Clarifying Intentions** (CA5, 1:19:58–1:20:16)

At the end of their session, C3 wants to open the ‘SVN Commit’ dialog to submit their code changes to the central repository, but accidentally hits the IDE’s ‘SVN Synchronize’ button which triggers a potentially long-running comparison of the local and the remote state. He immediately notices his mistake, no harm is done, and C4 only mockingly asks “why?”, but apparently C3 still feels the need to explain his behavior.

C4: “Then let’s commit this.”

C3: “<clicks on ‘Synchronize’\*> We can briefly talk about this, about testability (!...!) Now I clicked on this. I didn’t mean to.”

C4: “<acted outrage> Why would you do such a thing? (!!Yes!!) Bringing ‘synchronize’ into the game.”

C3: “That’s (!...!) misclicked (!...!) I’m used to (!...!) I’m used to clicking a button down here, that looks like this, because I have it down there in the QuickView, you know? And then (!...!) was a reflex.”

C4: “<acted forgiveness> Well-well.”

C3: “<laughing> Yes.”

The degree of **Togetherness** determines the properties of the pair’s base activities: high **Togetherness** allows to understand the thinking behind **non-expecting** ★s, ▲s are **prompt** and **appropriate** because the ★s are fully understood; with medium **Togetherness**, ★s need to be more **expecting** to engage the partner, ▲s are sometimes **delayed** or **misled** because of misunderstandings; low **Togetherness** also leads to **non-expecting** ★s but these are *not* understood by the partner, so there are more ◀s, ▲s are more often **non-evaluative**, and there also also longer pauses and missing reactions (♦s).

A pair’s **Togetherness** may change over the course of a session. In the next Section 6.4.2, I discuss the factors that appear to have positive and negative effects on it. I discuss cases of pairs accidentally or deliberately working with a lowered **Togetherness** in Section 6.4.3, and strategies employed to **Maintain Togetherness** in Section 6.4.4.

### 6.4.2 Facilitators and Inhibitors of Togetherness

By comparing situations of pairs who easily understand subtleties of their partner’s actions with those riddled with non- or misunderstandings, I identified five factors that make it easier (or harder) for a pair to achieve **Togetherness**, i.e., that enable (or hinder) them to understand even implicit intentions under ambiguous conditions and react properly. On the following pages, I discuss their role and provide examples for illustrating positive and negative cases.

### 6.4.2 a) Factor: Shared Understanding of the System

Pair programmers work on technical tasks in a software system of some sort. They discuss *design* proposals, formulate *hypotheses* or *findings* about its aspects, etc. In their technical discussion, the pair members refer to some part (such as a module, a class, or a method) or aspect (such as a faulty behavior, a requirement, or its performance)—none of which are tangible objects (unlike, say, a sewing machine, see Section 2.4.1b). For a productive conversation, the partner then needs to understand the reference, what the proposal, idea, etc. is about.

#### *Enabling Togetherness*

It is not necessary for either partner to have a full understanding of the system. What matters is a *shared* mental model of the relevant parts and aspects. In Example 6.1, C5 refers to “*an object, where you can get the IColumns, where you might get to a Provider, or something*”. Such Providers are neither visible on-screen nor do they play any role in the session. But C2 still responds “*M-hm. Sure thing.*”. The pair clearly has a shared understanding of the system at this point, and it enables C2 to understand what C5 is talking about.

#### *Hindering Togetherness*

In Example 6.8, C5 proposes to use one method over the other and asks C2 about it, who admits he has “*no idea what it does*”. C2 understood his partner’s intention, but knows too little about the method in question, so his reaction is *non-evaluative*. C5 expected the method’s purpose to be part of their shared understanding, and afterwards *Maintained Togetherness* by explaining it to C2: “*That’s for getting a unique name for the attribute*”.

In the first discussed *Breakdown* case, Example 6.16, O3 did not understand the “matcher” assertion in the test case and the technical reason for the assertion failure (see [1] and [2] in the overview on page 211)—but O4 did *and* assumed that O3 does, too. They did not have a shared understanding of their test logic, which made it more difficult for them to understand each other’s proposals and findings.

### 6.4.2 b) Factor: Shared Understanding of Software Development

Pair programmers rely on troves of knowledge on how software development works, how to use certain tools, how to use idioms of programming languages to approach different types of problems, etc. The general argument is the same as for the shared system understanding: To understand what, say, a *design* proposal entails, the partners need to have a shared understanding of what can be done in software development in principle.

#### *Enabling Togetherness*

Although the pair J1/J2 from the running Example 6.13 spends much time on a tiny detail (the size of the polling interval), J2’s initial explanations of the system’s architecture and purpose are all easily understood because both J1 and J2 have a shared understanding of a central control architecture, distributed systems, event-driven vs. polling solutions, etc.

#### *Hindering Togetherness*

In first *Breakdown* case, Example 6.16, O3 mistakenly rejects O4’s hypothesis: O4: “*getEnd() is not defined in the test, I guess*”—O3: “*But if it calls this function (!...!)*”. This is probably because O3 is not aware of how the existence operator “?” works: It first checks whether a method is callable. This mismatch between O4 and O3 is never cleared up, and was one reason for why they did not really work together as a pair.

In session DA2 with the pair D3/D4, the effects were milder: D3 does not fully grasp the idea of the refactoring his partner D4 proposed (the Template Method design pattern), so the

pair falls back to an asymmetrical mode with low **Togetherness** in which D4 does most of the work. (Their case will be discussed in more detail in Example 6.22 on page 228.)

#### 6.4.2 c) Factor: One Shared Plan

Having an agreed upon plan for what to do in their PP session appears to make it easier for both developers to understand the underlying intentions because every proposal and action can be considered against the backdrop of the overall plan.

##### *Enabling Togetherness*

In generally **fluent** sessions, such as AA1, CA2, or CA5, the developers talk about what they want to achieve early in the session as well as occasionally mid-session with *strategy* and *state* activities. The following example shows the strategy decision of C3/C4 in session CA5 that happened before their **Focus Phases**.

##### **Example 6.20: One Shared Plan (CA5, 17:28–18:20)**

The pair decides on a strategy on how to introduce a new class similar to an existing one while avoiding needless duplication.

C3: “I’m not sure which parts we can actually reuse.” *explain\_gap in knowledge*  
 C4: “Me neither.” *agree\_gap in knowledge*  
 C3: “Either we try to extract a method here and try to call it. Or blunt, straight on, copy it. Check one by one, ‘Makes sense? Do we need it? Where are the differences?’ And only then we look how to bring it back together, are there similarities to use.” *propose\_strategy*  
 C4: “[...] OK, let’s copy that thing to execute (). Let’s see what we need.” *decide\_strategy*  
 C3: “M-hm.” *agree\_strategy*

This strategy is the foundation for the next 10 minutes of their session during which **Focus Phases** #1, #2, and #3 happened (see Table 6.3).

##### *Hindering Togetherness*

In problematic session OA1, the developers hardly have any *strategy* or *state* activities (see second **Breakdown** case, Example 6.17, for an excerpt). In session OA8, developer O4 *did* have a plan—he wanted to test his hypothesis that more methods needed to be mocked—but it was not *shared* as he was not able to effectively explain it to O3, so its execution took way longer than necessary (see Example 6.16).

#### 6.4.2 d) Factor: Workspace Awareness

Working on one computer (as in most analyzed sessions) is easier than working on two synchronized machines (e.g., session JA1, a distributed session mediated by Saros), which in turn is easier than working concurrently on two computers that are not synchronized (as in session OA1). Depending on their setup, the two developers’ individual realities are more or less similar and therefore provide more or less shared context to rely on.

##### *Enabling Togetherness*

Refer back to the **Focus Phase** in Example 6.15: The utterances of both partners are full of *deictic* references: “*You can just return here*”, “*We don’t actually need it*”, “*We need that*”, and “*still a bracket too much, at the end*”. Looking at the same physical screen, seeing the same content and cursor allows for such short utterances to be perfectly understood by the partner.

***Hindering Togetherness***

In Example 6.17, O3's references are not understood because O4 looks on his own screen (and follows a plan of his own): O3: "Ah, what about *this filterSelectedTeam?*"—O4: "Sorry?" and later again O3: "*I think it's this guy that we want.*"—O4: "Sorry?".

**6.4.2 e) Factor: Language Barrier**

Not being able to speak one's native language makes it arguably more difficult (a) to express oneself precisely as a speaker and (b) to interpret the partner's utterances as a listener. In most analyzed sessions, the developers spoke their first language, with sessions MA1, OA1, OA5, and OA8 being exceptions: the developers spoke English for most of the time, which is a second language for all of them.

But even if both developers can use their first language, *idiolects* might still be an issue. The same words, even when uttered in comparable contexts, can have different meanings depending on who speaks them. Both developers J1 and C2 use the same words "*Wait a second*" (German: "*Warte mal kurz*") on multiple occasions to reserve some time to process a thought (a *propose\_step*). In C2's case however, this utterance always has the connotation of '*Give me mouse and keyboard and let me drive*', which his session-CA1-partner C1 appears to understand and sits back and releases the mouse, but his session-CA2-partner C5 does not and keeps talking until C2 slightly pulls the keyboard towards him.

***Enabling Togetherness***

The extent of mutual understanding and concurrent speech of the *Focus Phase* in Example 6.15 is hardly imaginable between two developers who are below native speaker proficiency.

***Hindering Togetherness***

In the beginning of the *normal* PP illustration in Example 6.14, O3 and O4 (both non-native English speakers) try to infer the meaning of the variable "offsetDays" from its name and argue—in English—about the semantics of the English word "offset" (see turns 3–6). Between two native speakers, there would probably be one level of confusion less.

**6.4.2 f) Factors' Interplay**

I usually do not have enough evidence from the session recordings to pin down exactly *why* a developer does or does not make her ★s *expecting* and her ▲s *evaluative*. I identified these five factors by systematically comparing situations where partners understood each other easily with situations of misunderstandings. The factors are not necessarily independent and I cannot say much about their relative importance. Asking the developers themselves for an explanation in the post-recording interviews was not possible, as my analyses had not yet reached this point when the sessions were recorded. The next example illustrates how four of the factors come together (see Example 6.3 for more context):

**Example 6.21: High *Togetherness*** (CA2, 28:14–28:23)

The pair agreed on changing one parameter type of a method in a Java interface. C5 wants to use Eclipse's "Change Method Signature" refactoring for this, which is suitable for renaming and reordering parameters, or introducing new ones with a static default value. Changing parameter types, however, does result in compilation errors if the new type is not compatible with the old one, which is the case here. C5 still wants to use this tool, possibly to get a definitive list of compilation errors they can go through one by one. The only thing he asks his partner C5, however, is:

## Example 6.21 (continued)

★ C5: “Erm, do you know the <\*right-clicks on method name to open context menu\*> how I access the feature to change a method?”

▲ C2: “That doesn’t get you anywhere.”

C2’s ▲ is prompt, coming less than a second after C5 finished his ★, and it indicates the pair’s high **Togetherness**.<sup>a</sup> C2 apparently relies on a number of factors to interpret C5’s action and intentions:

1. **No Language Barrier:** C5’s utterance was a question to which he expects an answer.
2. **Workspace Awareness:** C5 wants to apply a refactoring (opened context menu) on some method (cursor position).
3. **Shared Understanding of Software Development:** C5 refers to the refactoring “Change Method Signature”, which is known to both developers for changing a method’s signature in all places where it is declared, implemented, or called.
4. **One Shared Plan:** C5 wants to apply this particular refactoring probably with the expectation to save some manual editing on their way to change the Java interface.

<sup>a</sup>Technically, C2’s reaction is only indicative of his “half” of the pair’s **Togetherness**. In the full exchange (see Example 6.3) it becomes clear that this depth of understanding is mutual.

To summarize, **Togetherness** is the degree to which pair programmers are able to interpret each other’s activities. High **Togetherness** enables a PP process with high **Fluency**. **Togetherness** is facilitated by having a reliable *common ground*, which in pair programming situations boils down to a *shared understanding of the system*, *shared understanding of software development*, *one shared plan*, *workspace awareness*, and not having to deal with *language barriers*.

In practical scenarios, pair programmers may find pragmatic ways to deal with any limitations regarding these factors. In the following Chapters 7 to 11, I characterize different ways how pairs **Maintain Togetherness** with regard to the factors *shared understanding of the system* and *shared understanding of software development*. I summarize such behavior as **knowledge transfer**. How pairs agree on *one shared plan*, how they deal with *workspace awareness* and *language barriers*, however, is beyond the scope of this thesis. In Section 6.5, I discuss some related work in this regard.

### 6.4.3 Not Maintaining Togetherness

Before I explain how pairs go about **Maintaining Togetherness**, I will first discuss cases of pairs who effectively gave up their **Togetherness**, either by accident or by choice in order to enable one developer to technically proceed with the task without giving the partner’s grasp of things a high priority.

#### 6.4.3 a) By Choice

Example 6.22 shows how a pair managed a situation where *one shared plan* could not be established—because one pair member lacked the technical skills for understanding it—by switching into a half-detached mode.

#### Example 6.22: Splitting Up (DA2, 40:03, 1:24:25, 1:35:42, & 1:41:17)

Session DA2 was probably planned as a training session, as this is D4’s first week with the company. D3 explains the rough structure of the system before the pair starts implementing a seemingly small feature. They soon encounter some design flaws in the existing system, talk to senior developers D7 and D6, and pivot the session’s goal towards a simple, but far-reaching refactoring.

After having spoken to the senior developer D6, junior developer D4 has a good understanding of the refactoring which D6 proposed: It boils down to extracting an abstract superclass from two

## Example 6.22 (continued)

dozen implementation classes using the Template Method design pattern (Gamma et al., 1995, pp. 325–330).<sup>a</sup> The senior D6 happily stops his explanation and leaves the group as D4 begins completing his sentences and apparently got the idea. D3, although being at the company longer than his new colleague, is rather cautious:

D3: “<laughs> Erm, okay? It seems you were able to follow this better than me. Well, go ahead then!”

D4 performs the refactoring in one class after the other while D3 sits by and provides the occasional class name or proposal for a parameter name. After forty minutes, D4 asks whether D3 is yet comfortable enough to perform some changes on his own:

D4: “Care to take over? <\*nudges keyboard\*>”

D3: “I think you’re more into this whole thing. I’m quite out of my depths here.”

And again another 10 minutes later:

D4: “So, you say when you want to take over, right?”

D3: “Nope, you go on now. I say when I’m back on track. I’ll give a shout.”

D3 apparently trusts D4 who had convinced senior developer D6. D4 continues with the refactorings increasingly mechanically, as more and more classes look similar to already refactored ones. D4 seizes the opportunity to explain the design pattern to D3 (see Example 7.14), and another six minutes later, proposes to let D3 take over again. Now D3 gladly accepts:

D4: “Yeah, if you want to take this?”

D3: “Yep, let’s do this.”

<sup>a</sup>Although the pattern name was not mentioned at this point in the session, it is clear that both D6 and D4 know and refer to the pattern.

The pair in Example 6.22 did not invest any effort in making sure they *both* fully understand the refactoring procedure, i.e., they did not have *one shared plan* and did not *Maintain* their *Togetherness*. One could argue that this is not *real* pair programming. But: First, for most of their session, D3 and D4 are textbook examples of the classical *driver and navigator* roles—one of them writing the source code, the other looking for defects—so, Example 6.22 would be ‘pair programming’ by definition. Second, and more relevant than clinging to such a definition, it shows how two software developers made a conscious decision about how to proceed as a pair, and even found a way to make D4’s very first programming experience in this company rather fruitful (see Example 11.10 for further discussion).

## 6.4.3 b) By Accident

Not *Maintaining Togetherness* is not usually a seriously considered option but an accident. Drastic cases can be seen in the *Breakdowns* in Examples 6.16 and 6.17. There are, however, milder forms. The next example illustrates how inadvertently low *Togetherness* can arise and how the pair did not notice it for 40 minutes.

**Example 6.23: Reading Documentation** (KC2, 14:12–15:04 & 53:54–54:24)

For writing an integration test of an auto-completion feature, developer K2 and K3 want to programmatically enter characters into an input field with a JavaScript library. They open the online documentation: K2 controls the mouse and scrolls around as he sees fit, while K3 only appears to read the fragments K2 points to:

## Example 6.23 (continued)

K2: “<\*opens documentation\*> (,,,,,) That’s event binding, but I want to trigger it <\*scrolls down\*> (,,,,,) Ah, <\*selects text at bottom of the screen\*> (#keypress#) and then <\*scrolls selected text to the middle of the screen\*> it gets triggered.”

K3: “<\*reads text behind selection\*> (#without an argument#), okay.”

K2: “<\*scrolls up\*> But, how can I (!..!) <\*moves cursor to example code with eventData\*>”

K3: “<\*reads code near cursor\*> (#EventData#)”

K2: “We have to tell it somehow, what it should do. <\*scrolls down\*> Maybe we just have to google it. Google certainly knows.”

K3: “Or we take a look at (~) how it’s done there, in the code.”

K2: “In the code <\*switches to IDE\*>”

At this point, K2 knows that “keypress(handler)” is used to bind a handler to the keypress event and that “keypress( )” (“*without an argument*”) can be called to trigger said event, but K3 does not. This discrepancy only becomes apparent 40 minutes later (during which the pair followed different approaches, none of which involve that particular piece of knowledge): K2 the uses the event-trigger idiom and K3 is puzzled by the statement, thinking it is faulty as it would bind an empty event handler:

K3: “But, what you just wrote there [...] but, I don’t think that we trigger an event this way. It’s more like an event handler that we (.) don’t implement.”

K2: “When it’s empty (!..!) no, when it’s empty, we looked this one up <\*opens documentation\*> (#to trigger the event manually#)”

K3: “A-ha, (#without an argument#) (,) but (,,,)”

It is possible that K3 did, in fact, fully understand the event-trigger idiom at 15:04 and completely forgot it by 53:54. A more likely interpretation of the above events, however, appears to be that K2 read the documentation at his own pace and was scrolling too fast such that K3 could only read out loud fragments in the proximity of K2’s cursor without really digesting them. This way, K2’s understanding of the relevant technology increased while K3 lagged behind, thus weakening their *shared understanding of software development*, thus decreasing the pair’s *Togetherness*.

The damage of *not Maintaining Togetherness* in the particular situation of Example 6.23 is probably negligible: First, there cannot have been any related defects which K3 missed because K2 did not use the newly learned `keypress()` idiom in the 40-minute period; second, clearing up K3’s misconception took K2 less than 30 seconds; and, finally, the resulting interruption did not appear to throw the pair off track. However, it points to a mechanism that could have bigger consequences, an *anti-pattern* that reduces *Togetherness* and which will be discussed later in Section 9.5.2: *Parallel Production* instead of *Co-Production*.

#### 6.4.4 Maintaining Togetherness

Pair programmers may *Maintain Togetherness* with regard to all five factors.<sup>2</sup> While all of these five involve some kind of knowledge, for the rest of my thesis (Chapters 7 to 11), I focus on knowledge transfer in a narrower sense. First, I exclude *language* issues as they are too fundamental. Second, I exclude *workspace awareness* and *one shared plan* as they are too short-lived. Finally, I exclude the developers dealing with opinions. What is left is how pair programmers *Maintain* a shared understanding of factual information about the *software system* and *software development in general*, while the rest is beyond the scope of this thesis

<sup>2</sup>In an earlier publication (Zieris & Prechelt, 2016), *Maintaining Togetherness* was called “*resynchronization*”.

and calls for further investigation. As a starting point, I share a number of observations in the following sections.

#### 6.4.4 a) Excluded Factor: Language

There were only few sessions in my data where the pair members did not have a common first language. In sessions MA1, OA1, OA2, OA5, and OA8, the respective developers agreed to use English as the strongest common language. Occasional difficulties to express oneself were coped with by adding gestures (e.g., O3 mimicking a distance with her two index fingers, see Example 6.14).

Being able to communicate fluently in a spoken language certainly involves *knowledge*, at least in the sense of the cognitive sciences (see Section 2.2.2). I exclude activities that deal with differences between the pair members pertaining to this type of knowledge from my research not because it is not relevant, but because it is a more fundamental aspect of human social activity than what practitioners have in mind when they refer to *knowledge transfer* in pair programming.

#### 6.4.4 b) Excluded Factor: Workspace Awareness

The pair J1/J2 has a reduced workspace awareness in sessions JA1 to JA9 since they employ distributed pair programming. Schenk (2018) studied how they deal with this (see page 84).

##### Example 6.24: Maintaining Workspace Awareness (JA1, 53:56–54:34)

J1 and J2 do not share the same physical workspace. Since they cannot physically point to each other's screens and say *there*, the pair members developed a habit of routinely selecting portions of the source code (which their tool mirrors to the other side) or mentioning concrete line numbers when referring to them. J1 has selected lines 88 to 105 and his viewport shows lines 72 to 102 while he makes the following proposal:

J1: "Ah, so that means this <\*selects lines 88 to 105\*> (!...!) that means this case we can pull into that try [lines 76 to 79], right?"

J2's screen is larger and shows lines 42 to 108. He wants to make his partner J1 aware of an `if`-statement in 51, which happens to be beyond J1's current viewport.

J2: "We can (, ,) this, no! No-no-no-no-no-no-no. We can not, because <\*cursor to line 51, selects `if`-keyword for J1 to see\*> please be aware of line fifty-one <\*selects whole `if`-statement for J1 to see\*>, the `localNewsFile`."

J1: "Line fifty-one, what?"

J2: "We can not do this."

J1: "Ah, you mean <\*starts scrolling up to J2's selection\*> that we (!...!)"

J2: "What we can do, however (!...!!)"

J1: "Aargh-ha-ha-ha urgh <resignating>, oh god. Yes, you are right, you are right."

Note that neither developer actually spells out the problem that J2 noticed, but with the limitations of their workspace awareness mitigated, both developers reach a high *Togetherness*.

Being aware of what the current state of the (virtual and physical) workspace of each of the pair's members is a type of knowledge. Unlike language-related knowledge, it is presumably even mostly explicit knowledge that can be verbalized, as the developers in the example above demonstrated. It is, however, rather short-lived since knowing the current position of the text cursor or the content of a displayed error message are relevant for taking part in the ongoing

pair process, but is not something that developers take out of their session. It is not what practitioners mean when they talk about *knowledge transfer* in pair programming.

#### 6.4.4 c) Excluded Factor: One Shared Plan

One advantage of having two developers working on one task is that even if one of them forgets something, such as the original plan, the other can remind her:

##### Example 6.25: Maintaining One Shared Plan (AA1, 25:39–28:00)

The pair wants to work through four related defects and just arrived at case number three. A1 makes a start to look into the implementation of list no. 3 (turn 1) when A2 makes a discovery that makes the pair take a detour to look into a different aspect of the system (turn 2).

- (1) A1: “Let’s see how it’s rendered.”
- (2) A2: “<disgusted> Why does it have its own `isActive()`? <sucks air through teeth>”
- (3) A1: “No clue. And I don’t want to (!..!)”
- (4) A2: “No, I want to fix this, because <laughing> otherwise this one does something different than the other and then it’s screwed up again.”

Both partners follow A2’s initiative. They start to read and understand that the class has “*its own `isActive()` [implementation]*” for a good reason, have a short discussion on the pros and cons of moving such logic to the backend, and they agree on deferring this until the actual task is done (25:51–27:20, not transcribed). In turn (6), A1 wants to get back to the original issue (i.e., understand the rendering of list no. 3), but A2 at this point lost track of the latest open topic and their once-shared plan and he wants to start at the beginning again (list no. 1):

- (5) A2: “Well.”
- (6) A1: “Yes, we need to see how it’s rendered [= list no. 3]. Because have to see what it’s (!..!)”
- (7) A2: “Well, let’s start with the first page again. <\*switches to browser, look at A1’s handwritten notes\*> The first page you wrote down was ‘*Finish Tasks*’, right?”

A1, however, insists on continuing with list no. 3 and it requires multiple attempts of convincing A2 before they arrive at a shared plan again.

- (8) A1: “Why don’t we finish off the version page [= list no. 3] now?”
- (9) A2: “Because we did not finish the other ones either.”
- (10) A1: “Hello?”
- (11) A2: “We did not finish the other ones either. <\*switches back to IDE\*>”
- (12) A1: “We started this one [= list no. 3]. Why wouldn’t we finish it now? You only need to change the rendering of the text.”
- (13) A2: “<exhales audibly>”
- (14) A1: “Not there <\*points to screen\*> but in the `ContentVersionsView` thingy [= list no. 3]. Then we can complete that thing at least (!..!) (~) is working at least.”
- (15) A2: <\*Opens `ContentVersionsViewPage`, i.e., list no. 3\*>

Both developers continue inspecting the source code around the third defect.

An agreed-upon plan is relevant information in software development and therefore a type of knowledge. But similar to workspace awareness, most of the in-session plans are too short-lived to consider talk about them *knowledge transfer* in the practitioner sense, so I exclude it from my work.

How pairs **Maintain** one shared plan is an interesting topic for further work (Section 14.3). In Example 10.7, for instance, I discuss on how one pair used TODO comments for this.

#### 6.4.4 d) Excluded Type: Opinions

Maintaining a shared understanding of software development is not necessarily knowledge transfer in a narrow sense, but can be more a matter of opinion. I have seen too few instances of developers trying to consolidate different perspectives to properly discuss to which degree it can be considered *knowledge transfer*. In most instances, expressed opinions are accompanied by pieces of factual information. What follows is an example of one pair dealing with a well-known difference in their understanding proactively.

##### Example 6.26: Dealing With Conflict (CA5, 23:20–24:20 & 43:34–43:57)

C3 and C4 mostly jokingly deal with their different preferences of coding styles (the first segment was analyzed by Harms, 2017, pp. 28–31, whose Bachelor’s thesis I supervised).

C3: “You know what’s coming now.”

C4: “No. No!”

C3: *<\*types Ensure\*>*

C4: “Please please don’t. *<starting to smile>* Let’s write tests. (...) Then you won’t like these Ensures there. *<\*turns to C3\*>* *<laughs>*”

C3: “Do they hurt you, if I put them there now?”

C4: *<-laughing>* Yes, totally. But fine, if it’s not hurting you when I remove them for testing”

C3: “If you for testing (!...!) if they are in your way for testing (!...!) but as long they are not in the way (!...!) *<\*starts typing ensure call\*>*”

C4: *<-still smiling>* Argh, they are in the way.”

C3: *<\*introduces four calls of ensureArgumentNotNull, one for each method parameter, then looks at C4 smiling\*>*

C4: “We’ll never reach an agreement. Never!”

C3: “When there is a solution which does this all better, possibly with AspectJ?”

Later in the session, after the pair created a new class and C4 is typing, they pick up their play fight again:

C3: “Hey, you? Will you give me an Ensure?”

C4: “Naargh. *<\*begins insert Ensure statements\*>* My god, you are a sadist. (,,,,,) Shall we discuss this sometime? When we are not in the midst of programming?”

C3: “If you like, we can do that.”

Although knowing how the partner wants to approach certain types of tasks may be knowledge that is relevant for pair programmers, I exclude attitudes and opinions from my further analysis because the example above and Example 4.3 are the only instances that caught my attention (without a systematic search, that is) where developers expressed their opinion *without* also including factual information to satisfy some knowledge need.

## 6.5 Discussion of Related Work and Summary

In this chapter, I extended the base concepts with a new property. Each base activity can have a **conversational role** of which I distinguish ★ initiatives from ▲ pair-referential, ◀ self-referential, and ● corrective activities. Additionally, each base activity and also the lack of an activity can be a ♦ conversational defect. The ideas of conversational turns belonging together, e.g., as ★-▲-▲ rather than all being isolated ★s, and of an utterance being ‘wrong’ or even ‘missing’ after the partner said something, are long known to linguistics under the terms of *adjacency pairs* and *conditional relevance* (see Section 3.2.1b).

In their work on creative collaboration in pairs, Bryan-Kinns et al. (2007) distinguish three levels of mutual engagement indicated by how the pair members deal with their partner’s contribution: An *acknowledgement* indicates a basic engagement, *mirroring* indicates a medium engagement, and *transforming* indicates a high engagement. In my terminology, such actions correspond to **non-evaluative** and **evaluative** ▲s. In the original study, the pairs had to compose ringtones with a distributed music app without directly seeing or hearing each other, so ‘dealing with the partner’s contribution’ amounts to things like repeating a motif of three notes. Nevertheless, Plonka et al. (2012a) used the indicators (and added *questions/reponses* as another indicator for the medium level) to identify and compare episodes of basic engagement and disengagement in pair programming (see my discussion on page 84). A disengagement episode is characterized by the absence of these indicators, which in my terms would be ♦s. There are no indicators corresponding to my notions of ★, ◀, and ●.

The five activity types (★, ▲, ◀, ●, and ♦) and their properties (**expecting**, **prompt**, **evaluative**, **appropriate**, see Table 6.1) are the building blocks to operationalize three qualitative levels of pair process **Fluency**. Most pairs are at **normal Fluency** most of the time. Some pairs occasionally speed up in **Focus Phases**, other pairs suffer from **Breakdowns**. **Focus Phases** were mentioned under different names before by Belshee (2005, Sec. 1.2), Chong & Hurlbutt (2007, Sec. 5.1.2), and Salinger et al. (2008, p. 20), but not described or discussed in detail. A pair’s process quality in general has been of research interest before, but not addressed in depth: Mathieu et al. (2000) and Domino et al. (2003) merely scored pair communication, coordination, cooperation, and conflict handling with rating scales (my discussion on pages 72 and 101, respectively).

A pair’s **Fluency** is influenced by its **Togetherness**, which is a central concept of this chapter and, in fact, of pair programming in general as it is what makes two software developers work *as a pair*. **Togetherness** in itself is not observable. It characterizes how easy or difficult it is for the pair members to understand each other. From my data, I identified five other factors that influence a pair’s **Togetherness**—(1) a shared understanding of the software system and (2) of software development in general, (3) one shared plan, (4) workspace awareness, and (5) language barriers—all of which may be addressed by pair programmers to **Maintain Togetherness**. Recurring to the terminology of small group research (see Section 2.4.2a), **Togetherness** is a form of *shared cognition* or coordination-as-a-state, and **Maintaining Togetherness** is the process of creating such a shared cognition or coordination-as-a-process. Observations from some pair programming studies can also be put under the heading of individual factors:

- **(1) Shared Understanding of the Software System:** Bryant et al. (2008, Sec. 6.3, my discussion on page 72) observed that switches between the driver and navigator roles “*appear to be very fluid with little accompanying explanatory conversation*”. They conclude that the navigator must have “*maintained a clear mental model of their current state*” which enabled her to take over the driver’s role at a moment’s notice.
- **(3) One Shared Goal:** Cao & Xu (2005, discussed on page 78) describe setting a session goal and adjusting it along the way as a property of highly competent pairs.

- **(4) Workspace Awareness:** Flor & Hutchins (1991) noted in their analysis of one programmer pair through the lens of distributed cognition that the *shared physical workspace*—together with the pair’s *shared goal*—provides enough context to allow for efficient communication (see also my discussion on page 79).

There might be additional factors contributing a pair’s **Togetherness** and their ability to **Maintain Togetherness**. A blog post by developer Maaret Pyhäjärvi called *Power dynamics in pairs and mobs* describes a PP situation that could also be characterized as a **Breakdown** due to pair members not fully understanding each other’s intentions:

“ So we sat down together, I guided him first to get to the functionality just to see it was there. The functionality included a new dialog, and as it popped open, my first words were “I wonder what size it is...”. *It wasn’t intended a question, but very much taken as one.* What I meant is that we have an agreement on the resolution where things still must be usable on the screen without scrolling. We were clearly on a higher resolution, and still the dialog was big and clunky. But before I got in another word, Llewellyn *picked up on the cue and started showing me tools* that enable me to measure the dialog size in centimeters, pixels you name it.

I didn’t even understand right there and then that I was uncomfortable. That I felt over-ridden. That none of the stuff I was trying to show—my work in exploratory testing—got done. [...] I tried enjoying the results we ended up with, not miss the results that were hijacked from me by a power dynamic.

After this experience unfolded [...], we have had great pairing sessions and learned to explore in a pair and mob a lot better. He actively worked against the power dynamic, paying attention to listening instead of talking over me, thinking their way is always correct.

Pyhäjärvi (2018, emphases added)

Labeled with base concepts, Maaret probably uttered an *explain\_finding*, a *non-expecting* ★, which her colleague Llewellyn understood as an *ask\_knowledge*, an *expecting* ★, and reacted with ▲ *misled* explanations. The “*power dynamic*” which Maaret sees in this PP session is due to a structural “*belief system on how men and women interact*” that places female software developers in a listening rather than a contributing position. This dynamic consists of two parts: On the one hand, it primed her male colleague and led to his false interpretation and ▲ *misled* explanations; on the other, it prevented herself from clearing up the misconception with with ● activities.

Although I did not systematically search for instances of such a power dynamic at play in my data, such problems appear less pronounced as only few candidates come to my mind:

- In session OA5, senior (male) developer O1 explains many aspects of general software development to his novice (female) colleague O3. Not all of these explanations were specifically requested by O3; some of which she appreciates (e.g., “*That’s a very nice strategy*” at 19:05), others were unnecessary (e.g., “*Yeah, tests should be atomic*” after a one-minute explanation by O1 at 16:04), for others it is unclear (e.g., “*Hm. <stares at screen, lips screwed up, nodding\*>*” after a one-minute explanation by O1 at 06:48).
- In (same-gender) session PA3, backend developer P1 provides some unwanted explanations to frontend developer P3. In contrast to session OA5, however, the receiving partner makes his discomfort explicit (see Example 9.23).

For any further research it should be noted that my type of data and method of data collection—in particular the lack of separate *individual* accounts of the pair members—may not be suitable for determining whether one partner felt uncomfortable during the session.

Williams (2000, p. 53) describes the transition from two developers “*considering themselves as a two-programmer team [to] considering themselves as one coherent, intelligent organism working with one mind*” as **pair-jelling**. In the analysis of the difference in development time between solos and pair programmers,<sup>3</sup> she considered only the second and third experimental rounds because that is “*after the pair-jelling has occurred*” (*ibid.*, p. 64). She describes jelled pairs as more productive than unjelled pairs, but it is not clear what the difference actually is: Is it both developers individually getting used to not working alone anymore (Williams et al., 2000, p. 22) or is it the union or “*bonding*” of two developers (Williams, 2000, p. 101)? The first would only need to happen once per developer, but the second would be necessary for every new pair constellation.

Considering the concept of **Togetherness**, I can see aspects of both views. How to **Maintain Togetherness** during pair programming regarding the five factors may be a general learnable skill which may be practiced with any partner with whom there are differences regarding at least one factor. In addition, how to then *jell* with a specific partner may then be a matter of getting to know each other’s idiosyncrasies, e.g., to become able to distinguish an actual question from a rhetorical one, or a humorous remark from actual criticism. From what I have seen in my data, I would subsume the latter under the *Language Barrier* factor (see C2’s peculiar “*Wait a second*” discussed on page 227). I did not systematically study fresh pairs or developers new to the pair programming work mode over a longer period of time to observe any *jelling*.

In the following chapters, I will discuss how pair programmers actually transfer knowledge; or, in the terminology of this chapter, how they **Maintain Togetherness** with respect to a shared understanding of their software system and software development in general.

---

<sup>3</sup>I did not address this experiment (descriptions of which can be found in Williams, 2000, pp. 38–65, Williams et al., 2000, and Williams & Kessler, 2001) in particular in my related work discussion in Section 2.3, because it was part of the meta-analysis by Hannay et al. (2009, discussed on page 63).

## Chapter 7 Knowledge Conceptualized

---

7.1	Purpose and Structure of this Chapter . . . . .	237
7.1.1	Three Situation-Based Knowledge Concepts. . . . .	237
7.1.2	Discussion of Recurring Example . . . . .	239
7.2	<b>Knowledge Want.</b> . . . . .	240
7.2.1	Properties of Knowledge Wants . . . . .	240
7.2.2	Internal Knowledge Wants . . . . .	241
7.2.3	External Knowledge Wants . . . . .	242
7.2.4	Collective Knowledge Wants . . . . .	243
7.3	<b>Topic and Target Content</b> . . . . .	244
7.3.1	Types of Topics and Target Contents . . . . .	244
	<i>S knowledge: System-Specific Knowledge • G knowledge: Generic Software Development Knowledge • Other Types of Knowledge • Application Domain-Specific Knowledge?</i>	
7.3.2	Hypothetical Target Contents . . . . .	254
7.4	Summary and Discussion of Related Work . . . . .	254

### 7.1 Purpose and Structure of this Chapter

Verbally, it makes sense to say things like ‘*developer A knows about X*’, ‘*developer B has a knowledge gap regarding X*’, and ‘*A transfers knowledge to B*’. But what do I mean by that?

In the previous chapter, I excluded different kinds of knowledge which do play a role in pair programming, but not in the sense that is commonly referred to as *knowledge transfer* by practitioners. What remains are developer activities that pertain to factual information concerning their concrete software system and software development in general. Before I dive into the mechanics of these activities in the following chapters, I first describe three concepts that together characterize a knowledge transfer situation in pair programming as well as different types of relevant knowledge.

#### 7.1.1 Three Situation-Based Knowledge Concepts

I already discussed theoretically the class of knowledge I can empirically access as a researcher: Roughly speaking, it the part of the pair members’ *explicit* knowledge which can be *reconstructed from their activities* (see Figure 4.5). I now shift the perspective to what practically matters in concrete pair programming situations. Here, there is more than just ‘dry’ information. There is a process which is driven by a lack of knowledge—or more precisely: A *perceived* lack.

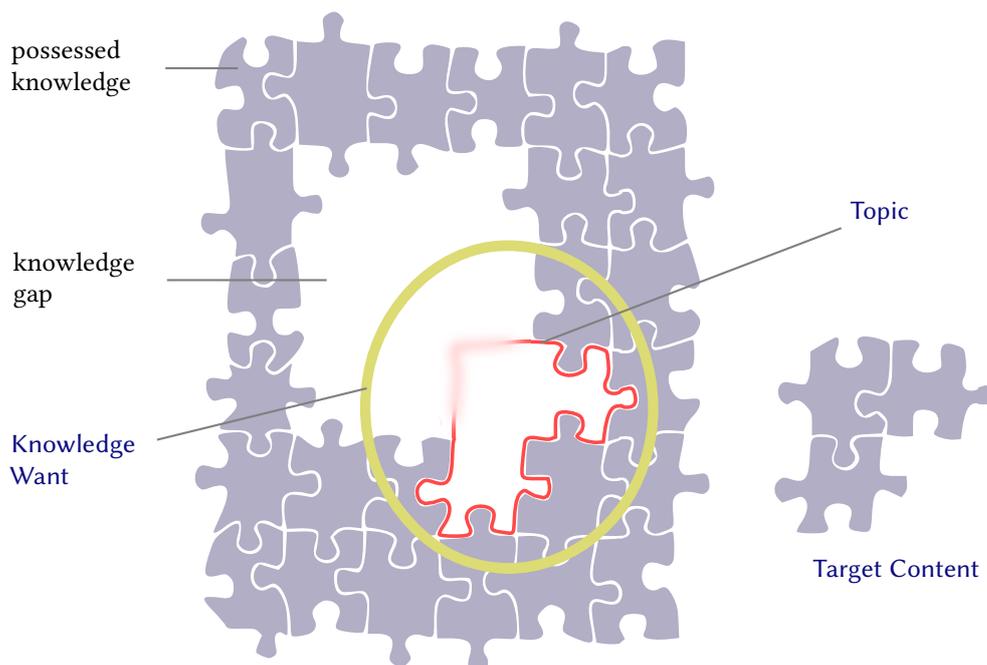
Pair programmers do not always talk explicitly about what they know and do not know. Nevertheless, some of their actions such as asking questions or providing explanations can be attributed directly to dealing with their own or their partner’s knowledge gaps (as I described in Section 4.5.2c). The underlying knowledge gap, however, remains elusive—both for me as a researcher and the pair member who also cannot fully grasp her partner’s state of mind. A **Knowledge Want**, the motivation to address a knowledge gap at some point during a pair programmings session, pertains to *perceived* knowledge gaps. This means that (1) there might

be actual knowledge gaps for which no pair member develops a **Knowledge Want**; (2) not all perceived **Knowledge Wants** are actually addressed; (3) not all pursued **Knowledge Wants** necessarily lead to successful acquisition or exchange of knowledge; and (4) not all **Knowledge Wants** necessarily correspond to an actual knowledge gap.

The **Target Content** is the canonical and abstract information which can fill the underlying knowledge gap. I introduce this concept to be able to speak about task-relevant information as such, independent from any developer knowing or talking about it at some point during a PP session. The pair may or may not possess the **Target Content** for a knowledge gap, they may even possess it without being aware of it (as in Example 7.1), or one pair member may consider some information to be the **Target Content** while her partner wanted to know something different or even nothing at all (in which case there is no **Target Content**).

The third concept to describe knowledge transfer situations is the **Topic**. Just like the **Knowledge Want**, the **Topic** ‘belongs’ to the pair member who perceives a gap in knowledge; it is what the **Knowledge Want** is about. Usually, but not always, both pair members need to understand the **Topic** for effective communication.

Metaphorically speaking, if a developer’s body of knowledge is a jigsaw puzzle, a knowledge gap is an empty area with missing pieces, a **Knowledge Want** is the intention to fill out a part of that area, the **Topic** is the shape of area to be filled out, and the fitting pieces are the **Target Content**. I summarize the relationship of non-observable knowledge aspects (possessed knowledge and knowledge gaps) and these three concepts in Figure 7.1.



**Figure 7.1:** Knowledge concepts illustrated with a jigsaw puzzle metaphor. Possessed knowledge and the knowledge gap itself is not directly observable, but a pair member may develop a **Knowledge Want** to address some part of it. The **Topic** is what the knowledge transfer, i.e., the exchange of existing knowledge or the acquisition of new knowledge, is about; the **Target Content** is the information which is able to fill the knowledge gap that gave rise to the **Knowledge Want**.

Jigsaw background based on image by user “OpenClipart-Vectors” used under Pixabay License. (URL: <https://pixabay.com/vectors/jigsaw-puzzle-game-shape-puzzle-152865/>)

## 7.1.2 Discussion of Recurring Example

I apply all three concepts—**Knowledge Want**, **Target Content**, **Topic**—to the recurring example from the beginning of session JA1 below. This is also another illustration of how the reconstruction of explicit knowledge from the pair’s activities works.

### Example 7.1: Knowledge Wants, Topics, and Target Contents (JA1, 02:29–06:15)

In the first minutes of session JA1, developers J1 and J2 deal with one knowledge gap: J1 not knowing how the software currently works and how it is embedded in its environment. The knowledge gap itself is not observable, but the actions of the developers dealing with it are. In this instance, each of the pair members develops a **Knowledge Want**: J1 is concerned with a design detail and J2 provides him with answers, and J2 is additionally concerned with explaining the system as a whole and its context. See Example 5.1 on page 186 for more background information and the full transcript with the same line numbers.

#### Aspect #1: Design Detail

##### Knowledge Want

When J2 mentions the polling mechanism, something appears to catch J1’s attention, which is presumably the moment he develops his **internal Knowledge Want**.

- (11) J2: “If so, the most recent file is selected and it starts checking how the file changes size-wise.”  
 (12) J1: < \*stops nodding, looks to his upper right\* >

##### Topic

The **Topic** which J1 wants to clarify is only hinted at in his question and obscured by his suboptimal choice of words: “*In what time window are you looking?*” (turn 16). A clearer formulation would have been something like “*What is the length of the polling interval?*”.

##### Target Content

The **Target Content** to satisfy J1’s **Knowledge Want** is ‘30 seconds’—he eventually explicitly confirms “*that’s what I wanted*” (turn 29).

However, J1’s partner J2 misunderstands the **Topic**. Over multiple conversational turns (16–27), J1 keeps pursuing the clarification of the **Topic** to satisfy his **Knowledge Want**, until he finally makes J2 understand the **Topic**. Note that J2 was in possession of the **Target Content** the whole time, i.e., he knew the polling interval was 30 seconds (turn 28), but did not understand that this was what J1 was after.

Up until the end of the excerpt, J2 appears to have understood the **Topic** as something like ‘*For how long is the polling going on?*’ to which the **Target Content** would have been: “*start[ing] two minutes after the full hour*” (turn 17) and with a variable end (turns 19 & 21), but at most until “*five [minutes] before the [full] hour*” (turn 26).

#### Aspect #2: Software Design and Context

##### Knowledge Want

J2 (who originally wrote the software) appears to have expected a relevant knowledge gap on part of J1 since he opens the conversation by explicitly asking J1 about his *standard of knowledge*. Based on J1’s reluctant answer, J2 develops his **external Knowledge Want**.

- (1) J2: “Do you know the NewsPlugin, or don’t you know it?” *ask\_standard of knowledge*  
 (2) J1: “<exhales audibly> Just show to it me again.” *explain\_standard of knowledge*<sub>PT</sub>

##### Topic

J2 goes on to explicitly state the **Topic** in turn (4): “*OK, I can give you the big picture of what this plugin does, overall.*”

Example 7.1 (continued)

### Target Content

J2 provides explanations on the context which the software operates in, which go beyond what J1 explicitly asks for. In particular, J2 explains (or at least addresses) the following aspects of the software system (turn numbers in parentheses):

- The system’s purpose is to produce news recordings (6), i.e., raw data is retrieved from radio stations and then converted to a different format (15).
- On a structural level, there is a central plugin which controls multiple processors, each of which is responsible for one radio station (8).
- The radio stations write their news broadcasts directly to files which continually grow until the broadcast is over (10, 11, 13, 22).
- News segments do not have a precise pre-determined start, end, or length; but at two minutes after the full hour any broadcast would have started and usually ends seven minutes later (17, 19, 21).
- There is no event mechanism of any kind, just files on a remote file system with their size and modification date (10, 11, 22).
- The software relies on a file size-based polling mechanism to determine the end of a news broadcast (11, 13).
- Any polling activity is stopped at 5 minutes before the full hour (26).

Note that since J2 both follows his own **external Knowledge Want** and reacts to J1’s questions (which in turn result from following an **internal Knowledge Want**), each individual utterance contains at least in part something J2 thinks J1 *wanted* to know **and** something J2 thinks J1 *should* know. I do not claim that the above delineation is the only possible way to interpret the situation.

## 7.2 Knowledge Want

A **Knowledge Want** is the motivation to fill a perceived knowledge gap. It is indirectly observable if and when the developer perceiving the **Knowledge Want** acts upon it.<sup>1</sup>

### 7.2.1 Properties of Knowledge Wants

**Knowledge Wants** can be **internal**, when a developer wants to know or understand something, or **external**, when a developer wants her partner to know something. Note that **external Knowledge Wants** do not necessarily correspond to actual knowledge gaps: Thinking that the partner ought to, but does not yet know something, although she ‘objectively’ already does, is still a **Knowledge Want**. As a third option, **Knowledge Wants** can also be **collective**, i.e., it is clear to both developers that either of them has the same **internal Knowledge Want**.

For characterizing such PP situations, I am more concerned with the variety of the phenomenon (i.e., *open coding*, leading to the three types of **Knowledge Wants**), than with a systematic analysis of the circumstances which led to the situation (i.e., *axial coding*). A causal analysis is limited (a) by the available data (e.g., the recorded sessions do not include the developers’ backstory and their reasons for wanting and needing to know certain things) and (b) by the inability to accurately interpret the developers’ cognitive states (e.g., in Example 7.1, I can only *presume* that the moment J1 stops nodding is when he develops his **Knowledge Want**). However, in the next three sections, I characterize a number of situations where **internal**, **external**, and **collective Knowledge Wants** arise, respectively.

<sup>1</sup>While writing Zieris & Prechelt (2016), the concept of **Knowledge Need** (which is a developer’s *actual* lack of knowledge with respect to a task, see Section 1.1.2) was not yet developed. Perhaps confusingly, the *perceived* lack of knowledge which I now call “**Knowledge Want**” was labeled “*knowledge need*” back then (short for “*need for knowledge transfer*”).

### 7.2.2 Internal Knowledge Wants

An **internal Knowledge Want** becomes observable when a developer starts asking questions or performing actions to find something out. As Salinger & Prechelt already observed, questions are usually the result of a recent insight that made the developer aware of a relevant knowledge gap (BL, p. 162). Such an insight may come at any point during a session. One situation to develop an **internal Knowledge Want** is that during the implementation of an agreed-upon code change either pair member ponders the potentially problematic consequences of the changes that are about to happen, as the next example shows.

#### Example 7.2: Internal Knowledge Want During Implementation (AA1, 16:20–16:42)

The pair agreed upon retrieving a business object (`MiniObject`) from a proxy object (`objectHandle`). As A2 is about to implement the necessary changes, information shown in the IDE's auto-completion reminds him of the fallback objects (`MicroObjects`). The autocompletion offers two options: One method that will return such a fallback object in case of an error, and another which will either return the real object or throw a runtime exception.

A2: "We do `fetchMiniObject` anyways, so (,,,) <\*starts typing, auto-complete opens\*>"

```
return objectHandle.fetchMiniObject();
```



As A2 hesitates, A1 proposes to choose the selected (fallback) option.

A1: "Yes!"

A2's **internal Knowledge Want**, however, is not about which option to choose, but how to go about the error handling. He knows that the `MicroObject`-based fallback mechanism will be removed in the future (he is, in fact, surprised that it still exists). With very few words the pair reaches a common understanding: Although the `MicroObjects` will be removed soon, using them is still good enough for now. The **Target Content** is transferred and the **Knowledge Want** is satisfied.

A2: "Do the `MicroObjects` still exist?"

A1: "They do here, on your machine. It's on the working branch (!...!) true!"

A2: "That's enough?"

A1: "Yes, think so."

A2: <\*types 'true'\*>

A1: "M-hm."

Another scenario is an incidental insight. The developer's attention meanders around (e.g., because the partner executes some simple task) and things she does not understand catch her attention:

#### Example 7.3: Incidental Internal Knowledge Want (AA1, 1:05:29–1:05:38)

A2 is completing some changes while A1 looks through the neighboring lines in the, to him, less familiar Objective-C code. A1 notices a reference to another programming language (TCL), which the company uses to generate test scenarios. The resulting **internal Knowledge Want** is not resolved by transferring any **Target Content**, but by convincing A1 to no longer pursue the matter.

A1: "<amused> (#TclCode#)?"

A2: "<grinning> You don't wanna know."

A1: "Ok <snorts> (...) it's all fine then."

### 7.2.3 External Knowledge Wants

An **external Knowledge Want** is what motivates an explanation of a pair member. For pair programmers to develop an **external Knowledge Want**, there are several possibilities. Here are two cases:

1. Her partner specifically asked for something. In Example 7.2, A1 developed an **external Knowledge Want** after understanding what puzzled A2. In Example 7.3, however, A2 did not develop an **external Knowledge Want** although he presumably understood what his partner found confusing: He did not consider the **Topic** worthy of a detailed discussion.
2. One developer appears uncertain, possibly her reactions are **delayed** or **non-evaluative** **▲**s, or even missing altogether (**◆**), indicating a lack of knowledge. Or she is confident, but her **▲**s appear to be **misled**, indicating a misconception. Her partner may decide to explain what she deems necessary, as C5 did in Example 6.8 after his partner C2 could not assess his design proposal, or as O4 did in Example 6.14 in turn (17) to explain to his partner O3 how to approach the definition of a test case.

In session MA1, programmer M1 could develop an **external Knowledge Want** in a rather unique way: His partner M2 wrote down what he learned in text form, giving M1 more time and options to detect missing elements in M2's understanding:

#### Example 7.4: Explicit Topics and Target Contents (MA1, 05:13–11:16)

M2 wants to understand the purpose of all database tables and has prepared a file with a number of SQL SELECT-queries and his questions as comments. Throughout their 25-minute session, M2 would execute one query, ask M1 about the result, and write down his insights in the same file. This way, M1 always had a written form of M2's **Topic** and could assess whether he understood the **Target Content** by reading M2's formulation. Occasionally, M1 would add some more explanation or clarification, as in this excerpt:

M1: "We have a lookup table, which is this one. If you could open it?"

M2: "Actually, I have it here. <\*highlights and executes next SELECT query in file\*>"

M1: "So here we have, based on the componentName, we have these three columns."

M2: "OK, got it."

For the next four minutes, M1 explains different details and M2 listens (05:30–09:43). Then, M2 summarizes his understanding in a code comment which triggers M1's **external Knowledge Want** (probably through M2's use of the singular in "the table"):

M2: <\*writes comment 'take values from the lookup table'\*>

M1: "By the way: This is one lookup table, but there will be one or two more, similar like that, but for other websites. Because this lookup table is for one website."

M2: "Ah <\*slow nodding\*> okay, got it."

M1 again explains more details for about one minute (10:13–11:01), before M2 amends his code comments and M1 is content:

M2: <\*adds 'Lookup to be done based on componentName and fk\_website'\*>

M1: "M-hm."

M2: "Ok, cool. Understood."

### 7.2.4 Collective Knowledge Wants

The **Topic** for a **collective Knowledge Want** is often not communicated explicitly, but is understood by both developers from context—a result of the pair’s **Togetherness** (see Section 6.4). I discuss one such case later in Example 7.13.

There are, however, some cases where the steps of the thought process become observable: From one developer’s **internal Knowledge Want** to a communicated **Topic** to a **collective Knowledge Want** shared by both developers and to concrete actions to acquire the **Target Content**.

#### Example 7.5: From Internal to Collective Knowledge Want (AA1, 08:58–11:08)

The pair looks at the web frontend of their software in a browser where some parts are crossed out. A1 wants to find out how the strikethrough is implemented (**Topic**) and immediately proposes a hypothesis of how the current implementation might look like. He follows up with a procedure to perform the validation—inspect the generated HTML output of the current site—all of which A2 silently agrees to as he puts the proposal into action.

- (1) A2: “Around the texts we can simply put a span `inactive` and decide with a toggle to strike it or not strike it. Meaning, this question can be answered at the end.”  
*propose\_design + propose\_strategy*
- (2) A1: “< \*looks at A2\* > Yes. OK.”  
*agree\_design*
- (3) “< \*looks at screen\* > How is it currently done in there, the strike through?”  
*ask\_knowledge*
- (4) A2: < \*looks at screen, raises eyebrows, purses lips\* >
- (5) A1: “Is it a span tag or something? Maybe it’s already the case.” *propose\_hypothesis*
- (6) “Can you check the sources somehow? ‘Frame source’ or something?” *propose\_step*

The pair has to switch browsers as their current browser does not have a ‘Show Source’ feature. Shortly after, they can read the source code and both developers find what they looked for.

- (7) A2: < \*selects code <span class="inactive">test</span>\* >
- (8) A1: “Ah, (`#span class inactive#`), okay.” *explain\_finding*
- (9) A2: “Exactly.” *agree\_finding*

A1 develops his **internal Knowledge Want** after he looks at the screen where he sees text being crossed out and realizes there has to be some implementation already, and formulates his **Topic** immediately (turn 3). A2 appears to develop the same **internal Knowledge Want** after looking at the screen, and they now have a **collective Knowledge Want** (turn 4).

Not all **collective Knowledge Wants** can be addressed by the partners themselves and they may ask a third developer for input. This happened in several of my sessions:

- In session CA1, the pair C1/C2 asks lead developer C3 for his opinion on whether they should start a new feature in the light of an upcoming release (see Example 9.3 where this is still C1’s **internal Knowledge Want**).
- In session DA2, the pair D3/D4 talks to two more senior developers to discuss design ideas (see also session description in Section 4.4.3).
- In session OA1 and OA2, the pair O3/O4 asks O6 for help (who knows more about the employed technology, but nothing about the system part the pair is supposed to write a test for, so he also cannot prevent the pair’s **Breakdown** discussed in Example 6.17).

I did not further analyze these temporary extensions of the pair status.

### 7.3 Topic and Target Content

The **Topic** of any particular knowledge transfer episode is what the developer who is pursuing her **Knowledge Want** intends to clarify. Usually, the **Topic** of a particular knowledge transfer is known to both pair members, e.g., because it is clarified along the way<sup>2</sup> or communicated explicitly in the beginning of the exchange. A developer asking a normal question often phrases the **Topic**: Here, the question is the **Topic**, though not necessarily in a verbatim sense, but what the developer intended to ask. Similarly, an explaining developer may lead in with a brief what-I-am-about-to-say and thus setting the **Topic**. From a researcher's perspective, it appears to always be possible to paraphrase the **Topic** of a knowledge transfer episode as a question; see Example 7.1 for both cases of J1 asking a question that can be paraphrased into his **Topic** and J2 announcing his explanations by stating his **Topic**.

Any knowledge transfer has the goal to acquire or exchange some information to fill a perceived knowledge gap. The very piece of information which fills the gap is the **Target Content**. For **internal Knowledge Wants**, i.e., a developer wanting to close a knowledge gap of her own, the **Target Content** is determined by what the developer wants to know as opposed to what the partner is capable of providing. For **external Knowledge Wants**, i.e., a developer wanting to close a knowledge gap that her partner supposedly has, the gap-perceiving developer may be mistaken if there is in fact no gap and consequently no **Target Content**, no information to close it.

#### 7.3.1 Types of Topics and Target Contents

Both **Topics** and **Target Contents** can be characterized by the type of knowledge they pertain to. There are two large areas of knowledge from which software developers acquire and exchange pieces during pair programming sessions. On the one hand, there is knowledge which is specific to the software system the software developers are operating in. I call this **S knowledge** (see Section 7.3.1a). On the other hand, there is more generic knowledge about software development in general, which I call **G knowledge** (see Section 7.3.1b). In my data, there were other types of **Topics** as well (see Sections 7.3.1c and 7.3.1d), but **S** and **G knowledge** were the most prevalent ones.

The following types of knowledge are not meant as a definitive taxonomy of knowledge which is relevant for software development, but is rather to be understood as a proof of existence of the form: *'Pair programmers do indeed transfer knowledge of that type'*. For brevity, I provide only one example for each type here. Further examples can be found in subsequent examples throughout this thesis.

##### 7.3.1 a) **S knowledge: System-Specific Knowledge**

**S knowledge** includes various types of knowledge which are specific for the software system the pair operates in: Its requirements, its architecture, design rationale, used technology, particular elements, known defects, and testing procedures.

---

<sup>2</sup>Note that these are two different meanings of the phrase "to clarify a **Topic**": On the one hand, pairs may need to clarify what they actually want to talk about, that is, to make the **Topic** itself clear. On the other hand, pairs will work on filling knowledge gaps and clearing up misunderstandings that relate to a **Topic**. In communication practice, however, these two aspects of 'problem specification' and 'implementation' are interwoven anyway (see Section 8.2.1c for details), which is why I think it is adequate to use the same phrase for both meanings.

**Requirements**

Desired properties of the system which the pair wants to (partially) fulfill or needs to consider in their session, also called “*desiderata*” (Ralph, 2013).

**Example 7.6: Uncovering Task Requirements** (DA2, 09:23–18:57)

In the beginning of session DA2, transferring the Target Content for the Topic ‘*What is our task?*’ takes almost ten minutes because the pair deals with several other Topics in between. The relevant pieces of their conversation (net time: 30 seconds) are:

D3: “In principle, there should be a toolbar up here < \*hovers blank area in GUI, closes application\* > I’ll show you how what it looked like in the old calendar. < \*changes code to load the old calendar view\* > [...]”

D4: “But where should it go? Here, or what? < \*points to screen\* >”

D3: “Actually, I was thinking, yes, it should go about here. < \*hovers narrow area above calendar view\* > [...]”

< \*puts back in the new calendar view, switches to other view with a similar toolbar\* >

“In general, a toolbar like this one. < \*hovers the other toolbar\* >”

D4: “But, shouldn’t it be the goal to always have these thingies up here? < \*points to buttons in the toolbar\* >”

D3: “Yes, exactly.”

D4: “But then < \*points to screen\* > for the calendar, it’s (!...!!)”

D3: “Yeah, that was nonsense what I told you before < \*switches to calendar view\* >”

D4: “Then it should be up there < \*points to screen\* >, right?”

D3: “We don’t put it here in this narrow bar < \*hovers narrow space in calendar view\* >, but up there, too < \*hovers space above calendar view\* >”

Note that the pair does not treat this matter as something they could decide on. Rather, first D3 and then D4 treat the state how the software “*should*”/“*should*” be as something that exists outside their PP session and which they need to figure out.

**Architecture**

Static and dynamic aspects of the system’s high-level design, e.g., which parts it is comprised of and how they interact.

**Example 7.7: Clearing Up Architectural Misconception** (CA2, 19:30–20:37)

Developers C5 and C2 discuss two approaches for implementing a new feature: C2 favors a simple solution which requires relevant classes and interfaces to reside in the same module; C5 already began with more indirect design which keeps all parts in their respective modules. Module pro can access module basis, but not the other way around.

In his explanation of the current state, C5 mixes up the class name and the interface name, but immediately corrects himself so the architectural Target Content is eventually transferred to C2.

C5: “I did it this way, because I wanted to move as little as possible to basis. And the interface which I moved only knows things that are available in basis.”

C2: “And the VirtualAttribute, where is it?”

C5: “That one I moved.”

C2: “Ah, so we’re clear.”

C5: “Not the Virtual! The IVirtualColumn < \*points to screen\* > I moved. The Virtual Attribute is here in pro. < \*looks at C2\* >”

C2: “In pro is the VirtualAttribute? OK. < \*opens file VirtualAttribute\* >”

C5: “Yes.”

C2: “Yes, ok, yes, right. I left it there because we didn’t need it anywhere else yet.”

**Example 7.7 (continued)**

Prior to the above excerpt, C5 tried to explain and defend his design decisions to C2 whose reluctance to accept C5's ideas can be explained by being mistaken about the package location of `VirtualAttribute`. It took C5 about 10 minutes of frustrating and unfruitful design discussions until he made the critical remark which triggered C2 to become aware of his knowledge gap, to develop an **internal Knowledge Want**, and to formulate the architectural Topic "*Where is the `VirtualAttribute`?*" which, until it was resolved, impeded the pair's session. Later, in Chapter 11, I introduce the concept of **Knowledge Needs** to also describe such latent knowledge gaps.

**Design Rationale**

Explanations for why the system is designed a certain way.

**Example 7.8: Rationale for Data Type (KB1, 15:54–16:27)**

K3 proposes to store the price of a subscription as a numeric attribute. K2 explains that an indirect way with an enumeration type and a lookup table is more desirable because it allows for bulk changes that the business department ("*they*") needs for experimentation. (Although K2's explanation appears incoherent, K3 understands him.)

K3: "What about simply putting the price in there?"

K2: "Well, the idea was (!...!) with the price, they [= business department] want to experiment a lot. That's why there are two configuration variables for now, one for LOW and for HIGH, in the price. Depending on that, what you have in the Enum here, it's fetched accordingly. (...) That's why we didn't want to put the price directly in there."

K3: "OK. Fine. You could also put the price in there. As a number. And then you'd have as many variants as you like."

K2: "Hm."

K3's last remark is not an actual proposal on how they should design their software, but more of a general remark on the limitation of the business decision to only have two tiers.

**Technology**

Which external software products are used in the technology stack.

**Example 7.9: Inquiring about GUI Technology Stack (DA2, 01:54–02:46)**

D4 is new at the company and wants to know about the GUI technology stack of their main software product.

D4: "How is this implemented in general, because (!...!) is it more of an SWT user interface than Eclipse, or what?"

D3: "Well, <\*> product name\*> as such is based on SWT."

D4: "Yes"

D3: "The calendar, there we use this (~) calendar component."

D4: "That's Swing?"

D3: "Nope, yes, so actually AWT."

D4: "Ah, ok. AWT even."

D3: "Yeah, there is this SWT-to-AWT container gizmo and that's how it's embedded in the end. How this SWT-to-AWT thingy works I can't tell you much about."

D4: "OK. But can you embed an SWT context menu or something in there?"

D3: "Can't tell you for sure."

**Example 7.9 (continued)**

AWT, Swing, and SWT are different GUI toolkits for the Java programming language. The Eclipse IDE uses SWT and makes heavy use of XML-configuration files. D4 probably asked about the difference between hand-crafted SWT GUI elements vs. reused XML-configured Eclipse elements. The **Topic** could be phrased as ‘*What is the technological basis for the component?*’ to which the **Target Content** would include ‘*AWT inside SWT*’. D3 could not answer D4’s follow-up question, so the full **Target Content** is not possessed by any of the pair members.

**Source Code**

Particular source code elements such as classes, most fundamentally their mere existence (in an explanation such as ‘*There is a class X which does Y.*’) but also more detailed information about their inner workings.

**Example 7.10: Getting to Know Relevant Classes (KA1, 51:17–53:37)**

K1 already knows the particular API which K2 is calling and the type of data it provides, but not yet the classes which are involved in processing that data, so K2 explains them to K1.

K2: “I can you show what we have so far. Then we can compare with the financial exposé.”

K1 asks to finish some task on his phone first, which takes about 90 seconds.

K2: “So, we have `ExposeApiClient` <\*opens that class\*>. It has a `getAsJson` where we can get the exposé JSON, which comes from the API. It ain’t pretty, since you get hundred things you don’t need, all nested and stuff.”

K1: “Right, and there are a thousand different cases and the documentation sucks, I noticed. It’s only documented for one property type, but there are about twenty.”

K2: “Exactly, right. Yes, right, there are twenty, yes. Anyway, then we have the `ExposeApiService` which uses a thing <\*opens that class\*>, I mean the `ExposeApiClient`, gets the JSON and puts it in our Mapper. And the Mapper then maps it [...] on the objects we need. <\*selects a number of data classes in overview\*>.”

The **Topic** here is ‘*How do we retrieve data from the API?*’ to which the **Target Content** is something like ‘*`ExposeApiClient.getAsJson()` gets all data, `ExposeApiService` then simplifies it*’.

**Defects**

Possible or actual known defects in the system which may pose a problem for the pair to look out for or to work around.

**Example 7.11: Open Bugs? (KA1, 59:23–1:00:01)**

The pair is designing a facade to simplify different data formats used by external systems. K1 already wrote a mock JSON file which represents the structure of the data he would like to receive for his application. K2 goes through the entries in the mock file and makes an assessment for each to determine whether the data K1 wants is already available and, if not, how easy it can be acquired.

K1: “Then I need all of these, some things, the `floorSpace` and so on and how many rooms the property has, and then all these tax values.”

K2: “We have that <\*selects next JSON line\*> right. We only need to check <\*selects next JSON line\*> the question is, where they all come from or how we find them and whether there are differences between the types. I recall a meeting where they said there was a problem in one of them. I don’t know whether they fixed it already. They did say they have no open bugs.”

**Example 7.11 (continued)**

As the pair does not further this strand in their conversation, the **Topic** remains ambiguous for an outsider. One way to phrase the **Topic** would be ‘Do we have access to all necessary data?’ (with the **Target Content**: ‘Mostly, yes.’), another way would be ‘Is there an open problem?’ with unknown **Target Content**.

**Routines**

Procedures for running and testing the system.

**Example 7.12: How to Start a Manual Test (CA2, 44:08–45:52)**

Developers **C2** and **C5** decided to perform a manual GUI test before they commit their changes. In their system, there are so-called “demos” which are fast-loading environments for testing individual GUI panels without the complete application underneath. **C2** is not sure how to start the demo and **C5** helps out.

**C2**: “Then we try out the GUI [...] where was it? <\*scrolls through source tree looking for a test class\*> Where were we?”

**C5**: “You have to, this demo down here <\*points to screen\*>”

**C2**: “This one? <\*expands one of many demo packages\*>”

**C5**: “It should be in there.”

They find and start a demo, which appears to work. After a minute **C5** still has reservations and wants to see the whole application in action. **C2** is not sure how to start the application and which of the over 40 run configurations to choose. **C5** helps out again.

**C5**: “OK, but what we did more affects the Action than the GUI. I mean, I’d like to see it from the application.”

**C2**: “Sure. <\*hovers IDE’s “Run” menu\*> Oh god, what’s the deal again? <\*hovers IDE’s “Run” button\*> Here I have to do (~)?”

**C5**: “Yes, and then <\*\*\*name of run configuration\*\*>.”

The **Topics** for these examples would be ‘How do I start the demo for feature X/the whole application?’ with the demo’s and the run configuration’s names and locations in the IDE as the respective **Target Contents**.

**State and Configuration**

Current state of a machine or a system, e.g., which packages are installed, the extent of network connectivity, and others. This is especially relevant for debugging, it is short-lived knowledge for local problem solving. If the developers are not in the midst of dealing with such a problem, it is not relevant. When it gets relevant, it usually needs to be found out and is not something readily known.

**Example 7.13: Understanding Failed Network Calls (BA1, 01:33–04:00)**

Developers **B1** and **B2** adapted a test script to run it against their local setup but it does not execute cleanly.

**B1**: “(#Couldn’t connect. cat: header.txt#) cURL still can’t connect. <\*switches to IDE and back to shell, executes just the curl command instead of whole script\*> [...]”

**B2**: “But cURL is on here, right? Sure, we wouldn’t see the error otherwise.”

**B1**: “(#Couldn’t connect#)”

**B2**: “Try wget, to simply fetch it.”

**B1**: “<\*tries wget on dev-intern, which fails\*> It failed.”

## Example 7.13 (continued)

B2: “Try localhost instead of dev-intern”

B1: “< \*tries wget on localhost, which fails\* > Hm, what’s going on here?”

B2: “Firewall?”

B1: “(#Resolving hostname ‘localhost’#)”

B2: “Possibly (~)? Nope, he did resolve it.”

B1 then proposes to not use the test script but to call the functionality in the web browser, which works and allows them to carry on with their actual task.

The **Topic** is: ‘What is the state of the current machine leading to cURL not running cleanly?’ The **Target Content** remains unknown, but some pieces are gathered: The cURL command is installed, host name resolution works, and the problem is not limited to the cURL command.

7.3.1 b) **G knowledge: Generic Software Development Knowledge**

**G knowledge** is system-independent. It is knowledge about programming languages, reusable frameworks and technology stacks, design principles, testing and debugging methods, methods for program understanding, and tool usage.

Unlike for most **S knowledge**, exchanges pertaining to **G knowledge** can be verified by an outsider more often, because there are resources which are external to the session recordings (such as books or other documentation) which are part of the presumably shared reality of the pair programmers and the researcher who has access to these materials. In Example 7.14, for instance, D4’s explanation of a well-known design pattern is far from perfect and, in a pedantic sense, even wrong. For understanding the concrete pair programming situation from a researcher perspective however, such a deviation from the ‘objective truth’ does not matter: D3 appears to understand everything that D4 said, and the knowledge transfer ends successfully.

As for **S knowledge** in the previous section, I provide one example per specific type of **G knowledge** that pair programmers actually dealt with in my data.

**Design and Programming Patterns**

Pair programmers talk about the classic Gang-of-Four design patterns, but also about other idioms and best practices beyond from their current code.

**Example 7.14: Template Method Design Pattern (DA2, 1:36:35–1:37:58)**

The pair used the Template Method design pattern for one hour in a dozen instances (see Example 6.22) before D4 mentions the pattern name for the first time and explains it to D3.

D4: “You know that pattern? [...] It’s a kind of Template Method. So, here I extracted the shared logic. < \*navigates to abstract class, selects call of abstract method\* >”

D3: “M-hm.”

D4: “[...] internalExecute is basically my Template Method. [...] And you can neatly extract those parts that are general, and for all things you don’t know yet, you make an abstract method which your superclasses then implement.”

D3: “M-hm.”

D4: “It’s pretty neat, because you don’t have to copy-paste.”

D3: “Right.”

Note that D4 probably meant to say ‘subclasses’ instead of “superclasses” in his explanation. Also, technically speaking, “the Template Method” is the algorithm implementation in the abstract

**Example 7.14 (continued)**

superclass which defines the general structure and calls one or more abstract methods or “*primitive operations*” such as `internalExecute` (Gamma et al., 1995, pp. 326–327).

Either way, ‘*How does the design pattern Template Method work?*’ is the **Topic** of the above exchange; D4’s explanations along with the highlights he sets in the source code point to the **Target Content**. Even though D4’s explanations alone were rather suboptimal, D3 appears to understand the pattern probably because he can relate the explanations to his own development experience.

**Programming Languages**

Details of the used programming language

**Example 7.15: Inner Classes in Java (DA2, 1:29:55–1:30:25)**

In the Java programming language, the keyword “`this`” refers to the ‘current’ object. Java allows to define classes within classes. A non-`static` inner class is bound to an instance of the outer class. To access the instance of the surrounding outer class from within an inner class, the class name needs to be prefixed, e.g., “`OuterClass.this`”. D4 knows about the idiom, but is not aware that he is currently inside such an inner class (in this case: an anonymous inner class that implements `SelectionAdapter`), and is puzzled by the prefix in `AbstractList.this`.

D4: “Why is it `AbstractList` before? `<*` deletes the prefix, the IDE marks a compilation error`*`” Nope `<*` undoes the deletion`*`” ah, I see. Since we are in an anonymous class.”

D3: “In an anonymous class? What’s an anonymous class?”

D4: “`<*` selects the new statement which marks the beginning of the anonymous class`*`” When you, for example implement such a `SelectionAdapter`, then it’s an anonymous class `<*` selects the body of the anonymous class`*`” since the class has no name.”

D3: “Ah, yes, right..”

D4: “That’s what puzzled me, I didn’t see that. And when I say ‘`this`’, then it’s the `SelectionAdapter`, or its implementation.”

D3’s question sets the **Topic** of their exchange; transferring the **Target Content** is easy because D3 already knew the idiom, but not the technical term “anonymous class”.

**Development Tools**

Tools to support the development process, such as version control systems.

**Example 7.16: Deleting Folders Under Version Control (CA2, 26:11–27:01)**

C2 wants to know whether SVN—the version control system used for their current source code tree—behaves like the older CVS which is file-based<sup>a</sup> and makes no distinction between empty and non-existing directories, e.g., after deletion. SVN does in fact support versioning of directories,<sup>b</sup> so the pair does not actually have a problem here, but they do not reach that conclusion explicitly.

C2: “How is it with SVN and deleting directories, does it work? (Should work.)”

C5: “You should be able to delete a package, right?”

C2: “Well, in CVS it doesn’t work.”

C5: `<*` looks at C2, puzzled`*`”

C2: “In CVS it doesn’t work.”

C5: “To delete a package?”

C2: “Yes, not possible in CVS. The directory always stays.”

C5: “It stays in the CVS (!!...!!)”

**Example 7.16 (continued)**

C2: “And there is no difference in deleting it or not, in CVS. <grinning> [...] It doesn’t show up when it’s empty”

C5: < \*turns to screen again\* >

C2: “regardless of whether you deleted it or not. [...] How it’s done in SVN, I don’t know. Might be totally different there.”

C5: “M-hm, we will see.”

C2: “I’d be curious.”

Again, the opening question hints at the **Topic** of the exchange; it could be phrased like ‘*How do version control systems deal with deleting directories?*’ for which C2 offers a partial **Target Content**, clarifying the **Topic** for CVS but not for SVN, leaving his **Knowledge Want** somewhat unsatisfied. C5, however, does not seem to share this particular **Knowledge Want** since he already turns back to the task while C2 is still explaining the CVS case.

<sup>a</sup>CVS project homepage: <http://www.nongnu.org/cvs>

<sup>b</sup>SVN feature overview: <https://subversion.apache.org/features.html>

**Technology**

Frameworks, libraries, and other reusable technology that are possibly used to build a software upon.

**Example 7.17: OSGi Class Loading (DA2, 1:30:49–1:35:15)**

The software system consists of over 50 OSGi bundles. A first attempt to introduce a new dependency could not be achieved by a simple Java `import` statement, but a second approach succeeded. D4 expects his partner D3 to not know why the first attempt failed and proceeds to explain a number of concepts and best practices of the OSGi technology for about five minutes. He provides some examples using the current source code, but much of his explanation (shown below) is independent from their current software system and thus constitutes **G knowledge**.

D4: “Do you know about OSGi class loading?”

D3: “Class-what? Not really, no.”

D4: “Should I tell you?”

D3: “Sure.”

D4: “[...] Each bundle [...] has its own `ClassLoader` which can only load classes from other bundles if [...] the other bundle from where you want to get the class does export the package of the class and if your own bundle explicitly imports.”

“You always have these `manifest` files [...] there you can do `Import-Package` [...] to say that you want that package.”

D3: “M-hm.”

D4: “[...] normally, you should always do `Import-Package`, <laughs> here it’s always `Require-Bundle`. You set the logical name of the bundle you want to import and depend explicitly on this bundle. You can not say, I remove this bundle and swap it with another with the same package for OSGi to use that one. [...] That’s why `Import-Package` is usually nicer. [...]”

The **Topic** may be framed as ‘*How can OSGi class loaders access classes from other bundles?*’.

### 7.3.1 c) Other Types of Knowledge

In addition to *S* and *G* knowledge, pair programmers also ask for and transfer other types of knowledge, which are not specific to the current software system as such and not universal enough to count as general software development knowledge. In my data, however, such instances were mostly limited to session DA2 featuring developer D4 in his first week at the company. Some of these knowledge types can also be expected to be transferred between more ‘senior’ developers, while others appear more typical for a first contact with a project. In particular, there were the following *Topics* and *Target Contents*:

- Information about a **pair or team member’s development background**, which is possibly relevant for assessing what to expect from one another or who to ask for additional information (e.g., for establishing a *transactive memory system*, see page 98).

#### Example 7.18: Talking About Developer Backgrounds (DA2)

The pair talks about their experiences and refers to other team members multiple times.

- D4: “How long have you been here? Three months?” (04:18)
- D3: “I’ve been programming some years before that. Started with HTML, PHP, and then over to Delphi.” (04:26)
- D3: “In the beginning, I had no clue about Java and had to start somehow.” (17:42)
- D3: “Did you ever do such a thing, such a toolbar?” (19:27)
- ↳ D4: “I just read this book here. I mean, these are at least the standards steps.”
- D4: “Who did this? I mean, the todo list in general?” (21:23)

- Information about the **company** the pair works in, its departments, technical equipment, and company culture, which is possibly relevant for a new developer to *naturalize* in a software project (see page 37).

#### Example 7.19: Talking About the Company (DA2)

The pair talks about different parts of work life in the company.

- D3: “In two months time, I’m back in VB [department], then I go to PS [...] ‘Professional Services’, they also do some software development.” (05:05)
- D3: “You have to know, the SVN repository is on <\*\*”current machine name\*\*”>. <laughs> It’s just moving files around!” (05:40)
- D3: “We don’t even have a fiber connection to the Internet. It’s just three DSL lines.” (06:20)
- D3: “Actually, <\*\*”current machine name\*\*”> is not that bad. What does it have? Two cores? [...]” (06:50)
- ↳ D4: “Yeah, (#2.4#) [GHz], two times.”
- D4: “That’s not my usual approach. Normally, I always write a test first.” (1:14:48)
- ↳ D3: “We don’t do that here. Although we’re supposed to. Anyway.”

### 7.3.1 d) Application Domain-Specific Knowledge?

Another major type of knowledge I expected to find in addition to *S* and *G* knowledge would have been application domain-specific knowledge. In my data, however, developers did not speak about their domain independent of their system. An explanation for this could be that both pair members were equally and sufficiently familiar with the domain as necessary for

the task that there were no knowledge gaps that could result in a **Knowledge Want** that could have led to observable knowledge transfer activities. Pair programmers *do* talk about aspects of their application domains, but these are often not their actual **Topics**:

**Example 7.20: Domain Knowledge as Background Information** (JA1, 05:03–05:19)

The application domain is radio news broadcasting. J2 explains a number of properties of hourly news segments: They do not have a precise starting time, but waiting two minutes is enough; and they also vary in length, but are not longer than seven minutes. Or in J2's own words (line numbers from Example 5.1):

(12) J2: "I start looking two minutes after the full hour, because then it's guaranteed that news files exist if any exist."

(14) J2: "And then monitor this file as long as needed until it's ready. That can take up to seven minutes, depending on the wave."

Note that the application domain was not the actual **Topic** here, but was merely mentioned as additional information to the **Target Content**: There is a polling mechanism which starts at some point in time and then goes on as long as necessary.

The next example is a rare case in my data of pair members actually talking about domain concepts, if only briefly. A few minutes later (Example 7.22), the same pair has again an opportunity to talk about a domain concept explicitly, but they treat it merely as an identifier and do not discuss its meaning.

**Example 7.21: Application Domain Knowledge Explained** (KA1, 54:00–54:29)

K1 and K2 are outlining an API to be built between their respective subsystems, and K2 explains the available data classes to his colleague. The application domain is real estate. The example below is a rare case of a pair member actually explaining a domain concept (more precisely, the *difference* between two domain concepts) to the partner. (The spoken language was German: words in monospaced font are identifiers from the source code; everything else was said in German.)

K2: "In the RealEstate we only have the data we need <\*opens class RealEstate\*>. That's title, an address, the price, livingSpace, plotArea (!!...!!)"

K1: "What is plotArea?"

K2: "Erm, siteArea, so property price (!!OK!!) no, not 'property price': property area. (!!OK!!) livingArea then is floor space."

K1: "Yes"

K2: "And then we have the marketValue, also some kind of price. And then we have constructionYear and modernizationYear."

K2's system talks to an external API. What is called plotArea in the foreign data model is called siteArea in their own model (as can be seen 90 seconds later in the source code)—this is probably why K2 starts his explanation the way he does.

**Example 7.22: Domain Concept as Identifier** (KA1, 1:00:05–1:01:24)

Same session as Example 7.21, a few minutes later. The pair opened the data model of the foreign service from where K2 already receives data which should be eventually passed down to K1's system. Another example of how domain concepts have an echo in the source code, but are merely treated as identifiers.

K2: "Here we have such a HoLder <\*open class ExposeHoLder\*> there is the Data element already. [...] Should all be in here. courtage, you need that?"

K1: "What is 'courtage'?"

## Example 7.22 (continued)

K2: “No idea, <\*opens online dictionary\*> it should be <\*enters ‘courtage’, result appear, K2 reads them\*> (#broker’s commission, broker’s fee#)”

K1: “Yes, (#broker’s commission#). Yes, that’d be nice. [...] Yeah, ‘broker’s commission’, that’s how it’s displayed later (!!M-hm!!) so that’s alright, I need that, too.”

The pair does not talk about the actual meaning of a broker’s commission, and it does not become clear whether K1 understands more than the syntactical level.

### 7.3.2 Hypothetical Target Contents

In some cases, the pair members deal with a **Hypothetical Target Content**, which, too, is a piece of information which could fill a knowledge gap and satisfy a **Knowledge Want** if it turns out to be correct. Recurring to the jigsaw metaphor, a **Hypothetical Target Content** is a puzzle piece which has the right shape, but perhaps the wrong picture part. In my data, this occurred in the following situations:

- A developer with an **internal Knowledge Want** can explicitly put forth a **Hypothetical Target Content** with the expectation that her partner is able to validate it. This is what J1 did in the beginning of session JA1.

**Example 7.23: Hypothesis to Satisfy Internal Knowledge Want** (JA1, 06:00–06:12)

Excerpt from Example 5.1, where J1 lets his partner J2 (in)validate a **Hypothetical Target Content** (“10 seconds”) after J2 did not understand J1’s **Topic** for one minute.

(21) J1: “[...] I mean (!...!) You wait for 10 seconds, then after 10 seconds you decide: In those 10 seconds nothing has changed, so the file appears to be ready.”

(22) J2: “Ahhh, that’s what you mean. No, 30 seconds.”

(23) J1: “30 seconds, that’s what I wanted.”

This can effectively allow to make clear what the **Topic** is meant to be. I call such knowledge transfer activities **Proposition** (see Section 8.2.6 for details and more examples).

- A **collective Knowledge Want** can be addressed in a rather pre-structured way by formulating a hypothesis that the pair together tests. Recall Example 7.5, where A1 hypothesized how the source code might look like (“*How is it currently done in there, the strike through? Is it a span tag or something? Maybe it’s already the case.*”), so all the pair had to do was inspect the source code.

In the first case, the **Hypothetical Target Content** is contained in an *ask\_knowledge* which includes a possible answer (BL, Sec. 16.2.14); in the second case, it is part of a *propose\_hypothesis* of type ‘can-check’ (BL, Sec. 13.1.3). The relevant difference between the base concepts lies in whether the speaker expects the partner to be able to provide an answer (BL, Sec. 16.3.6);

## 7.4 Summary and Discussion of Related Work

In this chapter, I introduced three *knowledge* concepts: (a) The **Knowledge Want** that motivates developers to engage in knowledge transfer activities (which may or may not correspond to an actual gap in knowledge); (b) the conversational **Topic** that both partners need to understand to some degree, and (c) the informational **Target Content** that can be acquired, possessed, and transferred. In discourse analysis (see, e.g., Baker & Ellece, 2011, pp. 122, 151), the “*theme*” of a conversation (the **Topic**) is distinguished from what is said about the topic, the “*rheme*” (transferred parts of the **Target Content**). I am not aware of any other comparable distinctions.

Additionally, I distinguish different types of knowledge by categorizing different types of **Topics** and **Target Contents**. The two most common were system-specific **S knowledge** and more generic **G knowledge**. A number of studies I discussed as related work in Chapter 2 make similar distinctions, some of them were a priori taxonomies, others were empirically derived:

- Jones & Fleming (2013, discussed on page 83) categorized 43 episodes of student pair members teaching each other by knowledge type. They distinguish four types (which occurred between 6 and 16 times each) which they group into two categories: Development tools and programming language are summarized as “*general development knowledge*” (both categories are similar to my **G knowledge** types with the same names); knowing how to reproduce the bug and code structure knowledge are labeled “*project-specific knowledge*” (similar to my **S knowledge** types of *Routines/Defects* and *Source Code*, respectively). They have no corresponding categories for the other types I found (i.e., **S knowledge**: *Requirements, Architecture, Design Rationale, Technology, State and Configuration*; **G knowledge**: *Design and Programming Patterns, Technology*). This is probably because the bug-fix task the students worked on for 110 minutes did not offer many opportunities in this regard.
- Sillito et al. (2008, see my discussion on pages 38 and 82) investigated what developers want to know about their system during a change task—**S knowledge** in my terminology. Each of their 44 identified question types (*ibid.*, Table 4) can be understood as an **S Topic** template. To give an example, a specific question/Topic such as “*how does [MAssociation] relate to [FigAssociation]?*” is an instance of question type #22 “*how are theses types or objects related?*” (*ibid.*, Sec. 4).

Most of their question types relate to my **S knowledge** types *Source Code* (e.g., types #1–#20) and *Architecture* (e.g., #21–#25). Others relate to *State and Configuration* (e.g., #27, #30, and #31) and *Requirements* (e.g., #39). There appear no question types for the **S knowledge** types *Design Rationale, Technology, Defects, and Routines*, nor any type for **G knowledge** at all.

- Hulkko & Abrahamsson (2005) and Vanhanen & Korpi (2007) do not explicitly discuss different knowledge types, but based on their group interviews on how PP was employed in the teams and observations of actual PP sessions (discussed on page 71), they emphasize the significance of **S knowledge**, while other types of knowledge, such as **G knowledge**, are not mentioned: “[K]nowing what the code did” was necessary to be a useful pair member (Vanhanen & Korpi, 2007, Sec. 5.3); pairing was useful “to get a clear understanding on the system” and beneficial for “code which had many dependencies with other parts of the software” (Hulkko & Abrahamsson, 2005, Sec. 3.2.1).

Other types of knowledge, in particular regarding the *application domain*, did not play a role in the sessions I analyzed. This might be due to the circumstance that all companies A to P develop their own software product. In consulting companies, this might be different, as the practitioner report by Rasmusson (2003) on pair programming in a ThoughtWorks project suggests: “it [...] spread domain knowledge throughout the team”. I come back to this issue in Chapter 13.

The above concepts are useful to characterize concrete PP situations. I use them as follows:

- Chapter 8 covers individual knowledge transfer activities which I distinguish into **Explanations** (which deal with **external Knowledge Wants**) and **Explanation Elicitors** (dealing with **internal Knowledge Wants**).
- Chapter 9 is about **Episodes** of knowledge transfer during which the developers deal with individual **Topics**.
- Chapter 11 explores the influence of lacking **S knowledge** and **G knowledge** on the overall dynamics of PP sessions.



# Chapter 8 Knowledge Transfer Activities: Asking and Explaining

---

8.1	Purpose and Structure of this Chapter . . . . .	257
8.1.1	Discussion of Recurring Example . . . . .	257
8.1.2	Overview of Knowledge Transfer Activities . . . . .	259
8.2	Asking Questions with <b>Explanation Elicitors</b> . . . . .	260
8.2.1	General Properties of <b>Explanation Elicitors</b> . . . . .	261
	<i>“Trigger” or “Elicit”? • Role in the Conversation • Precedence of Elicitors: Clarification Cascade</i>	
8.2.2	Improper Asking . . . . .	264
8.2.3	Direct Asking . . . . .	265
8.2.4	Refer to Common Ground . . . . .	266
8.2.5	Entice to Simple Step. . . . .	268
8.2.6	Make Proposition. . . . .	270
	<i>Optimistic Propositions • Pessimistic Propositions</i>	
8.3	Providing <b>Explanations</b> . . . . .	272
8.3.1	Present New Fact . . . . .	272
	<i>Present New Fact to Fill Gap • Present New Fact to Correct False Understanding</i>	
8.3.2	Refer to Common Ground . . . . .	274
8.3.3	Entice to Simple Step. . . . .	275
	<i>Enticing to Simple Step as Reaction to an Elicitor • Entice Simple Step as Argument in Decision Making</i>	
8.4	Summary . . . . .	277

## 8.1 Purpose and Structure of this Chapter

In the previous chapter, I discussed that pair programmers perceive knowledge gaps either in themselves or in their partner and develop **Knowledge Wants** to do something about it. The activities which then constitute the actual knowledge transfer mainly fall in two categories: Making clear the **Topic** to the partner through different forms of *asking* and transferring the **Target Content** through different ways of *explaining*.

This chapter introduces many concepts for both the *asking* side and the *explaining side*. I will start with a refresher on the recurring example, which happens to cover most of them.

### 8.1.1 Discussion of Recurring Example

The conversation in the first minutes of session JA1 illustrates the variety of ways how pair programmers *ask questions* and *provide explanations*. I annotate and explain my concepts in the example below. To make the concepts and their annotation easier to understand, I also comment on the pair members' (presumed) expectations for all their activities.

**Example 8.1: Knowledge Transfer Activities** (JA1, 04:15–06:15)

J2 begins his explanations by **Presenting New Facts**, which contain pieces of information he expects his partner J1 to not know yet. (J1's back-channel responses are not shown here, see Example 5.1 on page 186 for the full transcript and background information.)

- (6) “In the end, the news recording of every hour pops out.”
- (8) “[...] there is the central plugin and multiple processors which each handle one wave.”
- (10) “For most of them, right after the full hour there is a check, if there is a new file on the remote share.”
- (11) “If so, the most recent file is selected and it starts checking how the file changes size-wise.
- (13) “[...] it looks until the file does not get bigger anymore, then it is apparently ready.”
- (15) “And then it is fetched and handed over to transcoding.”

After these explanations, J1 develops an **internal Knowledge Want** and a dialog with interwoven explanations and questions unfolds.

- (16) J1: “In what time window are you looking?” J1 starts with a plain question. He expects J2 to understand the question and thus the **Topic**, and then to deliver the **Target Content**.  
**Direct Asking**
- (17) J2: “I start looking two minutes after the full hour, because then it's guaranteed that news files exist if any exist.” J2 provides another piece of information he expects J1 to not know yet. J2 thinks he was asked about the time window in which the software checks for changes and provides information on its start ...  
**Present New Fact**
- (19) J2: “And then monitor this file as long as needed until it's ready.” ...and its end: First qualitatively as a reference to what he said earlier († 13), which he expects J1 to pick up on, ...  
**Refer to Common Ground**
- J2: “That can take up to seven minutes, depending on the wave.” ...and then quantitatively with a concrete number, which he expects to be new information again.  
**Present New Fact**
- (20) J1: “Hm ya, but mh, the time window for the change?” J1's sees his **Topic** misunderstood. He rephrases his question with a different emphasis, expecting that this makes J2 understand the **Topic**.  
**Direct Asking**
- (21) J2: “Yes, right, that is, er, time window for the change is variable, depends on how the news go.” J2 considers the answer to be common ground, and repeats what he already said († 19).  
**Refer to Common Ground**
- (22) J2: “I can't know that, right?” J2 expects J1 to understand that there is no way to know in advance for how long a news segment will run.  
**Entice to Simple Step**
- J2: “You know, they always start a new file. When the news are over, again a file is created. I mean, I basically never have more than the news.” J2 adds more background information which he expects J1 to not know yet: He *has* to poll because there is no event mechanism.  
**Present New Fact**
- (23) J1: “Yes, no, I mean 'cause you said you look for so long until the size stops changing, right?” Since J2 still did not understand the **Topic**, J1 refers to their common ground, too, by repeating pieces of J2's explanation († 13).  
**Refer to Common Ground**
- (24) J2: “M-hm” J2 acknowledges that reference.

## Example 8.1 (continued)

- |  |                       |  |
|--|-----------------------|--|
| (25) J1: “Then you need to plan for a time window in which a change <u>could</u> happen, right?”   | Entice to Simple Step | Now J1 expects J2 to come to a certain conclusion: J2 should understand that a polling interval cannot be arbitrarily small.   |
| (26) J2: “Yeah, well, until up to five before the hour. I really take my time.”  | Present New Fact      | J2 does not come to the intended conclusion, but instead now thinks that he was asked about the end of the time span and provides information on when the system will stop looking for changes.  |
| (27) J1: “<laughs> No, I really mean the size now, the size of the time window, I mean (!...!) You wait for 10 seconds, then after 10 seconds you decide: In those 10 seconds nothing has changed, so the file appears to be ready.” | Make Proposition      | Although the last piece of information was indeed new to J1, it is not part of the Target Content. J1 understands that his partner still does not understand his Topic, and he no longer directly tries to make J2 provide the Target Content, but instead formulates a Hypothetical Target Content (HTC) which he expects J2 to (in)validate. |
| (28) J2: “Ahhh, that’s what you mean. No, 30 seconds.”   | Present New Fact      | Posed with a simple yes/no-question, J2 now understands the Topic, he invalidates J1’s HTC, and provides the actual size of the polling interval.  |

## 8.1.2 Overview of Knowledge Transfer Activities

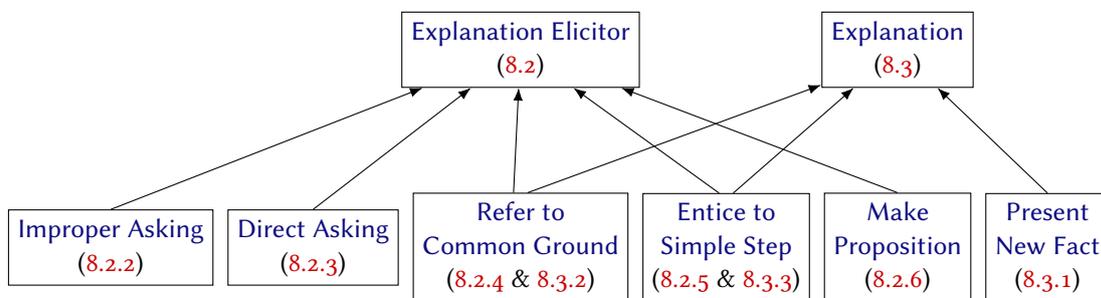
The excerpt above illustrates five of the overall six types of knowledge transfer activities which pair programmers choose depending on their own current understanding and their expectation of their partner’s:

- **Improper Asking** (*not part of the example above*): The speaker is puzzled by something and signals her confusion along with her readiness to receive information. She might say something like ‘Erm?’, which is not a proper question. Nevertheless, her partner may be able to infer to the Topic, though the speaker does not expect that she does.
- **Direct Asking**: The speaker has understood enough to formulate an actual question, such as J1: “*In what time window are you looking?*”. The speaker expects their partner to possess the relevant knowledge, to understand the question and thus the Topic, and then to deliver the Target Content.
- **Refer to Common Ground**: The speaker says something she expects her partner to already know and to recognize as a reference (e.g., by repeating something that was said earlier). J2, for example, repeats his own phrase ‘*looking/monitoring until the file is ready*’ (turns 13 and 19); J1 also repeats J2’s words verbatim: “*you said you look for so long until the size stops changing, right?*” (turn 23). If successful, it makes sure that both pair members zone in on a smaller part of their shared reality.
- **Entice to Simple Step**: The speaker says something that has an implicit conclusion and she expects her partner to take a simple mental step to reach that conclusion. In the example, J2 wanted J1 to understand that the runtime of a news segment is variable and not known in advance by saying “*I can’t know that, right?*”. J1 was similarly implicit in nudging J2 towards understanding that polling intervals cannot be arbitrarily small (“*Then you need to plan for a time window in which a change could happen, right?*”). Metaphorically speaking, while Referring to Common Ground goes back to square one, a Enticing to a Simple Step nudges the partner into a new direction.
- **Make Proposition**: The speaker formulates something with the expectation that her partner can validate or invalidate it. At first, this makes it unnecessary for the partner

to understand the speaker’s state of mind, as she can focus entirely on a simple *yes/no* question (see J1: “*I mean (!...!) You wait for 10 seconds, then after 10 seconds you decide: In those 10 seconds nothing has changed, so the file appears to be ready*”). If this is successful, i.e., if the partner understands the proposition and can validate it, the partner may also understand the speaker’s state of mind (J2: “*Ahhh, that’s what you mean.*”).

- **Present New Fact:** The speaker explains something as a fact (this contrasts with a **Proposition**, which the speaker is decidedly unsure of). She expects her partner to not know about it yet (unlike **Referring to Common Ground**), and the new information is to be understood as-is (unlike when **Enticing to a Simple Step**).

These six types of activities can be grouped into two categories: **Explanation Elicitors** are used to make the partner provide information to satisfy one’s **internal Knowledge Want**, while developers provide **Explanations** to deal with their **external Knowledge Want** which follows as a reaction to their partner’s **Explanation Elicitors** or which even goes beyond what was explicitly requested (see also Section 7.2.3). I discuss **Elicitors** in Section 8.2 and **Explanations** in Section 8.3.



**Figure 8.1:** Knowledge transfer activities overview. Pair programmers **Refer to Common Ground** and **Entice to Simple Step** in the context of both **Eliciting** and providing **Explanations**.

As is depicted in Figure 8.1, pair programmers **Refer to Common Ground** and **Entice to Simple Steps** in both contexts. This is not surprising as both activities are meant to make the partner understand something: From the perspective of an **external Knowledge Want**, the partner should understand some new fact by seeing its connection to existing knowledge. From the perspective of an **internal Knowledge Want**, the partner should understand one’s own **Knowledge Want** by repeating the same mental steps to reach the “edge” of the knowledge gap. In fact, the understanding of a knowledge gap is possessing *meta-knowledge*, which is also knowledge. One way to *ask a question* is to *explain one’s knowledge gap*.

## 8.2 Asking Questions with **Explanation Elicitors**

I first discuss general properties of **Explanation Elicitors** (Section 8.2.1) and then go through the characteristics of all five types (Sections 8.2.2 to 8.2.6). I discuss the contexts in which they occur and what pair programmers want to achieve with them. Some **Elicitors** have subtypes. There are probably more subtypes which I have seen in my data but not analyzed, but not likely any further **Elicitors**. I did not further my analysis beyond the level of detail presented here as it was enough to characterize the different ways how developers deal with their **internal Knowledge Wants** on an activity level.

Concept	Description/Characterization
Explanation Elicitor	Dealing with an <b>internal Knowledge Want</b> : Utterance or action that signals readiness to receive information to the partner to deal with an incomplete understanding. I distinguish four different local outcomes in a conversation:
– successful	Partner delivers the <b>Target Content</b> ; the <b>Knowledge Want</b> gets satisfied.
– insufficient	Not yet <b>successful</b> , additional attempts to transfer <b>Target Content</b> are made.
– frustrating	Partner is aware of <b>Knowledge Want</b> ; but no further attempts to satisfy it.
– ignored	Partner is not aware of the <b>Knowledge Want</b> .
Clarification Cascade	Prototypical order in which the five following types are performed:
– Improper Asking (Section 8.2.2)	Signaling the <b>internal Knowledge Want</b> in some way; requires high <b>Togetherness</b> for the partner to understand the <b>Topic</b> .
– Direct Asking (Section 8.2.3)	Formulating the <b>Topic</b> as an actual question; requires medium <b>Togetherness</b> for the partner to understand the intention.
– Refer to Common Ground (Section 8.2.3)	Referring to the basis from which to understand the <b>Knowledge Want</b> . With high <b>Togetherness</b> , such a reference alone may be enough to reconstruct the <b>Topic</b> .
– Entice to Simple Step (Section 8.2.5)	Making the partner understand one’s <b>Knowledge Want</b> by enticing her to perform a certain mental step with an implicit conclusion.
– Make Proposition (Section 8.2.6)	Formulating a <b>Hypothetical Target Content</b> for the partner to validate; requires less <b>Togetherness</b> as the partner does not need to understand what the speaker’s <b>Knowledge Want</b> . Can be <b>optimistic</b> (HTC is expected to be correct) or <b>pessimistic</b> (HTC is expected to be incorrect).

Table 8.1: Properties and types of Explanation Elicitors

## 8.2.1 General Properties of Explanation Elicitors

All **Explanation Elicitors** share a number of properties:

1. The speaker is aware of her incomplete understanding in some area and wants to do something about it, i.e., she perceives an **internal Knowledge Want** (see also Section 7.2.2).
2. The speaker expects her partner to be helpful in that area, be it in the sense of providing information or validating ideas.
3. The speaker makes an utterance that is at least in part directed to her partner to signal her *readiness to receive information*. These utterances are ultimately intended to make the partner help completing her understanding through communicating the **Topic**.

### 8.2.1 a) “Trigger” or “Elicit”?

In an early publication (Zieris & Prechelt, 2014), the **Elicitors** were called “*Explanation Triggers*”. I renamed them as I found the intentional aspect to be relevant, i.e., the intent to elicit an explanation is more relevant than the effect of actually triggering one (in speech act theory, this is the difference between illocution and perlocution, see page 112).

From a researcher perspective, the intention behind a pair member’s activity is occasionally ambiguous to the point that it could have been unintentional behavior: A developer may make some noise without any intention to elicit an explanation from the partner, but still trigger one and be content with it. While developing the concepts, I found it impractical to draw a clear line between triggering an explanation (regardless of whether it was intended) and eliciting an explanation (regardless of whether it is successful).

The next example illustrates the distinction of eliciting or triggering an explanation, which is especially relevant for acts of **Improper Asking**, as some of the according utterances are too unspecific to attribute an intention to them:

**Example 8.2: Intentional Elicitation or Incidental Trigger?** (CA2, 47:10–47:21)

The pair wants to perform a manual test to see whether their last refactoring broke the functionality they are concerned with (the “attribute table”). The application is already running and C2 looks for a way to call the functionality. C2 is lacking the according **S knowledge**, which would be to open the context menu and click on the button labeled “Attribute Table”.

C2: “<\*looks at GUI, stops moving the mouse cursor\*> Erm.” Improper Asking

C5: “Just ‘Attribute Table’”

C2: <\*double-clicks, after 4 seconds another windows opens\*>

“Ah, sure, ‘Attribute Table’. Of course.”

<\*right clicks, chooses entry “Attribute Table”\*>

C2’s “Erm” together with his mouse movement indicate that he is not sure how to proceed, but—because of their pair’s high **Togetherness**—he does not need to formulate an actual question for C5 to understand the **Topic** and provide him with guidance on how to test their application through the GUI.

C2’s actions are not explicitly directed at his partner, so it is not clear whether he originally intended to *elicit* an explanation. Certainly, his actions *trigger* an explanation by (from C5’s perspective) signaling a **Knowledge Want** and his readiness to receive information. Since C2 is not confused by C5 telling him how to call the functionality and he tries to put the advice into action, his original actions constitute an act of **Improper Asking**. (Had C2 refused C5’s explanation, however, I would not consider C2’s “Erm” to be an **Improper Asking** because that would contradict C2’s intention.)

### 8.2.1 b) Role in the Conversation

Depending on its aftermath in the ongoing conversation, I characterize an **Explanation Elicitor** as either **successful**, **insufficient**, **frustrating**, or **ignored**.

#### **Successful Elicitors**

An **Elicitor** is **successful** when the partner delivers the **Target Content** and the underlying **Knowledge Want** gets fully satisfied. See Example 8.2 above, where C2’s **Improper Asking** is **successful** as he gets the information he needs from his partner C5.

#### **Insufficient Elicitors**

An **insufficient Elicitor** made the partner aware of the **Knowledge Want**, but the **Target Content** is not yet transferred and additional attempts are made. The recurring Example 8.1 illustrates multiple **insufficient Elicitors** on the part of developer J1.

#### **Frustrating Elicitors**

If the speaker no longer attempts to **elicit** an explanation even though the **Target Content** is not transferred, the last **Elicitor** in the sequence is considered to be **frustrating**, because she is discouraged to pursue the **Knowledge Want** any longer. In the next example, J1 is not able to make his partner J2 understand the **Topic** of what he wants to know:

**Example 8.3: Frustrating Improper Asking** (JA1, 08:27–09:19)

J2 scrolls over a function which has 100 lines and 11 if-statements (nested up to five levels deep) and proposes to extract subroutines from it. Upon seeing the complex source code, J1 develops an **internal Knowledge Want**.

J2: “The function is extremely long, but splitting it should be possible. [...] Wouldn’t you agree?”

J1: “Yes? Erm, is there (!...!) can (!...!) <frustrated> nah (!...!)” **Improper Asking (insufficient)**

J2: “We can go through it, if you like.”

J1: “Yes? Erm, how does it (!...!) Yes, let us go through.” **Improper Asking (frustrating)**

J2: “Yes. Well, here, we have [...]”

J1 obviously wants to ask *something*, but is lost for words. While his first attempt is **insufficient**, he gives up after the second. Nevertheless, J1’s **Improper Askings** make J2 aware of an underlying **Knowledge Want** and he proposes to offer a general explanation of what the function does, allowing J1 to formulate specific questions later on once he understood better what he wants to know. (J1 asks many questions in the following hour, but it does not become clear what the **Topic** of this early **Knowledge Want** was and whether it was ever addressed.)

**Ignored Elicitors**

As all activities in pair programming, **Elicitors** may be **ignored** (which is a ♦, a conversational defect, see Section 6.2.2e). Then, the partner is presumably not yet aware of the **Knowledge Want** and hence does not react to the **Elicitor**. The next example shows how a developer’s attempt to **elicit** an explanation is completely **ignored**:

**Example 8.4: Ignored Improper Asking** (DA2, 12:52–13:21)

Developer D3 is looking for a piece of code he wants to show to his new colleague D4. As D3 searches, a statement involving some `LicenseKey` is visible on screen for less than three seconds but catches D4’s attention. D4 already knows at this point that the system uses some third-party library (“*the component*”), and he develops an **internal Knowledge Want**.

D3: < \*opens a file, reads file name, places cursor into file in line with `LicenseKey` and scrolls down immediately\* >

D4: “<chuckles> (#LicenseKey#)?” **Improper Asking (ignored)**

D3: “Exactly, and these are called < \*selects three methods\* > and from here it goes on < \*selects single line\* > to the respective class. I mean, in this case into the `TestCalendar` < \*scrolls up again, `LicenseKey` is visible again\* > [...]”

D4: “Mh, `LicenseValidator` is from the component, isn’t it?” **Make Proposition (successful)**

D3: “Yes, correct.”

D4: “Ok.”

D4’s first utterance indicates his focus of attention and his further actions make clear his intention to **elicit** an explanation from D3, which is why I consider this an act of **Improper Asking**. D3, however, does not react to this at all: His “*exactly*” (German: “*genau*”) is merely a self-directed exclamation that he found the right spot he wanted to show to his colleague—a *discourse marker* (see Jucker & Ziv, 1998) which D3 uses often throughout the session. Since D4’s **Knowledge Want** is not satisfied (and D3 takes his time looking for interesting pieces of code), D4 comes up with an idea of his own of what the `LicenseKey`-statement could be about—a **Proposition** that D3 immediately validates.

### 8.2.1 c) Precedence of Elicitors: Clarification Cascade

For a pair member with an **internal Knowledge Want**, there is an order in which the five types of **Elicitors** are usually employed until one of them is *successful*: **Improper Asking** comes before **Direct Asking**, which comes before **Refer to Common Ground**, which comes before **Entice to Simple Step**, which comes before **Make Proposition**. I call this progression **Clarification Cascade**. It consists in bringing first oneself and then the partner to clarity about what one wants to know. Developers may skip steps, but they generally do not go backwards.

There appear to be different thought processes behind each ‘level’: An **Improper Asking** is just about the experience of not understanding something, while **Direct Asking** involves formulating something that another developer can be expected to understand. While the formulation of such a question may or may not be tailored to the specific partner, explicitly **Referring to Common Ground** then definitely involves considering what is shared knowledge among the pair. **Enticing to Simple Step** goes even further in that it requires a notion of the *differences* of the two developers’ understandings. Finally, **Making a Proposition** requires to come up with some hypothesis all alone.

A long **Cascade** can be seen in the recurring Example 8.1 where J1 goes from **Direct Asking** through all levels to **Making a Proposition**; Example 8.4 above features a short one where D4 starts with **Improper Asking** and (after 20 seconds) follows up with a **Proposition**. In Example 8.16 on page 270, I discuss a rare case of a pair having to ‘reset’, where O4 **Makes a Proposition** which his partner O3 does not understand as an **Elicitor** so O4 backs down to **Direct Asking**.

A long **Cascade** going up through multiple levels may be a sign of low **Togetherness**, as it takes several attempts to make the **Topic** understood to the partner, while pairs with high **Togetherness** may infer each other’s **Topic** even from seemingly incoherent mumbling.

The **Clarification Cascade** is not likely something developers are aware of or which could be consciously employed when encountering a problem in a PP session.

### 8.2.2 Improper Asking

With an **Improper Asking**<sup>1</sup> the speaker does not explicitly state what she wants to know but merely signals an **internal Knowledge Want** to the partner along with her readiness to receive information. Such utterances are not real questions, but can still **elicit** a response like a question could. In cases where the developers’ intentions are not clear (as in Example 8.2), the ongoing conversation may provide some clues for the researcher: If the partner reacts to such an utterance and the original speaker in turn does not appear to be confused by her partner’s reaction, I characterize the original action as an **Improper Asking**.

With an **Improper Asking**, the speaker does not (yet) say explicitly what her **Knowledge Want** is about (as in Example 8.3), but the area may be hinted at (as in Example 8.4). Either way, her partner might still guess the **Topic** correctly (see Example 8.2), which depends on the pair’s **Togetherness**. Pairs with high **Togetherness**—i.e., with a shared understanding of the system and of software development, with a shared plan, good workspace awareness, and no language barriers (see Section 6.4)—may understand the **Topic** even without a properly formulated question. With lowered **Togetherness**, as in the next example, an **Improper Asking** may be **insufficient** for transferring the **Target Content**:

---

<sup>1</sup>This concept was called “*finding*” in an early publication (Zieris & Prechelt, 2014).

**Example 8.5: Insufficient Improper Asking With Low Togetherness** (JA1, 28:44–29:18)

The radio broadcast system written by J2 has multiple `NewsProcessor` implementations which deal with the peculiarities of different radio stations in their respective `execute` methods. Before the excerpt below, J1 wanted to know whether these are *all* different. J2 explained that, indeed, all radio stations are somewhat unique in their file handling, except for stations Alpha and Beta which follow the same process. In the past, J2 therefore extended class `Beta_News` from `Alpha_News` such that it inherits the `execute` implementation. J1 is not aware of that inheritance and deems all `NewsProcessors` to be siblings, which leads to some confusion.

- (1) J2: “It is the same for Alpha und Beta. [...]”
- (2) J1: “So, if I would compare `Alpha_News` and `Beta_News`, it would not be different.”  
Make Proposition (ignored)
- (3) J2: “Have a look at `Beta_News`, please. <\*opens class `Beta_News`, scrolls to middle of the file\*> I just opened it. You see, there is no `execute` method.”
- (4) J1: “<\*jumps to J2’s position\*> There is indeed no `execute` method.”
- (5) J2: “Nope. Since it is the same for both.”
- (6) J1: “Meaning?”  
Improper Asking (insufficient)
- (7) J2: “‘Meaning’ what?”

J2 does not understand J1’s attempt to elicit an explanation in turn (6), as J1 does not make clear the **Topic** of his **Knowledge Want** and J2 is not able to infer it due to momentarily low **Togetherness**. (This exchange is continued in Example 8.15.)

The pair’s **Togetherness** is not only impaired by a lack of shared system understanding, but also in two other ways: First, their workspace awareness is limited, since this is a distributed pair programming session. J1 jumped directly to J2’s cursor position in the middle of the subclass `Beta_News` and thus could not see that it extends `Alpha_News`. Second, there is also a language barrier of sorts, since the domain expert J2 talks about two actual radio stations Alpha and Beta, while the programming expert J1 refers to the corresponding classes `Alpha_News` and `Beta_News`.

**8.2.3 Direct Asking**

In contrast to **Improper Asking**, with an act of **Direct Asking** the speaker explicitly says what she wants to know. So in addition to just signaling an **internal Knowledge Want** and the speaker’s readiness to receive information it also phrases the **Topic**. The according utterances are typically questions, usually in the syntactic form of a *wh*-question (*Why? Where? What?*, etc.). If the utterances take a different shape in the pair’s dialog, they can at least be rephrased as a question.

The speaker expects her partner to be able to answer the question. In this sense, **Direct Asking** is similar to the base concept *ask\_knowledge*. As Salinger & Prechelt (2013, p. 158) already described, such requests may be open questions or already contain possible answers, as the next two example boxes illustrate respectively:

**Example 8.6: Direct Asking with Open Questions** (JA1, JA2)

Examples of **Direct Asking** from developer J1 who asks for rationales and other information not directly visible from the code in sessions JA1 and JA2:

- “*Is there an important reason for all `NewsProcessors` to get the same (time)?*” (JA1, 10:31)
- “*Can that (!...!!) is this an expected case, I mean, can it happen?*” (JA1, 14:15)
- “*Why is the encoder stored as a `String` instead of a constant?*” (JA2, 17:53)
- “*Why is there a timeout?*” (JA2, 25:21)

**Example 8.7: Direct Asking With Possible Answers** (DA2, 01:54–02:06)

D4 just joined the company and wants to understand the GUI technology stack. He does not ask a completely open question, but includes two possible answers he could think of (see Example 7.9 for more context).

D4: “How is this implemented in general, because (!...!) is it more of an SWT user interface than Eclipse, or what?”

While the base concept *ask\_knowledge* is limited to verbal requests for information, *Direct Asking* also includes cases where the speaker asks the partner to show something (see the following two examples), which would be considered *propose\_step* in the base layer.

**Example 8.8: Asking to Show Application** (DA2, 09:23–09:37)

D4 wants to know about the requirements and current implementation status of the application—which is more than D3 appears to have in mind in his *Explanation*.

D3: “In principle, there should be a toolbar up here <\*hovers blank area in GUI, closes application\*> I’ll show you how what it looked like in the old calendar.” Present New Fact

D4: “Yeah, also show me what it can do already and what it should be able to do.”

Direct Asking

D4’s turn could be rephrased as two *wh*-questions: ‘*What functions does the toolbar already have (show me, please)? What additional functions should it have?*’

**Example 8.9: Asking to Show Source Code** (CA2, 10:07–10:47)

C5 added a new method to an existing interface `IFeatureAttributeConfiguration` (which C2’s knows about). That new method returns an array of `IVirtualColumn`—an interface which C5 added recently and C2 does not know about yet.

C5: “Well, I better show you what I did already. <\*opens IFeatureAttribute Configuration\*> What I did [...] is extend this `IFeatureAttributeConfiguration` with an `IVirtualColumn` [...]” Present New Fact

C2: “Show me, what they look like.”

Direct Asking

C5: <\*opens `IVirtualColumn`\*>

C2’s utterance could be rephrased as a *wh*-question: ‘*What do they look like (show me, please)?*’

**8.2.4 Refer to Common Ground**

A pair programmer may repeat something that has been said earlier (sometimes even verbatim) or emphasize something obvious. In general, the propositional content of such utterances is not new to the partner. When *Referring to Common Ground*, the speaker expects the partner to know the piece of information already and to understand the reference as such.

Explicitly referring to the *Common Ground* this way is done both to *elicit* an explanation (this section) and to provide *Explanations* (Section 8.3.2). By *eliciting* an explanation by *Referring to Common Ground*, the speaker may clarify an earlier stated question, e.g., from an *insufficient Direct Asking*. The next example illustrates how the repetition can be meant as an amendment to the original question, that is, the context in which it should be interpreted:

**Example 8.10: Refer to Common Ground as Context for Question** (JA1, 05:00–05:48)

After J2 provided an explanation that did not meet J1's expectation, J1 Refers to Common Ground (see recurring Example 8.1 for more context):

- (16) J1: "In what time window are you looking?" Direct Asking
- (20) J1: "[...] the time window for the change?" Direct Asking
- (22) J2: "[...] You know, they always start a new file. When the news are over, again a file is created. I mean, I basically never have more than the news."
- (23) J1: "Yes, no, I mean 'cause you said you look for so long until the size stops changing, right?" Refer to Common Ground

The last statement can be placed *before* the Direct Asking that turned out to be insufficient as an attempt to make its context clear:

- J1: "[Context:] You said you look for so long until the size stops changing.  
[Question:] In what time window are you looking? [...] The time window for the change?"

In other cases, however, there is no 'original question' to be amended and the speaker expects the partner to understand the Topic by Referring to Common Ground alone:

**Example 8.11: Imply Question by Referring to Common Ground** (JA2, 13:41–14:25)

J2 walks his colleague through an API he designed earlier. J1 does not yet know that the method `setInputFile` can be called multiple times to accumulate input files. He is confused by the method name (in fact, the pair later agrees on renaming it to "addInputFile") and wants to know how multiple input files can be handled, but the Topic of his Knowledge Want is only implied by Referring to Common Ground.

- J2: "Basically, you have a `TranscodeJob` you want to create. There, you can set arbitrary `InputFiles`. Yes? You can set arbitrarily many. [...] So you have an `InputFile`, then everything that should be done to it, then `OutputFile`. Then again an `InputFile`, everything that should be done, `OutputFile`. [...] Ok, erm (!!!...!!)"
- J1: "Yes, hold on, hold on, hold on. You said, 'you can set arbitrarily many'?" Refer to Common Ground
- J2: "Yes, in saying <\*duplicates existing statement\*> (##job.setInputFile##) another `Input File`."

In both cases, that is, Referring to Common Ground as context for an explicit or an implicit question, the mechanism is to direct the pair's attention to a smaller part of the shared reality in their discourse. Just as for the partner to understand the Topic from an Improper Asking alone, being able to infer the implicit question of a partner who only Refers to Common Ground arguably requires high *Togetherness*.

A different use of Referring to Common Ground can be seen in Example 7.10 on page 247: Here, K1 receives a long explanation from his partner K2 and makes clear he already understands something of the Topic. K2 in turn is reassured that his Explanations are understood and does not need to go into more detail at this point. This goes beyond normal *back channel* utterances (which are typically rather short and would not interrupt the other speaker's turn, see Section 3.2.1b), because K1 effectively elicits the next explanation from his partner as if to say 'I already know X: Tell me something new'.

### 8.2.5 Entice to Simple Step

When the speaker entices her partner to take a **Simple Step**, she states something she expects to be easily digestible for her partner, either because that thing might be already known to her or at least easy to understand. In contrast to **Referring to Common Ground**, a **Simple Step** not only entices the partner to think about what was said and agree with it, but also to draw an implicit conclusion from it.

In the German language, there happen to be speech particles associated with these expectations. In particular, the unstressed modal particle ‘ja’ is “typically used when the speaker wants to indicate that the proposition is, should be or can be evident for the hearer” (Bross, 2012, p. 192). By **Referring to Common Ground**, the speaker wants to indicate the proposition is or *should be* evident for the partner, whereas **Enticing to Simple Step** indicates it is not yet, but *can* become evident. The particles ‘doch’ and ‘eigentlich’ can assume similar roles. There is no direct equivalent in English. In the transcripts of German utterances, I use ‘you know’ or ‘right?’ where appropriate. Although some of the analyzed sessions were in English, none of the recorded developers was a native English speaker. Example 8.25 on page 276 shows a **Simple Step** from an English session which contains the marker ‘as you know’.

#### Example 8.12: Translating German Modal Particles (JA1, 05:48–05:53)

Original German utterance from the recurring Example 8.1, with modal particle in bold face:

(25) J1: “Musst du **ja** noch ’nen gewisses Zeitfenster noch einplanen, in der immer noch ’ne Veränderung stattfinden **könnte**.”

English translation with appended “right?” to indicate that this is not an imperative:

(25) J1: “Then you need to plan for a time window in which a change **could** happen, **right?**”

Note, however, that the German original was a declarative utterance, not a (grammatical) question.

Usually for **Simple Steps**, the conclusion is left to the partner and not made explicit by the speaker. In the next example, however, the two parts of the partner agreeing first and then drawing the conclusion afterwards become visible as separate turns between which the original speaker feels compelled to spell out the conclusion:

#### Example 8.13: Simple Step with Almost Revealed Conclusion (DA2, 18:17–18:57)

D3 described to his partner the desired state of the application (which is not completely shown below, see Example 7.6 for more context) and D4 noticed an inconsistency in that description. In the excerpt below, D4 wants to make sure he understands the design requirements correctly and formulates his question in the form of a **Simple Step** (turn 2). At first, D3 does not understand what D4 is going for, and just agrees to D4’s statement. Consequentially, D4 goes on to make the conclusion of his **Simple Step** explicit (turn 4), but then D3 notices that his explanations did not add up, interrupts his partner, and sets the record straight.

- (1) D3: “In general, a toolbar like this one. <\*hovers toolbar in different module\*> [...]”
- (2) D4: “But, shouldn’t it be the goal to always have these thingies up here? <\*points to buttons in the toolbar\*>”  
Entice to Simple Step
- (3) D3: “Yes, exactly.”
- (4) D4: “But then <\*points to screen\*> for the calendar, it’s (!...!!)”
- (5) D3: “Yeah, that was nonsense what I told you before <\*switches to calendar view\*>”
- (6) D4: “Then it should be up there <\*points to screen\*>, right?”
- (7) D3: “We don’t put it here in this narrow bar <\*hovers narrow space in calendar view\*>, but up there, too <\*hovers space above calendar view\*>”

The common reaction to a **Simple Step**—at least for a pair with high **Togetherness**—is to draw the conclusion quickly. After all, that is what the speaker intends. The next example shows a *successful* **Simple Step** in the context of a **Clarification Cascade** with three stages:

**Example 8.14: Conclusion Drawn From Simple Step** (JA1, 09:32–10:01)

J1 ponders the rationale behind the method parameter `currentTime` and needs three attempts to get the response he wanted from J2. Note that it is not the response he expected: J1 apparently assumed the parameter was useless.

- (1) J1: “We enter with the `currentTime`” Improper Asking
- (2) J2: “Exactly, `currentTime`, that’s the timestamp on invocation.”
- (3) J1: “On invocation. Why is it needed (from outside)?” Direct Asking
- (4) J2: “I need it several times. For example at `getLastFile <selects statement*>`, where it gets the (!..!) to have something to compare the time against, right?”
- (5) J1: “Yes, but, I mean, why does it need to be passed in from the outside? You know, the function knows what time it is on its own.” Direct Asking + Entice to Simple Step
- (6) J2: “For each `Processor` to get the same timestamp.”
- (7) J1: “OK (. ) I see.”

The conclusion J1 is aiming for is something like: *‘The parameter is an unnecessary dependency, so it should be removed’*—J2, however, is disagreeing with it and explains its necessity. The intended conclusion builds on the pair’s **Togetherness**, especially their shared understanding of software development and their shared plan to improve the structure of the code at hand.

With too low **Togetherness**, however, the penny may not drop, as J2 clearly states with dismay at the end of another three-stage **Clarification Cascade** from the same session:

**Example 8.15: Conclusion Not Drawn From Simple Step** (JA1, 28:57–29:47)

Continuation of Example 8.5, with turns (3) to (7) repeated for context. J1 is not aware of the fact that `Beta_News` extends `Alpha_News` and that it inherits its `execute` method.

- (3) J2: “Have a look at `Beta_News`, please. *<opens class Beta\_News, scrolls to middle of the file\*>* I just opened it. You see, there is no `execute` method.”
- (4) J1: “*<jumps to J2’s position\*>* There is indeed no `execute` method.”
- (5) J2: “No, since it is the same for both.”
- (6) J1: “Meaning?” Improper Asking (insufficient)
- (7) J2: “‘Meaning’ what?”
- (8) J1: “Meaning, erm, where is the `execute` method for `Beta`?” Direct Asking (insufficient)
- (9) J2: “Well, the `execute` method of `Beta` is equivalent to that of `Alpha`.”
- (10) J1: “Yeah, sure. But there needs to be made a connection somewhere, right? *<scrolls up to the top\*>* [..]” Entice to Simple Step (insufficient)
- (11) J2: “I don’t understand the question. I’m sorry. [..]”
- (12) J1: “*<selects extends clause of class declaration\*>* Ahh, (#extends `Alpha_News`), ok.”

The conclusion of J1’s **Simple Step** was something like: *‘Class `Beta_News` has no `execute` implementation of its own, so it is not possible for instances to have an implementation equivalent to that of `Alpha_News` at runtime’*. In the exchange above, all three **Elicitors** were **insufficient**: J2 could not understand the **Topic**. J1 eventually understood that he overlooked the possibility of inheritance, produced the **Target Content** himself, and satisfied his **Knowledge Want**. The pair’s **Togetherness** was impeded because J1’s and J2’s respective understanding of the software system differed in a relevant way. As a consequence, they talked past each other for a while.

### 8.2.6 Make Proposition

**Direct Asking** requests the partner to supply the **Target Content**. But instead of asking an open question, the speaker may also reduce the matter to a closed *yes-no* question by **Making a Proposition** for which she does not know the truth value but assumes her partner is capable of making an assessment. The partner is basically handed a **Hypothetical Target Content** (see Section 7.3.2).

There are *optimistic* and *pessimistic Propositions*, depending on whether the speaker expects the partner to validate or invalidate it. *Indifferent Propositions*, like the one by J2 in the recurring Example 8.1, are the exception. Here, for the propositional content being right or wrong is not the actual point of the conversation.

#### 8.2.6 a) Optimistic Propositions

In making an *optimistic Proposition*, the developer thinks she understood something and expects her partner to agree and validate it. I can conceive of three different reasons for such utterances. They are not validated due to lack of insight into the developers' minds.

1. The *inner*-perspective: The speaker wants to be sure her understanding is correct. This perspective appears reasonable for developer D4 in Example 8.4 above.
2. The *we*-perspective: The speaker wants to maintain the pair's common ground and make sure both of them know what they know. This is probably what O4 had in mind here:

**Example 8.16: Misunderstood Optimistic Proposition** (OA8, 10:35–10:49)

The developers try to understand the reason for a test case starting to fail. O4 ponders the meaning of an attribute value and wants O3 to validate his hunch by uttering an **optimistic Proposition**. His partner O3 does not understand it as a question but as a hypothesis for what might have caused the test failure (which she rejects). Then, O4 backs down to **Direct Asking** and makes his intention to **elicit** an explanation as well as the **Topic** explicit.

O4: “offsetFraction means in the middle <\*hovers offsetFraction: 0.5 in test case\*>”  
Make Proposition (optimistic)

O3: “The offsetFraction, we didn't change that. So, I'm assuming this is okay. But we did change the offsetDays (!...!!)”

O4: “I want to know what the meaning of offsetFraction is in this test.”

Direct Asking

This is a rare case of a pair not progressing according to the **Clarification Cascade** (usually, **Direct Asking** comes before **Make Proposition**). Developers O3 and O4 had some difficulties with their **Togetherness** which makes interpreting the partner's intentions difficult; a long **Breakdown** (Example 6.16 on page 210) starts shortly after the above exchange.

3. The *outer*-perspective: Another way to interpret some pair members' behavior is that she wants to ‘show off’, as if to say ‘*I found a nice way to put this, wouldn't you agree?*’. Consider session JA1, which is all about original author J2 explaining the more experienced J1 what the software does. J1 summarizes his current understanding of the software by **Making Propositions** 17 times during the first half hour, 13 of which were *optimistic*. One instance can be seen below, where his understanding turns out to be wrong and he then produces two further *pessimistic Propositions*:

**Example 8.17: Propositions to Demonstrate Understanding** (JA1, 27:47–28:42)

J1 starts with an **optimistic Proposition** where he thinks he understood something and wants J2 to validate it, which J2 does not. Then, J1 follows with two **pessimistic Propositions** thus giving his partner the chance to say ‘*Sorry, I was mistaken*’ and is again surprised he does not.

J1: “The whole functionality up here is the same over all News plugins, **Make Proposition** am I right?” (optimistic)

J2: “No, this is not correct. Since they are all a little bit different.”

J1: “A little bit different, ok. <with rising voice>”

J2: “Yes, it’s a bit problematic around here [...]”

J1: “That means that each single class with underscore News at the end **Make Proposition** does something on its own in its execute method.” (pessimistic)

J2: “Yes, it is overwritten. As you can see here (!...!!)”

J1: “Yeah, I see that. But, erm, the stuff it gets overwritten with is really **Make Proposition** (!...!) I mean, if you take and compare two arbitrary files, they are always (pessimistic) somehow different.”

J2: “They are different, yes.”

**8.2.6 b) Pessimistic Propositions**

In a **pessimistic Proposition**, the speaker thinks the partner got something wrong and offers the partner something that is easy to disagree with, to understand where she was wrong, and to correct her mistake.

In Example 8.17 above, J1’s expectation with his two **pessimistic Propositions** was that J2 would realize and admit that he was mistaken two times. In the next example, another developer also appears to give his partner the chance to correct himself.

**Example 8.18: Pessimistic Proposition in Disbelief** (CA2, 11:32–11:55)

In the middle of C5’s explanation on his recent code changes which involve a generic type (Column Attribute), C2 suddenly asks about some other code and whether that type is used there as well. This questions takes C5 by surprise. C5 thinks that C2 suggested to use a more specific type (VirtualAttribute) and he wants to make sure that C2 meant it that way:

C2: “What do the existing data structures look like, those used for the GUI? Is there a ColumnAttribute? Otherwise, I would simply use those.”

C5: “I don’t know what you are talking about right now.”

C2: “About what I did, with the GUI.”

C5: “You only had a VirtualAttribute?” **Make Proposition** (pessimistic, ignored)

C5 formulates his question as a **pessimistic Proposition**, which gives C2 the chance to say something like ‘*No, I did not mean it that way*’. However, neither does C2 verbally react to this **Proposition**, nor does C5 persist. After four seconds of looking at each other silently, C2 continues his proposal:

C2: “(....) I thought that we use this for the data structure, what I did.”

The camera angle only captured their chins, but not the rest of their faces (see *Discussion of Data Collection* on page 156), which makes interpreting their somewhat absurd behavior even more difficult.

### 8.3 Providing Explanations

An **Explanation** can be one's own initiative or a reaction to an **Explanation Elicitor**. The different types of **Explanations** share the following properties:

1. The speaker perceives an **external Knowledge Want**, i.e., she thinks her partner has a relevant gap in knowledge that should be addressed (see also Section 7.2.3).
2. The speaker sees herself in the position to help closing that (perceived) gap.
3. The speaker makes an utterance that is directed to her partner with the intention to make the partner understand something.

Note that the speaker may wait for the partner to signal her readiness to receive information, but this is not always the case and can then be problematic (as I will discuss in the next chapter, in Example 9.23).

In contrast to the **Elicitors**, I did not find a particular order in which developers use the different **Explanation** types of telling, making their partner recall something, nudging, and plain hoping. There is no '**Explanation Cascade**' as a counterpart to the **Clarification Cascade**.

Concept	Description/Characterization
<b>Explanation</b>	Dealing with an <b>external Knowledge Want</b> : An utterance or action that is directed to the partner to help with their incomplete understanding. There are three types:
– <b>Present New Fact</b> (Section 8.3.1)	Presenting something as a fact, expecting it is new to the partner; either to fill a gap or to correct a false understanding.
– <b>Refer to Common Ground</b> (Section 8.3.2)	Referencing something that is considered common ground in order to make an explanation or thought easier to understand.
– <b>Entice to Simple Step</b> (Section 8.3.3)	Enticing the partner to take a certain mental step to come to an implicit conclusion; can be a step <b>to failure</b> (as in ironic speech) or a step <b>to success</b> (as in hoping the partner will understand the point despite a suboptimal explanation).

**Table 8.2:** Types of Explanations

#### 8.3.1 Present New Fact

**Presenting a New Fact** is the straight-forward form of an **Explanation**. The speaker presents something as a fact with the general expectation for it to be new information to her partner. In base layer terminology, instances of **Presenting New Facts** are thus a small subset of *explain\_ - knowledge*, as they are not about opinions and evaluations.

The new information may serve different purposes: To provide information that the partner lacks or correct a false understanding. I discuss both cases below.

##### 8.3.1 a) Present New Fact to Fill Gap

There are different types of knowledge gaps which **Presenting a New Fact** can attempt to fill. In the example below, the receiving developer did not even know he had an according knowledge gap until he reached a point in the code where a decision needed to be made:

**Example 8.19: Present New Fact After Improper Asking** (AA1, 59:12–59:25)

In a “Task” class, the pair is writing a call to a method that takes two boolean arguments: `isInactive` and `isMirror`. A2 enters an expression for the first argument and then hesitates for the second. A1 quickly informs him that the entities that the Task class represents are never mirrored, so A2 hardcodes `isMirror` as `false`.

A2: “(##!isActive##) (..) oh, well (!!...!!)” Improper Asking  
 A1: “There aren’t any tasks on mirrors.” Present New Fact  
 A2: “(..) There aren’t?”  
 A1: “Nope.”  
 A2: “<\*sets second argument to false\*> Right.”

In another session, the knowledge gap itself was possibly already known to both the developers—one developer already worked on the code alone and his partner does not know the content of these changes—but it was not relevant until the pair set the goal for their session:

**Example 8.20: Present New Fact After Direct Asking** (KB1, 02:39–02:58)

K2 and K3 set the goal for their session by defining that they do not want to alter any functionality yet. In order to assess whether K2’s unfinished changes from before interfere with this goal, K3 needs to know what these changes were. K2’s summarizes his changes by [Presenting New Facts](#):

K3: “Ok, we first only do the refactoring, right?”  
 K2: “I’d say so.”  
 K3: “Yes, right? And then we continue.”  
 K2: “If we touch this <\*hovers overview of locally changed files in IDE\*> along the way, it does no harm, it can be included easily.”  
 K3: “What is it?” Direct Asking  
 K2: “Well, there is <\*opens one changed file\*> the Controller for once. There, Present New Fact  
 I split the date field into two date fields, like date time field.”  
 K3: “Yes, ok.”

K2’s evaluation that his changes ‘*do no harm*’ is not [Presenting a New Fact](#) as it is a matter of opinion—which K3 could have followed along but wanted to know the facts instead.

**8.3.1 b) Present New Fact to Correct False Understanding**

[Presenting a New Fact](#) may also be intended to ‘replace’ some false understanding:

**Example 8.21: Present New Fact to Correct Understanding** (AA1, 28:26–28:33)

A1 refers to a (possibly completed) process of adapting a number of list views in their CMS to a new design. The current list uses the old design and A1 figures it should be adapted, too. A2, however, appears to recall an explicit decision to keep the old design here and corrects A1.

A1: “It should be rebuild to the new design (or something, right?)” Make Proposition  
 A2: “Nope, nope, this one should stay this way.” Present New Fact  
 A1: “It should stay?”  
 A2: “Yes.”  
 A1: “A-ha.”

**Example 8.21 (continued)**

Changing any list’s design is *not* the topic of their session. A1 is off, which makes reconstructing his intention difficult. Either way, this excerpt possibly illustrates two instances of information flowing in the team during a PP session: First, A1 noticing a possible defect and intending to bring it to his partner’s and later to the team’s attention; second, A1 learning about details of the ongoing redesign progress.

Note that I did not analyze how pairs deal with hypotheses specifically, but only instances where developers invoke factual knowledge, possibly in the context of dealing with unknown or uncertain aspects.

**8.3.2 Refer to Common Ground**

If a plain **Explanation** in the form of a **Presenting a New Fact** is not understood by the partner, the speaker may **Refer to Common Ground** in order to give her partner another chance to understand (e.g., turns 19 and 21 in Example 8.1). In other cases, developer may attempt to answer a question solely by **Referring to Common Ground** without any new information, as in this excerpt:

**Example 8.22: Refer to Common Ground for Obvious Explanation (CA2, 10:58–11:26)**

C2 questions a design choice in the code in front of them. C5 alludes to a larger pre-session design idea with a **Simple Step**, but C2 does not reach the intended conclusion that the local design follows from the larger idea. As a consequence, C5 refers to the design idea more explicitly, thus referring to their **Common Ground** and eventually answering C2’s question.

C2: “Do we actually have a `ColumnAttribute` here? Is that so?” **Direct Asking**

C5: “I did (!...!) we have, we do need an `I`, a `ColumnAttribute` afterwards, Entice to  
right? To be able to put this in these, erm, all (!...!) in these, where you Simple Step  
fetch them all.”

C2: “Is that so? Do we need this at all for the visualization in the attribute **Direct Asking**  
(table)?”

C5: “We do need, for the visualization in the attribute table, if we want **Refer to**  
to do it through `getAllAttributeColumns`, we do need an `IColumn Common Ground`  
`Attribute`.”

C2: “If we want to do it that way (yes, ok.)”

It might not be clear from the above transcript, but when the two talk about whether or not they “*have*” or “*need*” a dependency to `ColumnAttribute`, they talk about whether it is *necessary* rather than whether they *like* it. In this sense, the pair talks about facts and not about opinions.

It appears that C5 wanted to **Refer to Common Ground** explanation the first time around, but was lost for words. Additionally, C2 appears to have no memory of the pre-session design idea, so **Referring to Common Ground** does not work in the way intended by C5, but only because C2 knows the software system well enough to reconstruct the design idea on the fly.

Pair programmers not only **Refer to Common Ground** in an **Explanation** when there is an immediate point they want to get across, but also in the aftermath of an already finished topic, as if to make sure the original point really settles in. In session JA1, for example, J2 referred back multiple times to some topic the pair had already finished (e.g., a few minutes after Examples 8.1 and 8.14), and showed J1 the various places in the source code he was talking about before. J1 did not ask to see them, J2 just casually **Refers to Common Ground** in the ongoing conversation.

### 8.3.3 Entice to Simple Step

The speaker states something she expects to be easily digestible for her partner because that thing is easy to understand based on her current knowledge.

I found the following situations in which such **Simple Steps** are made: In the context of *knowledge transfer*, as part of an *elicited Explanation*, but also in the context of *decision making* with regard to the source code design or the pair's next steps. Roughly analogue to the *optimistic/pessimistic* distinction of **Propositions** (see Section 8.2.6), **Simple Steps** come in one of two flavors: They can be *to success* or *to failure*, the latter of which also includes ironic remarks.

#### 8.3.3 a) Enticing to Simple Step as Reaction to an Elicitor

When the partner asks a question and the addressed developer thinks the answer is obvious, she might have troubles to formulate a plain answer and instead hopes the partner can figure it out on her own. I call these **Simple Steps to success**, because the speaker wants the partner to understand the point despite a suboptimal explanation. This can be seen in C5's **Simple Step** in Example 8.22 (“*we do need [...] a ColumnAttribute afterwards, right?*”), in J2's turn (4) in Example 8.14 (“*I need it [...] to have something to compare the time against, right?*”), or with J2 in the following example:

#### **Example 8.23: Conclusion Not Drawn From Simple Step** (JA1, 14:35–14:58)

The code handles the default case (`Mode.FileTracking`) in an `if`-block from which the method returns, and it handles error case (`Mode.Error`) after the `if`-block. In the exchange below, J1 wants to know more about the external circumstances that lead to an error case, while J2 refers only to the internal control flow.

J1: “In this case here, with the `Error` it should never slide into, if everything goes well, right?”

J2: “If everything goes well, never slides into, right.”

J1: “(OK)”

J2: “I mean, if it found a file, it does not slide in there. Yes? <\*selects default-case return statement\*> Because then it is in this `FileTracking` mode. (...) Yes? (.) Or no? Why don't you say anything? <laughs>”

Entice to Simple Step (to success)

J2 expects his partner to take the **Simple Step** which consists of reading the source code and understanding the control flow. He is clearly confused by his partner not reporting back his success. J1, however, does not appear to be interested in the control flow. As far as I can tell, the pair never resolves this misunderstanding.

In principle, it is conceivable that the addressed developer knows the answer and *could* provide a plain explanation, but still decides to merely nudge her partner in the right direction with a **Simple Step**. However, I have not seen such behavior in my data.

What I have seen, though, is a developer combining information with nudges (i.e., **Present New Facts** and **Entice to Simple Steps**), hoping the partner connects the dots:

#### **Example 8.24: Entice to Simple Step After Presenting New Fact** (JA1, 24:16–24:44)

J1 noticed that a variable is only used within a single method, and wonders why it is nevertheless stored as an instance variable instead of a local variable. The class in question implements a state automaton. J2 first **Presents a New Fact** (the method is called periodically), and then hints at the implication (the automaton's state needs to be retained from one call to the next)—a **Simple Step**.

J2: “Well, let me put it this way: I have to retain it, since this execute method [...] is called every 30 seconds.”

Present New Fact

## Example 8.24 (continued)

J1: “Yeah?”

J2: “This means I have to save the state, right? Once I fetched the file from the remote and start the `FileTracking`, I need to save a reference to it. (...) I don’t want to start new every time, right? I can’t, right?”

Entice to Simple Step (to success)

J2 wants his partner to connect the dots after the new information has been presented.

The excerpt below has a similar structure, but here the explaining developer **Refers to Common Ground**, i.e., his conclusion does not remain as implicit as with a **Simple Step**:

**Example 8.25: Present New Fact and Refer to Common Ground** (MA1, 13:26–13:41)

M2 wanted to know whether there are other things besides a ship’s position that change over time. In M1’s explanation, he **Presents New Facts** and **Refers to Common Ground**:

M1: “Everything that is not in the `Vessel` table can be changed along the time.”

Present New Fact

M2: “OK.”

M1: “But it’s more likely that the `Machinery` and the `Equipment` won’t change, as you know, adding a crane to a ship is not something you do every day.”

Present New Fact + Refer to Common Ground

All the **Simple Steps** I analyzed in the context of reacting to an **Elicitor**, i.e., in the context of *knowledge transfer*, were **Simple Steps to success** as the speakers expected their respective partners to understand some relationship they were not aware of before.

**8.3.3 b) Entice Simple Step as Argument in Decision Making**

As discussed in Section 7.2.3, **external Knowledge Wants** do not only arise when the partner signals her **internal Knowledge Want**, but also when the partner appears as if she *should* learn something, e.g., if she makes a **misled** proposal. Instead of plainly rejecting the proposal, some developers explain *why* the proposal is problematic. An explanation provided by means of a **Simple Step** has a number of **benefits** in such a situation:

1. The proposal-rejector does not need to confront her partner directly, potentially saving some embarrassment if the proposal turns out to be good after all.
2. If the proposal is problematic, the proposer gets a chance to understand the reasons.
3. If the proposal is good, the proposer gets a chance to defend it effectively since the rejector made her thought process clear.

The next excerpt illustrates a **Simple Step** that one developer employs to criticize a (seemingly) problematic proposal:

**Example 8.26: Entice to Simple Step as Indirect Criticism** (OA8, 50:12–52:52)

O4 argues against O3’s proposal. He seems to think that O3 wants to make the test green by recording the current (faulty) behavior and simply declaring it ‘*correct*’ in the test case, which goes against his general understanding of software development. The conclusion O4 wants O3 to draw is ‘*Your proposal is not a reasonable way to approach writing a test case*’. As can be seen in the full exchange in Example 6.14 on page 201, O3 understands the intended criticism. (Furthermore, O3’s original proposal was, in fact, not problematic but just misunderstood by O4.)

(12) O3: “We can check that, if you console-log, erm, so this, the real variable value, here in the `<*>production code file*>`, we can like console-log the (!..!)”

(17) O4: “In the test, we have to think about what the right value is before we start the test. `<*>looks at O3*>`”

Entice to Simple Step (to failure)

Criticism of proposals can be rather direct. Below, the rejector formulates a **Simple Step** after the proposer has already begun to put his idea into action and defended it:

**Example 8.27: Entice to Simple Step in Direct Criticism** (AA1, 19:44–20:13)

The system uses fallback `MicroObjects` and more complete `MiniObjects` (see also Example 7.2). The current code tries to fetch the latter but allows to potentially use the other one (with a parameter `allowMicroObject=true`). Knowing that the surrounding code checks the type of the result, A2 proposes to fetch a `MiniObject` instead and to get rid of the parameter. A1 strongly disagrees and reminds A2 of the fallback mechanism with a **Simple Step**.

A2: “And now, we use a proper `MiniObject` instead `<removes parameter allowMicroObject=true, changes variable type to MiniObject*>`”

A1: “No!”

A2: “Yes, sure! We did already see that it only checks for `isActive` and then disregards the `editedContent`.”

A1: “Eh, sure! So what? What happens if the task is on a non-readable object?”  
Entice to Simple Step (to failure)

A2: “(..) Ah, then I get an exception.”

A1: “Yep. Exactly.”

A2: “Ah, that’s the fallback, right. `<undoes changes*>` Right.”

While **Simple Steps** in the context of an elicited knowledge transfer are *to success* (see Section 8.3.3a), **Simple Steps** in context of a design or process discussion appear to be *to failure*, which consist in attempting to make the partner stumble into a problem, to then see the issue and chuckle: ‘*Oh yes, that’s true. Silly me!*’

In sessions JA1 and JA2, the experienced developer J1 criticized many of J2’s design ideas through **Simple Steps**, yielding all of the above mentioned benefits (see, e.g., Example 8.14). My research was not focused on decision making in pairs, so I did not deepen these inquiries.

## 8.4 Summary

The key aspect of the low-level mechanisms of knowledge transfer in pair programming in both *asking* and *explaining* appears to be reconciliation of the two partners’ mental states.

The main difficulty in *eliciting* an explanation is to make the partner understand the **Topic** of one’s **Knowledge Want**, that is, to ask the right question in the right way. In pairs with (momentarily) high **Togetherness**, the **Topic** may be understood even through **Improper Asking** before the developer in need was even able to formulate an actual question, or by simply pointing something out that both developers know about anyway, i.e., by **Referring to Common Ground**. With low **Togetherness**, however, a **Clarification Cascade**, which includes different types of **Elicitors**, is necessary: Going from an initial **Topic** statement with **Direct Asking**, to securing the **Common Ground**, taking a **Simple Step** together, and potentially even reducing the matter to a simple *yes/no*-question with a **Proposition**.

The partner’s mental state is not any more transparent from an **Explanation** perspective either. It is not surprising then that **Presenting New Facts** to close a gap or to correct some false understanding are combined with references to the **Common Ground** and with **Simple Steps** if the speaker wants to gently make the partner understand something without confronting her directly or if she simply does not know how to explain something that is obvious to her.

In the next chapter, I zoom out one level to see how these fundamental building blocks come together to form **Episodes** of knowledge transfer.



# Chapter 9 Episodes of Knowledge Transfer

---

9.1	Purpose and Structure of this Chapter . . . . .	280
9.1.1	Discussion of Recurring Example . . . . .	281
9.2	Properties of <b>Episodes</b> . . . . .	282
9.2.1	Starting an <b>Episode</b> . . . . .	282
	<i>Start Pursuing an Internal Knowledge Want • Start Pursuing an External Knowledge Want</i>	
9.2.2	Ending an <b>Episode</b> . . . . .	285
	<i>Stage: Started Initiative • Stage: Acknowledged Initiative • Stage: Understood Topic • Stage: Transferred or Acquired Available Target Content • Stage: Satisfied the Knowledge Want</i>	
9.2.3	Defining Characteristics of an <b>Episode's Mode</b> . . . . .	288
9.3	 <b>Pull Mode</b> . . . . .	289
9.3.1	Properties of  Pull Episodes. . . . .	289
9.3.2	Short  Pull Episodes for Factual Information. . . . .	289
9.3.3	 Pulling for More Than Explanations . . . . .	290
9.4	 <b>Pioneering Modes</b> . . . . .	291
9.4.1	Properties of  Pioneering Episodes . . . . .	291
9.4.2	 Silent Pioneering Mode . . . . .	291
9.4.3	 Talking Pioneering Mode . . . . .	293
9.5	 <b>Co-Production Mode</b> . . . . .	295
9.5.1	Properties of  Co-Production Episodes . . . . .	295
9.5.2	 Parallel Production Mode . . . . .	296
9.6	 <b>Push Mode</b> . . . . .	297
9.6.1	Properties and Context of  Push Episodes . . . . .	297
9.6.2	Transfer or Construction?. . . . .	298
9.6.3	 Push is not just the Inverse of  Pull. . . . .	299
9.7	Summary and Discussion of Related Work . . . . .	300

## 9.1 Purpose and Structure of this Chapter

In Chapter 7, I described that pair programmers perceive **Knowledge Wants**, which can be *internal* (a developer wants to know something), *external* (a developer thinks her partner should know something), or *collective* (which both developers share). In Chapter 8, I described the individual activities of pair programmers to deal with these **Knowledge Wants** by making sure that the partner understands one's **Topic** and that the **Target Content** is transferred and understood. There are multiple forms of enticing the partner to provide an explanation—the **Elicitors**—and also different ways to formulate **Explanations**. Pair programmers also read source code and documentation, and interact with the running system and tools to learn more about their system's behavior and about relevant technology in general.

The focus of this chapter are **Episodes** which are the *sequences* of activities from when a **Knowledge Want** arises until it is satisfied by transferring or acquiring the **Target Content**. The developer who perceives the **Knowledge Want** also is the one who actively pursues the clarification of the **Topic** while the partner merely reacts. That active developer is the **Propellor** of the **Episode**.

Depending on the type of the underlying **Knowledge Want** and other factors, such as personal preferences and what the partner is expected to know about the matter, the pair members use different information sources and communicate the **Topic** and (parts of) the **Target Content** in different ways. The way *how* an **Episode** is carried out is its **Mode**, of which there are four main ones (with subtypes). Since the **Modes** are a central concept in my thesis, I use a color code to refer to them in several schematics later on.

-  **Pull** (Section 9.3): In a **Pull Episode**, the **Propellor** employs **Elicitors** to satisfy her **internal Knowledge Want** by explaining the **Topic** to her partner and make her deliver the **Target Content**.
-  **Pioneering Production** (Section 9.4): Here, the **Propellor**—called **Pioneer**—also wants to satisfy an **internal Knowledge Want**, but relies more on source code, the running system, and documentation (rather than her partner) to *produce* the **Target Content**.  
There are two forms here: First, the **Talking Pioneer**, who announces her **Topic** and/or makes her progress in producing the **Target Content** visible for her partner to jump in. Second, the **Silent Pioneer**, who works as if she was alone and communicates neither **Topic** nor **Target Content**, which, for better or worse, excludes her partner.
-  **Co-Production** (Section 9.5): A **Co-Production Episode** involves both developers who share a **collective Knowledge Want** such that both are **Propellors** for the common **Topic**. By going through the source code and other artifacts together and by consolidating their insights, they produce the **Target Content** together.  
An accidental variant of this is **Parallel Production** in which the developers do not make sure they end up with a common **Target Content**.
-  **Push** (Section 9.6): A **Push Episode** is **propelled** by a developer with an **external Knowledge Want**, who keeps offering not-explicitly requested explanations until she thinks her partner received the **Target Content** for a **Topic**.

Regardless of the **Mode** in which a particular **Topic** is addressed, both partners are involved *somehow*. The **Propellor** may dominate the conversation with her actions temporarily, but her partner's reactions are just as important. Conceptually speaking, I therefore consider these reactions part of the same **Episode**. For simplicity, I also disregard any developer's **external Knowledge Want** that only reflects her partner's **internal Knowledge Want**, i.e., I only speak of an **external Knowledge Want** when the developer follows an agenda of her own and wants to explain *more* than was asked for by her partner.

### 9.1.1 Discussion of Recurring Example

As for the whole main part of my thesis, I turn to the first minutes of session JA1 to illustrate the above considerations and relevant concepts of this chapter.

#### Example 9.1: Illustrating Episodes (JA1, 04:15–06:15)

The beginning of session JA1 features two Episodes: J2 pursues a long-running Push to bring J1 up to speed what their module is all about (left-hand side of the following transcript). J1 engages in a Pull Episode regarding an implementation detail (right-hand side). In each of these Episodes, one pair member is the active part who propels it forward, while the respective partner merely reacts (shown as indented utterances). Occasionally, J2 provides more explanations than J1 asked for and thus falls back into his own Push Episode.

#### J2's Push Episode

(Topic: Software Design & Context)

(6) J2: “In the end, the news recording of every hour pops out.”

(7) J1: “M-hm.”

(8) J2: “The way this works is that there are multiple processors, so there is the central plugin and multiple processors which each handle one wave.”

(9) J1: < \*nods\* >

(10) J2: “For most of them, right after the full hour there is a check, if there is a new file on the remote share.”

(11) J2: “If so, the most recent file is selected and it starts checking how the file changes size-wise.”

J1: < \*stops nodding, looks to his upper right\* >

(Knowledge Want presumably arises →)

(13) J2: “I mean, it looks until the file does not get bigger anymore, then it is apparently ready.”

(14) J1: “M-hm”

(15) J2: “And then it is fetched and handed over to transcoding.”

(16)

(17)

J2: “because then it’s guaranteed that news files exist if any exist.”

(18)

(19)

J2: “That can take up to seven minutes, depending on the wave.”

(20)

(21)

#### J1's Pull Episode

(Topic: Size of Polling Interval)

J1: “In what time window are you looking?”

J2: “I start looking two minutes after the full hour”

J1: “OK”

J2: “And then monitor this file as long as needed until it’s ready.”

J1: “Hm ya, but mh, the time window for the change?”

J2: “Yes, right, that is, er, time window for the change is variable, depends on how the news go.”

## Example 9.1 (continued)

(22)	J2: “I can’t know that, right? You know, they always start a new file. When the news are over, again a file is created. I mean, I basically never have more than the news.”	
(23)		J1: “Yes, no, I mean ’cause you said you look for so long until the size stops changing, right?”
(24)		J2: “M-hm”
(25)		J1: “Then you need to plan for a time window in which a change <u>could</u> happen, right?”
(26)		J2: “Yeah, well, until up to five before the hour. I really take my time.”
(27)		J1: “<laughs> No, I really mean the size now, the size of the time window, I mean (!...!) You wait for 10 seconds, then after 10 seconds you decide: In those 10 seconds nothing has changed, so the file appears to be ready.”
(28)		J2: “Ahhh, that’s what you mean. No, 30 seconds.”
(29)		J1: “30 seconds, that’s what I wanted.”
(30)		J2: “That’s 30 seconds long the time window. Now I got you.”

## 9.2 Properties of Episodes

Knowledge transfer **Episodes** have a number of relevant properties. Individually, they each have a **Topic**, that is, the (sometimes literal) question which one or both developers want to clarify. In most instances, a single pair member ‘owns’ the **Topic** and is the **Propellor** of the **Episode** who drives it forward.

While the **Topic** makes each **Episode** unique, three other properties make **Episodes** comparable across developers, sessions, and contexts:

- **Start of an Episode:** How does the pair go from one developer perceiving a **Knowledge Want** to both of them being engaged in clarifying the **Topic**? I discuss this in Section 9.2.1.
- **End of an Episode:** Not all **Episodes** end with a fully transferred **Target Content** and a satisfied **Knowledge Want**. When does the **Propellor** stop pursuing her **Knowledge Want**? I discuss this in Section 9.2.2.
- **Mode of an Episode:** The degree to which the two developers are involved in the **Episode**, the information sources they use, and about what they communicate how much are together characterized by its **Mode**. The details of which I discuss in Section 9.2.3. The central contribution of this chapter are then the four main **Modes** of knowledge transfer that are  **Pull** (Section 9.3),  **Pioneering Production** (Section 9.4),  **Co-Production** (Section 9.5), and  **Push** (Section 9.6).

### 9.2.1 Starting an Episode

Every **Episode** of knowledge transfer starts with one developer perceiving a **Knowledge Want**. While the pursuit of a **collective Knowledge Want** involves high **Togetherness** and appears to often simply happen (which I discuss in Section 9.5), both **internal** and **external Knowledge Wants** pertain to one developer only and the partner needs to be addressed somehow.

Concept	Description/Characterization
Episode	Sequence of activities to satisfy a <b>Knowledge Want</b> ; can end in various states (depending on acknowledgement and satisfaction of <b>Knowledge Want</b> and transfer of available <b>Target Content</b> ): <b>ignored</b> , <b>resignation</b> , <b>unnecessary</b> , <b>postponed</b> , <b>partial success</b> , <b>needs investigation</b> , or <b>successful</b> .
– Propellor	The pair member who pursues the clarification of the <b>Topic</b> .
– Mode	The way in which the pair carries out the <b>Episode</b> . Four main <b>Modes</b> :
□ Pull (Section 9.3)	<b>Propellor</b> pursues an <b>internal Knowledge Want</b> by eliciting explanations from partner who understands <b>Topic</b> and delivers <b>Target Content</b> .
▣ Pioneering (Section 9.4)	<b>Propellor</b> pursues an <b>internal Knowledge Want</b> by experimentation, reading in source code and other artifacts, thus acquiring the <b>Target Content</b> herself.
▣ Co-Production (Section 9.5)	Both developers <b>propel</b> by pursuing a <b>collective Knowledge Want</b> by experimenting, reading source code, and integrating their ideas and hypotheses.
▣ Push (Section 9.6)	<b>Propellor</b> pursues an <b>external Knowledge Want</b> by providing <b>Explanations</b> to her partner without being requested to do so.

Table 9.1: Properties of knowledge transfer Episodes

### 9.2.1 a) Start Pursuing an Internal Knowledge Want

For an *internal Knowledge Want*, the **Propellor** may wait for her partner to finish her turn before she starts a □ **Pull Episode**. In Example 9.1, J1 appears to perceive his **Knowledge Want** in turn (12), but waits until turn (16) to ask his question—after J2 finished his explanation.

In other instances, developers appear less considerate. In the example below, one partner does not seem to notice or care about that his partner is in the middle of explaining something:

#### Example 9.2: Start Episode With Oblivious Interrupt (CA2, 10:46–11:56)

C5 is about to explain the rationale of the changes he did prior to the session. He just opened up a new interface to show it to his partner (see Example 8.9). In the next minute, C2 starts two □ **Pull Episodes**, both by interrupting C5’s explanation:

C5: “More (!...!) ain’t more in it yet because (!!...!!)”	} ▣ Push (ignored)
C2: “O-kaaay.”	
C5: “Because, erm <*waves hands*> it is, I, erm, more (!!...!!)”	} (ignored)
C2: “Do we actually have a ColumnAttribute here? Is that so?”	
[...]	} □ Pull (Example 8.22)
C2: “If we want to do it that way (yes, ok.)”	
C5: “If we want to do it that way, that’s correct. But that’s how I understood our agreements so far. I didn’t yet move more in (!!...!!)”	} (ignored)
C2: “What do the existing data structures look like, those used for the GUI? Is there a ColumnAttribute? Otherwise, I would simply use those.”	
[...]	} □ Pull (Example 8.18)
C2: “I thought that we use this for the data structure, what I did. (Anyway, ok. But show me.)”	
C5: “We can go on, for now. <*starts looking for other file for one minute*>”	

C5 does not get to fully explain the rationale of his changes until 19:30 (shown in Example 7.7).

In the following example, the same developer knowingly interrupts his partner’s current turn to formulate his question:

**Example 9.3: Start Episode With Careful Interrupt** (CA1, 00:55–01:22)

In the very beginning of the session, C1 is about to summarize what he did prior to the session. C2 cuts him short in order to settle an overarching question first: Should they be working on this task right now at all? C2’s gestures and choice of first words indicate that he is fully aware that he is interrupting his partner C1.

C1: “As I said, I already started, programmed for an hour. I started with the GUI. I’ll show you quickly. <\*opens overview of recent changes\*> What I did with the (!!!)”

C2: “<\*raises hand\*> First, really short question. In how far is this whole thing destabilizing, with regards to the coming branch, and that we only should do consolidation on the trunk?” ] Pull

**9.2.1 b) Start Pursuing an External Knowledge Want**

I already discussed a number of cues programming partners may pick up on to infer a knowledge gap in their partner in Section 7.2.3. With such an external Knowledge Want, the developer may start an explanation right away, for instance if the knowledge gap is about to lead to a bad design choice which the speaker wants to prevent. This is what A1 did in Example 8.27 with his Simple Step: “No! [...] What happens if the task is on a non-readable object?”

Alternatively, the developer may also jump on signs of readiness to receive information, such as having read some source code and then leaning back, as can be seen here:

**Example 9.4: Start Pushing at the Right Moment** (CA1, 12:10–12:47, 13:57–14:32)

C1 worked on the task alone prior to the session and C2 already inspected the recent changes in the running application. He now wants to take a look at the source code (a long Talking Pioneer Episode), and C1 waits patiently for the right moment multiple times before he adds more information in short Pushes.

C2: “Then I’d take a look at this FeatureLayerPropertiesPanel thing.” ] Talking Pioneer

C1: “M-hm.”

C2: “<\*closes application, switches to IDE, searches for the class, opens it, slowly scrolls stepwise down through file (,,,,,,,,,,,,,,,,,,,,,)\*>” ] Talking Pioneer

C1: “And here, there is now a LabelAttributesPanel.” ] Push

C2: “<\*scrolls back up\*> Ah yes, I see.”

And about a minute later:

C2: “Let’s see where this is being used <\*opens list of field usages, hovers entries one by one (,,,,,,,,,,,,,,,,,,,,,)\*> Okay.” ] Talking Pioneer

C1: “Yes, so I already started to extend the data, I mean the model. [...]” ] Push

C2 was a Talking Pioneer who made his Topic explicit both times, so C1 could follow his actions more easily.

Example 7.17 on page 251 shows an explicit form of starting a Push Episode: D4 first asked about his partner’s knowledge level and then whether he should proceed with the explanation: “Do you know how this works with the OSGi class loading? [...] Shall I explain it?”

### 9.2.2 Ending an Episode

In general, an **Episode** ends when the **Propellor** no longer pursues her **Knowledge Want**. A *successful* **Episode** is one that ends with a satisfied **Knowledge Want** because the underlying knowledge gap has been filled. However, not all **Episodes** turn out this way, but stop at various points before. As is shown in Figure 9.1, each **Episode** conceptually goes through five stages but may stop at each of them if the **Propellor** does not persist.

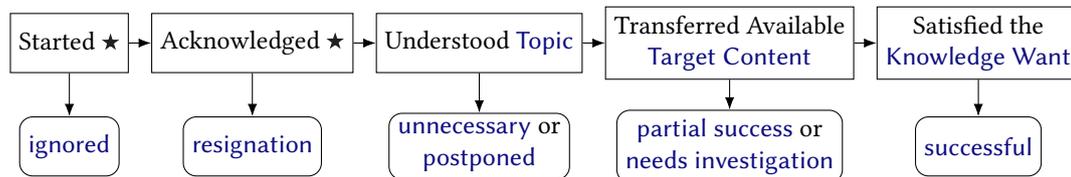


Figure 9.1: Lifecycle of an Episode: Stages and Outcomes

#### 9.2.2 a) Stage: Started ★ Initiative

Each **Episode** starts with some initiative action, or ★ (see Section 6.2.2a), such as an **Elicitor** or an **Explanation**, or some non-verbal computer interaction such as silently opening a certain file with the intention to read it. If the pair does not get past this stage, i.e., if the partner does not appreciate the ★, the **Episode** remains *ignored*:

##### Example 9.5: Ignored Episode (JA1, 40:29–40:42)

The pair has agreed to extract some portion of the code into a new method. As this is a distributed session, J2 only knows that J1 is about to call the refactoring tool, but is not able to see it. J2 is about to **push** something to his partner, but J1 struggles with the refactoring and does not react to J2 at all. J2 makes two further attempts to start his **Push**, but does not get J1's attention.

J1: < \*starts "Extract Method" refactoring with selected code block\* >

J2: "This is why I wanted up here < \*selects some lines\* >, in order to make splitting it up easier (!!...!!)" ] **Push** (ignored)

J1: " < \*error pops up, cancels refactoring\* > Does not work this way."

J2: "I mean, here (!!...!!)" ] (ignored)

J1: " < \*reading code\* > Ah, I see, it does return here, right."

J2: "I mean (!!...!!)" ] (ignored)

J1: "This does not work yet."

Although the pair's audio connection might have a part in this particular instance, Example 9.2 shows that *ignored Episodes* are not limited to distributed settings.

#### 9.2.2 b) Stage: Acknowledged ★ Initiative

A knowledge transfer **Episode** typically involves both developers to some degree. However, a **Propellor** may find it too difficult to communicate her **Topic** to her partner and the **Episode** ends in *resignation*. This is the fate of J1's **Pull** attempt in Example 8.3 who gave up after two **Improper Askings** and let his partner take over with a **Push**. In the example below, the speaker is able to formulate a proper question (**Direct Asking**), but his partner makes clear he knows too little about the technology to understand him:

**Example 9.6: Resigned Episode With Explicit Topic** (DA2, 10:11–10:26)

D4 already asked some questions regarding the technology stack, which D3 could only answer in general terms. Ten minutes into the session, D3 not being able to answer D4’s questions appear to be have become an inside joke for the pair. Now, D3 indicates he does not even understand the question properly (and thus, the **Topic**) and D4 does not persist but **resigns**.

D4: “Is this still an Eclipse View, or what? (No.)”

D3: < \*shrugs, looks at D4 smiling\* >

D4: < \*snorts\* >

D3: “<laughing> I really don’t a have clue regarding ExtensionPoints and stuff. I don’t get along with it. I mean, I can’t give you great information here.”

D4: < \*clicks ball pen, stares at screen\* >

□ Pull

] (resignation)

This is a case of a developer giving up on a **Pioneering Episode** where he did not even try to communicate the **Topic**:

**Example 9.7: Resigned Episode With Tacit Topic** (CA2, 1:03:46–1:04:12)

In the middle of debugging their recent changes, C2 cuts off a proposal of his partner in order to start  **Pioneering** until he **resigns** after about 12 seconds and sets a breakpoint for the debugger.

C5: “Let’s select getVirtualAttributes (!...!!)”

C2: “Hang on a sec, I’d just really quick take a look < \*reads code ( , , , , , , , ) \* >”

“Really quick < \*hovers keyboard\* > (..)”

“Screw it”

< \*sets breakpoint ( , , , , , , , ) \* >

“And now, I’d debug this thing”

< \*starts application in debug mode\* >

C2 remains a **Silent Pioneer** here, his **Topic** does not become clear.

 Pioneer

] (resignation)

**9.2.2 c) Stage: Understood Topic**

With the **Topic** understood by both developers, they may still decide to not completely transfer or acquire the **Target Content**. On the one hand, there are **unnecessary Episodes** which the pair aborts and does not plan to pick up again later; on the other hand, some **Topics** are merely **postponed** to some later point.

The **Topic** of an **unnecessary Episode** appears irrelevant for the current progress as can be seen in Example 7.3 on page 241 as well as in this example:

**Example 9.8: Unnecessary Episode** (DA2, 15:02–15:15)

When the pair tries to compile their code, an error message pops up, and D3 engages in a  **Talking Pioneering Episode** with the (implicit) **Topic**: ‘Why does the build process prompt an error?’ After a few seconds, D3 proposes to not dig into the issue and to just live with it.

D3: “< \*open details of error message ( , , , ) \* > Hm?”

“Checkstyle doesn’t work ( , , , )”

“< \*closes error message\* > OK, I’d say: Get lost, Checkstyle! <laughs>”

D4: “<laughs>”

 Pioneer

] (unnecessary)

Other not particularly pressing, but still relevant **Topics** may be considered, but then **postponed**:

**Example 9.9: Postponed Episode** (CA2, 1:05:29–1:05:38)

C2 and C5 are circling in on the cause of a failure. After C2 understands C5's **Topic**, he proposes to address it later.

C5: "Let's have a look whether the delegates are correct."	] <input type="checkbox"/> Co-Production  (postponed)
C2: "They are, you see <*hovers inspection view in debugger*>"	
C5: "Yes, the the, erm (!...!) methods, whether they are (!...!!)"	
C2: "We'll look in a moment."	
C5: "M-hm."	

If, however, developers decide to address the **Topic** now, they transfer or acquire the **Target Content** as far as the current **Mode** allows it.

**9.2.2 d) Stage: Transferred or Acquired Available Target Content**

If all parts of the **Target Content** that are readily available to the pair in the current **Mode** are transferred or acquired but the **Knowledge Want** is not yet satisfied, the pair may decide the **Topic** still needs investigation and switch to a different **Mode** such as  **Pioneering** (as in Example 9.10) or  **Co-Production** (as in Example 9.11).

**Example 9.10: Pull Episode Switching to Pioneering** (CA2, 52:26–53:44)

C2 is not sure about the commit message convention and starts to  **Pull** about it. C5 is not sure about it either, so C2 decides to look into his e-mails, probably because the convention was introduced in the team recently.

C2: "<*types cad-507*> That's what it should be like, with the hyphen?"	] <input type="checkbox"/> Pull  (needs investigation) <input checked="" type="checkbox"/> Pioneer
C5: "Not sure, hyphen (!...!) I think, the cad in all caps? (..) Or has it to be lowercase?"	
C2: "Erm, I have to look in my e-mail. Don't know it from the top of my head."	
<*leaves desk for one minute, then comes back*>	
"I'll write all caps <*writes CAD*>"	

**Example 9.11: Pull Episode Switching to Co-Production** (JA1, 22:59–23:48)

Experienced developer J1 wants to know why a variable is in the class scope, instead of just being local. Module author J2 cannot answer the question right away, so the  **Pull Episode** ends and both of them start looking for possible reasons in the code.

J1: "You store this in a class variable [...] Why does remoteNewsFile have to be defined in the class? Why is not enough to do it in the method?"	] <input type="checkbox"/> Pull  (needs investigation) <input type="checkbox"/> Co-Production
J2: "Erm, the remoteNewsFile? You mean, why it's not enough in here?"	
J1: "Yep, the remoteNewsFile."	
J2: "Let's think about it, whether <*J1's name*> has a point here. <*starts scrolling*>"	
J1: "<*starts reading code*> There is one occurrence [...]"	

For less important **Topics**, the **Episode** may remain a *partial success* in case the pair reaches a point where some questions are still open, but pursuing them is not deemed worth doing. This can be seen in Example 7.9, where D4 asks a few times about the used technology stack, but D3 can only answer general questions, and in Example 7.16, where C2 wants to know more about a tool’s behavior, but C5 loses interest in the **Topic** after making clear he does not expect any upcoming impediments on part of the tool.

### 9.2.2 e) Stage: Satisfied the Knowledge Want

If the **Propellor’s Knowledge Want** is satisfied after all available **Target Content** has been transferred or acquired, the **Episode** is *successful*. This type of outcome is nicely illustrated in Example 9.1, where J1 explicitly says: *“That’s what I wanted.”* Many of the **Episodes** shown in the numerous examples are *successful*, such as the very brief one in Example 7.2 on page 241: *“Do the MicroObjects still exist?” – “Yes. They do here.”*

### 9.2.3 Defining Characteristics of an Episode’s Mode

Roughly speaking, the **Mode** of an **Episode** is the fashion how the pair deals with the underlying **Knowledge Want**. I discuss the **Modes** individually in Sections 9.3 to 9.6, but will give an overview about the idea and their commonalities here. The aspects characterizing the **Mode** of an **Episode** are:

1. **Type of the Knowledge Want:** Do the developers’ actions primarily pertain to one pair member’s *internal* or *external* **Knowledge Want**, or is there a *collective* **Knowledge Want**?
2. **Topic Handling:** Is the **Topic** itself known to both developers? Or is it a ‘private’ **Topic** of one pair member?
3. **Target Content Source:** From where does the relevant information, the **Target Content** primarily come? Is it one developer’s memory or is it artifacts such as source code and documentation?
4. **Target Content Handling:** Is the transferred and acquired **Target Content** consolidated and integrated such that it is then part of the pair’s common ground? Or are the pair members not aware of what their respective partner knows about the **Topic**?

Mode	Type of Knowledge Want	Topic Handling	Target Content Source	Target Content Handling
□ Pull	internal	shared	developer	consolidated
▣ Talking Pioneer	internal	shared	artifacts	consolidated
▣ Silent Pioneer	internal	private	artifacts	not consolidated
▣ Co-Production	collective	shared	artifacts & developers	consolidated
▣ Parallel Production	collective	shared	artifacts	not consolidated
▣ Push	external	shared	developer	consolidated

**Table 9.2:** Characteristics of the different knowledge transfer **Modes**. The above properties pertain to the idealized concepts.

For any particular **Episode**, the answers to the four questions above will not always be perfectly clear, such as a □ **Pull Episode** in which the **Propellor** also reads bits of code instead of relying upon verbal explanations alone. Considered as concepts, however, the **Modes** each have

some ‘center of gravity’ as summarized in Table 9.2. More fruitful here, however, may be the consideration of the relevant *differences* between them:

- **Push vs. Pull:** In both cases, existing knowledge is transferred from one pair member to the other. The difference is who of the two is **propelling** the exchange. Does one developer keep asking until her knowledge gap is closed, or is it her partner who offers explanations for as long as she sees fit?
- **Pull vs. Pioneering:** Both satisfy an **internal Knowledge Want**. The difference is whether the partner or the source code (and other artifacts) are the main source of information.
- **Co-Production vs. Parallel Production:** Both developers are engaged in reading source code and documentation, but they may or may not take care of making the produced **Target Content** part of their common ground.
- **Silent Pioneer vs. Talking Pioneer:** In both cases, the **Pioneer** reads source code and documentation. The difference lies in how much she tells her partner about the **Topic** and the produced **Target Content**.

**Modes** are the most interesting property of **Episodes**. I discuss their details in the following sections.

## 9.3 Pull Mode

### 9.3.1 Properties of Pull Episodes

During a **Pull Episode** the **Propellor** is trying to make her partner do things that help satisfy her **internal Knowledge Want**, which the partner primarily does through verbal **Explanations** (see Section 8.3), but also through demonstrating features or source code, or by providing directions on how to get there. The **Propellor** uses the different **Explanation Elicitors** which are meant to indicate the **internal Knowledge Want** and to signal that the speaker is now ready to receive information, as well as to make the partner understand the **Topic** (see Section 8.2).

An **Episode** encompasses both partners’ actions that pertain to the **Topic**, i.e., in the case of **Pulling**, the **Propellor’s Elicitors** and her partner’s reactions. Note, however, that there may be other pair activities *during* a running **Episode** that do not belong that **Episode**, such as in Example 9.1 where developer J2 fell back to his overarching **Push** three times while J1’s detailed **Pull** was still running (second halves of turns 17 and 19, and turn 22).

### 9.3.2 Short Pull Episodes for Factual Information

Many **Pull Episodes** are rather short, consisting of just a question and a quick answer, such as the **successful Pull**: “*Do the MicroObjects still exist?*”—“*Yes. They do here.*” (see Example 7.2); or the short **unnecessary Pull**: “*{#tclCode?#}*”—“*<grinning> You don’t wanna know*” (see Example 7.3). Others are longer because they take more turns, such as in the recurring Example 9.1 during which the pair goes through a **Clarification Cascade** (see Section 8.2.1c).

The difference between short, efficient **Pull Episodes** and longer ones that require more clarification is, again, the pair’s **Togetherness**, which makes the process more **fluent** such that simple **Topics** do not take much time and effort to deal with.

Another reason for many **Pull Episodes** for being short is that they have clear factual **Topics**. Once that **Topic** is understood, they reach either a **successful** outcome or at least a **partial success** and end quickly. In the next section, I discuss other cases where verbal explanations are not enough and the **Propellor** asks for more.

### 9.3.3 ■ Pulling for More Than Explanations

Developers with an **internal Knowledge Want** who start a ■ **Pull Episode** expect her partner to possess the **Target Content**. However, developers may also decide to start ■ **Pioneering** despite their partner being an available information source. In the example below, it appears as if one developer *would* have ■ **Pioneered** but, having no control over the keyboard, he resorted to ■ **Pulling**:

#### Example 9.12: Asking for Code, Non-Verbal Reaction (CA2, 10:42–10:54)

C2 wants to know more about the newly introduced interface `IVirtualColumn`. He could ask C5 for a verbal explanation, but he asks to see the code instead. He probably would have started ■ **Pioneering** instead, but the keyboard was out of reach. (This is an example of *backseat driving*, Jones & Fleming, 2013.)

C5 complies and silently opens the according file which only contains one method signature and no comments. After C2 can be expected to have ‘read’ its content, C5 sets to explain the rationale, but C2 cuts him off. C2 is the **Propellor** of this short ■ **Pull Episode** and he decided to stop pursuing his **Knowledge Want** here.

C2: “Show me, what they look like.”

C5: < \*opens interface `IVirtualColumn` >

< \*looks at C2\* > More (!...!) ain’t more in it yet because (!...!!)”

C2: “O-kaaay.”

■ Pull

] ■ Push (ignored)

C5’s reaction (opening the file) is part of C2’s ■ **Pull Episode**; his attempted explanation, however, is part of an (ignored) ■ **Push Episode** (in which he is interrupted multiple times, see Example 9.2).

In other instances, a developer may have control over the keyboard, but is unsure where to go. She pursues her **Knowledge Want** in ■ **Pull Mode**, i.e., she uses her partner as the main information source to close the knowledge gap. This can be seen in Example 7.12 from session CA2 (on page 248), and in the next example from session CA1:

#### Example 9.13: Pulling for Guidance (CA1, 06:17–07:05)

C1 knows the current implementation better than C2 who wants to understand which existing GUI parts may be considered for reuse. He makes C1 provide verbal explanations and guidance around the running application.

C2: “Isn’t there a complete, isn’t there a complete panel already? Did you take the complete preassembled panel already? Or just the individual parts?”

C1: < \*points to screen\* > The two selection boxes are one panel. The checkbox is not part of it (..) erm. There is a bigger component, too, but is also has a button and other stuff that aren’t exactly related to this task. (!...!!)”

C2: “Mmmh, but I’d like anyway, like to see it anyway (!...!) simply because so I (!...!) Do you, do we still have a running application? < \*skims task bar\* >”

C1: “Yes < \*points to task bar\* > we still do.”

C2: < \*restores application\* >

C1: “Alright, it would be the ‘Scale’ tab.”

C2: < \*opens ‘Scale’ tab\* >

C1: “Alright, and this < \*shows area on screen\* > would basically be the component with the button and the label on top, I’d say.”

C2: “M-hm.”

■ Pull

## 9.4 🧩 Pioneering Modes

🧩 **Pioneering** is another way for a developer to deal with an **internal Knowledge Want** in addition to 🟦 **Pulling**: Instead of making the partner do things to close one’s gap, the **Propellor** performs these actions herself—mostly reading source code and interacting with the running system.

### 9.4.1 Properties of 🧩 Pioneering Episodes

There are different reasons why developers choose to 🧩 **Pioneer**. In each particular **Episode**, at least one of the following reasons appears to play a role:

- The partner is not expected to possess the **Target Content**, so 🟦 **Pulling** would not be possible.
- The partner may possess the **Target Content** but is not expected to be good at giving explanations, so 🟦 **Pulling** would take too long.
- The partner is not expected to possess the **Target Content** and not interested in the **Topic** or does not understand it, so 🟩 **Co-Production** is not an option.
- The **Propellor** *wants* to read source code directly, at her own pace.

The **Propellor**’s actual motivation, however, is difficult to reconstruct. So I differentiate two forms of 🧩 **Pioneering** based on *observable behavior* instead. Depending on how transparent the **Propellor** makes her process, I either call her a **Silent** or a **Talking Pioneer**. The differences can be in three aspects:

1. **Knowledge Want**: Is the intention to **Pioneer** communicated to the partner?
2. **Topic**: Is the **Topic** communicated to the partner?
3. **Target Content**: Is the intermediate **Target Content** shared with the partner?

A **Silent Pioneer** falls short on sharing her **Topic** and intermediate **Target Content**, and possibly even her intention; a **Talking Pioneer**, in contrast, shares **Topic** and/or **Target Content**. Put differently: The difference between a **Silent** and a **Talking Pioneer** is whether or not she is **Maintaining Togetherness** (see Section 6.4.4). Sharing intermediate **Target Content** helps **Maintaining Togetherness** with regard to a *shared understanding of the software system* (although it may be possible, I have not seen 🧩 **Pioneering Episodes** pertaining to **G knowledge**), while sharing the **Topic** does so with regard to *one shared plan*. If the pair’s **Togetherness** is high, especially regarding a *shared understanding of software development* and *workspace awareness*, then remaining **Silent** may be hardly problematic because the partner can still reconstruct what is going on (examples follow below).

One advantage of being a **Talking Pioneer** is that a knowledgeable partner may validate newly created **Target Content** or intersperse short 🟨 **Pushes** when the **Pioneer** can actually digest them. This is difficult for the partner of a **Silent Pioneer** who has to guess what she is doing right now. Another advantage is that a **Talking Pioneer** may also enable to partner to develop an **internal Knowledge Want** as in Example 7.15: “Ah, because we are in the anonymous class.”—“What is an anonymous class?”

### 9.4.2 🧩 Silent Pioneering Mode

🧩 **Silent Pioneering** may be beneficial, e.g., if the partner does not feel compelled to follow along and can relax for some time (I have not seen such cases of disengagement). In the worst case, however, the partner is not even aware of the intention to 🧩 **Pioneer**, e.g., because a ★ (an initiative activity) was nonverbal or not understood as such. Then, the partner may think the **Pioneer** is fully attentive and thus ignore her **Episode**:

**Example 9.14: Peril of not Sharing Pioneering Intention** (CA2, 1:15:59–1:16:43)

The pair has a design discussion and C5 closes with a piece of information coming from a conversation C2 is probably not aware of. C2, however, already started a  Pioneering Episode without making it clear, and C5's Push gets ignored.

C5: “We don’t need to change anything in the FeatureProxy.”

C2: “In the way it works we do! Maybe not the class itself, but in the way it works.”

C5: “We need some Provider to pass the FeatureProxy to, in order to get the values. That’s a change in the model.”

C2: “We will see. < \*starts reading source code \* >”

C5: “The only thing is, that < \*lead developer \* > and I discussed yesterday, that we change the FeatureProxySet. There should be a getTableModel and it should be replaced with a factory or something.”

C2: “(,,,,,) I’d like to know (,,,) where this thing is called < \*selects class constructor \* >”

 Silent Pioneer

 Push  
(ignored)

 Talking Pioneer

C2 becomes a Talking Pioneer the moment he shares his Topic, which allows the pair to increase its Togetherness by developing *one shared plan* again.

The next example shows that “silent” does not necessarily mean that the Pioneer goes mute. Although he keeps saying things, he keeps his partner in the dark about the Topic and any newly produced Target Content:

**Example 9.15: Silent Pioneer Who Talks** (CA1, 19:57–21:01)

The pair decided to let their class implement an interface and just found out that they need to implement two methods: `getComponents()` and `setEnabled()`. C2 now goes through the class’s current implementation and appears to have some doubts but does not make his Topic and newly acquired Target Content explicit: This an Episode of  Silent Pioneering. This excludes C1 from the process, and his questions in turns (2) and (8) remain unanswered proposals in turns (4) and (6) go unheard.

(1) C2: “The problem is, it doesn’t fit with `getComponents`. < \*scrolls through file \* >”

(2) C1: “Why doesn’t it fit?”

(3) C2: “I think so, I think so. I could be wrong. I mean (!..!) < \*continues scrolling \* >”

(4) C1: “We only need to get the individual component from the panel, right? (Is that complicated?)”

(5) C2: “Ah, it has a `getContent`. It already has a `getContent`, I just noticed.”

(6) C1: “Ok (..) and a `PanelBuilder`, can we possibly (!..!) get the other panels from there?”

(7) C2: “< \*continues scrolling \* > (,,,,,) I’m not sure whether this all will work (,,,,,)”

(8) C1: “God, we have to (..) hope that there is a (!..!) a `JPanel` on its own, can we deactivate it?”

(9) C2: “< \*continues scrolling \* > (,,,,,) OK, I’d say (,,,,,,,) shall we simply try to implement the methods?”

C1 tried to understand what C2 did, but the pair did not reach a *shared understanding of the software system* and thus a lowered Togetherness.

In the examples above, the Pioneer’s partner at least had the benefit of good *workspace awareness* to help reconstructing what is going on. The next example shows how  Silent Pioneering can be irritating for the partner if it is not clear what the Pioneer is after.

**Example 9.16: Irritating Silent Pioneer with Low Togetherness** (JA1, 19:13–19:39)

In their distributed PP session, J1 deals with an internal Knowledge Want in  Silent Pioneering. He communicates his *intention* (turn 2 in the transcript), but neither his Topic nor any intermediate Target Content. This irritates J2, who originally wrote the software and was already accustomed to being  Pulled by J1 on many different Topics.

- (1) J2: “< \*moves cursor around the code lines while he talks \* > Ok, then we have here (!...!) Yes, it gets copied. You see it here, line 101. Into the file localNewsFile. (...) Copies it here (!...!!)”
- (2) J1: “Hold on, hold on, hold on.”
- (3) J2: “Hm?”
- (4) J1: “Just looking < \*reads code (,,,)\* >”
- (5) J2: “Pardon?”
- (6) J1: “OK. I just needed to read that line.”
- (7) J2: “OK.”
- (8) J1: “< \*continues reading (,,)\* > OK, yes.”

**9.4.3  Talking Pioneering Mode**

A Talking Pioneer makes her Topic explicit to the partner and/or summarizes her current understanding, hypotheses, and such (parts of the Target Content). This verbalization may help the partner to follow along and provide additional information at the right moment.

**Example 9.17: Necessary Pioneering w/o Knowledgeable Partner** (AA1, 13:10–14:13)

Through a  Co-Production Episode, A1 gathered enough understanding to make a concrete design proposal (first line in the excerpt below), which A2 does not understand. A1 starts to explain the currently visible source code ( Push) which helps A2 a little, but he is perplexed about the rationale behind it. He starts to read the surrounding source code and keeps uttering his insights ( Talking Pioneering), allowing A2 to follow him and add further explanations. A few moments later, A2 understands both the current code and A1’s design proposal.

A1: “Sure, we need to override `getIconPrefix` in `TaskNode` (.....)  Co-Production  
it’s missing.”

A2: “Uh? Is it reasonable to (!...!!)”

A1: “See, the icon name is plugged together (.....) and the `TaskNode` simply delegates this to the `ObjectNode`, that’s why these are all static classes there, in order to not duplicate the code.”  Push

A2: “Ah! But why is it built so cumbersome?”

A1: “Who knows.”

A2: “Well, I want to know. < \*opens call-hierarchy of current method \* > See, it’s called from `browse.html` by `renderIcon()` < \*opens that method \* > and here it’s done through the `ViewConfig`”  Talking Pioneer

A1: “Sure, it’s done the same way now, and it’s right that way. We only need to overwrite that method.”

A2: “Ah, it’s because of the on/off, the showing and hiding and stuff.”

A1: “Exactly.”

A2 becomes a Pioneer because A1 does not care for the Topic (“Who knows”). However, A2 utters his insights, so A1 can provide relevant and timely information, and Togetherness is kept up.



the **Target Content** anyway. Although the developer did not say a word while  **Pioneering**, he is still no *Silent Pioneer* because (a) his partner (dealing with the interrupt) would not have been attentive anyway and (b) the **Pioneer** is going to summarize the **Target Content** when appropriate, thus allowing for a seamless progression.

**Example 9.19: Dealing with Interrupts Seamlessly** (AA1, 1:43:40–1:47:42)

A1 and A2 are in the middle of  **Co-Producing** new system understanding. A third developer approaches the pair multiple times to ask technical questions. A1 would try to help her, while A2 continues reading the source code ( **Pioneering**). Whenever A1 turns to A2 again, A2 would then summarize the **Target Content** he produced in the meantime, and the pair would continue with  **Co-Production**. Here is A2’s summary after the first interrupt (from 1:43:57–1:44:17):

A2: “It already supports that <*selects constructor call*>”	}  Talking Pioneer }  Co-Production
A1: “Who?”	
A2: “The Reminder. There are indeed cases where it knows already that it’s a mirror.”	
A1: “No kidding? [...]”	

A2 **Pioneers** again during the second interrupt (from 1:45:11–1:46:31). Then, his summary consists of highlighting relevant parts of the source code, which, due to the pair’s high **Togetherness**, is enough to make A1 understand an issue found in the meantime:

A2: “<*selects propertyClassMap*> In the Reminders?”	}  Talking Pioneer }  Co-Production
A1: “Whoops!”	
A2: “Why is there a propertyClassMap? Completely unnecessary.”	
A1: “No idea. [...]”	

The summary after the third interrupt (from 1:47:00–1:47:17) is then more verbose again:

A2: “I don’t really think this is used in any way. Somebody just copied the ReminderAccessor from the other Accessor [...]”	}  Talking Pioneer }  Co-Production
---	--

## 9.5 Co-Production Mode

In  **Co-Production Episodes**, both pair members are involved in creating new knowledge, the **Target Content**. Their **Knowledge Want** is *collective* and the **Topic** is clear to both.

### 9.5.1 Properties of Co-Production Episodes

Although one pair member may make the **Topic** of her **internal Knowledge Wants** explicit such that it can become *collective* (as seen in Example 7.5), pairs may also enter a  **Co-Production Episode** without explicit discussion, as was illustrated in Example 7.13, where B1 and B2 were trying to execute a test script: Here, the pair had a high **Togetherness** due to its *shared plan* (which was: execute the test script before making changes to the production code), had a *good workspace awareness* (since both see the script fail and read the error message), and they have a *shared understanding of software development* (which leads them to similar hypotheses for the failure, such as network issues).

The **Target Content** is created by both developers and they **Maintain** their *shared understanding of the software system*—or in fewer cases: *software development in general*; most

▣ Co-Production Episodes have *S* Topics. Since their Knowledge Want is collective, they both recognize when the Target Content is created.

**Example 9.20: Co-Production with High Togetherness** (AA1, 11:19–11:45)

The pair chooses one faulty page to work on first (it is titled “Complete Editing”). The Topic of the ▣ Co-Production Episode below is ‘Which Java class corresponds to the displayed website?’ That Topic itself remains implicit. In fact, after their decision, A2 does not say anything and A1 does not type anything, yet their actions appear to come from ‘one mind’:

A2: “That means (!...!) (#Complete Editing#), for example.”

A1: “Shall we start with that one?”

A2: “Yes, I’d say so.”

A1: “Fine.”

A2: <\*switches to IDE, opens dialog ‘Open Type’.\*>

A1: “This is the (!...!)”

A2: <\*switches to web browser\*>

A1: “What is it called, the page?”

A2: <\*right-clicks in browser, chooses ‘Page Info’ in context menu\*>

A1: “(#FinishTasksPage#), ok.”

A2: <\*switches to IDE, types ‘FTP’ in ‘Open Type’ dialog, and selects ‘Finish TasksPage’ from the results\*>

▣ Co-Production

There appear to be different styles to structure a ▣ Co-Production Episode. The developers might follow the call-hierarchy (as A1/A2 did many times in AA1), or go through some piece of source code from top to bottom (as C1/C2 did in CA1, or J1/J2 in JA1). I did not analyze these different styles in depth.

### 9.5.2 ▣ Parallel Production Mode

Just like there are two types of ▣ Pioneering in which the pair is or is not Maintaining Togetherness by sharing Topic and Target Content, there is also a non-Maintaining form of how pair programmers deal with a collective Knowledge Want. I call it ▣ Parallel Production, because both partners seem engaged in some activities and may learn or understand something, but do *not* make sure they end up with a *shared* understanding.

Refer back to Example 6.23 where K2 and K3 both looked up how to programmatically simulate a key press event for an integration test. They shared a collective Knowledge Want and also the Topic was clear to both. K2, who controlled the mouse and scrolled through a library documentation, did learn the necessary idiom, but did not make sure his partner did, too (who also did not protest). About 40 minutes later in their session, K2 had used that idiom and K3 was puzzled by it, indicating their avoidably lowered Togetherness with regard to a *shared understanding of software development*.

In general, when there is little dialog, there are also only few cues to tell from the outside whether the pair has a shared understanding or not. What can be seen, however, is the lack of *explicit* activities to Maintain Togetherness. The pair may still be ‘in sync’, but there is no way of knowing. The next example illustrates this:

**Example 9.21: Together or Not?** (AA1, 1:23:27–1:25:13)

For two minutes, it is not clear whether the pair still has *one shared plan* and a *shared understanding of the software system*. They are not actively **Maintaining Togetherness**; this is an **Episode** of  **Parallel Production**. The transcript below begins with with A2 having an insight where in the backend the API needs to be amended.

A2: “Ah, it’s not Link but LinkChecker. <\*open source file\*> Ah, there is a test for that <\*open test file\*>”

A1: “M-hm <\*leans back, looks around office\*>”

A2: “<\*reads in test code\*> (#unreachableUrls#)”

A1: “<\*looks back at screen\*> hm hm hm hm hm”

A2: “<\*silent reading (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,) \*> (#testFetchUrls#) (#checkUrlsAreUnique#) (#unreachableForUrlRow#) (,,,,,,,,,)”

A1: <\*turns away\*>

A2: “Ah, here it’s testing unreachableUrls”

A1: “<\*turns back to screen\*> multiple times, right?”

A2: “<\*continues reading\*> Not here, not here, but here <\*reads test case names\*> (#testFailedUrlsAreReachable#) (#testInternalValidLinksIsNotUnreachable#) (#testLinksToInactiveObjectsFromInactiveAreNot#) (!!!!!)”

A1: “<\*turns away\*> M-hm, yes”

At this point, it seems that A1 has difficulties concentrating. After this excerpt, A1 proposes to finish the session, but A2 convinces him that “*there is not much left*” and they go on for almost another hour.

## 9.6 Push Mode

Unlike for all the other **Modes**,  **Push Episodes** deal with *external Knowledge Wants*: The **Propellor** keeps providing information until she is satisfied that her partner understood enough.

### 9.6.1 Properties and Context of Push Episodes

On an activity level, the **Propellor** of such an **Episode** may employ all types of **Explanations** (i.e., **Present New Fact**, **Refer to Common Ground**, and **Entice to Simple Step**, see Section 8.3) or present a piece of source code or some feature in a running application. The *partner*, then, also has a variety of different options for how to react to the **Propellor**’s actions:

- Backchannel utterances to keep the discourse flowing (see Section 3.2.1b, e.g., J1: “M-hm” in Example 9.1 or D4: “Yes” in Example 7.9).
- **Refer to Common Ground** to signal the **Propellor** she may ‘fast forward’ in her explanation (e.g., K1: “Right, and there are a thousand different cases and the documentation sucks” in Example 7.10).
- **Make Proposition** to summarizing one’s own understanding (e.g., D4: “That’s Swing?” in Example 7.9).
- Start a new  **Pull Episode** to inquire about a detail (e.g., J1: “In what time window are you looking?” in Example 9.1). I discuss the phenomenon of such **Sub-Episodes** in Chapter 10.

I already discussed the cues on which a developer may rely to infer a knowledge gap in her partner and develop an *external Knowledge Want* (Section 7.2.3) and how a **Propellor** may then actually start her explanations (Section 9.2.1b). There are a number of different contexts in which a  **Push** may be embedded, such as:

- A known or assumed knowledge gap in the partner (e.g., C5’s long-running **Push Episode** in the beginning of session CA2 on his recent changes which his partner cannot know about, the start of which can be seen in Example 6.1).
- A design discussion, during which an argument for or against a proposal is made (e.g., in Examples 7.7, 7.8, and 8.27).
- Making an assessment of the current state of the software, to make one’s reasoning explicit (e.g., in Example 7.11).
- In relaxed situations, with no pressing issue (“stable understanding”), with the time to reflect on recent problems that were due to a gap in G knowledge (e.g., in Examples 7.14, 7.15, and 7.17).

The context conditions for transferring knowledge (not just in **Push Mode**) and the overall trajectories of PP session are a complex topic on its own. I come back to this in Chapter 11.

### 9.6.2 Transfer or Construction?

Not all relevant **Target Content** is communicated directly through **Explanations**, but may also be constructed at the recipient. In Example 7.14, D4 provided imperfect explanations of what the Template Method design pattern is which D3 could nevertheless understand because of his own development experience: He knew which type of undesirable situations D4 was referring to and understood how the design idea D4 described can help with it.

The next example is another case in point: The **Propellor** noticed a (potential) problem, but instead of pointing it out directly to the partner, he asks a number of questions that superficially may look like a **Pull Episode** but are actually a series of **Simple Steps** (discussed in Section 8.3.3) to make the partner construct the **Target Content** himself:

#### Example 9.22: Socratic Issue **Push** (JA2, 18:38–19:43)

J1 is not content with J2’s API design and attempts to lead J2 to a better idea. Instead of the current string-based call `setEncoder("wmav2")`, he would favor an enumeration type such as `setEncoder(Format.WMA)`. Despite J1’s many grammatical questions, this is a **Push Episode**.

J1: “Based on which parameter do you want to (!...!) I mean, when you write ‘*setEncoder*’ of what? ‘*Encoder.getEncoderBy*’ and then what?” Entice to Simple Step (to failure)

J2: “No, listen. Here is how it works. You can get a list of all available audio encoders.”

J1: “<satisfied, expectant> Yes. And how do I know which one is the right one?”

Entice to Simple Step (to failure)

J2: “( . . . ) Erm, well. That’s (!...!) I mean, you can choose any. There is not ‘the right one.’”

J1: “Yes, sure, but do I take ‘any’? Or the first, or the fifth, or what?”

Entice to Simple Step (to failure)

J1: “I probably want to choose one that generates WMA files.” Refer to Common Ground

J2: “Exactly (!...!)”

J1: “That’s the WMA encoder ‘*wmav2*’.”

Refer to Common Ground

J2: “Yes, right, that’s just its name (!...!)”

J1: “Exactly, ‘*that’s just its name*’. And now you say that this name may change in later versions of `ffmpeg`, right?” Refer to Common Ground

J2: “Yes, we don’t know.”

J1: “‘*We don’t know*’, right, so we don’t want to get it by name, but by it being ‘*the WMA encoder*’. So it might be a good idea to (!...!)” Entice to Simple Step (to success)

J2: “An enum with a `String` in the constructor, yes. Hm.”

### 9.6.3 Push is not just the Inverse of Pull

The notion of  Push is not just looking at a  Pull Episode from the other end. In a  Pull Episode, the developer with the Knowledge Want keeps asking until she is satisfied or gives up; but in a  Push Episode, the presumably more knowledgeable partner keeps providing information as long as *she* sees fit.  Push may be the *Mode* that makes pair programming difficult for inexperienced pairs. I want to explain why I think so in this section.

In solo programming, the  Pioneering Mode to deal with internal Knowledge Wants is presumably the most prevalent. Depending on the environment of the solo software developer, she might be able to ask nearby colleagues for information or for help (possibly resulting in  Pull or  Co-Production). For a  Push Episode, however, another developer needs to develop an *external* Knowledge Want, for which there needs to be enough opportunity for cues to pile up (see Section 7.2.3). External Knowledge Wants and  Push Episodes may therefore be unique to pair programming sessions longer than just a few minutes.

On the one hand, the occurrence of  Pushes can be an advantage, because the former solo developer now has a partner to provide her with information she did not even know she needed. On the other hand, the partner might misinterpret some cues and provide unwanted information. In  Pull Mode, a developer-in-need cannot do much to *propel* the Episode if her partner does not cooperate (as C5 did in Example 7.16 by turning away from C2). This power balance is reversed in  Push Mode: The partner may have difficulties *stopping* the Propellor from explaining.

One way a  Push Episode can be unwanted is a lack of **necessity**, meaning that there is, in fact, no knowledge gap and the partner already knows everything the Propellor is going to explain. An *unnecessary*  Push Episode irritates the partner, because the Propellor apparently ‘violates’ Grice’s second maxim of Quantity and the maxim of Relation (“*Do not make your contribution more informative than is required*” and “*be relevant*”, see Section 3.2.1b). The partner may start to think about the reasons for the Propellor’s behavior: A simple explanation would be that the Propellor underestimated the extent of the *common ground* and has the intention to *Present New Facts*. But if the Propellor explains something by *Referring to Common Ground*, then there is presumably a particular point to the explanations *beyond* transferring knowledge. It is irritating for developers to think there might a ‘true purpose’ but not being able to understand it, as the next example shows:

#### Example 9.23: Why Push? (PA3, 29:53–31:37)

Developers P1 and P3 just extracted multiple occurrences of the value `0.01` that is used in several percentage calculations into a new constant called `OFFSET_PERCENTAGE`. P3 just successfully ran the test suite when P1 starts a  Push Episode that massively irritates P3.

P1: “It’s important to make clear that the last two `0.01` have no relationship. Because they might have no relationship and someone comes along and says ‘*Look, it says `0.01`*’ (!...!!)”

P3: “Which last two?”

P1: “The last two in lines 31 and 32, for example. Assuming the two numbers would have no relation and someone who only sees the implementation with raw numbers thinks ‘*Oh, there is a relation, I’ll introduce a constant*’. And then another comes along and introduces it everywhere. Now all have the same relation. Now you know that they should explicitly be converted this way.”

P3: “If it’s the same relation, you can treat them as such. You adapt it the moment it changes.”

P1: “Yes, but the one seeing the code doesn’t know when you have only raw numbers, with the same values. What about 3660?”

P3: “When did we ever have 3660 as a percentage?”

## Example 9.23 (continued)

P1: “Or 3600! With 3600 it’s an example. That’s the conversion from hours to minutes, but also from seconds to minutes. Depending on the context two identical number can mean two completely different things.”<sup>a</sup>

P3: “But applied to our case this has no relevance.”

P1: “Yes, it has. Because it is a Magic Number, and Magic Number means (!...!!)”

P3: “But it is no longer ‘magic’. We just named it.”

P1: “<annoyed> Yes, we named it because it now creates a relation between these individual numbers. Before, it was not clear (!...!!)”

P3: “I don’t understand what you want, right now.”

P1: “I wanted to explain why we are doing this (!...!!)”

P3: “<annoyed> I got that.”

P1: “Good. It’s alright then.”

P3: “<nervous laughter> I tried to understand what you still wanted to change.”

P1: “Nothing. I didn’t want to change anything.”

P3: “<relieved> Ok.”

P1: “I only want to clarify that it’s important to (!...!!)”

P3: “<annoyed> Got it.”

P1: “make the relation with this renaming.”

P3: “<annoyed> <\*stares at screen\*> So.”

P1: “Not only to rename the variable.”

P3: “<annoyed> It’s ok.”

P3 understood that P1 was  **Pushing** but deemed himself already knowledgeable concerning ‘Removing Magic Numbers’ (which is **G knowledge** of type *Design and Programming Patterns*, see page 249). He also assumed that P1 could not mean his explanations as **New Facts** (because they talked about this already), so he considered them as **Referring to Common Ground** or **Enticing to Simple Step**. He did not understand, however, to which end. And that irritated him.

After the session, P1 said, he kept explaining because he was missing an affirmative signal that P3 got the message. P3 said, he did not give any, because he was waiting for the punch line, the ‘ingenious code change’ which P1 was about to propose, but there was none. (See also evaluation discussion in Chapter 13).

<sup>a</sup>P1 is mistaken. Of course, both the conversion from seconds to minutes as well as from minutes to hours has a factor of 60, not 3600.

## 9.7 Summary and Discussion of Related Work

The main concept of this chapter is the **Episode**: One developer pursues either an **internal** or **external Knowledge Want** in  **Push**,  **Pull**, or  **Pioneering Mode**, or both partners engage in  **Co-Production** to satisfy their **collective Knowledge Want**. The **Episode** concept offers a higher level of abstraction to talk about the process structure of a PP session than the utterance/activity level of the previous chapters. They cover longer time frames than individual activities (up to several minutes, although most **Episodes** are shorter) and capture *pair* behavior rather than just an individual developer’s intentions.

The idea of conceptualizing different ways of how pairs interact—as the *Modes* do—is also used by some other researchers:

- Walle & Hannay (2009, Sec. 3.3.3, discussed on page 73) distinguish different pair programmer *interaction patterns* that are conceptually similar to the notion of a *Mode*:
  - In *consensual interaction*, only one developer contributes substantively, while the other follows along—this may be both  *Pushing* or  *Pioneering*.
  - *Stonewalling* and being *non-responsive* is not an extra *Mode* for me, but is captured by the notion of *ignored Episodes*.
  - *Cross-purpose* may be represented by  *Parallel Production* or by two concurrent *Episodes*.
  - *Responsive* and *elaborative* interactions can be  *Push*,  *Pull*, or  *Co-Production*.
- Flor (1998, my discussion on page 80) also uses the terms “*push*” and “*pull*”, but he uses them as a generalization of writing/speaking and reading/listening, respectively, and are thus not to be confused with the  *Push* and  *Pull Mode* presented here.
- Okada & Simon (1997, pp. 132–136, my discussion on page 96) speak of a pair that “*co-constructed new knowledge*”. Based on the 33-line transcript, the pair’s interaction could be characterized as a  *Co-Production Episode*. The authors do not provide further terminology, which is not surprising given the  *Produce* nature of their task: The pairs had little relevant pre-existing knowledge they could  *Push* or  *Pull*.

The idea to structure a (pair) programming process on an episode-like level is also used by some other researchers. Their concepts compare to my notion of an *Episode* (i.e., the pursuit of a *Topic* in a constant *Mode*) as follows:

- Both Kissinger et al. (2006, Sec. 3.1) and Xu et al. (2005, Sec. 3.6, my discussion on page 77) divided their transcripts into “*stanzas*” and “*episodes*”, whenever there are “*shifts of topic*” and the developers “*start the discussion of a different concept*”, respectively. This leads to data segments that are coarser than my *Episodes*, since changes in the conversational *Mode* are not considered.
- Walle & Hannay (2009, Sec. 3.3.2, my discussion on page 73) divided their audio recordings into “*interaction sequences*” using on a similar, topic-based approach, but also considered changes in the “*interaction pattern*” (e.g., from *consensual* to *stone-walling*). The resulting segmentation is probably comparable, since, as mentioned above, their *interaction patterns* bear some resemblance with my *Modes*.
- Kubelka et al. (2018, Sec. 5) state they adopted my *Episode* notion (as published in Zieris & Prechelt, 2016) to structure and qualitatively analyze recorded sessions of developers working on tasks with instant feedback while programming. However, it is not clear whether they considered both *Mode* and *Topic* changes.

All of the above studies speak of data segmentation, which is *not* what my *Episodes* are about. First, there are gaps between my *Episodes* because pair programmers do not always transfer knowledge. Second, each partner may simultaneously pursue their own *Topic*, and there might be higher-level *Topics* that are pushed back for a while, but should be reconsidered later. A linear segmentation of a programming session is not adequate for this. I discuss the matter of concurrent *Episodes* and *Sub-Episodes* in the next chapter.



## Chapter 10 Patterns of Episodes

---

10.1 Purpose and Structure of this Chapter . . . . .	303
10.2 Anti-Patterns . . . . .	304
10.2.1 Branching Wildly . . . . .	304
10.3 Positive Patterns . . . . .	307
10.3.1 Return Explicitly . . . . .	307
10.3.2 Scope Limiting . . . . .	310
10.4 Summary and Discussion . . . . .	314

### 10.1 Purpose and Structure of this Chapter

Pair programmers’ **Knowledge Wants** are often not isolated, but appear to be connected. The different ways how pairs deal with multiple **Knowledge Wants**—e.g., acting on them and start a knowledge transfer **Episode**, to postpone addressing them to a later point in time, or to ignore them—are the topic of this short chapter.

I introduce two simple concepts here to distinguish two cases that are common when a pair shifts their attention from one **Topic** to another: **Sub-Episodes** and **Catalyzed Episodes**. When a pair member develops a new **Knowledge Want** during an **Episode** which the developer feels should be addressed first *in order to* satisfy the other **Knowledge Want**, she starts a **Sub-Episode**. The **Topic** of a **Sub-Episode** is not necessarily a sub-**Topic**—the **propelling** developer just thinks it should be addressed *first*. A **Catalyzed Episode** is triggered by something the partner did or said, or that was visible on the screen (analog to the base-layer notion of a catalyzed *finding*, see page 134). While **Sub-Episodes** are more like sub-routines in programming, **Catalyzed Episodes** are like interrupts.

**Example 10.1: Discussion of Recurring Example: Multiple Wants** (JA1, 04:15–06:15)

The explanations in J2’s long-running **Push Episode** trigger an **internal Knowledge Want** in J1. He does not expect his partner J2 to address the issue on his own and it appears relevant enough to J1 to be addressed right away before J2 continues with any further explanation: J1’s question “*In what time window are you looking?*” starts a **Sub-Episode** in **Pull Mode**.

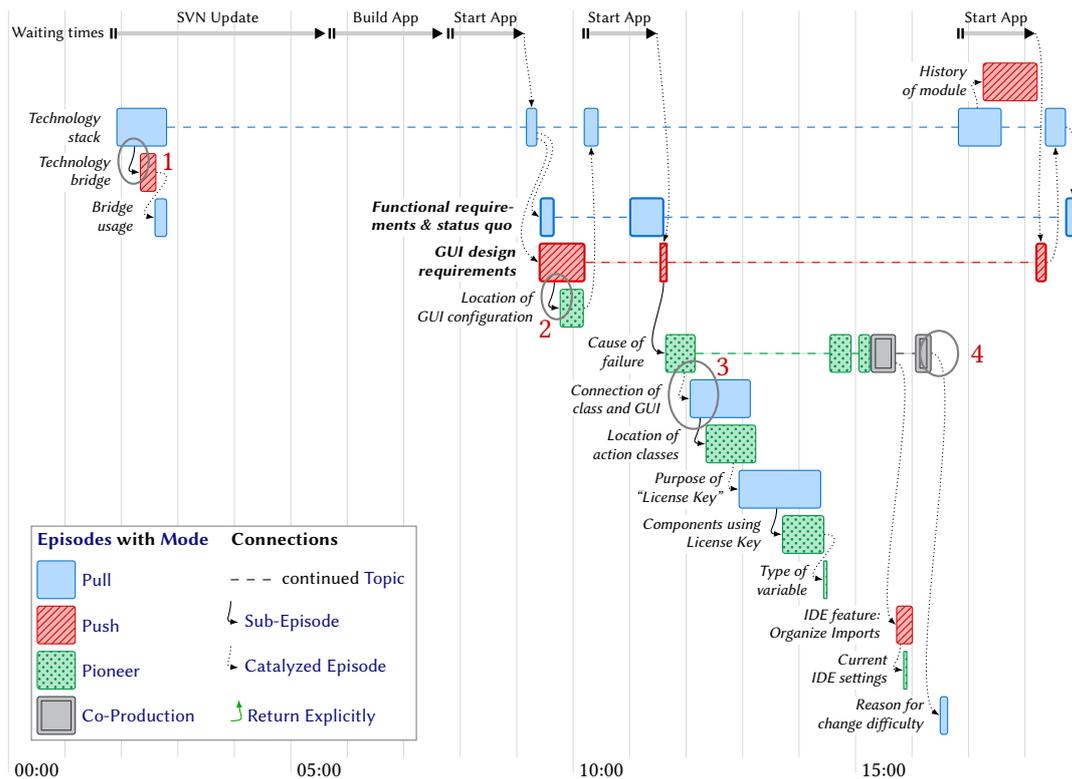
There is no clear-cut line between **Catalyzed** and **Sub-Episodes**: After all, the **propelling** developer considers the new **Topic** to be relevant *enough* to be shared with the partner in both cases. I may be going out on a limb here, but I characterize a **Topic** switch as **catalyzed** when I got the impression that the **propelling** developer would agree with ‘*OK, I brought this up mostly because it just occurred to me, not because it’s necessary right now.*’ One indicator is the seriousness with which the new **Topic** is announced: D4 chuckles when he sees a `LicenseKey` (see Example 8.4); A1 asks about the `TclCode` with some amusement (see Example 7.3).

Neither **Catalyzed** nor **Sub-Episodes** are inherently good or bad: Pair programmers will have insights or make observations during their session and some of them will lead to **internal** or **external Knowledge Wants**, which potentially result in the transfer of relevant knowledge, regardless of whether the pair jokes about the **Topic** or not. Nevertheless, pairs or pair members still exhibit behavior that has positive or negative consequences. I first discuss the case of pair **D3/D4** who kept opening new **Topics** without the previously started ones being brought to a satisfactory end in Section 10.2. In Section 10.3, I characterize two positive patterns: **Return Explicitly** from **Sub-** or **Catalyzed Episodes** and **Scope Limiting**, i.e., not going off on tangents in the first place.

## 10.2 Anti-Patterns

### 10.2.1 Branching Wildly

In my data, one pair stuck out in terms of how many new **Episodes** they began without the already started **Topics** being fully dealt with. The first 19 minutes of session **DA2** are an extreme case of nested **Episodes**, which I call **Branching Wildly**. The case is too complex to be discussed here in full. Instead, I provide a schematic representation of the **Episode-level** structure of these minutes in Figure 10.1 from which I will discuss four exemplary shifts between **Topics**.



**Figure 10.1:** Episode structure of the beginning of session **DA2**. Numbered ellipses (1 to 4) refer to sections in Example 10.2.

**Notation:** Each row represents one **Topic** the pair deals with in either ■ **Pull**, ▨ **Push**, ▤ **Pioneer**, or ■ **Co-Production Mode**. The **Episode** boxes are connected, reflecting how the pair came from one to another (**Catalyzed** or **Sub-Episode**, or picking up a previous **Topic**). Elements pertaining to the pair’s main **Topic**—here: their task requirements—are set in bold.

**Example 10.2: Long, Winding Way to Understanding Requirements** (DA2, 01:54–18:57)

Overall, developer D3 wants to explain the task *requirements*, and D4 also wants to understand the system’s status quo. These two related issues form the ‘main’ Topics of the first minutes of their session (set in bold in Figure 10.1). D3 wants to illustrate the target state of the implementation by loading an older, but functionally complete version of the calendar module. The switch is more difficult than anticipated and while the pair attempts to figure that out, they fall into a rabbit hole. Since both pair members develop and pursue multiple additional Knowledge Wants along the way, they go on and off on their main Topic for almost 10 minutes. I do not discuss all digressions, but focus on four moments to highlight both necessary (1 and 2) and avoidable detours (3 and 4). See Figure 10.1 for context.

**1. Sub  Push in Order to  Pull**

In addition to the task requirements, D4 also wants to understand the technological basis of the product, which his partner D3 cannot say much about. Conceptually speaking, D4 asks whether technology A or B is used. D3 knows there is no simple answer to this and thus starts a  Push Episode to clarify that technologies B and C are used with a special B-C connector in-between (see also Example 7.9).

D4: “How is this implemented in general, because (!...!) is it more of an SWT user interface than Eclipse, or what?”  Pull

D3: “Well, <\*> product name\*> as such is based on SWT.”

D4: “Yes”

D3: “The calendar, there we use this (~) calendar component.”

D4: “That’s Swing?”

D3: “Nope, yes, so actually AWT.”

D4: “Ah, ok. AWT even.”

D3: “Yeah, there is this SWT-to-AWT container gizmo and that’s how it’s embedded in the end. How this SWT-to-AWT thingy works I can’t tell you much about.”  Sub-Episode  Push

D4: “OK. [...]”

**2. Sub  Pioneer in Order to  Push**

D3 wants to swap in an old version of the view part to demonstrate the GUI requirements. For this to work, he needs to find the place in the module configuration where the view is loaded and starts a  Pioneering Episode.

D3: “I’ll show you how it looked in the old calendar. <\*> closes application\*> I’ll show you where we’re heading. <\*> navigates to configuration of calendar view\*>”  Push

D3: “<\*> skims configuration dialog\*> Erm, erm. Just need to check where the menu items were”  Sub-Episode  Pioneer

D3: “<\*> opens list of configured navigation buttons, selects first entry\*> Exactly (#calendar#) nope, wrong. [...]”

The  Pioneering Episode continues for another 16 seconds. After the modified application is started, D3 notices it does not show the old version as he expected.

**3. Catalyzed  Pull interrupting  Pioneer**

While D3 is looking for the reason why the old version is not properly loaded, D4 notices something in the source code and asks his partner about it. It has nothing to do with the failure, but D3 nevertheless stops his search and deals with D4’s question, which he cannot answer right away. Instead of letting it be, he goes off track and starts another  Pioneering Episode to satisfy D4’s Knowledge Want.

## Example 10.2 (continued)

D3: <\*clicks through the configuration (,,,,,,)\*>

D4: “What are these navigation things for? Are these Actions? Where are they displayed?”

D3: “They are processed by this class, and then the according method is called.”

D3: “There is a (!...!) what’s it called again? <\*navigates through package tree\*> No, not here, but [...]”

☒ Pioneer (c’d)

Catalyzed

☐ Pull

Sub-Episode

☒ Pioneer

D3’s last ☒ Pioneering Episode continues for 18 seconds until the same pattern of D4 asking a question (Catalyzed: ☐ Pull) and D3 looking up things (Sub: ☒ Pioneer) repeats again, leading deeper into the rabbit hole at the end of which D3 wonders “OK. Now, where were we?” Fortunately, the failure’s stacktrace is still displayed on their screen, D3 sees it (“Ah, the exception, right.”), and picks up his ☒ Pioneering Episode from three minutes earlier.

#### 4. Catalyzed ☐ Pull after finished Topic

The pair eventually continues their investigation in ☐ Co-Production Mode and understands the failure (changing the configuration to the old class was not enough; there were also numerous static references in the source code that needed to be changed, too). D3 decides to *not* swap the view, but D4 is now curious why the swap was so difficult and starts a Catalyzed ☐ Pull Episode.

D3: “<\*hovers error icons on multiple files in package explorer\*> Ah, I see.” ☐ Co-Production

D4: “<laughs> OK.”

D3: “Hell no, I won’t go through all this. I’ll show you somewhere else.”

D4: “But, how is it [...] Did you already change this on your machine, or what?”

Catalyzed

☐ Pull

D3: “Well, you see, this is the old version, which I restored. [...]”

#### Discussion

From start to finish, it took the pair almost 10 minutes to transfer the Target Content for the “Requirements” Topic. The net time, however, is only 30 seconds, which already includes some clarification and discussion (see Example 7.6 on page 245). Figure 10.1 shows eighteen additional Episodes during that time, six of which were ‘necessary’ (i.e., the Sub-Episodes of D3 trying to figure out how to swap in the old version of the view and both trying to figure out why it failed) and twelve were ‘optional’ (i.e., the numerous Catalyzed Episodes and their Subs).

Considering session DA2 was in part meant to introduce D4 to the software system, even the ‘optional’ Topics possibly have some value for D4. So, what did (or could) D4 learn through them? What were the outcomes of all these Episodes (as introduced in Figure 9.1)?

- The eight ☒ Pioneering Episodes were all propelled by D3, who was too occupied to thoroughly explain what he was doing. It is unclear how much D4 understood during these, but it was apparently enough to catalyze three of his ☐ Pull Episodes.
- All nine ☐ Pull Episodes were propelled by D4, but only the last of the three pertaining to the main Topic was eventually *successful*. The others were *postponed* or ended either in *resignation* or as a *partial success* because D3 did not understand the Topic or could not provide the Target Content.
- The two ☐ Co-Production Episodes were eventually *successful*, but the Target Content became obsolete after D3 decided to revert his code changes.
- Both pair members each had one ☒ Push Episode: D3 *successfully* explained the history of the module to D4; D4’s only ☒ Push to D3 was *unnecessary*.

In summary, the Catalyzed Episodes probably were of quite limited value for D4. Additionally, although I cannot know how long it would have taken D3 to get back to the ‘necessary’ Topics had the stacktrace *not* been visible (“Ah, the exception, right”), such branching behavior arguably might throw other unexperienced pair programmers off the rails.

The above discussed first 20 minutes of session DA2 are exceptional. First, the remaining 2 hours of this session, that is, once the task was defined and the pair started the implementation, the Topic ‘nesting level’ is shallower: There are some **Co-Production Episodes** during which D4 starts a **Pull Episode**, but not much more. Second, no other analyzed PP session had such a branched **Episode** structure. See Figure 10.2, for example, which shows a similar time span from the beginning of session JA1. Here, J2 basically has one long-running **Push Episode** (displayed in the figure as multiple **Pushes**, each **catalyzed** by finished one) and J1 would do the occasional **Sub Pull** for details.

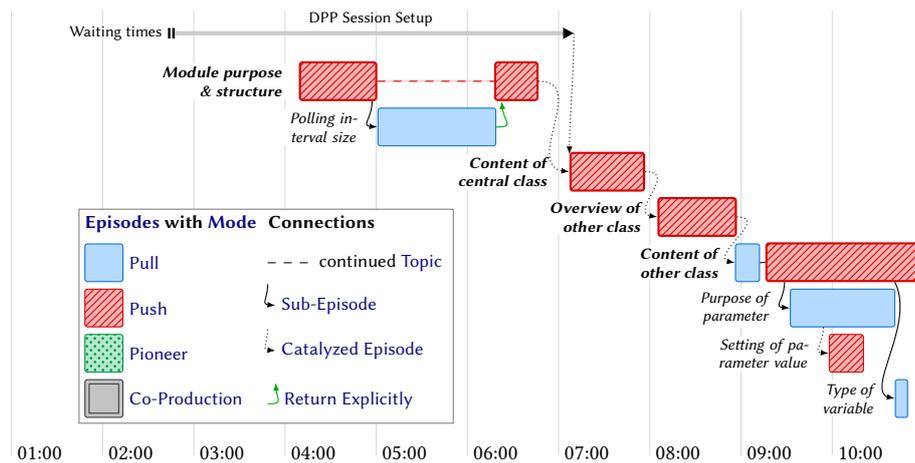


Figure 10.2: Overview of the beginning of session JA1. Notation is explained in Figure 10.1.

The reason for this difference might be that this was not only D4’s first week at the company (so he had many questions about basically everything), but session DA2 was also his first pair programming encounter ever. According to the pre-session questionnaires, his partner D3 also only started with occasional PP when he joined the company three months before.

## 10.3 Positive Patterns

I identified two behavioral patterns which are positive in the sense that the pairs deal with multiple **Knowledge Wants** without getting lost. One is to **Return Explicitly** at the end of a subordinate **Episode**, the other is **Scope Limiting** in order to keep the complexity manageable in the first place.

### 10.3.1 Return Explicitly

To **Return Explicitly** means to finish a subordinate **Episode** by explicitly referring back to the original **Topic** verbally or by explicitly returning the ‘control’ to the partner to continue **propelling** her original **Episode**. Pairs appear to do this when the **Sub-** or **Catalyzed Episode** was meant to be short, such as a brief question about some detail, but turned out to take much longer, e.g., because the partners had some misunderstanding. In Example 10.3, the supposedly simple question for the length of a polling interval took 1:20 minutes to answer. Both partners appear to have a need for closure and refer back to the main **Topic**:

**Example 10.3: Return After Finished Sub Pull** (JA1, 04:08–06:33)

In the first minutes of the session, J2 propels a long-running Push Episode which serves as the context for J1's Sub-Episode in Pull Mode. Here are the relevant utterances as a reminder (see Example 5.1 for the full exchange with the same line numbers):

- |   |  |
|---|--|
| <p>(8) J2: “The way this works is that there are multiple processors, so there is the central plugin and multiple processors which each handle one wave.”</p> <p>⋮</p> <p>(16) J1: “In what time window are you looking?”</p> <p>⋮</p> <p>(29) J1: “30 seconds, that’s what I wanted.”</p> <p>(30) J2: “That’s 30 seconds long the time window. Now I got you.”</p> | <p><span style="background-color: #f0f0f0; border: 1px solid #ccc;">Push</span></p> <p>Sub-Episode</p> <p><span style="background-color: #e0e0e0; border: 1px solid #ccc;">Pull</span></p> |
| <p>Immediately afterwards, both developers explicitly work their way back to the original Topic (see also their diagram in Figure 10.2): First J2 alludes to the source code he is about to show, then J1 summarizes his understanding and repeats bits of information from J2's original explanation (↑ 8).</p>  |  |
| <p>(31) J2: “I can show it to you in a minute.”</p> <p>(32) J1: “Yes. And the NewsPlugin is doing what in all of this? Does it do exactly this monitoring and the delegation to the individual wave plugins, or what?”</p> <p>(33) J2: “No. The NewsPlugin basically only does (!...!) it gets called periodically by the cron server [...].”</p>                   | <p>Return Explicitly</p> <p>Return Explicitly</p> <p>(back in <span style="background-color: #f0f0f0; border: 1px solid #ccc;">Push</span>)</p>  |

In the beginning of session CA2, the main Episode was ‘on halt’ for 1:23 minutes and Catalyzed Pulls led to confusion which neither partner wanted to resolve, so they cut them short and Returned to the main Topic.

**Example 10.4: Return After Resigning a Catalyzed Pull** (CA2, 10:07–11:58)

In the beginning of the session, C5 explains his recent code changes to C2 in a long-running Push Episode, but gets interrupted by C2 multiple times. This is the Episode diagram of these first minutes of their session:

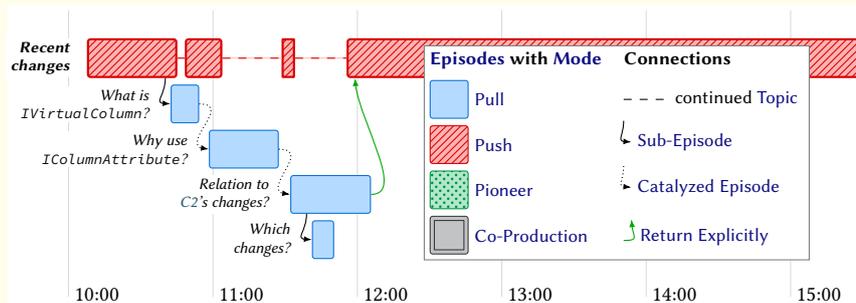


Figure 10.3: Overview of the beginning of session CA2. Notation is explained in Figure 10.1.

The Push Episode starts as follows:

C5: “Well, I better show you what I did already. <\*opens IFeature AttributeConfiguration\*> What I did [...] is extend this IFeature AttributeConfiguration with an IVirtualColumn [...]”

C2 asks detailed questions multiple times (Pull Episodes). The first of these starts a 12-second Sub-Episode relating to C5's concrete code changes (discussed in Example 9.12). Then come two Catalyzed Episodes relating to the overall design idea (discussed in Example 9.2).

## Example 10.4 (continued)

As shown below, these leave C5 rather puzzled and he starts another Sub-Episode asking for what C2 means. Both developers realize now that this exchange is not going to be very helpful and they both Return Explicitly to the original Push Episode:

C5: “I don’t know what you are talking about right now.”

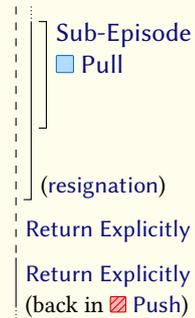
C2: “About what I did, with the GUI.”

C5: “You only had a VirtualAttribute?”

C2: “(…) I thought that we use this for the data structure, what I did. (Anyway, ok.)”

C2: “(But show me.)”

C5: “We can go on, for now.”



The sequence of Catalyzed Episodes can be considered the beginning of Branching Wildly, but both developers became aware of their problematic situation and returned to the main Topic. Note that neither developer explicitly says *what* they should go on with, but both intend to make C5 continue propelling his Push Episode. This indicates their relatively high Togetherness, in particular regarding having *one shared plan*.

The next example illustrates how returning even from a short Sub-Episode requires some orientation to get back to the main Topic:

## Example 10.5: Easy Return from Short Sub Pull (EA1, 04:23–13:45)

In the beginning of session EA1, developer E2 explains to his colleague E1 what he already knows about a display error of route segments on a map (long-running Push); E2 asks two questions (Pull Sub-Episodes) from which E1 Returns Explicitly:

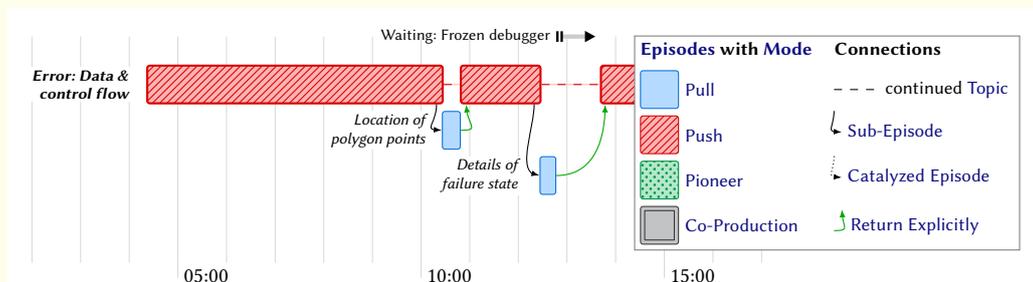


Figure 10.4: Overview of the beginning of session EA1. Notation is explained in Figure 10.1.

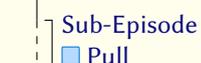
The software is running in debug mode, execution is halted at a breakpoint. E2 switches between source code and GUI to explain the observable failure and related code segments:

E2: “I’ll show you what I did […] The error is this […] you see, the last segment has an extra point. (!!Yep!!) The last point should have been this one, but it takes that one. <\*hovers two points in GUI\*> (!!Yes!!)”

E2’s Push Episode continues in this fashion for five minutes: Mostly a monologue, carried on by E1’s backchannel utterances. Then, E1 starts two short Sub-Episodes after each of which E2 Returns Explicitly to his original Topic.

E2: “<\*looking at source code\*> result is the point count […] which gets increased here. <\*opens inspector\*> And now it’s 131. Before it was 1, I guess that’s the start point. (!!M-hm!!) And then 130 were added. And now they are in this pPoints, there <\*opens inspector\*> they are.”

E1: “The polygon points of the route, they are still in there?”



## Example 10.5 (continued)

E2: “Yes, exactly. This `TraceFerry()` has an output parameter where the points get copied to. (!!M-hm!!) It’s called multiple times. It’s a big array, first for the stub, and later it says ‘*copy starting here*’. (!!M-hm!!) And in `TraceFerry()` they get copied.”

E1: “OK.”

E2: “Exactly. And, erm, now in this `pPoints` array, there are the points.”

E2: “The last point should be correct now. Or it’s not. We’ll see about that. [...] <inspects values> Yes, it is the one, it’s doubled. So, the error is somewhere in `TraceFerry()`. [...]”

E1: “So you say, it’s actually (!..!) we break the route (!..!) I mean <takes mouse> it goes here, goes here, goes back again, and takes that as the endpoint, or what?”

E2: “Exactly, it got all these points first, then this one, then that one instead of this.”

E1: “Yes, and goes back again. OK.”

E2: “[...] I first thought it goes wrong here, but it’s not. Instead it’s `TraceFerry()`, where it comes out wrong.”

After they reached the limits of E2’s existing understanding, the pair then continues in  Co-Production Mode, debugging the source code together.

Return Explicitly  
(back in  Push)

Sub-Episode  
 Pull

Return Explicitly  
(back in  Push)

### 10.3.2 Scope Limiting

**Scope Limiting** means to *not* address some **Topic** now—and possibly never. The pair may actively decide against starting a **Sub-** or a **Catalyzed Episode**. In successful **Scope Limiting**, both partners (a) appreciate the **Knowledge Want** and understand the **Topic** of the would-be **Episode**, and (b) reach an agreement on what to do about the **Knowledge Want**.

Refer back to Example 7.3 for a case where both partners quickly agreed on not addressing the **Knowledge Want** at all: While A2 performs some code changes, A1 notices a reference to the test code in the production code and develops an **internal Knowledge Want**. He starts a **Catalyzed  Pull Episode** by asking his partner about it (“*TclCode?*”). A2 immediately understands the **Topic**, and could possibly even transfer the **Target Content**, but performs **Scope Limiting** instead which A1 accepts: “<grinning> You don’t wanna know”—“Ok <snorts> (...) it’s all fine then”.

Similarly, the pair in session JA1 agrees to not start a **Sub-Episode** while going through the source code together:

#### Example 10.6: Negotiating the **Scope** (JA1, 13:15–13:43)

J1 is still getting an overview of the long method, so he gladly accepts the option to *not* go into too much detail.

J2: “And it sets the `remoteNewsFile`, which comes out of the function `getLastFile()`. That’s one function we may replace later on. [...] Shall we look into the function, or not?”

J1: “No, not now, please.”

Scope Limiting

J2: “Not now, ok.”

Scope Limiting

The remaining session is about the control flow inside a single class, in which that function has no role. The pair never considers it again.

While J1/J2 were fine with postponing some **Knowledge Want** to a vague point in the future, the pair C3/C4 employed **Scope Limiting** throughout the session, which was one of the reasons why they stayed ‘focused’ during their **Focus Phases** in session CA5:

**Example 10.7: Limiting Scope for Focus** (CA5, 19:12–20:11, 47:11–47:22, 1:21:57–1:22:20)

The first instance of **Scope Limiting** in **Focus Phase #1** is when the pair decides to *not* think about parameter validation now (see Example 6.15 for the complete **Focus Phase** with the same line numbers and technical context):

- (6) C3: “<\*cursor along line 32\*> Probably, this again turns out to be (.) it depends (!...!) (#validator#)”
- (7) C4: “Could be we need it, and then we can get it back anyway.” Scope Limiting
- (8) C3: “Yes, would get rid of it. <\*deletes lines 32–37\*>”

The second instance is when they ponder which **GeometryType** to use and decide to leave a **TODO** comment to deal with it later.

- (24) C3: “That remains to be decided whether we (!...!)”
- (25) C3: “Well, we can simply use POLYGON for now.” Scope Limiting
- (26) C4: “Let’s start with POLYGON.” Scope Limiting
- ⋮
- (30) C4: “Maybe leave TODO\_NOW here [...]”

Overall in their session, the pair writes *nine* **TODO** comments (the first at 05:46, the last at 1:10:16). Three of these are beyond the scope of their current task, but the pair completes the other six **TODO**s within their session (two immediately after writing the comment, and the others after 3, 14, 16, and 44 minutes, respectively). Conceptually speaking, these **TODO** comments are the manifestations of **postponed Topics**, which developers understood but did not want to address right away.

**Discussion**

This behavior is exceptional. No other pair in my data wrote nearly as many **TODO** comments, and no other pair picked up their **postponed Topics** with such consistency. The pair itself also reflects on their usage of **TODO** comments. Before writing the fifth **TODO** comment, C3 explains his motivation:

- C3: “Would you mind adding a **TODO\_NOW**? ‘Release lock here?’ Simply because (!...!!)”
- C4: “Not at all <\*starts writing the todo comment\*>”
- C3: “Then the information is off of my mind.”

In the last minutes of the session, the two reflect on their behavior and understand that their **TODO** comments were a symptom of them working as a *pair* and having to reconcile different development approaches:

- C4: “You leave a lot of **TODO\_NOW**s. I like to round off things, be safe, and go the next point.”
- C3: “M-hm. That’s interesting, because, had I worked alone on this, I would have completed things more often before moving on.”

The pair experienced the difficulty of **Maintaining** one shared plan and they discovered writing (and revisiting) such **TODO** comments in particular and **Scope Limiting** in general as an effective way to **Maintain Togetherness**.

In other cases, the [Sub-](#) or [Catalyzed Episode](#) may have already begun, and the pair decides to cut it short, as in session [BB1](#), where a technical decision made a [Topic](#) obsolete:

**Example 10.8: Limit Scope of Partner’s Pioneering** (BB1, 16:47–19:06)

As a starting point, the pair already copy-pasted about 45 lines of existing HTML template code to the location of the new feature. They are now in the process of determining which copied parts to delete and which to keep (eventually, just two lines of code), for which they need to understand the correspondence of HTML tags to visible GUI elements:

B2: “Whatcha thinking?”

B1: “What ends up where. `<*hovers template code*> (#slimcolbox#)`”

B2: “H2 is this nice red hue `<*looks at printout*>`”

B1: “OK, then this is the second heading `<*looks at printout, too*> [...]`”

Co-Production

The pair continues their [Co-Production Episode](#) for 1:20 minutes. Rather than just deleting the unnecessary code, B1 started to comment it out. He already opened a multi-line comment (`<!--`) in the third line of the pasted code and he takes some time in a [Pioneering Episode](#), scrolling, reading, and reformatting the (commented out) code to find the right place to close the comment (`-->`):

B1: “Where does it close that `div` tag?”

B2: “Here! `<*points to the last line of the copied code, which has two closing </div> tags*>`”

B1: `<*slowly moves cursor between two closing </div> tags, opens another comment with “<!--” [...]*>`

B1: “It only clears the inner one, right? [...]”

Sub-Episode  
Pioneer

B2 proposes to not ponder the inner structure of the obsolete code, but to disregard it altogether, which is [Scope Limiting](#) by cutting B1’s [Pioneering](#) short:

B2: “Just kick the whole block.”

B1: `<*undoes changes, then comments out the whole block*>`

Scope Limiting  
Scope Limiting

After validating the visual appearance in the frontend, B1 then deletes the commented out block about a minute later.

In all of the above cases, the pair members appreciated the [Knowledge Want](#), understood the [Topic](#), and agreed on whether or not to [Limit their Scope](#). The next example, however, shows a failed attempt by J1 to stop his partner J2 from pursuing a [Topic](#) that seems to be [catalyzed](#) and thus unnecessary from the perspective of J1.

**Example 10.9: Trying to Limit Scope of Partner’s Push** (JA1, 58:23–59:32)

J2 wants to explain a peculiarities the current design of the system which loads audio files from multiple remote systems and then processes them. The [Topic](#) could be phrased as ‘*Why does the data processing method have an additional boolean argument?*’ and the complicated [Target Content](#) could be phrased like this:

Usually, downloaded files are deleted after processing. However, two particular radio stations are flaky: They may not have the requested data when they are queried. Then, a fallback system will automatically provide a default news segment. Separate queries to the two flaky systems may thus return the *same files* in case they come from the fallback. Downloading files over the legacy network connection is slow. The solution: Files coming from the fallback system are cached to avoid downloading the same files multiple times (boolean argument `false`); files coming *directly* from the flaky systems should be deleted (boolean argument `true`).

J2 originally explained this [Target Content](#) as follows:

## Example 10.9 (continued)

J2: “Where should I start explaining this? It’s more complicated than you’d think.”

J1: “Doubtless.”

J2: “Because if <\*> or <\*> don’t have their own news, if they failed somehow, then they use these <\*> news. And depending on whether it’s their own, if that construct back here is true <\*>, then it is their own files, then it should delete them. If it’s <\*> news, it should of course not delete them, because there may be others who need them, too. You see? But that could all be done a bit different later (!...!!)”

J1: “<\*> Different kettle of fish. A different kettle of fish!<sup>a</sup> That is (!...!!)”

Scope Limiting (not successful)

J2: “Yes, there is another kind of fish! I did that only because the downloads took so long of those stupid news files. So I thought it need not take even longer so I’d use the same for them all. Of course, if it runs on a local file system that’s a whole lot faster and then it doesn’t hurt to download twice.”

At this point in the session, the pair already spent almost an hour reviewing the convoluted source code. Both developers seem tired: J2 had to pause multiple times to organize his thoughts, and J1 burried his face in his palms and rubbed the back of his nose. J1 probably felt J2 was about to follow one *Catalyzed Episode* after the other, explaining things that are not relevant to their session. He interrupted J2, trying to *Limit the Scope*. However, he could not know the actual extent of the *Target Content* so J2 ignored the *Scope Limiting* attempt.

<sup>a</sup>The original German idiom is literally “a different construction site”, which can refer to both an area where something is built and an area of interest or responsibility. A loose translation here would be ‘*Not today*’.

Concept	Description/Characterization
Sub-Episode	An <i>Episode</i> that is started to address a new <i>Knowledge Want</i> <i>in order to</i> satisfy an existing one.
Catalyzed Episode	An <i>Episode</i> that is started to address a new <i>Knowledge Want</i> which the developer developed because something caught her attention during a currently running <i>Episode</i> , but which does not need to be addressed in order to satisfy the current <i>Knowledge Want</i> .
Branching Wildly	Anti-Pattern: Starting new <i>Episodes</i> , mostly <i>catalyzed</i> ones, without finishing relevant open <i>Topics</i> first.
Return Explicitly	Positive Pattern: Finishing a <i>Sub-Episode</i> by handing back the control to the developer who <i>propelled</i> the original <i>Episode</i> .
Scope Limiting	Positive Pattern: Not starting a <i>Sub-</i> or <i>Catalyzed Episode</i> by disregarding its <i>Topic</i> temporarily or permanently.

**Table 10.1:** Elements and types of *Episode* Patterns

## 10.4 Summary and Discussion

Over the course of a session, pair programmers will perceive many **internal** and **external Knowledge Wants**, some of which are closely related to the **Knowledge Want** that is currently being addressed in some **Episode**, others are merely **catalyzed** by something the partner said or did. This mechanism appears to generally exist in pair programming, and how it unfolds depends on the particular situation and the pair members.

**Branching Wildly**, that is, to start **Sub-Episodes** or **Catalyzed Episodes** whenever a new **Knowledge Want** is perceived, comes with the risk of losing sight of not-yet-finished **Topics**, which in turn stresses the pair's **Togetherness** because **Maintaining one shared plan** gets more difficult. This behavioral anti-pattern appears to emerge in a session when the pair does not employ countermeasures, such as (a) to **Return Explicitly**, i.e., making sure to get back to the 'main' **Topic**, and (b) **Scope Limiting**, i.e., *not* starting subordinate **Episodes** by deferring **Topics** or by cutting irrelevant ones short. Circumstance of letting one's guard down appear to be:

- **Having no or only little pair programming experience.**

Consider the pair **D3/D4** in Example 10.2: **D4** had never pair-programmed before, so he could not have any first-hand experience of the dynamics of two people pursuing **Catalyzed Episodes** in a programming situation. His partner **D3** also only started pair-programming three months earlier.

The pair **C3/C4** (Example 10.7), in contrast, appears to have considerable PP experience, both individually and together (see, e.g., their pro-active approach to dealing with conflict in Example 6.26). Before and during their **Focus Phases**, they did **Scope Limiting**.

- **Dealing with the unknown.**

In the pair **D3/D4** (Example 10.2), one pair member, **D4**, knew nothing about the software system, so he could not assess which of his observations were relevant and therefore bombarded his partner **D3** with questions. Additionally, **D3** tied himself in knots when he tried to swap in the old module version which produced unexpected failures. This placed a double burden on him: Satisfying his own *and* **D4's Knowledge Wants**.

The pair **J1/J2** (Example 10.3), in contrast, experienced more smooth sailing as they basically reviewed one class authored by **J2** from top to bottom.

- **Being at the beginning of a session.**

Both pairs **D3/D4** (Example 10.2) and **C2/C5** (Example 10.4) encountered some **Branching** early in their sessions when one pair member explained his recent codes to the other, but had considerably more 'linear' progression later on.

I cannot say much more about the effect of PP experience because **D3/D4** were the only notably unexperienced pair in my data. The other two aspects I discuss in the next (and final) results chapter, where I analyze the session dynamics and overall progression that result from the pair members' **Knowledge Needs**, i.e., which task-relevant knowledge they do not yet possess.

## Chapter 11 Session Dynamics

---

11.1 Purpose and Structure of this Chapter . . . . .	315
11.2 Individual Developers' <b>Knowledge Needs</b> . . . . .	316
11.2.1 <b>S Need</b> – Need for System-Specific Knowledge . . . . .	317
11.2.2 <b>G Need</b> – Need for Generic Software Development Knowledge . . . . .	318
11.2.3 <b>Knowledge Needs</b> in Practice. . . . .	318
11.3 <b>Pair Constellations</b> . . . . .	319
11.3.1 Session Context and Goal: <b>Initial</b> and <b>Target Constellation</b> . . . . .	319
11.3.2 Constellation Changes. . . . .	320
11.3.3 Session Visualizations . . . . .	321
11.4 <b>Session Dynamics Prototypes</b> . . . . .	321
11.4.1 No Knowledge Gaps, No Opportunity . . . . .	322
11.4.2 Dealing with the <b>Primary Gap</b> . . . . .	322
<i>Proactive Explanations • Interview Mode • Pioneering</i>	
11.4.3 Dealing with the <b>Secondary Gap</b> . . . . .	326
11.4.4 The <b>G Opportunity</b> . . . . .	327
<i>Seizing or Not Seizing the G Opportunity • Complementary Pairs • When do Pair         Programmers Seize their G Opportunity?</i>	
11.4.5 <b>Two-Sided G Gaps?</b> . . . . .	331
11.5 <b>Summary and Discussion of Related Work</b> . . . . .	333
11.5.1 Related Work Discussion . . . . .	333
11.6 <b>Grounded Theory of Knowledge Transfer Session Dynamics</b> . . . . .	335

### 11.1 Purpose and Structure of this Chapter

In my analysis so far, I considered local process phenomena of developers working together as a pair (Chapter 6) and how they deal with perceived knowledge gaps on an activity and an episode level (Chapters 8 to 10), as well as the nature of transferred knowledge (Chapter 7). Now it is time to integrate the pieces to a theory of knowledge transfer in pair programming. In particular, this entails considering the roles of the two developers and the task they are working on. (Further research may consider larger contexts, such as the development team, see my scope discussion in Section 4.2.3.)

A common model in PP literature is that of ‘expert’ and ‘novice’ developers. It is, however, not clear what makes a developer an ‘expert’ (see Section 2.2.3a). Furthermore, such a dichotomy appears suspiciously simplistic to begin with (see Section 2.3.5d). In Section 11.2, I discuss that the developers’ individual knowledge levels *with respect to their task* are the relevant context conditions for what happens in a pair programming session. In particular, each pair member has session-specific **S Need** and **G Need** which characterize how much she does not know with respect to the current task about the system and about software development in general.

Typically, though not always, both pair members want to meet the *S Need* by understanding all task-relevant parts and aspects of their system. Through addressing and narrowing individual and collective knowledge gaps, the pair starts a trajectory from its *Initial Constellation* towards its *Target Constellation*. I explain these concepts in Section 11.3.

Considering the trajectories across all sessions then reveals three prototypical dynamics: If one partner has a larger *S Need* than the other, pairs start their session by addressing that difference, which I call the *Primary Gap*. After the *Primary Gap* has been closed, pairs address their *Secondary Gap* by acquiring system understanding which they *both* lack. These activities are limited by the pair's momentary awareness of these gaps and the *Target Constellation* acts as a moderator: In case not both developers want to meet their *S Need*, parts of the *Primary* and *Secondary Gap* may remain unfilled. Finally, some pairs seize a *G Opportunity* when one pair member has more task-relevant general software development knowledge and transfers it to her partner.

In Section 11.4, I summarize the multiple concrete forms for each of these phases (the pieces of which were presented in the previous chapters), which together form the overall dynamic shared by almost all analyzed sessions: First *Primary Gap*, then *Secondary Gap*, then *G Opportunity*. Different orders appear to be the exception, e.g., if, both partners have large *S and G Needs*, the pair may become overwhelmed by difficulty. The pair process then *breaks down* and no or nearly no progress happens.

I discuss the dynamics of the recurring example session JA1 in Example 11.9. For now, I give a brief overview of the pair's *Initial* and *Target Constellation*:

**Example 11.1: Recurring Example: Foreshadowing the Session Dynamics (JA1)**

In the first minutes of session JA1, the pair members allude their respective *Knowledge Needs*, which indicates that they are aware of their *Initial Constellation*. In particular, J1 has a (large) *S Need*: He does not know the software that his colleague wants to refactor and redesign. J2 also has an *S Need* since he has not looked at the source code in a while. In the first minutes of their session, though, J2's *S Need* does not yet show. More relevant is his *G Need*: Although J2 knows the software design is flawed, he lacks the relevant *G knowledge* to take systematic action against it. This is where J1 comes in, who is a more experienced Java developer and architect. The reason why the two work together is to apply J1's *G knowledge* to J2's software.

Their *Initial Constellation* can thus be summarized as follows: The pair has a *Primary Gap* between them (J2 knows more about the system), a *Secondary Gap* ahead of them (neither of them knows enough about the system yet), and *G Opportunity* waiting to be seized (J1 has more task-relevant general software development knowledge). Their *Target Constellation* allows to not fully close the *Primary Gap*, i.e., J1 does not need to fully meet his *S Need* because he will not work on the software without J2. J2's *G Need*, however, should be addressed: By the end of the session, he should possess a little more *G knowledge*.

## 11.2 Individual Developers' Knowledge Needs

I do not characterize software developers as such by labeling them as 'experts' or 'novices', but consider their role as *pair members* in a PP session. For such a session, they bring some body of existing knowledge to the table, which may or may not be enough to meet the specific knowledge demands of their task ahead. I already described how I reconstruct both technical information and the developers' understanding of it from their interaction in Section 4.5.2c. I do not address what a developer knows about arbitrary topics, but only those aspects that the pair touches during their session. In industrial PP sessions, the "task" is usually not well-defined (unlike in controlled settings) and may be modified (explicitly or implicitly) as the

session proceeds. Depending on the pair’s design decisions, different areas of knowledge may become more or less relevant. Their decisions, in turn, depend on what the developers know and do not know. What is relevant in a PP session is not some overall knowledge level, but the **Knowledge Needs**, the overall knowledge gaps of the pair members in the *particular situation* they happen or chose to be in.

There are two types of knowledge relevant across PP sessions (Section 7.3.1): specific **S knowledge** that pertains to the software system and generic **G knowledge** about software development in general. Along these lines, I separate two knowledge dimensions for characterizing a pair member in the context of a specific session: Her **S Need** and her **G Need**. In principle, these two could be split up further to allow a more detailed characterization of a pair with, say, a frontend and a backend expert (such as the pair **A1/A2** in session **AA1**). For comparing different contexts, however, I generalize such differences to the two dimensions that are relevant in all sessions.

There is an important difference between a **Knowledge Need** and a **Knowledge Want** (introduced in Section 7.2). A developer *perceives* a **Knowledge Want** and wants to *satisfy* it, e.g., by understanding something or providing an explanation (**internal** and **external Knowledge Want**, as in ‘*I want to understand it*’ and ‘*I want you to understand it*’). Developers are not necessarily aware of their **Knowledge Needs**: If they are, they may choose to *address* and eventually *meet* them. **Knowledge Wants** are local, **Knowledge Needs** possibly exist for whole sessions.

I discriminate three degrees of **Knowledge Need**: low, mid, and high. I use these degrees to conceptualize different pair constellations in Section 11.3. The degrees are fuzzy concepts used for illustrative purposes, but they are not to be mistaken for some quantity. Next, I provide for each degree an *operationalization* and brief *examples* from my data.

### 11.2.1 S Need – Need for System-Specific Knowledge

**S knowledge** is about the software system at hand: its requirements, architecture, design, design rationale, code base (including tests, scripts, etc.), known defects, etc.—see Section 7.3.1a for concrete pieces of **S knowledge**. The **S Need** is a pair member’s task-specific need for such knowledge. I characterize three degrees of **S Need** as follows:

**Low S Need** The developer provides **Explanations** about the current state to her partner, she alludes to things not yet seen in the session, and she **evaluates** findings, explanations, and hypotheses proposed by her partner. She does not ask questions about **S knowledge** and is rarely puzzled by new discoveries.

*Examples:* In several sessions, one developer already worked on the task prior to the session and had time to build a current mental model of the system, e.g., **C1** in **CA1**, **C5** in **CA2**, or **J2** in **JA2**.

**Mid S Need** The developer has some knowledge about the system in general, but not enough about the particular area relevant for the task. For instance, she may be not up-to-date with recent changes in that area. The developer may acknowledge her lack of knowledge, propose to ‘*look into things*’, or ask concrete questions. Alternatively, if she is not aware of her **S Need**, she might make proposals that are **misled** and which her partner rejects.

*Examples:* On the one hand, there are the partners who joined their colleagues and are not aware of the recent changes (e.g., **C2** in **CA1** and **CA2** or **J1** in **JA2**). On the other hand, sometimes both partners developers navigate through parts of the system they have not seen in a while or only know vaguely (e.g., **A1/A2** in **AA1** or **B1/B2** in **BA1**).

**High S Need** The developer knows little to nothing about the system’s relevant parts and aspects. She acknowledges her lack of **S knowledge** and asks her partner about the system. She does not refer to system parts or properties until the pair has looked at them. When her partner poses ideas or proposes hypotheses, her reactions are **non-evaluative**.

*Examples:* J1 has never seen the module his partner wants to talk about in JA1. O3/O4 are supposed to write a test for a piece of unknown functionality in sessions OA1 and OA2.

The degree of **S Need** depends on prior involvement with the relevant parts of the system (e.g., authorship), on forgetting details, and on many specifics of the current task.

### 11.2.2 G Need – Need for Generic Software Development Knowledge

**G knowledge** is generic, system-independent knowledge such as programming languages, frameworks and technology stacks, design principles, testing and debugging methods, methods for program understanding, tool usage, etc.—see Section 7.3.1b for concrete examples for all these types. The **G Need** is a pair member’s task-specific need for such knowledge. I distinguish three degrees of **G Need**:

**Low G Need** The developer is able to provide **Explanations** on the meaning of programming language idioms or how to use certain libraries or tools, if need be. She does not ask questions in this regard.

*Examples:* Developer D4 explains design patterns and technology details in session DA2. J1 explains coding best practices in JA1 and JA2.

**Mid G Need** The developer asks informed questions about the used technology or the development approach, and occasionally reads in the documentation.

*Examples:* Both developers K2 and K3 do not know how to use a library for an (important) detail of their integration test in KC2, but otherwise make progress.

**High G Need** The developer would not succeed in systematically solving the task without extensive access to other material (handbooks, documentation) or colleagues. The developer asks fundamental questions concerning programming language, standard libraries, or basic tools, and/or uses documentation extensively. She might also express uncertainty and verbalize a lack of ideas on how to proceed.

*Examples:* Both O3 and O4 know very little about the technological basis of the component they are supposed to test in sessions OA1 and OA2 (and make almost no progress).

### 11.2.3 Knowledge Needs in Practice

By these terms, ‘experts’ and ‘novices’ would be developers with low and high **G Needs**, respectively, for the majority of tasks in their job. For different tasks, however, the same developer will often have different degrees of **S Need** (and, in fact, **G Need** as well).

In practice, **G** and **S Needs** are not independent. While for an individual developer and a given task, having a low **G Need** with a high **S Need** is plausible, the combination of perfect system understanding (low **S Need**) and no applicable general development knowledge (high **G Need**) is unlikely: Understanding a system without having a grasp of the used technology is difficult. Furthermore, the knowledge types overlap: In session DA2 (e.g., Example 7.14 on page 249), understanding the instances where a design pattern is implemented in a software system (**S knowledge**) is not completely detached from understanding the design pattern as an abstract concept (**G knowledge**). Or refer to session OA8, where O4 learned that performing calculations on the value `undefined` yields `NaN` or *not-a-number* (see Example 6.16, on page 210, lines 34–42): It is not entirely clear whether he understood it on the conceptual level (**G knowledge**) or just in the concrete instance (**S knowledge**).

## 11.3 Pair Constellations

With respect to a specific development task, each developer has degrees of **S** and **G Need**, which possibly change over the course of a session. A PP situation can thus be characterized by each developer's momentary **S** and **G Needs** which together describe their current *constellation*. A pair has a **One-Sided Gap** if one developer has a higher **Knowledge Need** than the other. Such a gap lowers the pair's **Togetherness** with respect to a *shared understanding of the system or software development in general* (see Section 6.4). A **Two-Sided Gap**, then, exists if both developers share a lack of **S** or **G knowledge**. The relevance of either type of gap depends on the session context and goal. I discuss the role of a pair's **Initial** and **Target Constellation** in Section 11.3.1 and I characterize how a pair's constellation changes and how I visualize these changes in Sections 11.3.2 and 11.3.3.

### 11.3.1 Session Context and Goal: **Initial** and **Target Constellation**

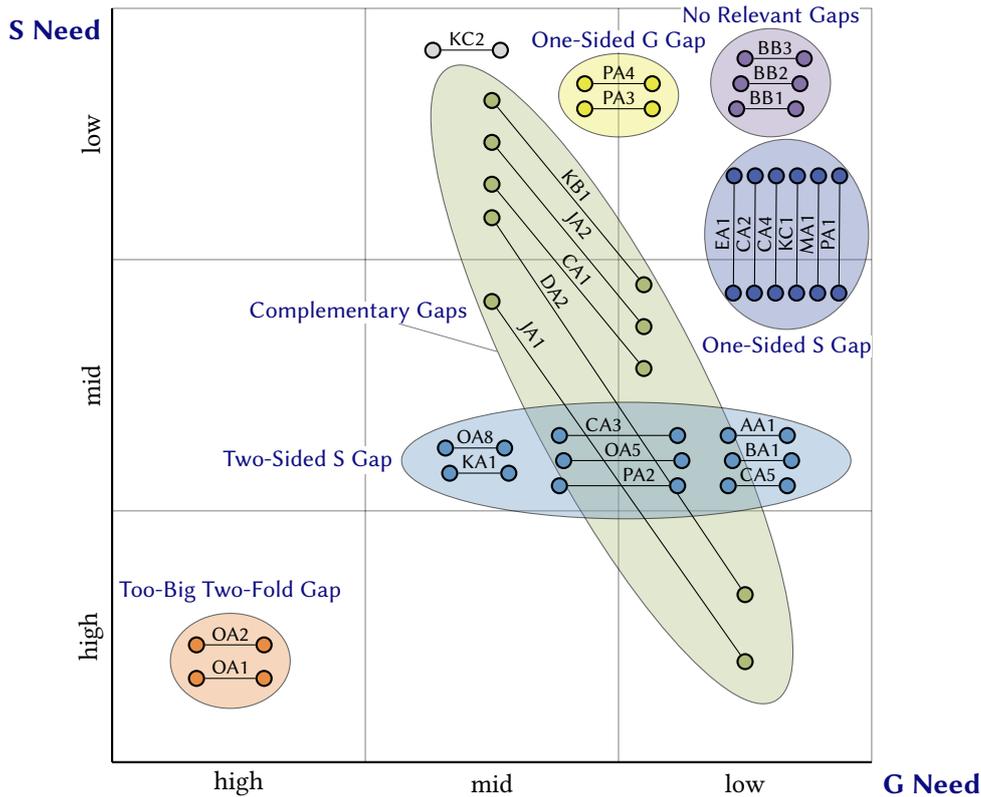
For systematically solving a task, the *pair* needs to address its overall **S Need** and attain a complete understanding of the system's task-relevant aspects. Depending on the goal of the particular session, the pair has one or more options to break this down to the individual level:

- Meeting the **S Need** for both developers is desirable, e.g., for working on similar tasks alone or with a different partner in the future. This is the case for almost all pairs.
- Other pairs are content with only one developer meeting her **S Need**, leaving a **One-Sided Gap** between the partners. Session JA1 is an example: Developer J1 did not need to understand all of the system, but just enough to provide ideas for how J2's code may be improved.
- Theoretically speaking, it could even be enough for the pair to maintain a *transactive memory system* (see page 98) where each partner relies on the other to remember certain things, so both partners can remain partially ignorant while still having all relevant **S knowledge** available together. Session AA1 with front-end expert A1 and backend-end expert A2 could have been such a case, but the developers made sure to synchronize all relevant knowledge both ways.

In contrast to **S Needs**, not all **G Needs** have to be addressed in a session. However, more complete **G knowledge** facilitates important steps such as addressing an **S Need**, designing a good solution, implementing and debugging that solution smoothly. Meeting a **G Need** may be part of the session goal, e.g., for training purposes. Again, session JA1 is an example: J1 was brought in on the task because of his general software development expertise to help refactor and redesign J2's code *and* improve J2's **G knowledge** in this regard.

Note that the above characterization is based on the analyzed PP sessions and their respective tasks (see Table 4.3). Other types of industrial PP sessions are conceivable. One developer at company R (where I did not record any PP sessions, see page 160) told me, that she had recently paired up to try out some new technology without any existing system and by producing throw-away source code only: Here, the **S** dimension supposedly has little weight and the session is all about acquiring **G knowledge** together.

Either way, pair programmers begin a session with an **Initial Constellation** of each partner having some **S** and **G Need**, plus a more or less clear idea of what they want to achieve with their session. I identified six recurring **Initial Constellations** in my data (see Figure 11.1); others are conceivable, but I have not seen them. The 'task' might be for example fixing a problem (for which **S Needs** have to be met) or educating the partner (addressing an **S** and/or **G Need** in some respect). The intended outcome of a session in terms of **S** and **G Needs** to-be-met denote the **Target Constellation**.



**Figure 11.1:** Six recurring **Initial Constellations** in terms of the pair members' **Knowledge Needs** regarding **S** and **G** knowledge. Each pair of points represents one analyzed session with its two pair members (see Table 4.3 for short characterizations of all 27 sessions).

### 11.3.2 Constellation Changes

Overall, knowledge gaps tend to shrink during a session. Pair members may become aware of a lack of knowledge and possibly develop new **Knowledge Wants** along the way, but their **Knowledge Needs** are not affected by such insights alone. In principle, pairs can take two approaches to deal with **Knowledge Needs** they are aware of:

The first approach is to limit the scope of the current task, thereby making some of their **Knowledge Need** obsolete. The pairs' *initial discussion* of their session scope is often not recorded. Before some session recordings, the developers filled out a pre-session questionnaire stating the purpose of the session (see Table 4.2). The process of filling out that questionnaire was (accidentally) recorded for sessions CA2 and EA1 (see Section 4.3.4c), but only in EA1, some discussion occurred during that time. In effect, I generally do not know how the pairs agreed on their scope or whether they were aware of their **Initial Constellation**.

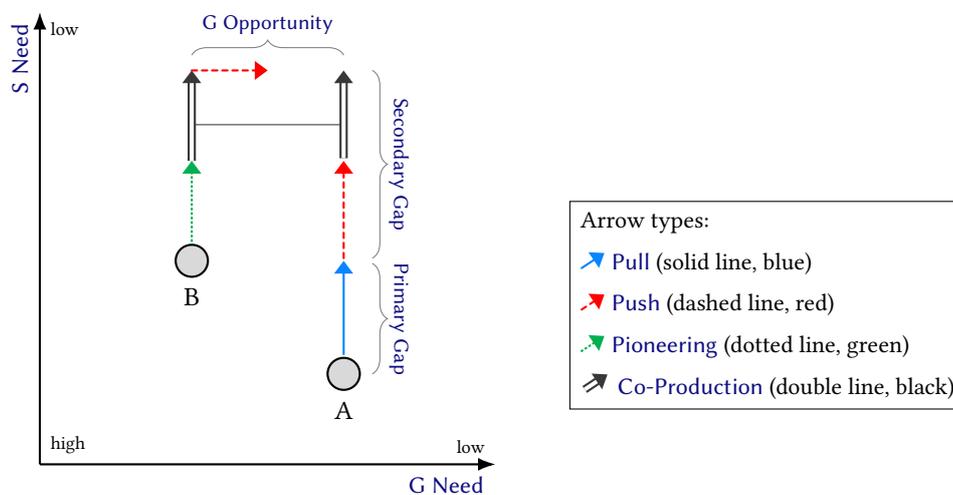
Developers sometimes decide *during their session* that some subtask is not mandatory and stop pursuing it (e.g., early in session BB1, see Example 10.8), or they may shift their focus mid-session, effectively changing what knowledge is relevant (e.g., in DA2 which should have been a feature implementation but pivoted to a large refactoring, see Example 6.22).

The second approach to deal with **Knowledge Needs** is to **transfer** existing or acquire new **knowledge**. This behavior sets the pair on a trajectory from their **Initial Constellation** towards their **Target Constellation**, the general dynamics of which I describe in Section 11.4 right after I introduced some notation in the next section.

### 11.3.3 Session Visualizations

I provide schematic representations of a pair's constellation and its trajectory to make the complexity of a whole PP session more digestible. Figure 11.2 serves as a demonstration. Each developer is represented by a point on a two-dimensional coordinate system, with the **G Need** decreasing from left to right and the **S Need** from bottom to top.

The pair's points are drawn at their **Initial Constellation**. The reduction of **Knowledge Needs** is indicated by arrows: upward for gaining **S knowledge**, to the right for **G knowledge**. Multiple arrows starting at the same height indicate multiple attempts to transfer or acquire knowledge. The trajectories do not depict technical progress or time; arrow length does not mean anything except the qualitative reduction of a **Knowledge Need**. Arrow color indicates the **Mode** of knowledge transfer (see Sections 9.3 to 9.6 for details).



**Figure 11.2:** Demonstration of a session trajectory visualization. In this (fictous) session, developer A first asks her partner about **S knowledge** ( $\uparrow$ ); then developer B investigates the source code alone ( $\uparrow$ ) and explains her findings to developer A ( $\uparrow$ ). Afterwards, they investigate the source code together ( $\uparrow$ ) and eventually, developer A explains something about software development in general ( $\rightarrow$ ).

Numbers in the trajectories to follow correspond to numbers in the main text. For readability, a single arrow might represent multiple **Episodes** pertaining to similar **Topics**. The arrows originate at the developer whose understanding improves: A **Pull**- $\uparrow$  means that the developer asks about something, a **Push**- $\uparrow$  means that *her partner* explains something.

## 11.4 Session Dynamics Prototypes

Each of the six **Initial Constellations** has a different combination of the pair members' **S** and **G Needs** which leads to a characteristic session dynamic. Together, these form a set of three session dynamics *prototypes* that characterize all analyzed PP sessions:

1. If one partner possesses more task-relevant **S knowledge** about the system and its parts (e.g., because she already worked on the task), the pair address this **One-Sided S Gap** first. I call such a difference the **Primary Gap**, which is shown in the corresponding session diagram as the height difference of the pair members (see Figure 11.2).

2. Pairs then proceed to acquire lacking **S knowledge** together to close the remaining **Two-Sided S Gap**. This is what I call the **Secondary Gap**. It is visualized as the shared distance of both partners to the top of the diagram.
3. Eventually, pairs with a **One-Sided G Gap** may also transfer **G knowledge** about software development in general which goes beyond their particular system. Such pairs seize their **G Opportunity**, which is visible as their horizontal distance in the diagram.

If both partners need to acquire **G knowledge** in order to work on their task, they have a **Two-Sided G Gap**. This is rare in my data, probably because software developers intuitively avoid partners and tasks that would lead to such a pair constellation. I have not seen enough instances to derive a session dynamics prototype and to analyze its relation to the three described above. It appears, however, that a *small Two-Sided G Gap* can be dealt with by acquiring the necessary **G knowledge** while the **Secondary Gap** is still open whereas a *large Two-Sided G Gap* in conjunction with a large **Two-Sided S Gap** poses a difficult situation for the pair.

I will now characterize how pair programmers deal with each of these situations.

#### 11.4.1 No Knowledge Gaps, No Opportunity

In many PP sessions, there is a point at which both partners have all necessary understanding to work productively on the task. This is the **No Relevant Gaps** constellation (see top right corner in Figure 11.1). As an **Initial Constellation**, I have only seen it with one pair who started the development of a completely new feature:

##### **Example 11.2: Greenfield Development (BB1, BB2, BB3)**

Developers **B1** and **B2** work on a new feature from scratch over the course of one afternoon in three sessions (BB1 to BB3) with short pauses in between. Both are proficient in the involved technologies (low **G Need**). They need to interact with existing code only through few and well-understood interfaces and the newly developed code stays small enough to be fully understood by both developers at all times (low **S Need**).

Apart from a short orientation phase in the beginning, when they decide on where to visually place the feature in the GUI (see also Example 10.8, where the pair effectively did **Scope Limiting**), they are in construction-only mode throughout the sessions, i.e., defining requirements, discussing design proposals, and writing code—with zero debugging.

Although the pair's **Initial Constellation** had neither **Primary** nor **Secondary Gap** (which is unique in my data), they still both acquired **S knowledge** about their new code along the way. There were, however, no explicit knowledge transfer activities: The pair discussed and agreed how to implement the feature, performed the necessary code additions and changes, and did not need to come back to look things up or to reconcile different mental models. In effect, they had high **Togetherness** without the need to **Maintain** it explicitly.

#### 11.4.2 Dealing with the Primary Gap

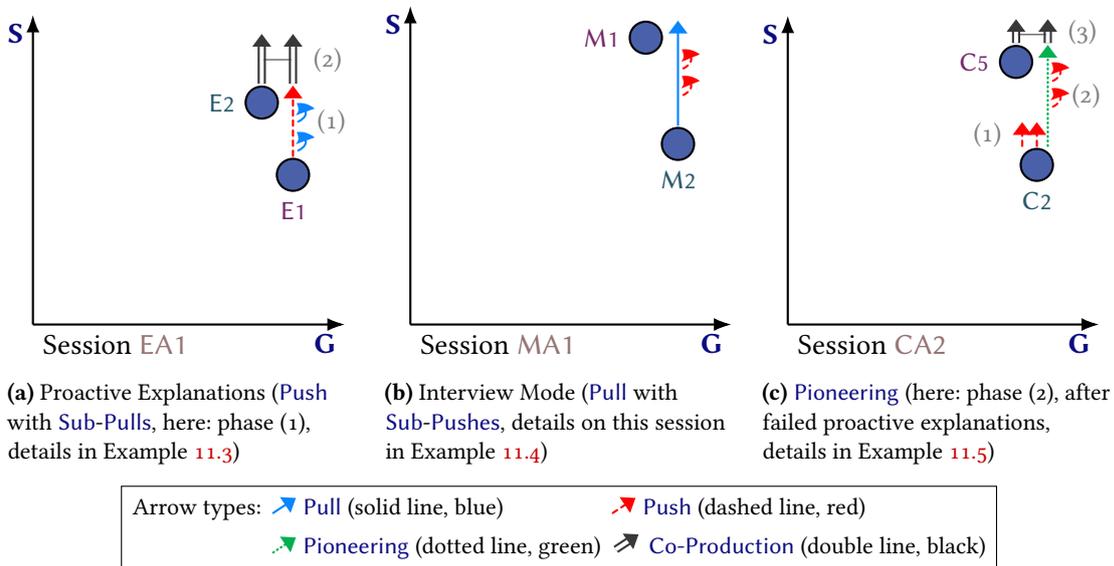
In some pair programming sessions, one developer possesses more relevant **S knowledge**, e.g., because she already worked on the task. Two constellations have this property: **One-Sided S Gap** and **Complementary Gaps**, each of which happened to be the **Initial Constellation** of five analyzed sessions (see Figure 11.1).

Whenever a **Primary Gap**—one developer having a larger **S Need** than the other—exists, the pair addresses it first. If a pair member is not aware of her larger **S Need**, she might make **misled** proposals which need to be identified as such. This can take some time and be frustrating for the developers (as session **CA2**, see Example 11.5 below).

Concept	Description
Knowledge Need	Extent of individual developer's knowledge gap resulting from her existing knowledge and the specific demands of the task. May pertain to either knowledge type and are then called <b>S</b> and <b>G Need</b> (see Sections 11.2.1 and 11.2.2).
One-Sided/Two-Sided Gap	Characterization of a pair: Either only one pair member or both partners have a <b>Knowledge Need</b> in some regard.
Initial/Target Constellation	Characterization of a PP session: Constellation of the developers each with her respective <b>S</b> and <b>G Needs</b> which they assume at the beginning of their session and which they want to achieve with their session. Six recurring constellations (see Section 11.3.1 and Figure 11.1):
– No Relevant Gaps	Neither partner has an <b>S</b> or <b>G Need</b> ; rarely an <b>Initial Constellation</b> , often the <b>Target Constellation</b> .
– One-Sided S Gap	One has a larger <b>S Need</b> than the other; e.g. when joining a partner who already started, sometimes also <b>Target Constellation</b> .
– Two-Sided S Gap	Both lack system understanding; <b>Initial Constellation</b> e.g. for debugging tasks, not a <b>Target Constellation</b> , but a common intermediate constellation.
– One-Sided G Gap	One has a larger <b>G Need</b> than the other; opportunity to transfer <b>G knowledge</b> in absence of <b>S Need</b> .
– Complementary Gaps	One has a larger <b>S Need</b> , the other a larger <b>G Need</b> ; satisfactory session possible due to mutual learning.
– Too-Big Two-Fold Gap	Both have high <b>S</b> and <b>G Needs</b> ; difficult and undesirable constellation.
Overall Session Dynamics:	Three prototypes of part-of-session dynamics describing pairs' trajectories from all <b>Initial Constellations</b> :
1. Close the <b>Primary Gap</b>	Narrow a <b>One-Sided S Gap</b> between the developers: either through pro-active explanations, an interview mode, or through solitary reading of the less knowledgeable partner (Section 11.4.2).
2. Close the <b>Secondary Gap</b>	Narrow a <b>Two-Sided S Gap</b> of both developers by building up understanding together and staying in sync (Section 11.4.3).
3. Seize the <b>G Opportunity</b>	Narrow a <b>One-Sided G Gap</b> between the developers after any <b>S Need</b> has been addressed sufficiently (Section 11.4.4).

**Table 11.1:** Concepts to characterize programming pairs and their session dynamics

There are three general strategies how pairs go about closing their **Primary Gap**: **Proactive Explanation**, which is a long-running  $\uparrow$  **Push Episode** into which the partner hooks with  $\curvearrowright$  **Sub-Episodes** for details (Section 11.4.2a); **Interview Mode**, which is a long-running  $\uparrow$  **Pull Episode** (Section 11.4.2b); or  $\uparrow$  **Pioneering** which the developer with the larger *S* Need starts (Section 11.4.2c).



**Figure 11.3:** Three different strategies of dealing with a **Primary Gap**

#### 11.4.2 a) Proactive Explanations

Often, when a developer has a relevant *S* Need, she is aware of it and the pair can address it right away in the beginning of their session. A recurring pattern here is a long-running  $\uparrow$  **Push Episode** of the developer who already worked on the task, into which the partner can hook with detailed questions ( $\curvearrowright$  **Sub-Episodes**). In sessions CA4, EA1, JA1, and PA1, such **Proactive Explanations** were enough to close the pair's **Primary Gap**.

While the  $\curvearrowright$  **Sub-Episodes** are helpful to close the **Primary Gap**, they may also result in the  $\uparrow$  **Pushing** partner losing sight of the main **Topic** and the pair starting to **Branch Wildly** (see Section 10.2.1). A countermeasure is to **Return Explicitly** when a  $\curvearrowright$  **Sub-Episode** is finished (see Section 10.3.1). Session EA1 is an example of this:

##### **Example 11.3: Bringing Partner Into Ongoing Work (EA1)**

The pair wants to fix a bug which E2 has already worked on. This means that his partner E1 has a larger *S* Need, and the pair has both a **Primary** and a **Secondary Gap**. The two major phases of their session are also depicted and numbered in Figure 11.3a:

- (1) The developers first close their **Primary Gap**: E2 steps through the code with a debugger, demonstrates the failure in the running application, and comments on the state of individual variables ( $\uparrow$  **Push**); his partner E1 asks questions regarding details ( $\curvearrowright$  **Pull Sub-Episodes**, which were discussed in Example 10.5).
- (2) They continue debugging together, engaging in  $\uparrow$  **Co-Production** of system understanding.

### 11.4.2 b) Interview Mode

If proactive explanations are not enough to close the **Primary Gap**, e.g., because the more knowledgeable developer does not provide suitable explanations, her partner may take the lead with a more interview-style **↑ Pull-driven mode**, as in session **DA2** (whose trajectory I discuss later in Example 11.10), where **D4** asks his partner about the technological basis of the software and its requirements. Many of these questions appear to be ad hoc, which contrasts sharply with session **MA1** where the **Interview Mode** appears to be the result of careful preparation:

#### Example 11.4: Prepared Interview Mode (MA1)

The whole session of 25 minutes followed mostly pre-structured **↑ Pull Episodes** (see Figure 11.3b for an overview of the session trajectory): **M2** wants to understand the purpose of all tables in the database (an **S Need**). Prior to the session, he prepared SQL SELECT queries for all of them and put his questions in comments. In the session, he executes the queries, asks his partner **M1** about the results, and writes down the answers.

This particular style allows **M1** to validate that the **Target Content** is correctly transferred to **M2** and to start a **▶ Push Episode** if there is something important missing (see Example 7.4 for details).

Similar to the strategy of proactive explanations, the partner's **Sub-Episodes** can help closing the **Primary Gap**. In the case of **DA2**, however, developer **D4** asked many questions and some of them resulted in **Catalyzed Episodes**: The pair did not contain these and started **Branching Wildly** (see Section 10.2.1).

Session **MA1** was the only one where the developers followed an explicit and pre-formulated plan, but the interview mode does not rely on such planning to work. In session **CA1**, for example, **C2** also attempted to close his **Primary Gap** with ad hoc **↑ Pulling** which was at least partially successful. I discuss the trajectory of session **CA1** later in Example 11.8.

### 11.4.2 c) Pioneering

If the interview mode is not enough to close the **Primary Gap** either, the partner with the **S Need** may switch to reading up the necessary information herself in **↑ Pioneering**. In session **DA2**, such a switch was necessary when developer **D3** could not explain well because he lacked relevant **G knowledge** to properly understand **D4**'s questions (see, e.g., Example 9.6 for how **D4** gives up asking questions).

Developer **C2**, however, appears to *generally* prefer to **↑ Pioneer** for closing a **Primary Gap**, even though his partner is willing and able to provide suitable information. He did this in both sessions **CA1** and **CA2** where his respective partner had already worked on the task alone.

#### Example 11.5: Closing the Primary Gap Painfully (CA2)

**C2** and **C5** want to implement a new feature for just one edition of their software. **C5** has already started the implementation and is familiar with the system modularization (low **S Need**). **C2** does not know **C5**'s recent changes; additionally, some aspects of the system's architecture have slipped his mind (mid **S Need**). It takes the pair a frustrating 11 minutes and multiple attempts to close their **Primary Gap** (see also Example 4.2). Their trajectory is depicted in Figure 11.3c:

- (1) **C5** tries to explain his recent changes and alludes to the underlying architecture that motivated them. **C2** does not engage in these **↑ Push Episodes**: he does not listen to **C5** at all and keeps hushing him (see, e.g., Example 6.12).
- (2) Instead, **C2** starts reading the source code (**↑ Pioneering**), which leaves him puzzled several times, because he is not aware of the underlying rationale. **C5** tries to follow **C2**'s mostly silent reading process and intersperses architectural explanations (**▶ S Pushes**). **C2**, however,

## Example 11.5 (continued)

appears to misinterpret these as a discussion of general design principles,  $\rightarrow$  **G Pushes**, and ignores them. This continues until **C2** eventually recognizes the underlying system structure and finally understands **C5**'s changes from before the session (see Example 7.7).

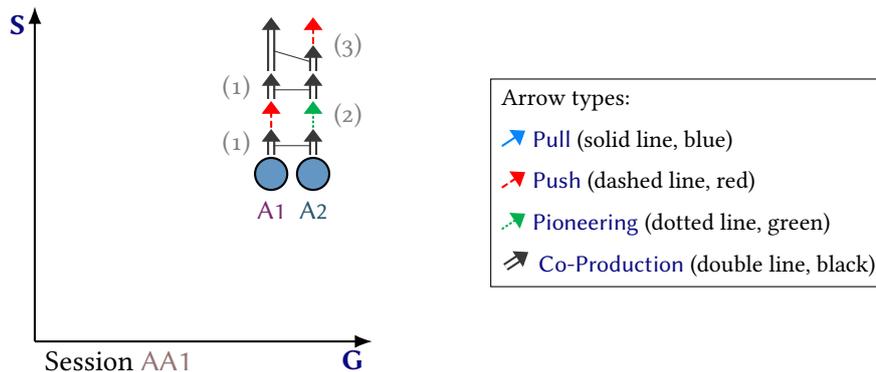
- (3) With their **Primary Gap** closed, the pair continues in a **Two-Sided S Gap** constellation and works on their **Secondary Gap** mostly in  $\uparrow$  **Co-Production** (see, e.g., Example 7.12).

**Sub-Episodes** can be helpful in all three strategies to close a **Primary Gap**, i.e., with proactive explanations, interview mode, and pioneering: The partner briefly shifts the attention to some detail in either  $\rightarrow$  **Pull** or  $\rightarrow$  **Push Mode**. While during proactive explanations and the interview mode *both* partners are engaged,  $\uparrow$  **Pioneering** does not inherently involve the partner. Being a **Talking Pioneer** rather than a **Silent Pioneer** (see Section 9.4) offers more opportunities for the partner to provide helpful information with  $\rightarrow$  **Sub Pushes**.

### 11.4.3 Dealing with the Secondary Gap

Pairs where both members have an **S Need** cannot simply exchange the necessary **S knowledge** but have to acquire it. Such a **Secondary Gap** appears to be common: eight of the analyzed sessions started with a **Two-Sided S Gap** constellation (see Figure 11.1), many others reached this constellation after closing their **Primary Gap**.

Similar to closing the **Primary Gap**, there are different strategies how a pair may close its **Secondary Gap**. The pair **A1/A2** exemplifies all three of them in their session **AA1**:



**Figure 11.4:** Three different ways of dealing with a **Secondary Gap** illustrated with the trajectory of **AA1**: (1) **Co-Production**, (2) **Pioneering** with subsequent **Push**, and (3) **Co-Production** with subsequent **Push**.

#### Example 11.6: Pairing-Up Throughout (AA1)

Developers **A1** and **A2** want to fix four similar bugs and need to work with two different sub-systems, neither of which is fully understood by either partner. Since they both want to meet their **S Need**, they keep their **S knowledge** in sync along the way. Their session illustrates three ways how pairs can deal with their **Secondary Gap** (see Figure 11.4 for the numbers):

- (1)  $\uparrow$  **Co-Production**: Most of the time, **A1** and **A2** address their **Secondary Gap** collectively by formulating hypotheses about the system, reading source code, trying out the application, and integrating their insights (e.g., in Examples 7.5 and 9.20).
- (2)  $\uparrow$ + $\rightarrow$  **Pioneer plus Push**: The developers disagree on the relevance of some **Topics**. **A2** occasionally pursues a  $\uparrow$  **Pioneering Episode** (see, e.g., in Example 6.25, **A2**: “*Why does it have its own [implementation]?*”—**A1**: “*No clue. And I don’t want to*”—**A2**: “*No, I want to*”) and afterwards explains what he learned ( $\rightarrow$  **Push**). **A1** hardly opposes **A2**'s initiatives, as some of them lead to task-relevant insights which the pair probably would have missed otherwise.

## Example 11.6 (continued)

- (3) **↑↑↑ Co-Production plus Push**: Sometimes one developer is faster at understanding something than the other during **↑ Co-Production**. In such cases, the faster developer **↑ Pushes** explanations for his partner to catch up (e.g., in the beginning of Example 9.17).

The subsequent **↑ Pushes** in phases (2) and (3) avoid the introduction of a **One-Sided S Gap** where one pair member knows more about the task-relevant parts of the system than the other. These **↑ Pushes** are how the pair **Maintains Togetherness** throughout the session, which ultimately enables their **Focus Phases** (1:53:20–2:00:55, shown in full in Appendix C.1.1).

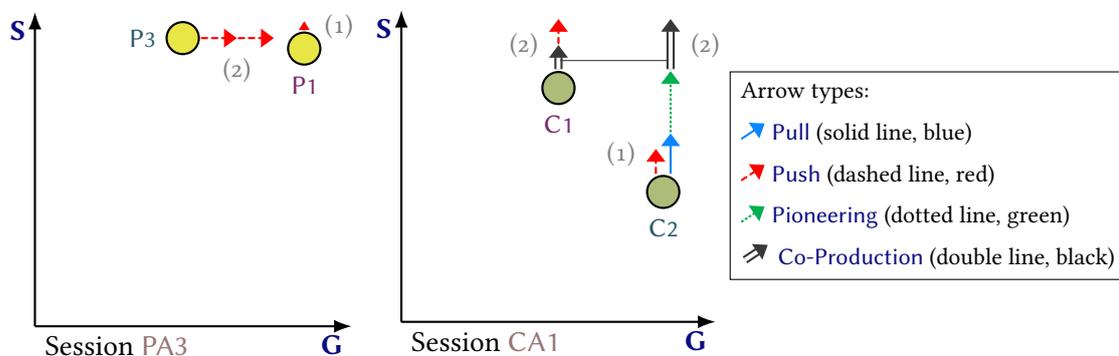
For closing a **Secondary Gap**, **↑ Co-Production** is common behavior in many sessions (e.g., CA1, CA2, and JA1). If one pair member understands faster, e.g., due to possessing more **G knowledge** (as in CA1, discussed below in Example 11.8) or more **S knowledge** in some code area (as in AA1 above), a **↑↑↑ Co-Production plus Push** makes sure the partner does not fall behind. If the developers have different goals or preferences, not all **Topics** need to be understood fully by both and the more invested pair member may **↑↑ Pioneer plus Push**. Occasionally, this also happened in other sessions, such as JA1 (discussed below in Example 11.9).

With the **Secondary Gap** closed, or at least contained by temporary **Scope Limiting**, a pair may enter a **Focus Phase**—given their **Togetherness** is high enough, i.e., they also have *one shared plan, good workspace awareness, and no language barrier* (see Chapter 6). The three **Focus Phases** of pair A1/A2 in session AA1 happened in the end of their session when no **Secondary Gap** was left (between 1:53:20 and 2:00:55, see Table 6.3). In session CA5, the **Focus Phases** were much earlier (between 19:12 and 28:06, see Table 6.3), but by then, the pair C3/C4 was at a similar point in their trajectory with only a small **Secondary Gap** left.

## 11.4.4 The G Opportunity

A difference in **G knowledge** between the partners can be an opportunity to transfer general software development knowledge. In my analyzed sessions, pairs only seized their **G Opportunity** after any known **Primary Gap** and **Secondary Gap** were closed. Some pairs started from a **One-Sided G Gap** (e.g., PA3 and PA4), others from **Complementary Gaps** (e.g., JA1 or DA2) or a **Two-Sided S Gap** (e.g., OA5).

## 11.4.4 a) Seizing or Not Seizing the G Opportunity



(a) Seizing **G Opportunity** (here: phase (2), details in Example 11.7)

(b) Not-seizing the **G Opportunity** (details in Example 11.8)

Figure 11.5: Trajectory of sessions with a **G Opportunity**

Although session PA3 was at times frustrating for P3, the pair nevertheless seized their G Opportunity:

**Example 11.7: Initially Misunderstood Teaching (PA3, PA4)**

In sessions PA3 and PA4, frontend developer P3 and backend developer P1 work in the backend of their system. Both know the relevant parts of the system well (no Secondary Gap), but P3 already started implementing a new API endpoint (small Primary Gap). Their knowledge trajectory has two phases (see Figure 11.5a for the corresponding numbers):

- (1) Since they are on P1's technological home turf (no G Need for him), he understands P3's  $\uparrow$  explanations quickly and the Primary Gap is soon closed.
- (2) P1 explains the newest PHP language features and how to employ test-driven design whenever he sees an opportunity ( $\rightarrow$  G Pushes).

In session PA3, P3 misinterprets these explanations as lead-in for unnecessary  $\uparrow$  S Pushes and gets confused (see also Example 9.23), but after talking to P1 about P1's intentions in a break he then acknowledges them as valuable lessons in session PA4.

The pair in session CA1, in contrast, had multiple occasions to become aware of C1's G Need, but did not seize the G Opportunity:

**Example 11.8: G Opportunity Not Seized (CA1)**

The pair wants to implement a new GUI feature that is similar to an existing feature. C1 already worked on it for an hour when C2 joins him. This gives creates a Primary Gap between C1 and C2, which needs to be addressed. C2 is more proficient with the object-oriented paradigm (more G knowledge). They deal with their Primary and Secondary Gap (see corresponding numbers in Figure 11.5b):

- (1) To close the Primary Gap, C1 first tries to explain what he did to C2 ( $\uparrow$  Push). This is not effective and C2 starts to ask specific questions about existing classes ( $\uparrow$  Pull), which C1 begins to answer (see also Example 9.13). But C2 quickly gives up on this in favor of trying out the new GUI elements and reading in the new code himself ( $\uparrow$  Pioneering), which eventually achieves the desired understanding (see also Examples 9.4 and 9.15).
- (2) Later in the session, the pair is able to address their Secondary Gap in  $\uparrow$  Co-Production Episodes. In these cases, C2 is always the first to understand (presumably due to his better G knowledge) and often explains his findings to C1 ( $\uparrow$  Push).

This pair does *not* use their G Opportunity. During the session, there are multiple occasions where C1's G Need becomes apparent, such as in the exchange below in which only C2 is able to quickly assess the implications of changing a class in an object-oriented design (18:46–19:54):

<p>C1: "This Panel needs to implement that interface, right?"</p> <p>C2: "[...] Exactly, this one. &lt;*opens Panel class implementation*&gt;"</p> <p>C2: "God &lt;*hovers class declaration*&gt;"</p> <p>C1: "&lt;*dictates*&gt; 'implements IEnableableComponentContainer' "</p> <p>C2: "(#AbstractDialogPanel#), that's interesting. [...] What's that now? &lt;*opens superclass, reads (,,,,), back to original class, types 'implements IEnableableComponentContainer'*&gt;"</p> <p>C2: "Let's see what it has. &lt;*opens interface*&gt;"</p>	<p>C1 makes a valid design proposal.</p> <p>C2 agrees with it and opens the class they want to change.</p> <p>C2 sees that the class extends another one.</p> <p>C1 is not aware of the super class; he thinks that his partner is stuck and needs to be reminded of the interface name.</p> <p>To C2 (low G Need), the presence of a superclass meant that there is more than one potential place to implement the interface. C1 seems oblivious to the issue (mid G Need) but C2 does not explain his reasoning.</p>
--	--

**Example 11.8 (continued)**

C1: “(#getComponents#), that’s cool. Looks easy.”      C1 sees that the interface declares only one method (getComponents), but he overlooks that it also extends another interface.

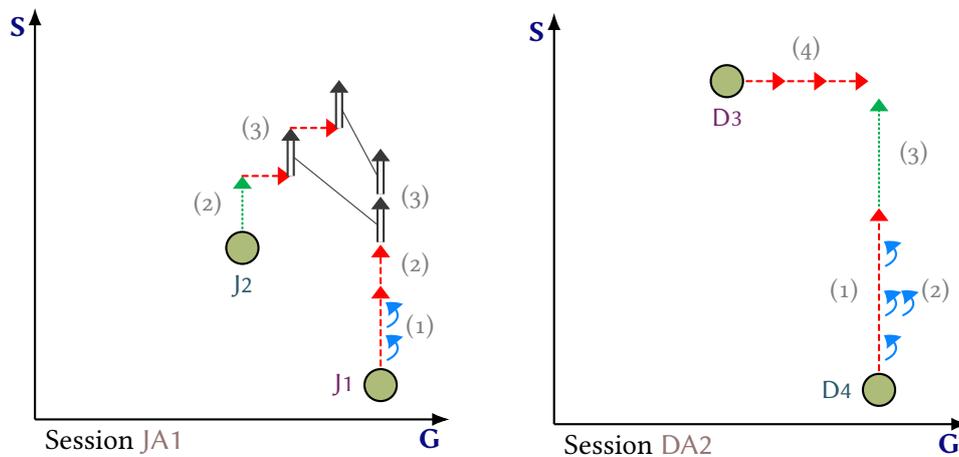
C2: “Ah, <\*hovers extends-declaration\*> and an IEnableable <\*open super-interface\*>”      C2 notices the super-interface and realizes there is more than just one method they need to implement.

In both instances, C2 *could* have picked up on C1’s G Need and make his reasoning explicit by pushing some G knowledge to C1 (i.e., -▶), e.g., by saying something like ‘*In object-orientation, we need to carefully consider any extended classes and interfaces*’.

A constructive pattern for a situation like the one above might be to ▶ Pull specifically for G knowledge whenever the partner does something ‘magic’ without ▶ Pushing.

**11.4.4 b) Complementary Pairs**

In pairs with a One-Sided G Gap, but little or no S Needs (as in PA3, see Example 11.7), the partner with the G Need may receive and appreciate explanations, but has no opportunity to provide some of her own. In contrast, a pair with **Complementary Gaps** such as in sessions KB1, JA1, or DA2 are mutually satisfying sessions: One developer has an S Need and her colleague can help with this; with her own advantage in G knowledge she may help her colleague in return. I discuss sessions JA1 and DA2 in Examples 11.9 and 11.10 on the next page.



(a) Complementary Gaps example where the Primary Gap is not completely closed in phase (1) and the G Opportunity is seized in phase (3) (see Example 11.9 for details)  
 (b) Complementary Gaps example where all Knowledge Needs are eventually met (see Example 11.10 for details)

Arrow types: ▶ Pull (solid line, blue)      -▶ Push (dashed line, red)  
 ▲ Pioneering (dotted line, green)      ↗ Co-Production (double line, black)

**Figure 11.6:** Trajectory of sessions starting with Complementary Gaps.

**Example 11.9: Embracing a Difference (JA1)**

Session JA1 is about improving the maintainability of a module that J2 wrote a year earlier and which J1 never saw before. The module is basically a state automaton implemented with deeply nested `if`-statements. In addition to a seized **G Opportunity**, this session also illustrates how one developer (J1) does neither need nor want to fully meet his **S Need** as he will only ever work on that module together with J2. The notable phases of their trajectory are as follows (see also Figure 11.6a):

- (1) The pair deals with its **Primary Gap** via a long running **Push** with hooked-in **Sub Pulls** (one of which is the recurring example in every results chapter of this thesis).
- (2) To address their **Secondary Gap**, J2 repeatedly reads through the complex low-level control structure and then explains the high-level states and transitions of the automaton (**Pioneer** plus **Push**). Both partners take care to keep the **Pushes** from going into too much detail (**Scope Limiting**, e.g., in Example 10.6).
- (3) After J1 got the big picture (only a mid **S Need** left), the pair starts reading source code together (reducing the **Secondary Gap** further through **Co-Production**). In doing so, J2 looks for code smells with which to explain possible refactorings (**G Pushes**) thus using the **G Opportunity** (see Example 9.22 for similar behavior of the same pair in session JA2).

**Example 11.10: Easy-Task Jump-Start with G Opportunity (DA2)**

In session DA2 the pair discovers that, in order to implement a new feature in a clean way, they first need to apply the Template Method design pattern multiple times. This is an easy task for developer D4 with no **G Need**: He starts with a high **S Need** but can easily acquire the necessary **S knowledge** for this narrow task. The session trajectory has the following phases (see also Figure 11.6b):

- (1) D3 explains the system's basic structure and technology stack (long-running **Push**).
- (2) D4 asks many questions about the system (**Sub** and **Catalyzed Pulls** hooked into the long **Push**, see, e.g., Examples 7.9 and 8.8). Due to D4's high **S Need**, not all of these questions are relevant for the current task, and as it happens, D3 does not have satisfying answers for all of them (unsuccessful **Pulls**, see also Example 10.2).
- (3) D4 ceases to ask questions and explores source code instead until he understands enough of the system to get productive (**Pioneering**). D3 offers only few explanations during this phase.
- (4) Later in the session, when D4 met his **S Need**, he takes charge of the whole session. He then actively seeks opportunities to explain general object-oriented design principles and IDE features (**G Pushes**), which D3 welcomes (see e.g., Examples 7.14, 7.15, and 7.17).

**11.4.4 c) When do Pair Programmers Seize their G Opportunity?**

Of the ten analyzed sessions in which the pairs had a **G Opportunity** (see Figure 11.1), only two had no transfer of **G knowledge** (CA1 and CA3). So, what are the context conditions for pairs seizing their **G Opportunity**? Since I had no chance to interview the respective developers about this, the following overview is based on their in-session behavior only.

First, **G knowledge** transfer does not happen early in a session. All analyzed pairs first addressed their **Primary** and **Secondary Gaps** which makes sure they know everything they need to know to work productively on their tasks. The reason for why **G knowledge** transfer does not happen earlier, however, does *not* appear to be that possessing **S knowledge** is a necessary *precondition*. Rather, it appears that pair programmers need to be in a somewhat 'relaxed' state of mind. Second, there needs to be some trigger for a **Knowledge Want** to arise that can lead to a transfer of knowledge. The following two cases illustrate these points:

- In session JA1, J1 starts his -> G Pushes while the pair is still in the process of closing the Secondary Gap. The trigger for his external Knowledge Wants are potential flaws he sees in J2's design. For this, J1 needs to understand *something* of the system, but not to the same degree as his colleague: J1 can relax.
- Similarly in session DA2, where the pair introduces the Template Method pattern in a number of classes, the pair still needs to understand the details of more than ten classes around the time when D4 starts his -> G Pushes. But at this point, he has already refactored a dozen similar classes and he is familiar with the steps. He, too, is relaxed, if not already a little bored. The triggers of his external Knowledge Wants are hard to pin down, but it appears as if brief moments of silence make him search for things to explain.

The two sessions with a not-seized G Opportunity had completely different contexts:

- The moments in session CA1 where C1's G Need becomes noticeable and *could* trigger an external Knowledge Want in C2, the pair is fully immersed in closing their Secondary Gap. The knowledgeable partner C2 is not relaxed: While reading source code, he says to himself "God [...] What's that now?" (see Example 11.8), barely reacts to C1, and does not take his eyes off the computer screen. All subsequent explanations concern S knowledge, and the pair does not seize their G Opportunity.
- The only other session with a not-seized G Opportunity is CA3. Here, pair member C7 is sometimes surprised by the refactorings that her partner proposes: She has a small G Need. The session, however, is very frustrating for the pair because their IDE repeatedly went unresponsive for 80 seconds totaling about one third of the whole session. The pair was clearly frustrated and not relaxed (they even took a break to buy some candies to cool off) and did not address their G Opportunity.

In summary, pairs seize their G Opportunity if and when one partner's G Need becomes apparent and the other pair member is relaxed enough to develop and pursue an external Knowledge Want, which is usually the case only after Primary and Secondary Gap are being dealt with.

### 11.4.5 Two-Sided G Gaps?

There are sometimes PP sessions where both pair members have a G Need—a non-routine situation. I have seen one instance of a pair attempting to acquire the relevant G knowledge:

#### **Example 11.11: It's not easy! (KC2)**

Developers K2 and K3 want to write a test case for an auto-completion feature which K2 implemented earlier. They already addressed their Primary Gap in session KC1 before lunch and now want to simulate keystrokes programmatically. Both have a mid G Need: they know their tools and where to look for help, but cannot implement a test case right away. Their trajectory is depicted in Figure 11.7a:

- (1) They attempt to read documentation together (=> G Co-Production), which helps K2 somewhat, but not so much K3.
- (2) Forty minutes later(!), they notice this one-sided G gap and close it (-> G Push)—see also Example 6.23.

However, they do not fully meet their G Need and give up after two hours. The next day, K3 said he found a simple solution alone.

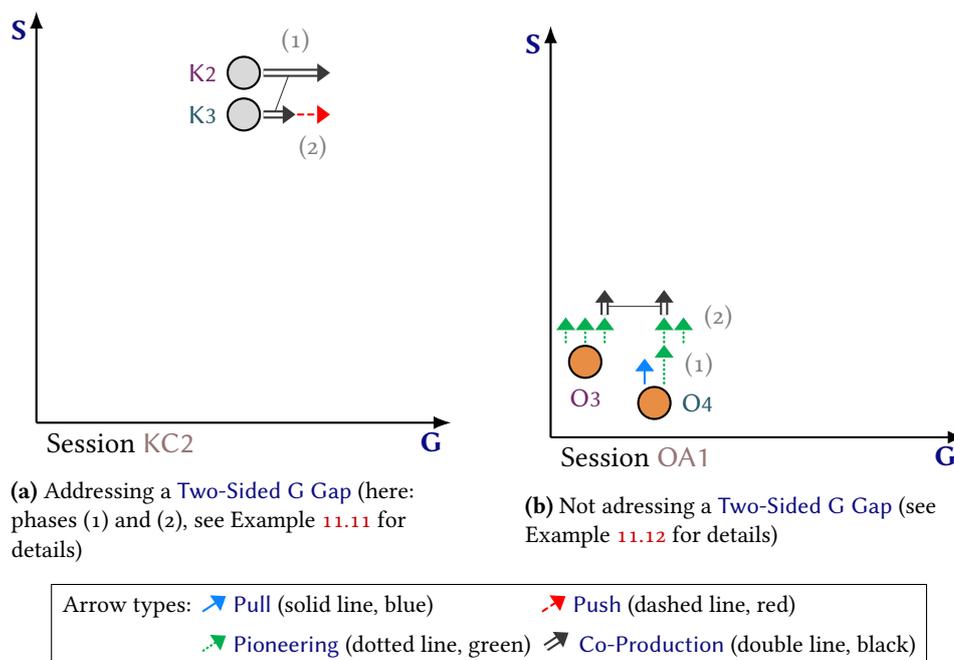
If both pair members have a high S Need *and* a high G Need, the pair faces a Too-Big Two-Fold Gap, lacking the technical background (G knowledge) to build up the required S knowledge. I have seen this constellation twice with one pair in sessions OA1 and OA2:

**Example 11.12: Breakdown (OA1, OA2)**

The developers had to write test cases for some new functionality they did not implement and which is built with a technology they are not familiar with. O3 and O4 have a high *S Need* and a high *G Need* at the same time. O3 has a slight *S* advantage, as she already opened and skimmed the relevant source code. O4 has a slight *G* advantage since he knows a bit more about the programming language. (See numbers in Figure 11.7b.)

- (1) To close the small *Primary Gap*, O4 has to  $\uparrow$  *Pioneer* since O3 does neither  $\uparrow$  *Push* nor react to O4's  $\uparrow$  *Pull* attempt.
- (2) For most of the session, they address their *Secondary Gap* and try to acquire *S* knowledge by individually reading in the source code ( $\uparrow$  *Pioneering*). At some point, and for lack of better ideas (remember the high *G Need*), they together resort to “printf” debugging ( $\uparrow$  *Co-Production*), but do not gain much *S* knowledge in this way either (see *Breakdown* discussion in Example 6.17).

The developers express confusion on fundamental issues (e.g., O3: “*Type? Function? I don’t even know what this is.*”), but never really attempt to address their *G Need*. The same pattern continues in session OA2 on the same day after lunch, where they *do* ask a colleague for help multiple times regarding the different types of tests which their system uses (*S* knowledge), but do not ask about the used technology. The pair eventually decides to not continue with this task.



**Figure 11.7:** Trajectory of sessions starting with a *Two-Sided G Gap*

A *Two-Sided G Gap* is too rare in my data to make much of it, but it appears to be difficult to resolve. In session KC2, the pair had no *S Need* to deal with and presumably were simply too tired to put their newly gained *G knowledge* to proper use. My interpretation of sessions OA1 and OA2 is that the situation was so difficult overall that the pair failed to manage the combined complexity of task solving plus coordinating the PP process. I do not expect a *Too-Big Two-Fold Gap* to be common in general software development practice, as developers likely anticipate and avoid such a situation. In the OA1/OA2 case, the pair only started this task because they were the only team members available and the task had high priority.

## 11.5 Summary and Discussion of Related Work

The purpose of *selective coding* in the Grounded Theory Methodology is to develop the “*core category*”, the central concept that integrates all concepts of interest. Strauss & Corbin (1990) propose that the researcher should write a *narrative*, a story to explain the core category. In my study, this is the **overall session dynamics** consisting of the three steps of closing the **Primary Gap**, closing the **Secondary Gap**, and seizing the **G Opportunity**. The full narrative where I integrate the core category with the concepts from the earlier chapters follows in Section 11.6 (without back references to not clutter the text), after I discuss work related to the findings in this chapter.

### 11.5.1 Related Work Discussion

The following three ideas are central to this chapter:

1. A mere *expert/novice* dichotomy based on software development experience is simplistic.
2. Each task or particular situation requires different knowledge.
3. *General* software development knowledge and *specific* system understanding have different roles in sessions, and in terms of productivity, the latter trumps the first.

Individually, all of these ideas were explicitly proposed or last tacitly assumed by different authors. Simon (1996, p. 111), for instance, characterized all professional practice as being “*aimed at changing existing situations into preferred ones*”. A similar idea is put more clearly by Schön (1983, pp. 39–41) who argues that the core of professional practice is not so much technical problem solving than understanding the situation first, which is characterized by its “*complexity*” and “*uniqueness*”, among other things. Schön’s notion of the “*Reflective Practitioner*” frames the essence of professional work as some kind of “*conversation*”:

“ [The professional practitioner] shapes the situation, in accordance with his initial appreciation of it, the situation “talks back,” and he responds to the situation’s back-talk. In a good process [...], this conversation with the situation is reflective.

Schön (1983, p. 79)

As far as I know, Schön was not concerned with software development, but the above description also fits it quite well. In my terms, *understanding the situation* is acquiring **S knowledge**, *changing the situation* (or *technical problem solving*) then requires **G knowledge**.

In Table 11.2, I summarize the stance of publications discussed in Section 2.2.3 regarding the three central ideas mentioned above. Here are some of the details:

- Beck (1999, pp. 59, 67) deems it necessary for developers to have a common understanding of their system (i.e., shared **S knowledge**) to avoid unfruitful discussions and slow progress during pair programming; partners should be chosen based on recent experience in task-relevant areas.
- Robillard (1999) argues that novice developers mostly possess knowledge from courses and textbooks, whereas experts can additionally draw upon their more important “*episodic knowledge*” which is built up through actual practice. While the textbook-knowledge falls into the **G knowledge** category, it is not clear what type of knowledge he expects experts to possess, be he sees it as superior.
- Soloway & Ehrlich (1984) subscribe to the expert/novice dichotomy. Their “*programming plans*” and “*programming discourse rules*” would be parts of **G knowledge** that only experts possess. They only studied coding activities and short algorithms, with no relevant **S knowledge** to consider.

Publication	1. “Expert/Novice” is Simplistic	2. Task Specificity	3. S over G knowledge
Schön (1983)	✓	✓	(✓)
Soloway & Ehrlich (1984)	✗	✗	✗
Beck (1999)	✓	✓	✓
Robillard (1999)	✗	(✓)	(✓)
Sim & Holt (1998)	(✓)	✗	✓
Zhou & Mockus (2010)	(✓)	✓	(✓)
Fritz et al. (2010)	(✓)	✓	(✓)
Kerr (2017)	(✓)	(✓)	✓

**Table 11.2:** Publications on knowledge-relevance in software engineering and their relation to three of my central ideas. Symbols: ✓ – idea is presented; (✓) – similar idea is presented; ✗ – contrary idea is presented (or no positive evidence for compatibility)

- Sim & Holt (1998) characterize “*software immigrants*” as developers with potentially much **G knowledge** but with high **S Needs**. Two of the practices they describe to ramp up such new team members are (a) selecting first tasks that are limited to a very narrow portion of the system which makes meeting the **S Need** easier (see developer **D4** in session **DA2**, Example **11.10**) and (b) assigning a mentor (who has more **S knowledge** and is possibly available for pair programming).
- Zhou & Mockus (2010) emphasize task specificity. Two of the dimensions they report as what makes a task difficult—*application* and *technology*—correspond to **S knowledge** and **G knowledge**, respectively. They do not weigh these dimensions against each other.
- The tool by Fritz et al. (2010) is meant to find a colleague with relevant system understanding based on their prior interaction with parts of the source code. It postulates the importance of **S knowledge**—although not relative to other knowledge types.

I close this discussion with a quote from developer and blogger Jessica Kerr. She, too, observed the importance of **S knowledge** for efficient software development:

“ Let’s talk about why some developers, in some situations, are ten times more productive than others. [...] When do we get that exhilarating feeling of hyperproductivity, when new features flow out of our fingertips? It happens when we know our tools like the back of our hands [**G knowledge**], and *more crucially*, when we know the systems we are changing [**S knowledge**]. [...] Know the contents of every module, both what they are and what we’d like them to be if we ever finish that refactoring. Know the edges, who uses every API and which changes will break whom, [...] which database fields are indexed and which are obsolete and which have quirky special values. [...]

It is extremely difficult to establish this level of intimacy with an existing system. [...] Pair program! By far the best way to transfer understanding of the system [**S knowledge**] to another human is to change it together.

Kerr (2017, emphasis added)

The apparent real-world importance of **S knowledge** over **G knowledge** is not reflected in the many PP studies in which the developers had to work with small and unknown “systems” (see, e.g., Section *Unrealistic and Unfair Comparisons* on page 67). First, using small systems ignores the dimension of **S knowledge** and thus cannot make any statement about its importance. Second, using systems unknown to the developers may create a high **S Need** for *both* partners,

while starting development from scratch leads to zero *S Need*. Both situations are unfamiliar or uncommon for professional developers, which might in part explain the observed between-study variance of PP effectiveness (see page 64).

Overall, my key observations ('expert/novice' is simplistic, knowledge needs are task-specific, and system understanding is more relevant than general development knowledge) are compatible with the experience of reflective practitioners and with observations of researchers working closely with practitioners in industry. What was missing so far is an explanation that integrates these pieces and that is based in empirical research and systematic analysis. In the next section, I present my Grounded Theory of how software developers actually transfer knowledge in pair programming sessions.

## 11.6 Knowledge Transfer in Pair Programming: A Grounded Theory of Session Dynamics

When two software developers decide to work together on some task, they may expect a number of benefits, such as to avoid introducing defects, to come up with a good design, to understand the software system better, or to learn something from their partner. To really work *as a pair* and for the benefits to possibly come into effect, the two need to understand their partner's actions and intentions. The ease with which they can understand each other is their *Togetherness*. This is what determines the *Fluency* of their pair process: Pairs with too low *Togetherness* risk a frustrating and unproductive *Breakdown* where the pair situation not only loses its benefits, but may even be worse than either developer working alone. Pairs with high *Togetherness* may enter an enjoyable and possibly very productive *Focus Phase* where they do not need many words and may even complete each other's sentences.

A pair's *Togetherness* (and thus *Fluency*) is influenced by a number of factors: (1) The pair's shared understanding of the software system and (2) of software development in general, as well as having (3) one shared plan, (4) good workspace awareness, and (5) no language barrier. Any of these factors may impede a pair's *Togetherness*. Moreover, the factors' impact may change over the course of a session, e.g., one developer losing sight of their plan, or the other gaining a better understanding of some module. For a *fluent* process, pair programmers therefore need to *Maintain Togetherness*.

No two developers know exactly the same. Each has their distinct technical background and knows more or less about different parts of the system. Depending on the task and how the pair chooses to approach it, neither of them usually possesses all relevant knowledge. Whether or not the developers are already aware of it, both have *Knowledge Needs* that are specific to their task. The two classes of knowledge most relevant in pair programming are system-specific *S knowledge* and general software development, or *G knowledge*. Usually, the developers need to understand some aspect or part of the existing system and have an *S Need*; sometimes, one or both partners lack general software development knowledge pertinent to the task and thus have a *G Need*.

Pairs may decide to not address all their *Knowledge Needs* in their session. Complete system understanding, for example, may not be necessary for a partner who just joins for quick help, making some unmet *S Need* tolerable for her. Pair programming sessions are driven by the developers trying to meet the *Knowledge Needs* relevant to them:

1. First, they close the **Primary Gap** which is the difference in system understanding between the developers.

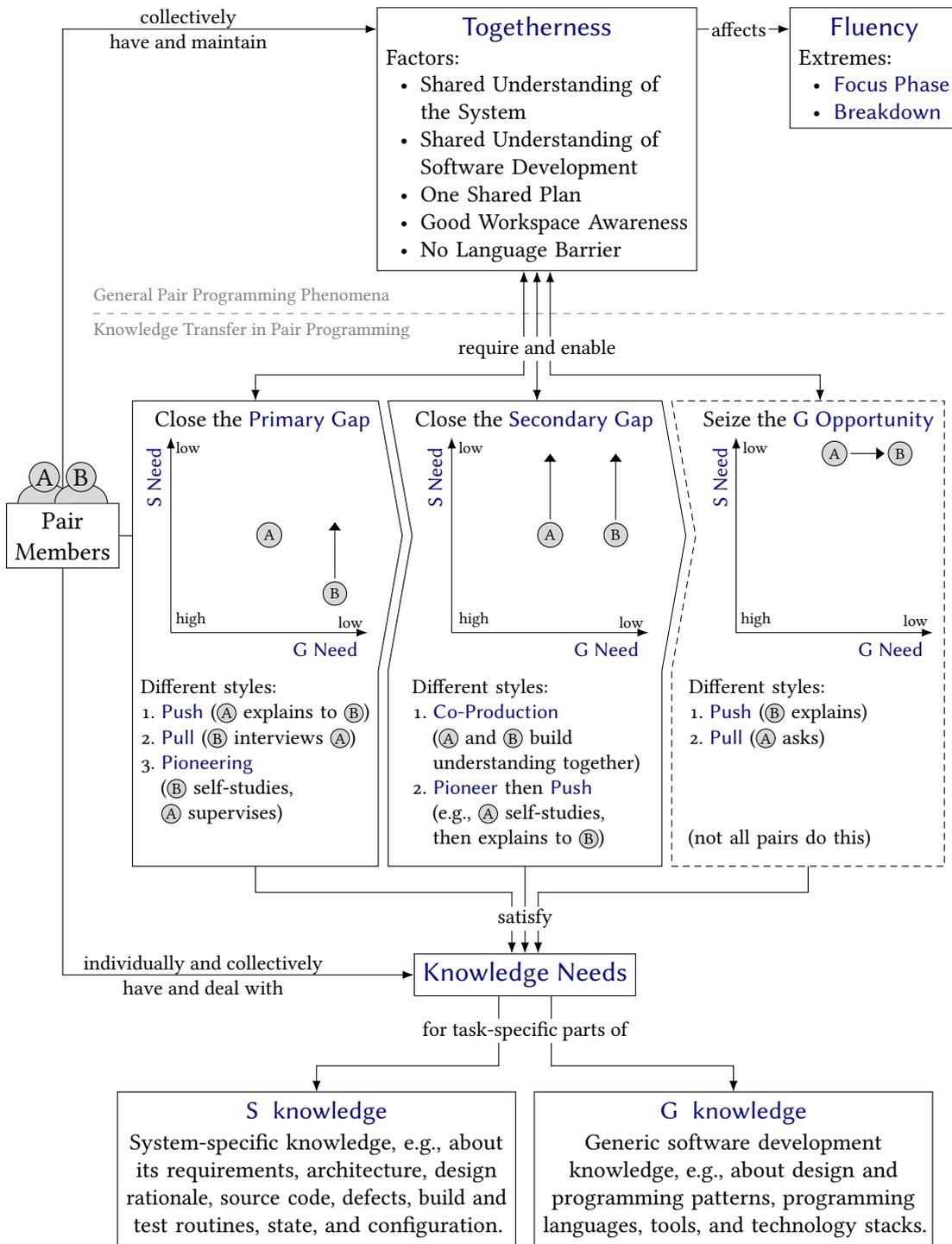


Figure 11.8: Grounded Theory of knowledge transfer session dynamics

The most common strategy is for the more knowledgeable developer to provide **Proactive Explanations** while her partner asks for details if need be: A long-running **Push** with **Sub Pulls**. During the **Pull Episodes**, the asking developer sets the pace and granularity of the explanations. The difficulty of **Pull Episodes** lies in making the partner understand one's particular **Knowledge Want**, i.e., what is the **Topic** of the request. In some cases, a whole **Clarification Cascade** is necessary to lead the partner to clarity in this regard. Afterwards, to commence the long-running **Push Episode**, the more knowledgeable developer can **Return Explicitly** to the main **Topic** (her partner may be of some help).

The second strategy for closing a **Primary Gap** is an **Interview Mode** during which the developer with the **S Need** conducts a long-running **Pull Episode**, e.g., when the more knowledgeable partner has some difficulty providing coherent explanations. A danger here lies in **Branching Wildly** when the provided explanations in turn lead to more questions and the pair follows each new idea.

The last resort for closing a **Primary Gap** is for the less knowledgeable partner to **Pioneer**, that is to read source code to acquire the relevant knowledge. For some developers, this may be a matter of personal preference over proactive explanations or the interview mode. Whatever the motivation, a **Silent Pioneer** leaves the knowledgeable partner in the dark, but a **Talking Pioneer** makes her thought process visible and thus allows the knowledgeable partner to provide brief **Pushes** in the right moment.

2. Pairs who closed their **Primary Gap** (or had none to begin with) then close the **Secondary Gap**, their *shared* lack of system understanding. Here, the most common way for pairs to proceed is the **Co-Production** of new knowledge by reading source code together, developing hypotheses, and integrating each other's ideas. Often, one pair member will be slightly faster than the other in understanding some aspect. If the partner does not catch up soon, the resulting gap is addressed yet again either by a short **Push** from the 'faster' partner or by **Pioneering** of the 'slower' one.
3. Eventually, when both **Primary** and **Secondary Gaps** are closed to the extent the pair is aware of them, they may seize their **G Opportunity**: Deliberate transfer of **G knowledge** from one to the other. This mostly happens in **Push Episodes** when the pair member with the **G knowledge** advantage—now at relative ease because there is little to no **S Need**—looks for opportunities to explain general concepts with concrete examples. There is, however, no **G Opportunity** when both partners have a **G Need**. Quite the opposite: The few instances in my data suggest that pair programmers are not good at handling such a situation. This is possibly due to a lack of according experience: Most knowledge transfer in PP sessions pertain to **S knowledge**.

The developers' **Togetherness** is crucial in all three stages. When pairs close their **Primary Gap**, they make sure they have shared understanding of the system. This establishes **Togetherness** which makes it easier for the pair to acquire lacking system understanding together. To then close their **Secondary Gap**, they need to maintain their **Togetherness** with each newly understood piece of information if they want to keep working as a pair. Pairs appear to only seize their **G Opportunity** when their **S Needs** are resolved and they have a satisfactory and shared understanding of the software system, that is, when they are in a state of high **Togetherness**.

In summary, software developers need to transfer knowledge in order to work together as a pair, and they need to work together as a pair in order to transfer knowledge.



— Part III —

# Evaluation and Conclusion



## Chapter 12 Actual Research Process

---

*The proper test is not that of finality, but of progress.*

– Alfred N. Whitehead

12.1	Phase 1: Initial Analysis of Base Activities . . . . .	342
12.2	Phase 2: Developing the <b>Episode</b> Concept . . . . .	342
12.3	Phase 3: Analysis of <b>Pull Episodes</b> . . . . .	343
12.4	Phase 4: New Knowledge Transfer <b>Mode: Produce</b> . . . . .	343
12.5	Phase 5: First Round of Data Collection . . . . .	343
12.6	Phase 6: Considering Practitioner Relevance . . . . .	345
12.7	Phase 7: Give Up Naturalistic Approach? . . . . .	345
12.8	Phase 8: Discovery of Second Knowledge Dimension . . . . .	346
12.9	Phase 9: Member Reflection and Selective Coding . . . . .	347
12.10	Phase 10: Finishing the Thesis . . . . .	347

I presented the results in Chapters 6 to 11 in roughly linear and bottom-up fashion starting from subtle properties of individual utterances to different types of knowledge transfer activities, **Episodes**, relationships between **Episodes**, and eventually whole pair programming sessions. However, my qualitative research process did not proceed in this order. Just as the three modes of *open*, *axial*, and *selective coding* of Straussian Grounded Theory Methodology are different perspectives rather than sequential phases (see Section 3.3.3), I shifted my analysis focus multiple times, revisiting my concepts multiple times to arrive—after several years—at the grounded theory I summarized in the previous chapter.

Qualitative research in general has the characteristic trait of an *emergent research design* (see, e.g., Flick et al., 2004, p. 8; Patton, 2002, pp. 43–45, and my discussion in Section 3.2.2). In Chapter 4, I characterized the strategic decisions I made for my research process and described what I did to analyze my data in a semi-abstract way. In this chapter, I present the timeline of my research. As I advance chronologically, I summarize my research activities (data collection, data analysis, literature study, etc.), what I specifically looked for in my data, what notable observations I made that shaped the further process, and which concepts I developed. As is common for a GTM study, my concepts evolved over time. I will highlight some of the notable developments and provide references to the chapters and sections of this thesis where some intermediate concepts found their place, possibly under a different name.

## 12.1 Phase 1: Initial Analysis of Base Activities

### *Data Collection*

I started my analysis on already available PP session recordings (see Section 4.3).

### *Literature*

At this point, I was already familiar with the ambiguous results of controlled experiments comparing solo and pair work regarding quality and duration (see Section 2.3.4a), and some of the qualitative work on PP published before 2012, including the work of Chong & Hurlbutt (2007) and Bryant et al. (2008, discussed on pages 72 and 80), and of course the work from my research group (Salinger et al., 2008; Salinger & Prechelt, 2013).

### *Data Analysis*

As proposed by Salinger & Prechelt (2013, p. 30), I started my analysis by looking for base activities that clearly involve pre-existing knowledge, mostly *explain\_knowledge* activities. I did this together with Stephan Salinger (who left the project after this phase) in the manner of *pair coding* (see Section 3.4.1d). We started with session CA2, which we both already knew quite well, so there was not much open coding for us to do to ‘break up’ the data. We soon switched to axial coding, considering more the causal conditions, context, and strategic behavior of the developers. These were among our first observations:

- Although it cannot be directly observed, there appears to be something that makes developers start knowledge transfer from one moment to another. We called this *causal condition* a “finding” (not to be confused with a base layer *finding*, which is a discourse object). Years later, this evolved into the *Knowledge Want* concept (see Section 7.2).
- Knowledge transfer is usually not achieved with a single base activity, but takes multiple somehow related activities. We called this *context* a “Knowledge Transfer Interaction Sequence”, which I later called *Episode* (see Chapter 9).
- Pair programmers may change the thematic orientation of such a “Sequence”: We saw *strategic behavior* of both “branching” and “returning”, now *Sub-Episodes* and *Return Explicitly* (see Chapter 10).

## 12.2 Phase 2: Developing the Episode Concept

### *Data Analysis*

In axial coding, while considering the context of individual explanations, I came across different knowledge types. To structure these, I went back to open coding but, at this point, only ended up with an unordered list of what are now mostly the subtypes of *S* and *G* knowledge (see Sections 7.3.1a and 7.3.1b).

I also developed a notion of the knowledge content of an *Episode* as a *whole* and the individual utterances as *puzzle pieces*. Eventually, this led to the concepts of *Topic* and *Target Content* (see Section 7.3).

At this point, I noticed that pair programmers have different ways of *asking a question* which set the direction of the further knowledge transfer. I initially called this concept “shaping” before it was incorporated in the “*Propellor*” concept, which denotes the developer who drives forward the clarification of a *Topic* (whereas her partner merely reacts). Together with the two *Modes* in which a *Propellor* can proceed— *Push* and  *Pull* of knowledge, i.e., asking and providing explanations—this led to my final *Episode* concept: The pursuit of clarifying one *Topic* in a constant *Mode* (see Chapter 9).

## 12.3 Phase 3: Analysis of Pull Episodes

### Data Collection

I supported Julia Schenk in recording the JA-sessions and then performed the quick analysis (see Section 4.3.2d) of session JA1 with her in *pair coding* manner (see Section 3.4.1d).

### Data Analysis

The first few minutes of JA1 were different from the other sessions I had analyzed so far, because there was a lot of knowledge transfer in a short amount of time (not by accident are these minutes the recurring example of the result chapters). At this point, I decided to focus on ■ Pull Episodes (see also a memo of that time in Figure 12.1), investigated their internal structure, discovered Explanation Elicitors (see Section 8.2) and the Clarification Cascade (see Section 8.2.1c). With my goal to eventually advise practitioners, I also started considering how Episodes end, e.g., which of them could be considered *successful* or not (see Section 9.2.2).

## 12.4 Phase 4: New Knowledge Transfer Mode: Produce

### Data Analysis

I discovered that pair programmers not only ▣ Push and ■ Pull existing knowledge in an entirely verbal manner, but also acquire new knowledge from reading source code and documentation, using a debugger, or interacting with the application. Since pairs also rely on pre-existing knowledge during these activities, sometimes even explicitly, I also included these phenomena in my analysis. This new ‘knowledge producing’ Mode allowed for both developers to be ‘in charge’ of the Episode, so I made distinction between ▣ Pioneering Production and ■ Co-Production (i.e., one or two propelling pair members).

### Writing

At this point, we wrote the article “On Knowledge Transfer Skill in Pair Programming” (Zieris & Prechelt, 2014), where the established concepts of the time were published: Target Content and Topic (see Section 7.3); Explanation Elicitors (called “explanation triggers” back then), the Clarification Cascade, and Explanations (see Chapter 8); and the notion of an Episode with a Propellor and a Mode (see Chapter 9). I did not yet have a concept to address a developer’s lack of knowledge. However, the original formulation from the article (“*The need [for knowledge] is difficult to observe directly and will hence not even be given a formal concept name*”, *ibid.*, Sec. 4.2) already hints at what would in Phase 9 become the Knowledge Need (see Section 11.2).

## 12.5 Phase 5: First Round of Data Collection

### Data Collection

I recorded eight pair programming sessions at company K and discussed some of my observations with a large number of K-developers (see Section 4.3.5b for details).

### Data Analysis

After systematic comparison of the four knowledge transfer Modes (▣ Push, ■ Pull, ■ Co-Production, and ▣ Pioneering), I introduced new concept to capture the motivation of developers to start an Episode. I initially called it “knowledge need” (short for “*need for knowledge transfer*”), of which I identified the three forms of *internal*, *external*, and *collective*. In Phase 9, I would later rename that concept to Knowledge Want to make clear the distinction between a

I noticed that it is easy to introduce new properties (especially property values). I cannot address all of them in new material and need to focus.

As of now, there are the following (somewhat relevant) properties of KT episodes and their values:

- \* information type & related information type: (many, with subtypes)
- \* level: keep/set, narrow, indifferent, (expand ?)
- \* shaping: intensional, extensional
- \* medium: verbally, demonstration, typing+verbally
- \* mode: facts/reasoning, go through
- \* scope qualification: most
- \* starter: (several)
- \* finisher: (several)

Additionally, there are multiple "markers":

- \* invalidation
- \* substantiate
- \* reservation
- \* stubby

My first idea was to group them by What?, How?, and Why?/When?:

- \* What?: (related) information type
- \* How?: level, shaping, medium, mode, scope qualification
- \* Why?/When?: starter, finisher

The "How?"s can be grouped even better:

- \* level and shaping are relevant for describing the interaction
- \* medium is more about the chosen "aid" for the KT
- \* mode is more about the "style" of the KT
- \* the scope qualification and the markers describe in various ways the quality of the KT

Assessment:

- \* (related) information type: leads to an arbitrarily fine-grained taxonomy, easy to code with. Currently (or even if it would be "complete") it serves no purpose, but exists only as end in itself. May be useful for filtering (only KT for certain type of knowledge). It appears reasonable for a study on KT to include the type of knowledge, but as of now there is no justification from the data.  
Conclusion: Don't throw it away, but don't put too much energy in it.
- \* medium: Very descriptive. Can be kept, but not worth much attention.
- \* level/shaping: Both useful for low-level descriptions of interaction. Possibly try a more high-level perspective and use the low-level perspective as a fallback.
- \* mode: Is a high-level perspective and way more interesting for my GT study. Should get more attention than the low-level things.  
Hypothesis: The mode or mode changes in KT are important properties of the overall process.
- \* qualification: Rather young property, may incorporate some of the current "markers". Content-wise this property seems somehow interesting, possibly for the style or success of a KT. But I don't expect wonders here, so I don't focus on this now.
- \* starter/finisher: rather descriptive, possibly later, as fallback.

So: Mode it is!

**Figure 12.1:** Example of a memo I wrote on 2013-04-15, about one year into my research. At this point, I still wrote my memos in German, so the above text is translated. "KT" stands for *knowledge transfer*. In the terminology of Strauss & Corbin (1990), this memo contains both *theoretical* and *operational notes* (see Section 3.3.3f).

developer’s motivation to transfer knowledge and her actual (potentially disparate) lack of knowledge or **Knowledge Need**.

A misunderstanding between the pair members in session KC2 (see Example 6.23) led to the introduction of the concept of **Parallel Production** as a less communicative form of **Co-Production** (see Section 9.5.2). Through constant comparison, I noticed a similar difference between some **Pioneering Episodes**, which led to the distinction of **Talking Pioneer** and **Silent Pioneer** (see Sections 9.4.2 and 9.4.3). It would take until Phase 8, however, for me to introduce a concept to capture the ‘less communicative’ property: These pairs did not **Maintain Togetherness**.

## 12.6 Phase 6: Considering Practitioner Relevance

### *Data Analysis*

I found that behavioral patterns can be identified, conceptualized, and operationalized—but not easily classified into *good* and *bad*, which I deemed necessary for my goal of formulating practically relevant advice. I decided to ‘zoom out’ and consider the role of **Episodes** in the course of a session by asking questions like *What are the relationships of individual episodes?* and *How do pairs choose which topics to address?*

I came back to the old concepts of “branching” and “returning” from Phase 1 and wrote my own visualizer tool to get an intuitive high-level overview of the super-structure of knowledge transfer **Episodes** (compare Figures 4.7a and 4.7b). Insights from experimenting with different data visualizations led to further qualitative analyses across sessions and ultimately to the concepts of **Branching Wildly**, **Returning Explicitly**, and **Scope Limiting** (see Chapter 10).

One additional pattern I discovered then was a long-running **Push** with **Sub-Pulls**. Through axial coding around this pattern, I identified the relation of the developers to the task as a relevant context condition and distinguished three levels:

- “Level 3”: The *task expert*, who possesses all task-relevant knowledge (I originally described this role together with Salinger et al. (2013) prior to my work on knowledge transfer; see page 84).
- “Level 2”: partial or outdated knowledge
- “Level 1”: no knowledge

Axial coding revealed that the above-mentioned **Episode** pattern occurred when one partner had a higher level than the other (i.e., 3-2, 3-1, or 2-1). This would later become the strategy of **proactive explanations** with which many pairs address their **Primary Gap** (see Section 11.4.2).

Although these were interesting analytic results, I knew that I would need to zoom out further to identify and describe phenomena that practitioners could find interesting.

## 12.7 Phase 7: Give Up Naturalistic Approach?

### *Data Collection*

It occurred to me that the distinction between ‘good’ and ‘bad’ patterns hinges on the *consequences* of the developers behavior. However, all the sessions I had seen so far were of pairs who may have struggled a bit along the way, but ultimately achieved something useful in a general software engineering sense. Without an alternative universe to compare to, pinpointing their relative success or failure to individual occurrences of behavioral patterns seemed unjustifiable (see also my discussion in Section 4.2.2).

All session recordings were naturalistic so far, i.e., the developers decided for themselves on which tasks to work on with whom. With the conjecture in mind that software developers are more or less *skilled* in pair programming, I briefly considered giving up the *naturalistic* nature of my inquiry (Patton, 2002, pp. 39–43) and to record developers who never pair-programmed before—or who *did* pair-program before, but abandoned it—to see how a ‘failure’ looks like.

### **Data Analysis**

On the side of data analysis, I switched to axial coding again: I tried to determine the importance of the *type* of knowledge developers are after in different phases of their session. However, there were not enough differences between the sessions available to me at this point: Basically in all cases, the developers needed to understand some parts of their system. I had not yet seen other cases, so the relevance of this observation was not yet clear to me.

### **Writing**

At this point, “Observations on Knowledge Transfer of Professional Software Developers During Pair Programming” (Zieris & Prechelt, 2016) was written, where we emphasized the existence of knowledge transfer in *all* pair programming sessions (not only ‘expert/novice’ constellations) and contrasted **Parallel Production** with ‘proper’ **Co-Production** where the pairs “resynchronize” (which would later be called **Maintaining Togetherness**). This was also the first time, the idea of **internal** or **external Knowledge Want** as the motivation behind a knowledge transfer **Episode** was published.

## **12.8 Phase 8: Discovery of Second Knowledge Dimension**

### **Data Collection**

At this point, the opportunity arose to record pair programming sessions in a self-proclaimed “all PP company” (company O). I did not yet let go of the plan from Phase 7 to record (presumably) bad pairs, but to the end of looking for maximal differences in PP process quality, recording developers who do not do anything else but PP seemed helpful as well. Ironically, the sessions I recorded next (OA1 and OA2) were the *worst* I have ever seen in terms of what the pair achieved in a software engineering sense.

### **Data Analysis**

My existing concepts could not describe, let alone explain, what happened in these problematic sessions. My first attempt was a general “overtaxation” concept: The task was simply too difficult for the pair. However, the same pair of developers had similar problems in another session (OA8), but *not* throughout the session. My existing three levels of task-familiarity from Phase 6 could not explain their behavior.

But before I looked for the causal conditions in the manner of axial coding, I first needed to understand the phenomenon through open coding. This is how I developed the **Fluency** concept as a characterization of a pair process (see Section 6.3) and soon after the role of a pair’s **Togetherness**. The pair O3/O4 had difficulties in *all five Togetherness* factors (see Section 6.4).

With these concepts in place for characterizing the pair process of all analyzed pairs, I could now approach axial coding to systematically search for the causal conditions. General software development skills were not an issue in prior sessions, but they were in session OA1 and OA2, so I introduced a new dimension of *developer knowledge level* to account for this difference: The many knowledge types from Phase 2 now came together and the task-familiarity concept from Phase 6 got split up to form two categories of **S knowledge** and **G knowledge**.

Through constant comparison, I noticed that the already analyzed pairs also had relative difference regarding that new **G** dimension, but it never hindered their progress. This observation led to the notion of pair constellations on a two-dimensional chart (see Figure 11.1) and was the first hint at the special role of the **G Opportunity** (see Section 11.4.4).

## 12.9 Phase 9: Member Reflection and Selective Coding

### *Data Collection*

A two-dimensional chart which provides a high-level view and categorization of pair programming sessions based on the two dimensions of pair member's respective task-relevant understanding of the system and software development was the first result I could expect practitioners to understand. I presented it to software developers and Scrum Masters of four different companies where it was met with resonance and only few open questions (see Chapter 13).

Up to this point, I referred to developers' understanding regarding **G** and **S knowledge** as "knowledge levels". However, after reflecting on the feedback from practitioners (see Section 13.2.2) and from reviewers (see Section *Writing* below), I found the notion of "**Knowledge Needs**" to be more appropriate because it emphasizes task-specificity over developer careers.

### *Data Analysis*

What was missing at this point was the integration of all the pieces to a grounded theory, so I switched to *selective coding*. I noticed that I had not yet completed the circle of how *knowledge* affects a pair's **Togetherness** and thus **Fluency**, which led me back to the utterance level of the PP process. I introduced the notion of **conversational role** (in particular the ● *corrective activity*) and could now properly describe the other **Fluency** extreme: **Focus Phases**.

While it was clear to me from its inception in Phase 8 that the two knowledge dimensions of a pair member in session are highly task-specific, only through systematic comparison of all sessions the importance of a pair's **Target Constellation** dawned on me. With this relevant context property, each pair's knowledge trajectory made sense, and I could identify the three parts of the overall session dynamics: Closing the **Primary Gap**, closing the **Secondary Gap**, and seizing the **G Opportunity**.

### *Writing*

After selective coding, I finally settled on the order and content of the result chapters of this thesis, and worked my way up from the utterance level, over activities, to **Episodes**, and then whole sessions. The publication "Explaining Pair Programming Session Dynamics from Knowledge Gaps" (Zieris & Prechelt, 2020a) is another result of this phase. Ultimately, the reviewers' feedback triggered the reframing of the concept of a developer's "knowledge levels" to "**Knowledge Need**".

## 12.10 Phase 10: Finishing the Thesis

### *Writing*

The last big phase of my PhD project was finishing this very document, in particular Chapters 2 to 4. I conducted an in-depth literature study (on PP in education, and involving more quantitative studies than just the meta-analysis of Hannay et al., 2009), looking for indicators that other researchers had also seen the phenomena I discovered and described, even if their reports only hinted at them as a side-note. For me, this led to an understanding of the difference and, more importantly, commonalities across very different research approaches on the same

topic, that allowed for a bigger and more complete picture of what pair programming actually is than would have been possible by an up-front literature analysis. The results of these efforts are Chapter 2 and the *Discussion of Related Work* sections at the end of the result chapters.

Similarly, I dived deep into the methodology literature to use the proper terminology to explain my own method in Chapter 4. Although qualitative approaches are becoming more common in software engineering research, there are still some reservations, misunderstandings, and misconceptions. This is why I wrote Chapter 3.

## Chapter 13 Evaluation

---

13.1 Purpose and Structure of this Chapter . . . . .	349
13.2 Member Reflection . . . . .	350
13.2.1 What to Validate? Ideas and Practices. . . . .	350
13.2.2 How to Evaluate? Interviews and Workshops. . . . .	351
13.2.3 Member Reflection Results . . . . .	352
<i>Ideas 1 &amp; 2: S vs. G knowledge and Knowledge Needs • Idea 3: Initial Constellations • Idea 4: S over G knowledge • Idea 5: Modes of Knowledge Transfer • Practice 1: Pair Forming • Practice 2: Set Session Goal • Practice 3: Reflect on Trajectory</i>	
13.2.4 Summary and Consequences . . . . .	355
13.3 Eight Criteria for Qualitative Research. . . . .	356
13.3.1 Worthy Topic . . . . .	356
13.3.2 Rich Rigor . . . . .	357
13.3.3 Sincerity . . . . .	357
13.3.4 Credibility . . . . .	358
13.3.5 Resonance . . . . .	358
13.3.6 Significant Contribution. . . . .	358
13.3.7 Ethics . . . . .	360
13.3.8 Meaningful Coherence. . . . .	360

### 13.1 Purpose and Structure of this Chapter

The goal of my work is to understand how knowledge transfer in pair programming works and to formulate my results in a way that is understandable and relevant for practitioners. I described my overall research approach in Chapter 4, how I arrived at my results in Chapter 12, and the results themselves in Chapters 5 to 11.

As explained in Section 3.3.5a, I use the eight quality criteria proposed by Tracy (2010, see also my discussion in Section 3.2.4) to evaluate my research. Tracy’s criteria of *resonance* and *credibility* are closely connected in my case: Findings that practitioners do not deem credible are unlikely to resonate with them, and vice versa. The most important method to evaluate my findings is *member reflection* with industrial software developers. I describe my approach, the results from discussions with practitioners in four companies, and how I incorporated the feedback I received in Section 13.2. I go through all eight criteria proposed by Tracy in Section 13.3.

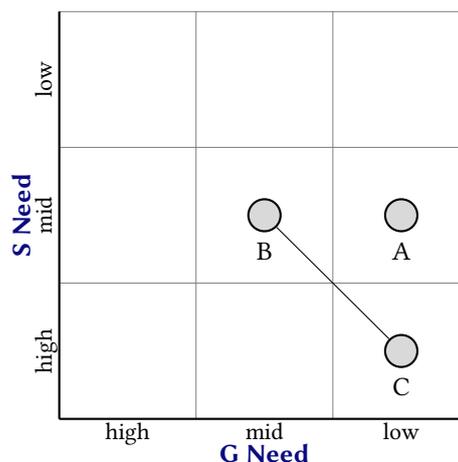
## 13.2 Member Reflection

Member reflection is collaborating with study participants to learn whether one’s research is comprehensible and perceived as meaningful, which in itself yields new data for further research (Tracy, 2010, p. 844).

### 13.2.1 What to Validate? Ideas and Practices

I introduced many concepts to characterize various aspects of how pair programmers transfer knowledge. Some concepts are building blocks for higher-level concepts, such as the *conversational roles* of individual utterances and activities (★, ▲, ▼, ●, and ◆) are for a pair’s *Fluency* (see Chapter 6) or as *Explanations/Explanation Elicitors* are for *Episodes* and their *Modes* (see Chapters 8 and 9). These are useful for fine-grained descriptions. However, they are probably of limited value for practitioners because they account for too small fractions of a PP session to be memorable enough to resonate with the developers when explained to them. The high-level concepts from Chapter 11, in contrast, refer to PP sessions as a whole and are probably more relatable. I therefore chose the following ideas for *member reflection*:

- **Idea 1: S vs. G knowledge** Two classes of knowledge are relevant in PP sessions. One is system-specific *S knowledge*, which includes requirements, system architecture, implementation details, configuration, scripts, and so on; the other is generic *G knowledge*, which is about programming languages, design patterns, frameworks, libraries, etc.
- **Idea 2: Knowledge Needs** Individual developers can be characterized according to how much *S* and *G knowledge* they do not yet possess for working on some specific task. For instance, one developer may have a medium *S Need* but only a small *G Need* for some refactoring task (see A in Figure 13.1).
- **Idea 3: Initial Constellations** The *Knowledge Needs* of two pair members form their *Initial Constellation*, of which there are only few different types (such as B-C’s *Complementary Gaps* in Figure 13.1), each with their characteristic session dynamic.
- **Idea 4: S over G knowledge** Within-pair differences in *S knowledge* affect a session’s dynamic more than *G knowledge* differences.
- **Idea 5: Modes of Knowledge Transfer** Pairs approach their individual and collective lacks of knowledge in different ways, e.g., they ▨ *Push* or ▨ *Pull* existing knowledge, or acquire new knowledge in ▨ *Co-Production* or ▨ *Pioneering Production*.



**Figure 13.1:** A G-S chart for visualizing individual’s *Knowledge Needs*, a pair’s *Initial and Target Constellation*, or their trajectory.

Additionally, I formulated practical ideas on how to make use of these concepts in everyday software development. Considering that reflecting on one’s pair programming process mid-flight might be difficult (especially for pairs with reduced *Togetherness*), I devised the following three practices to be done before and after a PP session:

- **Practice 1: Pair Forming** Consider *task-specific* knowledge when forming pairs. This is an extension of what Beck (1999, p. 59) proposed: “If you have responsibility for a task in an area that is unfamiliar to you, you might ask someone with recent experience to pair with you”. Specifically, teams could use a G-S chart to discuss the *Knowledge Needs* of

all team members for an important upcoming task to form a pair (and/or amend the task) to get a favorable **Initial Constellation**, e.g., one with a **G Opportunity**.

- **Practice 2: Set Session Goal** Once a pair is set, the two developers discuss their **Target Constellation** before working on the task: Do both pair members need to understand the task-relevant system parts enough to continue the work, or is some **Primary Gap** tolerable? Is there a **G Opportunity** that could be seized?
- **Practice 3: Reflect on Trajectory** After the session, the pair reflects on their trajectory with questions like: Did we reach our **Target Constellation**? Did we underestimate our **Knowledge Needs**, i.e., are there lacking pieces of **S** or **G knowledge** whose relevance we were not aware of? Should we discuss these with the rest of the team?

### 13.2.2 How to Evaluate? Interviews and Workshops

I performed *member reflection* in three ways: First, as part of the **reflective interview** with the pair members shortly after their session had been recorded (see Section 4.3.2e for details); second, in **workshops** during which I present my research and discuss it with groups of developers (as described in Section 4.3.2g); and third, in **1-on-1 interviews** that follow the same structure as the workshops (details follow), but involve only one practitioner.

A reflective interview after a PP session has the advantage that the developers have a concrete recent pair programming experience they can refer to and which I also know about. However, these interviews are rather ad hoc and last 30 minutes at most. The team workshops and 1-on-1 interviews take between 1 and 2 hours which gives me time to address more aspects of my findings and allows the participants to reflect longer. The general **structure** of my presentations in the workshops and 1-on-1 interviews is as follows:

- First, I explain **Ideas 1 and 2**, that is, the two dimensions of task-specific **S** and **G Needs** which each pair member brings into a session (as explained in Section 11.2). It is important to note, though, that I did not speak of “**Knowledge Needs**” yet, but of “**knowledge levels**”. I developed the terminology of **S** and **G Knowledge Needs** only in the last steps of selective coding based on the feedback I received. The underlying concept remained the same, but instead of saying ‘*Developer A has a high S Need*’, I would have said ‘*Developer A has a low S level*’, and vice versa. I come back to what motivated this change in Section 13.2.4.
- I continue with **Idea 3**, in particular with explaining the five **Initial Constellations** I had seen so far (the sixth constellation, **One-Sided G Gap**, I saw in company P for the first time, *after* the team workshop). To illustrate the dynamics of each constellation, I sketch out particular session trajectories: Session BB1 for **No Relevant Gaps**, session CA2 for **One-Sided S Gap**, session AA1 for **Two-Sided S Gap**, session DA2 for **Complementary Gaps**, and session OA1 for **Too-Big Two-Fold Gap** (see Figures 11.3c, 11.4, 11.6b, and 11.7b).
- While doing so, I ask the participants for their personal experience with the discussed constellations, e.g., whether they encountered them recently, how these turned out, what other dynamics they recall, and whether they can share further anecdotes. This is meant to evaluate **Idea 4**, i.e., whether the practitioners share my observation that **S knowledge** differences have more impact on a session than **G knowledge** differences and whether there are other types of relevant knowledge.
- When time allows, I provide further details on the different ways how pairs can address their knowledge gaps (**Idea 5**, **Modes** of knowledge transfer).
- Last, we discuss whether and how some of the proposed **Practices 1 to 3** would fit in the developers’ everyday work.

Context	Format	Length	Participants	Content
Company O	Workshop	1/2 h	2×SM, 1×PO	I1–I3, P1
Company P	1-on-1 Interview	1 h	1×SM	I1–I3 & I5
Company P	Workshop	2 h	6×D, 1×SM, 1×PO	I1–I3 & I5, P1
Company Q	1-on-1 Interview	1 h	1×TM	I1–I5, P1–P3
Company R	Workshop	2 h	10×D, 2×TM, 1×SM	I1–I4, P1
Company P	Reflective Interview (3×)	1/2 h	2×D/interview (total: 3×D)	I1–I3 & I5, P2–P3

**Table 13.1:** Member reflection activities with their format, length, numbers and roles of participants (SM - Scrum Master, PO - Product Owner, TM - Technical Manager, D - Developer), and discussed Ideas (I1–I5) and Practices (P1–P3).

I did not discuss all five Ideas and three Practices at all occasions; see Table 13.1 for a summary of all my member reflection activities. In particular, I was involved with four companies:

- **Company O:** About one year after recording sessions OA1 to OA10, I talked to two Scrum Masters and one Product Owner (who had formerly been head of development).
- **Company P:** At first, I had a 1-on-1 discussion with the Scrum Master (who has no technical background, but a sociology degree). He was intrigued by my concepts and organized a two-hour workshop with six developers, a product owner, and himself. Later, I recorded four sessions (PA1 to PA4) and conducted three reflective interviews, where I discussed the Ideas again and let the developers try out Practices 2 and 3 (*set a session goal and reflect on session trajectory*).
- **Company Q:** Company Q is a consulting company and all software development is done with pair programming. I performed a one-hour 1-on-1 interview (video call with presentation) with a technical manager.
- **Company R:** This, too, is a consulting company. The Scrum Master from company P got hired here and organized another two-hour team workshop. Ten developers, two technical managers, and the Scrum Master participated.

### 13.2.3 Member Reflection Results

Below, I summarize my insights from the member reflection. I refer to participants by their company and role (e.g., “P-SM” is company P’s Scrum Master). See Table 13.2 for an overview.

#### 13.2.3 a) Ideas 1 & 2: S vs. G knowledge and Knowledge Needs

The two dimensions of **S** and **G knowledge** were understood in all discussions. The O-SMs immediately started to characterize developers from their teams as to their typically available **S** and **G knowledge**; O-PO used the dimensions to characterize recent difficulties across all teams as a “*collective G gap*”. To Q-TM, the classic distinction of ‘expert vs. novice’ was “*too simplistic, too naive, offensive even*”, whereas **S** and **G knowledge** “*resonate better*”, they “*get better to the heart of the matter*”; he found thinking about ‘*Resolving an S-gap or a G-gap?*’ more compelling than ‘*Am I a novice?*’.

Overall, I therefore consider these concepts to be understandable for practitioners and useful for discussing pair programming. Nevertheless, I changed the name of one concept: The former “knowledge level” is now called “**Knowledge Need**”, which serves the same purpose as a concept, but has its polarity reversed. I explain this change in Section 13.2.4.

### 13.2.3 b) Idea 3: Initial Constellations

The five discussed **Initial Constellations** were also quickly understood. O-SM and O-PO independently identified **Complementary Gaps** as an interesting case, as the “*most real and valuable*” pairing; O-PO, one P-D, and Q-TM immediately recognized it from recent experience of working together with different roles (such as system administrator) or as a common consulting theme. Q-TM had recent experience with four constellations, but not with **Too-Big Two-Fold Gap**. Nevertheless, after just seeing the constellation **Too-Big Two-Fold Gap**, he immediately figured that the pair would have a difficult session ahead of them: “*Tricky, isn’t it? The developers do not get very far.*” I thus deem the concepts suitable for talking about pair programming and comprehending PP situations with and without first-hand experience.

O-PO found it very useful to have names for the constellations. Q-TM said it would be nice for developers to know these constellations in order to recognize them when they occur. He also remarked, however, that names like “**One-Sided S Gap**” are “*too technical*”. I did not change the names of the pair constellations, but the terms “**Primary Gap**”, “**Secondary Gap**”, and “**G Opportunity**” were introduced to speak more to practitioners.

### 13.2.3 c) Idea 4: S over G knowledge

With consulting companies Q and R, I wanted to assess the importance of application domain knowledge, a possible third type of “**D knowledge**”, which in my data only appeared in the form of identifiers in the source code (see Section 7.3.1d). I asked the respective groups for types of relevant knowledge in their setting *before* presenting my types of **S** and **G knowledge**.

I learned that **D knowledge** is seen to have little impact on PP session dynamics: Several R-Ds explained that there are only small **D knowledge** differences within their team, but sometimes gaps which only the (non-technical) client can fill. The far more serious issue is their overall lack of **S knowledge**, as due to the legacy system, a pair member with a mid **S Need** is already considered a rare ‘expert’. They pair-program to carefully build up and maintain **S knowledge**. Their scenario illustrates the practical relevance of pair programming not only to *transfer* existing knowledge but also to (re-)acquire new (or ‘lost’) knowledge together.

Q-TM ranked the problems imposed by **G knowledge** below those caused by lacks of **D** and **S knowledge** because **G knowledge** can be hired if needed or be distributed in the team through pair rotation.

Overall, I see support for the relative importance of **S knowledge** over **G knowledge** in PP sessions and the suitability of PP to build up such knowledge.

### 13.2.3 d) Idea 5: Modes of Knowledge Transfer

The participants in the workshops and 1-on-1 interviews appeared to understand the different **Modes** of knowledge transfer but did not seem to care much. There was, however, one incident through which the involved developers came to appreciate the utility of the concepts.

A quick reminder on session PA3: Developer P3 was irritated because he did not see the point of P1’s explanations which led to a tense situation (see **Push Episode** in Example 9.23). I talked to both developers about this directly after the session. P3 stated that P1 would often give such “*unwanted lectures*”, not only in this instance, but also in sessions PA1 and PA2 (which P3 witnessed as an outsider) and in planning meetings. P3 told his colleague, “*If I didn’t know you better, I’d perceive this behavior as arrogant. Better wait until the other one asks.*”, to which P1 replied, “*But there are topics for which the other one does not know yet that he should be asking questions!*”.

I pointed out to them that they just discovered the difference between an *internal* and *external Knowledge Want*, or between the  Pull and  Push Mode. P3 then acknowledged that many large and small explanations P1 provided (Push Episodes) were actually helpful, so shunning P1 from doing this is not a viable option. It was my impression that it helped both partners to have concrete terms to talk about what just happened. Their next session PA4 also had many Push Episodes from P1 to P3, and none of them was as tense as Example 9.23.

### 13.2.3 e) Practice 1: Pair Forming

Many participants *tended* to like the idea of forming pairs based on Knowledge Needs, but none of them reacted enthusiastically in this respect. Some raised *practical* concerns: O-PO and P-PO said their teams are too small to regularly offer more than one pairing to choose from. O-PO also believed all their pairings would have the same constellation.

Q-TM noted that developers would need to not only assess their Knowledge Needs but also feel comfortable communicating it to their partner or team: Admitting to a large S Need might be difficult for a seasoned developer. There was no opportunity to actually try out Practice 1, but I expect the issue of *social desirability bias*, i.e., the tendency to present oneself conforming to social norms (Zerbe & Paulhus, 1987), to be more of an issue than the inability to ‘correctly’ assess one’s Knowledge Needs. After all, the worst that could happen is becoming aware of a previously unknown lack of knowledge—which is an improvement.

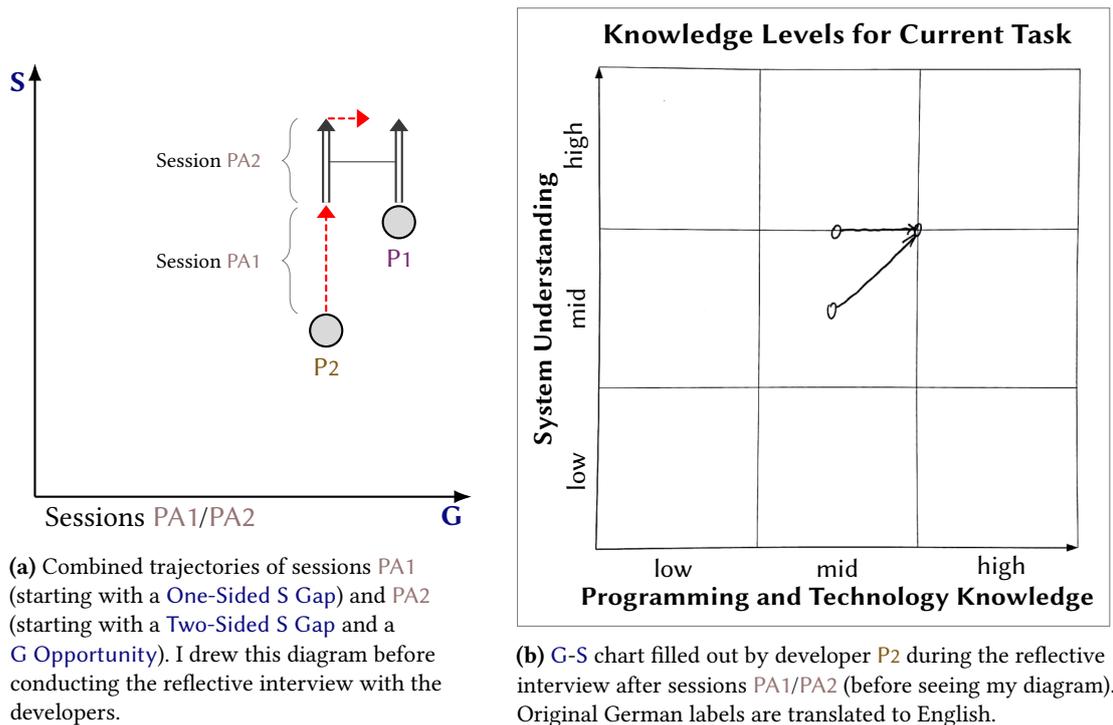
### 13.2.3 f) Practice 2: Set Session Goal

I asked the pair P1/P3 to discuss and plan their upcoming session PA4 by drawing onto a blank G-S chart their Initial and Target Constellation. They quickly agreed on a One-Sided G Gap (i.e., with no Primary and Secondary Gap) and the goal to work on P3’s G Need (regarding an object-relational mapper). After the session, both explained that filling out the chart did not affect their session, but *could* have, had there been discrepancies in their respective Knowledge Need assessments to be resolved. In other words, the pair had a high Togetherness which allowed them to correctly assess each other’s Knowledge Needs, and they considered the G-S chart to be useful for pairs who understand their own, but not each other’s Knowledge Needs.

### 13.2.3 g) Practice 3: Reflect on Trajectory

In the reflective interviews after sessions PA1/PA2 (with P1/P2) and PA3 (with P1/P3), I asked the developers to *individually* trace out and then discuss their trajectory without me intervening. Both times, the pairs had actually seized their G Opportunity but only remembered the Primary and Secondary Gaps consistently. The G knowledge transfer then resurfaced during the reflection (as P3 happily said when hearing P1’s version of session PA3: “*Right, I totally forgot about that. That was actually really cool.*”), showing that developers do not remember everything about their sessions and that reflecting on it can make it a more positive experience.

While the developers described their trajectories, I noticed that they understood the classes of S and G knowledge as I intended, but struggled with the task-specificity of the Knowledge Needs: They appeared to think of themselves having a relatively *fixed*, mostly experience-based knowledge level with little changes over the course of a session. In the reflective interview for sessions PA1/PA2, for instance, P2 said this about his drawing (see Figure 13.2b) after seeing his partner’s version: “*I drew this a bit like caricature, the arrows a bit larger.*” As can be seen in comparison with my tracing of the session (see Figure 13.2a), he did *not* exaggerate, but appeared to have accidentally used the degrees of low, mid, and high as intended.



**Figure 13.2:** Example of a developer reflecting on a recent PP session with a G-S chart (before I renamed “knowledge levels” to “**Knowledge Needs**”). He remembered closing the **Primary Gap** (S knowledge transfer from his partner), but not the **Secondary Gap** (collective acquisition of S knowledge). Instead of a seized **G Opportunity** (transfer from his partner), he remembered some collective acquisition of G knowledge.

### 13.2.4 Summary and Consequences

The only concept I felt was misunderstood is the *task-specificity* of a developer’s “knowledge level”: O-PO saw all team members at the same knowledge levels, which is highly unlikely (see Section 13.2.3e); P2 considered his trajectory to be exaggerated, which is was not (see Section 13.2.3g). After session PA4, developer P1 added another remark that made me question the notion of a “knowledge level”: He said he felt like “*moving left*” on the chart, which “*could not be*” since that would mean to forget G knowledge. What P1 meant to say was that he became aware of a knowledge gap where he did not expect one. From my analyses, I knew already that developers are not aware of all their knowledge gaps at once (that is the difference between a **Knowledge Need** and a **Knowledge Want**). But for P1, there was no way to fit this into the concepts I explained to him. To address this issue, I changed the concept name from “knowledge level” to “**Knowledge Need**” and flipped its direction (i.e., a ‘*high knowledge level*’ became a ‘*low Knowledge Need*’, and vice versa). I did not have the chance to ‘test’ the new notion, but I expect it to be often better and never worse than the old formulation.

Overall, the concepts were well received and mostly understood by the practitioners, and at least in one instance helped resolving a conflict between two developers who then had a more enjoyable follow-up session together. The proposed practice of *reflecting on a session trajectory* can help developers remember, appreciate, and possibly retain pieces of knowledge that were acquired or transferred through pair programming. Although I did not observe it, practitioners expect *forming pairs based on their Knowledge Needs* and *setting a session goal* to also have positive effects.

Aspect	Evaluation
Idea 1: S vs. G knowledge	✓
Idea 2: Knowledge Needs	✗
Idea 3: Initial Constellations	✓
Idea 4: S over G knowledge	✓
Idea 5: Modes of Knowledge Transfer	✓
Practice 1: Pair Forming	(✓)
Practice 2: Set Session Goal	(✓)
Practice 3: Reflect on Trajectory	✓

**Table 13.2:** Summary of member reflection. In particular: Which Ideas were understood and perceived as meaningful, which Practices are actionable and beneficial?

### 13.3 Eight Criteria for Qualitative Research

Tracy (2010) compiled eight general quality criteria for qualitative research. Each of the subsections to follow begins with a formulation of her criteria that is reduced to the aspects relevant for the goals of my work, that is, to identify and describe mechanisms of knowledge transfer in pair programming that are relevant for industrial practitioners.

#### 13.3.1 Worthy Topic

*The research topic needs to be relevant, significant, and interesting.*

The applications of software engineering research are often bound to some technology or type of software development (such as *artificial intelligence*, *open source software development*, or *mobile development*), or limited to a small range of software development tasks (such as *test case generation* or *build-infrastructure maintenance*). While many of these are or may become relevant to everyday industrial software development, their scope is yet limited.

**Pair programming**, in contrast, is a way of developing software that is amendable to every aspect of software development in all application domains, regardless of concrete technology. Although meaningful numbers are difficult to obtain, at least *one third* of all software developers work in pairs at least some of their time (see Section 2.3.1b). They expect a number of positive effects in terms of higher technical quality and developer abilities and team processes, many of which directly relate to the transfer and/or collaborative acquisition of *knowledge* (see Section 2.3.1c).

For researchers, the question of whether or not pair programming is ‘worth doing’ is answered with “it depends”. But on *what* it depends, what actually happens *during* pair programming, and how observable effects come to be is not fully understood (see Section 2.3.5).

Given the universality of the practice and the central role of knowledge in both software development in general and pair programming in particular, I deem *knowledge transfer in pair programming* to be a quite worthy topic.

### 13.3.2 Rich Rigor

*Analyze enough appropriate data to allow for complex and rich explanations.*

As Tracy (2010, p. 841) summarizes, *richness* comes from a variety of data sources and contexts. The basis of my in-depth qualitative analysis were 27 recorded pair programming sessions (plus dozens of questionnaire sheets and fieldnotes from observations and interviews) that were collected in ten different industrial contexts (see Section 4.3.4d and Tables 4.2 and 4.3).

In total, the session material has a length of about 40 hours. On the one hand, this is a limited data set which only represents a small fraction of the respective teams' social reality. On the other hand, however, this data is *rich* in another sense described by Tracy (*ibid.*, p. 841): It allows to “*see nuance and complexity*”. As the recurring example from the beginning of session JA1 illustrates, just a couple of minutes of pair programming contains many different phenomena, which become visible through the different analytic angles I took in the results chapters. Most of the over 100 other examples in this document I only used to illustrate a *single* type of phenomenon, but they have much more depth and could serve to illustrate many other phenomena as well. The verbatim parts of all examples cover about two hours; there are many more analyzed instances for the reported concepts, which I did not include in here.

The *nuances* and *complexity* of the examples are met by my concepts: Most of them have meaningful substructures, such as the five types of **Explanation Elicitors** (see Section 8.2) or the seven types of outcomes of knowledge transfer **Episodes** (see Section 9.2.2), and they are densely connected (as the many cross-references indicate).

### 13.3.3 Sincerity

*The researcher's role is transparent, including biases, goals, mistakes, etc.*

In Chapters 4 and 12, I discuss the blueprint of my research and the actual research process. I highlight some key aspects below.

**Goal, Bias, and Mistakes** My goal was to (a) understand *knowledge transfer* in pair programming and (b) to formulate relevant advice for *practitioners*. For some time during my research, part (a) was my main goal. This focus led to the frustration of not being able to find behavioral patterns and anti-patterns through analyzing pre-recorded data alone (Sections 4.2.2, 12.6, and 12.7). I cannot know whether shifting my attention to part (b) earlier would have helped me, but after collecting new data and spending more time with practitioners, the pieces began to come together (Sections 12.8 and 12.9). I am a software developer myself and enjoyed talking to my participants not only about pair programming but also about software development in general. This helped establishing trust for data collection (Section 4.3.2f) and provided me with more context information for interpretation (Sections 4.5.1b and 4.5.2c).

**Transparency** I described in detail ...

- ... where the analyzed data comes from, who else was involved in collecting it, and what the limitations and unintended effects of the data collection were (Section 4.3);
- ... how I analyzed the data (Section 4.5);
- ... the many steps I have taken to arrive at the Grounded Theory presented in this thesis (Chapter 12); and
- ... how practitioners reflected on my findings (Section 13.2).

### 13.3.4 Credibility

*Results are trustworthy enough for readers to act on them.*

**Thick Descriptions** The first of two approaches I chose to achieve this goal are *thick descriptions* (Tracy, 2010, p. 843) with plenty of concrete detail in the well over 100 examples throughout this thesis. This approach is geared towards readers of this document, who should be able to immerse themselves into the scenes I described. The underlying idea is to *show rather than tell*, allowing readers to come to their own conclusions (*ibid.*, p. 843). Of course, I report my conclusions as well, but I tried to keep them separate from the descriptions.

**Member Reflections** The second approach addresses the intended recipients of my research results: Software developers. I performed *member reflection* (*ibid.*, p. 844) by discussing theoretical concepts and three possible practical applications as described in Section 13.2. Most of the discussed concepts (S vs. G knowledge, Knowledge Needs, Initial Constellations, S over G knowledge, and Modes of knowledge transfer) were understood and considered valuable vocabulary for talking about pair programming. In the only instance where I had doubts the participants understood the nuances as intended, I renamed the concept accordingly from formerly “knowledge levels” to “Knowledge Needs” to better reflect its task-specificity.

Among the discussed practical applications, *reflecting on a session trajectory* helped developers remember and appreciate knowledge they had transferred in their PP session. Although I did not observe effects of two further ideas in practice, *forming pairs based on Knowledge Needs* and *setting a session goal*, developers expected them to be useful, too.

### 13.3.5 Resonance

*The research influences, affects, or moves the reader through transferability.*

**Transferability** As Tracy (2010, p. 845) puts it, the reader should be able to “*intuitively transfer the research to their own action.*”

From the workshops and interviews in four companies I know that practitioners find the topic of knowledge transfer in pair programming and my observations interesting, and want to embed them in their daily work (see Section 13.2). I can clearly say that my research *resonates* with practitioners.

As for this particular piece of writing, I took great care of making my thesis accessible despite its sheer volume and level of detail. Although I cannot assess the effect of these efforts, I went through multiple iterations on how to present my examples until I found the final style. I hope that the way of presenting concrete data allows readers with a practical background to see similarities to their own context, even if they did not take part in one of my workshops.

### 13.3.6 Significant Contribution

*The research extends knowledge, improves practice, or generates research ideas.*

**Theoretical Significance** There are a number of findings that may affect future PP research:

- Prior pair programming publications regarding industrial practice, such as surveys or reports written by practitioners, often suffer from unclear terminology. Future work should consider the following conceptual distinctions:

- Surveys on adoption rates are often difficult to interpret because two meanings of “pair programming” are conflated: Pair programming as a **work mode**, i.e., the occasional, possibly ad hoc collaboration; and pair programming as a **practice**, which is a strategic decision of when and how to pair (Sections 2.3.1a and 2.3.1b).
- Discussions of potential PP benefits are often vague—e.g., in treating “permanent review” and “higher quality” as conceptually equal and independent aspects—and confuse observable **effects** and underlying **mechanisms** (Section 2.3.1c).
- I applied and extended the ideas and concepts base layer for qualitative pair programming research (Salinger & Prechelt, 2013). I introduced *base coding* as a technique to augment *open coding* from the Grounded Theory Methodology (Section 4.5.2b) and I refined the base concepts by introducing a new property: the **conversational role** (Section 6.2.2).
- I found very little indication that *having the keyboard* affects the behavior of pair programmers. There was only one developer who would have preferred to look at the source code himself, but instead asked his partner sitting near the keyboard to open certain files (i.e., instead of **Pioneering**, he **Pulled**, see Section 9.3.3). This supports earlier findings by Bryant et al. (2008): ‘Driver’ and ‘navigator’ are not appropriate for explaining what pair programming is.
- My counterproposal to explain what makes two software developers a *pair* is the notion of **Togetherness** (Chapter 6). Higher **Togetherness** allows for a better pair process **Fluency**. Pairs may find themselves in situations where they need to actively **Maintain Togetherness** regarding the following five aspects: Shared understanding of the software system, shared understanding of software development, one shared plan, workspace awareness, and no language barrier. Communication is key for all of these.
- In pair programming sessions, the ‘transfer’ of existing knowledge and the collaborative acquisition of new knowledge are not separate concerns, but actually deeply intertwined (Chapter 9). Both occur across and throughout PP sessions (Sections 11.3 and 11.4).
- I have shown that the often-used notions of ‘expert’ and ‘novice’ are not appropriate for characterizing the pair programming behavior of software developers (Section 11.2).
- Finally, I can provide a two-part explanation for why previous PP studies, especially controlled experiments, yielded little insight despite decades of research (Section 2.3.5): First, acquiring and transferring system or **S knowledge** is a large and important part of industrial PP sessions (Section 11.4). This aspect of pair programming is completely missing when experimental subjects work on small and isolated tasks. Second, pairs’ process **Fluency** can vary between **Focus Phases** and **Breakdowns**, depending on how well the developers maintain their **Togetherness** throughout their session. Studies which only measure time and code quality could not see this.

There are further topics and open questions that arise from my research on pair programming which I deem relevant and interesting. I summarize them in Section 14.3.

**Practical Significance** For my research, *practical significance* was an important and explicit goal (see Section 4.2.2). I already discussed that my results were well received by practitioners in four companies. There are more practical results which go beyond the concrete feedback I received through the member reflection activities summarized in Section 13.2:

- My interaction with the developers made the teams aware of how little thought they had previously put into what their pair programming *practice* actually entails. The management of company O, for instance, considers themselves an *all-PP company*. They were in a period of hiring many developers, and saw mandating everybody to work in pairs combined with regular shuffling of teams as a replacement for a structured on-boarding process. However, what I saw over a four-week period were only some

PP sessions and a widespread lack of technology knowledge, which the Scrum Masters became aware of through my research activities.

In company P, product owner and developers expected pair programming to yield “*all the benefits*”, but had no priority: Better design, fewer defects, faster progress, transfer of knowledge, and enjoyable work were all equally important, and each PP session was expected to have these effects. My workshop made them reconsider their priorities.

- I sent the Scrum Masters in company O an early draft of my findings (in particular Chapters 6 and 11), and they asked whether they could forward it to select team members as a recommended reading for professional growth (which they did at least twice).
- Even the ad hoc feedback I could give in the reflective interviews shortly after a session recording appears to resonate well with practitioners: Although there were *six months* between the reflective interviews for sessions KB1/KB2 and KC1/KC2, both developers K2 and K3 could remember what we talked about the first time.

Developers from companies K, O, and P specifically asked me for recordings of the reflective interviews and/ or their own PP sessions (which I provided to them).

It is my impression that my discussions with the practitioners made them reflect on their work, motivated discussions on which PP effects they actually care about, and made them see forming pairs considerably, setting sessions goals, and reflecting on recent PP sessions as opportunities to achieve these effects.

### 13.3.7 Ethics

*Effects of research actions on subjects and others are considered.*

**Procedural Ethics** All participation was based on informed consent (see page 146); no company or developer is identified by name in any publication (Section 4.3.2h). In addition, no personal information from particular PP sessions or reflective interviews was conveyed to colleagues or superiors (Section 4.3.2e).

**Relational Ethics** The reflective interviews and workshops were meant to give something back to the developers, even before I performed any in-depth analysis (Sections 4.3.2e and 4.3.2g). Additionally, I made session recordings and recordings of the reflective interviews available to the respective developers in those cases they were interested (companies K, O, and P).

### 13.3.8 Meaningful Coherence

*Theoretical framework, research method, and goals are aligned.*

Straussian Grounded Theory Methodology as such is based on a positivist epistemology (see Section 3.3.4a) which contrasts with my constructivist epistemological stance (see Section 4.5.1b). However, as Charmaz (2006, p. 9) argues, many of the practical research elements of Straussian GTM are *neutral* to the researcher’s epistemology. In particular, I found the coding practices to break up the data, to consider behavior in context with causes and consequences, and to integrate everything to a meaningful whole (i.e., open, axial, and selective coding) to be well applicable for my research. The necessary methodological extensions are described in Sections 3.4 and 4.3.3.

To ensure coherence beyond my own work, I integrated my findings with existing concepts found in various literature at the end of most results chapters (Sections 6.5, 7.4, 9.7, and 11.5.1).

## Chapter 14 Conclusion and Further Work

---

My goal was to understand how *knowledge transfer* in pair programming (PP) works and to formulate results that are meaningful to practitioners. I qualitatively analyzed 27 industrial PP sessions from ten companies and developed grounded theoretical concepts starting from individual *utterances* over knowledge transfer *episodes* and to whole *sessions*. I validated the high-level concepts with practitioners in four companies, none of which pointed to missing relevant elements. I therefore consider my overall theory of knowledge transfer in pair programming to be theoretically saturated.

I summarize my  $\mathbb{A}$  research contributions and  $\mathbb{Q}\mathbb{Q}$  advice for practitioners in Sections 14.1 and 14.2; I propose directions for further work in Section 14.3.

### 14.1 $\mathbb{A}$ Research Contributions

- An extensive **review** of practitioner and scientific **literature** on PP effectiveness, influence of knowledge, task types, and pair constellations, showing that the often employed quantitative methods do not explain the observable effects and that *qualitative* approaches are needed to understand the underlying processes and mechanisms (Section 2.3).
- A refined **qualitative research process** for collecting and analyzing data to understand pair programming process phenomena based on video recordings (Sections 4.3 and 4.5).
- The first detailed description of **Focus Phases** of high productivity, which occur in some pair programming sessions and had been reported by other sources before, as well as their antagonist, the **Breakdown**, which had not been reported before (Section 6.3).
- The concept of **Togetherness** to describe what makes two software developers work together *as a pair*. Handling its five factors well can lead to **Focus Phases**—and to **Breakdowns** when not: Making sure to have a *shared understanding of the system* and of *software development in general*, as well as maintaining *one shared plan*, and possibly dealing with *workspace awareness* and a *language barrier* (Section 6.4). Prior PP research mostly did not consider the pairs' processes; differences in **Togetherness** might explain the effectiveness variations observed in experiments.
- A taxonomy of **Topics** that are actually addressed in industrial pair programming sessions: By far the *most* knowledge transfer pertains to system-specific **S knowledge** and *only some* to general software development knowledge, or **G knowledge** (Section 7.3.1).
- The notion of knowledge transfer during pair programming being structured in **Episodes**: During each, the pair pursues a **Topic** in one of six knowledge transfer **Modes**, i.e., **Push**, **Pull**, **Parallel** or **Co-Production**, **Silent** or **Talking Pioneering** (Chapter 9).
- A characterization of pairs that is not based on hard-to-agree-on global developer expertise (no 'expert/novice'), but on task-specific **Knowledge Needs** (Sections 11.2 and 11.3).
- A **Grounded Theory** of PP session dynamics of pairs acquiring and transferring knowledge: A relative difference in task-relevant **S knowledge** is addressed first; a relative difference in task-relevant **G knowledge** is hardly a problem, and may even pose an opportunity to transfer such knowledge *after* the pair acquired enough **S knowledge** to work on the task (Section 11.4). Prior PP research, in which subjects often worked with unknown systems or *none*, could not have observed this industrially relevant dynamic.

## 14.2 Practical Applications

These pieces of advice were condensed from my observations and address practitioners directly.

### 14.2.1 Maintain Togetherness

During pair programming, you and your partner want to work *as a pair* to possibly achieve the benefits of better design, fewer defects, faster progress, knowledge transfer, and more enjoyable work. This **Togetherness**, however, needs to be **Maintained** throughout a PP session. There are some **problematic signs** to look out for:

- You do not understand the intentions behind your partner's utterances and actions. Also: You feel like your partner does not understand your intentions, e.g., she may take longer than normal to react, her reaction may not match your actions, or she may not react at all.
- You cannot evaluate a proposal your partner made. Also: Your partner does not evaluate your proposal, or makes a proposal of her own without addressing yours.

These are all *conversational defects* and they are worth clearing up. Not all defects are necessarily problematic, but they may point to underlying problems such as:

- **Lack of Shared System Understanding:** You and your partner have no common mental model of the software system, no common way of referring to its parts and aspects.
- **Lack of Shared Understanding of Software Development:** You and your partner have no common toolkit (e.g., known libraries or tools) or way of approaching certain types of programming tasks and issues (e.g., writing tests before production code or using a debugger).
- **No Shared Plan:** You and your partner do not have a common conception of what steps to take to achieve which goal and where you are in the process.
- **Limited Workspace Awareness:** You and your partner cannot fully perceive each other's actions in the code editor, see and read the same things on the screen, or notice what each of you is looking at.
- **Language Barrier:** You and your partner may have difficulties expressing or understanding each other thoughts on a phonetic, lexical, or semantic level. Even within the same natural language, words and phrases do not mean the same to everyone.

All these problems reduce your **Togetherness**, but all can be mitigated by communicating explicitly. Not every PP session will have problems in each area, but too many unhandled problems might result in a **Breakdown** of the pair process, which then has none of the expected benefits and may be even worse than working alone. Addressing all the areas, however, might lead to a **Focus Phase** where you and your partner complete each other's thoughts and make fast progress.

**Further Reading:** Chapter 6 on *Process Fluency and Pair Togetherness*.

### 14.2.2 One Topic at a Time

Sometimes during a session, there are multiple things you want to understand or clarify at once, especially since there are two members in a pair who may want to pursue different topics. Experienced pairs manage to temporarily **Limit their Scope** such that only one topic is relevant at every given moment. Once they are done with it, they **Return Explicitly** to the original topic to make sure to not lose track. Unexperienced pairs, however, may start new lines of inquiry whenever something catches their attention, which leads to many expensive context switches and makes backtracking more difficult.

**Further Reading:** Chapter 10 on *Patterns of Episodes*.

### 14.2.3 Choose **Mode** of Knowledge Transfer

There are different styles for transferring existing knowledge and for acquiring new knowledge. Depending on the situation and your preferences, there are different **Modes** to choose from:

- **Pull vs. Push:** Knowledge which one partner already possess can be transferred either by a series of questions from the developer in need (**Pull**) or by explanations driven by the more knowledgeable pair member (**Push**). Pushing has the advantage that it may transfer knowledge whose lack the partner was not yet even aware of, which can also be confusing for her if the point of the push does not become clear soon. Some developers may also have difficulties giving pro-active explanations in push mode. Switching to an interview-style pull mode might help those pairs.
- **Co-Production vs. Pioneering Production:** Lacking knowledge can also be acquired through reading source code, using a debugger, or interacting with the application. Often, both partners are interested in the topic and engage in **Co-Producing** the knowledge. In case one partner is less interested, however, the other may **Pioneer** for a moment until she is satisfied and then continue to work together.
- **Silent Pioneer vs. Talking Pioneer:** Some developers prefer to read source code for themselves (**Pioneer**), even if their partner could explain it to them. If you decide to do this as a pair, the reader should be a **Talking Pioneer**, that is, to make clear what she is looking for and what she understood so far, so her partner can validate her findings and give useful pointers when the time is right. A **Silent Pioneer**, in contrast, makes it more difficult for the partner to follow along.

**Further Reading:** Chapter 9 on *Episodes of Knowledge Transfer*.

### 14.2.4 Embed Pair Programming Sessions in the Team Process

**Before the Session** Consider the technical task and the knowledge it requires about the software system as such and about software development in general (e.g., frameworks, technology stack, design patterns, testing strategies): What are your **Knowledge Needs**, i.e., what relevant knowledge does either of you *not* yet possess? Are there **One-Sided Gaps** where one of you knows more about some area, or **Two-Sided Gaps** where both of you lack knowledge?

A setting of **Complementary Gaps** can be mutually beneficial, because both partners can bring in some knowledge advantage. If your pair constellation is not yet complementary, maybe the task can be amended in a way that both partners' respective expertise can come into play, e.g., by keeping an eye open for code smells and possible refactorings.

A **Two-Sided Gap** regarding general software development knowledge, e.g., where both of you do not understand some technology, will probably not make for a good PP session if you also try to work on a technical task. Better choose a different task and/or pairing.

Discuss which **Knowledge Needs** you want to address in your session. Understanding the task-relevant parts of the software system is usually required for both of you, but sometimes only one needs to continue with the task and an unclosed **One-Sided Gap** may be tolerable.

**After the Session** Reflect on what either of you learned. Chances are that each of you remembers different episodes. Together, you get a fuller appreciation of which knowledge you transferred and acquired, and where newly discovered knowledge gaps are.

**Further Reading:** Chapter 11 on *Session Dynamics* and Section 13.2 on *Preparing and Reflecting on a PP Session*.

### 14.3 Further Work

My work is done, but there is more to do. The following two areas for further investigation occurred to me while considering the limitations inherent to my data (Section 4.3.4):

**Ad-hoc Pairings** I have no idea in which regards spontaneous pairings are different from the at least half-planned sessions that ended up in the recordings repository. Recording such sessions would probably require an always-on setup to get rid of session start-up times (similar to what Socha et al., 2015, 2016, did, see page 144).

**Application Domain Knowledge** Developers in consulting may encounter new domain concepts more often than my pairs, who work in their company's own product. **Knowledge Needs** of this type might influence pairs in different ways than **S** and **G knowledge** do.

Just as I used the *base layer* (Salinger & Prechelt, 2013), further pair programming research may build upon my methods and concepts. Based on interesting phenomena I have seen in my data but not analyzed, I deem the following two areas particularly relevant and insightful:

**Fluency and Togetherness** The **Fluency** of most analyzed sessions was **normal**, with both **Breakdowns** and **Focus Phases** being the exception. I do not think that splitting up the **Fluency** concept into more than these three levels is useful: Even though the appearance of **normal** PP process varies across sessions, these differences do not appear to matter much. Almost all analyzed pairs manage to achieve something useful in their sessions, with not many avoidable detours along the way. There are, however, three directions I consider useful:

- **Focus Phases** appear highly productive and enjoyable. But: Is there an actual difference between sessions with and without **Focus Phases**, or are they just an occasional side-effect of high **Togetherness**? If they are desirable, is there anything pair programmers can do (e.g., more strict **Scope Limiting**) to get more and longer **Focus Phases**?
- Similarly, further types of **Breakdowns** are worth investigating. I heard anecdotes about terrible pair sessions, but none of my recorded sessions came close to those stories. Assuming these stories are not over-dramatized, there appear to be practically relevant ways how PP can go 'wrong' which my concepts cannot describe.
- Finally, I only considered how pairs **Maintain Togetherness** regarding two factors: *Shared system understanding* and *shared understanding of software development*. I provide initial observations on how pairs deal with *language barrier*, *workspace awareness*, and *one shared plan* in Section 6.4.4. There might also be more factors influencing a pair's **Togetherness** besides the five I identified.

**Decision Making** The **Episode** concept can easily be transferred to other process aspects of pair programming, such as discussing design decisions. I also expect the **Propellor** concept to be applicable (i.e., one or possibly both developer(s) being active rather than reactive), and different **Modes** to exist, in particular something **Push**-like where one developer pitches an idea, and something **Co-Production**-like where both partner go back and forth on something. Similarly, the patterns of **Branching Wildly**, **Scope Limiting**, and **Returning Explicitly** strike me as not necessarily specific to knowledge transfer.

Years of qualitative research and building on the *base layer* led to the concept of **Togetherness**. Its five factors appear central to fully understanding how *pair programming* works: maintaining (1) a shared understanding of the system and of (2) software development, (3) one shared plan, (4) good workspace awareness, and (5) dealing with any language barrier. My Grounded Theory of *knowledge transfer* explains the mechanisms of the first two factors to a degree that is enabling and meaningful to practitioners; the foundation for the other three has been laid.



# Appendices



## Appendix A Own Publications

---

While working on this thesis, I (co-)authored a number of publications. Here, I list those that relate to my doctoral research and explain their relationship to parts of this document:

1. Stephan Salinger, [Franz Zieris](#), & Lutz Prechelt (2013). “Liberating Pair Programming Research from the Oppressive Driver/Observer Regime.” In: *Proc. 35th Int’l. Conf. on Software Engineering*. ICSE ’13 (NIER). IEEE, pp. 1201–1204. DOI: [10.1109/ICSE.2013.6606678](#)

We started developing a catalogue of empirically grounded roles for a richer characterization than the terms ‘driver’ and ‘navigator’ allowed (see my discussion page [83](#)). We only analyzed one session ([CA1](#)) and did not further this line of investigation. The idea behind the role of the “*task expert*”, however, would later resurface as the [S Need](#) of a developer (Section [11.2.1](#)): A *task expert* is a pair member with a low [S Need](#).

2. [Franz Zieris](#) & Lutz Prechelt (2014). “On Knowledge Transfer Skill in Pair Programming.” In: *Proc. 8th ACM/IEEE Int’l. Symp. on Empirical Software Engineering and Measurement*. ESEM ’14. ACM. DOI: [10.1145/2652524.2652529](#)

We introduce many of the basic concepts: [Target Content](#) and [Topic](#) (Section [7.3](#)); [Explanation Elicitors](#) (called “*explanation triggers*” back then), the [Clarification Cascade](#), and [Explanations](#) (Chapter [8](#)); and the notion of [Episodes](#) having a [Propellor](#) and being carried out in one of several [Modes](#) (Chapter [9](#)).

We also report the first elements of the “*subtle skill*” that is pair programming (in the spirit of Beck, [1999](#), p. 100) which would eventually lead the low-level concepts of [◆](#) conversational defects and [●](#) corrective activities (Section [6.2](#)), and also the high-level concepts of [Scope Limiting](#) and [Returning Explicitly](#) (Chapter [10](#)).

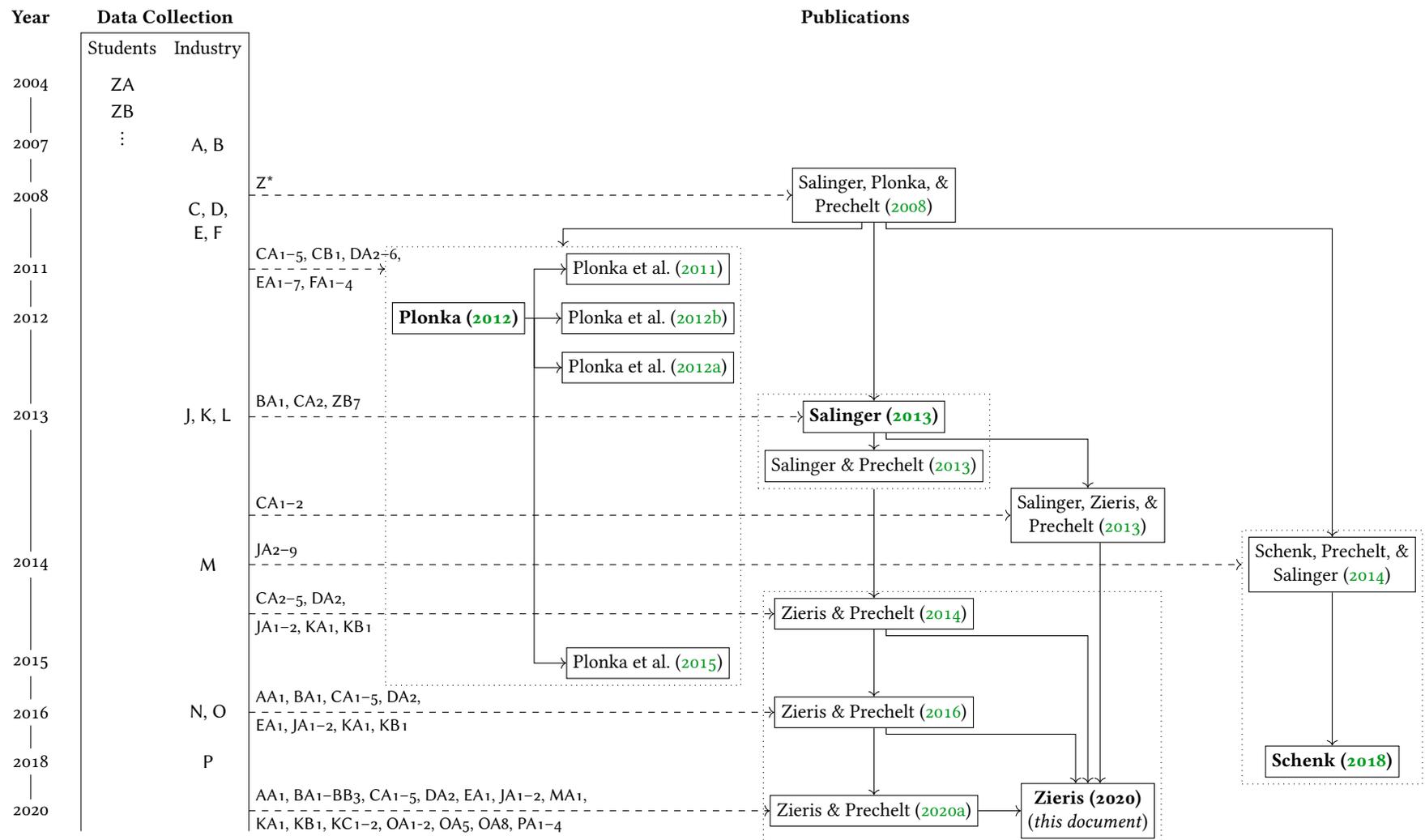
3. Lutz Prechelt, [Franz Zieris](#), & Holger Schmeisky (2015). “Difficulty Factors of Obtaining Access for Empirical Studies in Industry.” In: *2015 IEEE/ACM 3rd Int’l. Workshop on Conducting Empirical Studies in Industry*. CESI ’15. IEEE, pp. 19–25. DOI: [10.1109/cesi.2015.11](#)

We report on our experience of conducting pair programming research in the field, in particular the idea of recording naturalistic sessions, maintaining a session repository, and offering reflection interviews to the recorded developers (Section [4.3.2](#)).

4. [Franz Zieris](#) (2015). “Qualitative Analysis of Knowledge Transfer in Pair Programming.” In: *2015 IEEE/ACM 37th IEEE Int’l. Conf. on Software Engineering*. ICSE ’15 (Doctoral Symposium). IEEE, pp. 855–858. DOI: [10.1109/icse.2015.277](#)

I present an earlier version of my research process (Sections [4.3.2](#) and [4.3.3d](#)) and summarize the basic concepts of the time: the knowledge transfer [Modes](#) (Chapter [9](#)) and the [Clarification Cascade](#) (Section [8.2.1c](#)).

5. Lutz Prechelt, Holger Schmeisky, & [Franz Zieris](#) (2016). “Quality Experience: A Grounded Theory of Successful Agile Projects without Dedicated Testers.” In: *Proc. 38th Int’l. Conf. on Software Engineering*. ICSE ’16. ACM, pp. 1017–1027. DOI: [10.1145/2884781.2884789](#)  
We report a Grounded Theory on a team-level phenomenon. I supervised Holger Schmeisky’s master’s thesis, during which he collected data in three teams in two companies, and I advised him in qualitative data analysis. This line of work later led to the data collection and recording of pair programming sessions in company N (Section [4.3.1](#) and Tables [4.1](#) and [4.2](#)).
6. [Franz Zieris](#) & Lutz Prechelt (2016). “Observations on Knowledge Transfer of Professional Software Developers During Pair Programming.” In: *Proc. 38th Int’l. Conf. on Software Engineering Companion*. ICSE ’16 (SEIP). ACM, pp. 242–250. DOI: [10.1145/2889160.2889249](#)  
We introduce the notion of pair members being motivated to transfer knowledge by [internal](#) or [external Knowledge Wants](#) (called “*knowledge need*” then, Section [7.2](#)) and that pairs [Maintain Togetherness](#) (called “*resynchronization*” then, Section [6.4.4](#)).
7. [Franz Zieris](#) & Lutz Prechelt (2019). “Does Pair Programming Pay Off?” In: *Rethinking Productivity in Software Engineering*. Ed. by Caitlin Sadowksi & Thomas Zimmermann. Apress. Chap. 21. DOI: [10.1007/978-1-4842-4221-6\\_21](#)  
This is a non-scientific publication for a practitioner audience. We summarize the [Focus Phase](#) of pair [C3/C4](#) (see Section [6.3.3](#)), distinguish [S knowledge](#) and [G knowledge](#) (see Section [7.3.1](#)), and discuss three pair constellations [One-Sided S Gap](#), [Two-Sided S Gap](#), and [Complementary Gaps](#) (see Sections [11.4.2](#) to [11.4.4](#)).
8. [Franz Zieris](#) & Lutz Prechelt (2020b). *PP-ind: A Repository of Industrial Pair Programming Session Recordings*. arXiv: [2002.03121v3 \[cs.SE\]](#)  
This technical report describes the repository of all recorded PP sessions and how the data was collected. I wrote the report after Chapter [4](#) and reused several passages from my thesis in the report, in particular: Sections [4.2.3a](#) and [4.2.3c](#) on the basic decision to record and analyze PP sessions of professionals; Sections [4.3.2a](#) to [4.3.2c](#) on the general description of the data collection protocol; and Sections [4.3.1](#) and [4.3.4](#) on how different research interests reflect on otherwise neutral data collection.
9. [Franz Zieris](#) & Lutz Prechelt (2020a). “Explaining Pair Programming Session Dynamics from Knowledge Gaps.” In: *Proc. 42nd Int’l. Conf. on Software Engineering*. ICSE ’20. ACM. DOI: [10.1145/3377811.3380925](#)  
This is the paper version of Chapter [11](#) and Section [13.2](#). The paper was written after the thesis parts.
10. [Franz Zieris](#) (2020). “When Grounded Theory is Not Enough: Additions for Video-Based Analyses of Software Engineering Process Phenomena.” In: *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik, 24.–28. Februar 2020, Innsbruck, Österreich*. Ed. by Michael Felderer et al. SE ’20, pp. 153–154. DOI: [10.18420/SE2020\\_47](#)  
This is not a reviewed publication, but (the extended abstract for) a talk I gave at the SE 2020 conference. I presented my considerations of the limitations of Straussian GTM in the context of video-based analysis of software development process phenomena. Sections [3.3.5](#) and [4.6](#) formed the basis for that presentation.



**Figure A.1:** Timeline of data collection and major scientific publications originating in my research group, including my own work. Arrows between publications (—) indicate reuse of ideas or building on results; arrows from data collection (- -) indicate which PP sessions were analyzed. PhD theses are set **bold**.



## Appendix B Transcription Notation

---

In Section 4.5.2a, I described when and how I transcribed developer utterances from the pair programming sessions. My notation is based on that of Salinger (2013, pp. 451–459), but I amended it slightly. The following Table B.1 explains all notational elements used throughout the examples in this thesis.

Purpose	Notation	Example
Emphasis, notable stress in the utterance	Underlined	“Do we have to do it <u>now</u> ?”
Non-literal speech (quotations and figurative speech)	In single quotes	“You said ‘ <i>Let’s start over</i> ’, right?”
Low voice	In parentheses	“We could do this, (if we wanted).”
Unintelligible word(s)	Tilde in parentheses	“The (~) component.”
Speech pauses	Parenthesized dots (one per second)	“This is (...) weird.”
Computer interaction (e.g., typing and clicking)	Parenthesized commas (one per second)	“Meh (,,).”
References to source code identifiers	Typewriter font	“Let’s open class FooBar.”
Self-interruption	Three dots in parentheses with single bangs	“We need to (!...!) we have to change this.”
Interrupt by the partner	Three dots in parentheses with double bangs	“Could you open the (!!...!!).”
Cross-Talk, speaking over each other’s turns	Partner utterances in parentheses with double bangs	“This seems like (!!Why don’t we!!) like a bad idea.”
Quasi-utterance, reading screen contents aloud	In parentheses with pound signs	“And then (#deleteFile#), that one.”
Quasi-utterance, speaking while typing	In parentheses with double pound signs	“Here we need (##this.file##).”
Actions, things the developers do	In angled brackets with asteriks	“This <*looks to partner*> doesn’t work anymore.”
Paraverbal, almost like speech	In angled brackets	“Oh <laughs> we need to delete this as well.”
Anonymization	In angled brackets with double asteriks	“We should talk to <***lead developer***> about this.”

**Table B.1:** Transcription notation. Note that the developers spoke German in most sessions whereas identifiers in the source code are usually English. Since the developers incorporate them naturally in their speech as a means of “*conversational code switching*” (Gumperz, 1982, Ch. 4), they can actually ‘hear’ the mono-spaced font.



## Appendix C Pair Programming Sessions

---

Here I provide short descriptions for all 27 analyzed PP sessions (see also Table 4.3), additional annotated excerpts with more detail for some concepts discussed in the main text, and the untranslated transcripts (i.e., most of them in German, some in English) of all verbatim examples in chronological order with backlinks to the examples they were used in. Not all analyzed data has been transcribed (see Section 4.5.2a); see Appendix B for the transcript notation.

Part of this material was also analyzed by Plonka (2012), Salinger (2013), and Schenk (2018). I provide an overview in Appendix C.23.

### C.1 Session AA1

Experienced developers A1 and A2 work in a web-based content management system (CMS) for 2:30 hours to fix inconsistencies across different list views. They work both in the frontend code (written in Java) and in the backend code (written in Objective-C). They know their code base well, but still spend some time along the way to understand its peculiarities. See Section 4.4.1 for more details.

#### C.1.1 Focus Phases #4, #5, and #6

I discussed Focus Phases in Section 6.3.3 by using one example from session CA5 which mainly consists of deleting source code. Here, I provide three additional instances from the end of session AA1 in less analytic detail: Two revolve around editing existing code, the third is entirely verbal and no source code is changed.

##### Example C.1: Focus Phase #4 (AA1, 1:53:20–1:54:56)

Prior to this excerpt, the pair added a new parameter to a method signature. Now, they introduce the parameter at one invocation and then a replace fixed value with function call (two times). Both times, they (a) get an Object from a String-indexed map, (b) cast it to String and (c) convert the result to boolean. One index is already named while the other is not, so they introduce a named index in the form of a Java class constant.

```
274 return new CMReminder(manager.getObjectHandle(objectId), from, comment, objectType, path,  
275 false, recipients); // FIXME: isMirror
```

Figure C.1: Relevant excerpts of the Java code before Focus Phase #4

The pair starts with using an existing named index for the first case to get a value from the map (step a) and convert it to boolean (step c). Then, they consider the second case.

- (1) ★ A1: “Now we have the other, that’s still broken.”
- (2) ▲ A2: < \*navigates to other constructor call\* >
- (3) ▲ A1: “Yes, and we have to put something in there.”
- (4) ▲ A2: “It has to be both.”
- (5) ▲ A1: “It has to take care of both somehow.”
- (6) ★ A2: “We already had something (!...!)”

## Example C.1 (continued)

- (7) ★ A1: “Do we have to slam the Accessor in there?”
- (8) ◆ A2: “<cursor to fifth argument> XmlUtils was it, right? <types “XmlUtil.” then auto-complete opens>”
- (9) ▲ A1: “(#parseBoolean#) yes”
- (10) ▲ A2: “<chooses entry> (##parseBoolean##) (##data get CMReminderAccessor.KEY\_IS\_ACTIVE##)”

A1 immediately abandons his own ★ and follows A2’s idea. They first remember the necessity of casting to String (step b), and then go on to copy the code for steps (a) and (b) for the second case, define a name for the index in the second case, and finally consider the order of the two cases.

- (11) ★ A1: “<sees compile error, is surprised> Mhm.”
- (12) ▲ A2: “<introduces and deletes closing parenthesis> It’s not completely correct yet.”
- (13) ▲ A1: “Nope, you have to cast to String.”
- (14) ★◆ A2: “<copies current line to clipboard>”
- (15) ★ A1: “And the same again.”
- (16) ▲ A2: “<pastes line as sixth argument, changes fifth argument to “KEY\_IS\_MIRROR”, compile error is shown>”
- (17) ★ A1: “Was isMirror the first?”
- (18) ▲ A2: “Yes. <deletes left-over argument from before>”
- (19) ▲ A1: “OK.”
- (20) ★● “FIXME can go now, and cast to String”
- (21) ★ A2: “<opens tooltip on compile error on KEY\_IS\_MIRROR>”
- (22) ▲ A1: “Huh? Ah, I see, ok.”
- (23) ▲ A2: “<chooses “Create constant ‘KEY\_IS\_MIRROR’ in type ‘CMReminder’” which automatically creates the constant in that class>”
- (24) ★ “<moves created constant statement a few lines to the top, deletes constant value null>”
- (25) ★ A1: “Object’ is creative.”
- (26) ◆ A2: “<sets constant value to “isMirror”>”
- (27) ▲ “<changes type from Object to String>”
- (28) ▲ A1: “OK.”
- (29) ▲ A2: “<goes back to constructor call, opens help on other compile error, chooses “Cast argument 1 to ‘String’”>”
- (30) ▲ A1: “M-hm. Yes.”
- (31) ▲ A2: “<deletes FIXME comment>”
- (32) ▲ A1: “Not bad.”

Although some ideas by A1 do not get evaluated by A2, A1 is apparently satisfied with how smoothly the change went.

```

274 return new CMReminder(manager.getObjectHandle(objectId), from, comment, objectType, path,
275     XmlUtil.parseBoolean((String)data.get(CMReminderAccessor.KEY_IS_MIRROR)),
276     XmlUtil.parseBoolean((String)data.get(CMReminderAccessor.KEY_IS_ACTIVE)), recipients);

```

Figure C.2: Relevant excerpts of the Java code after Focus Phase #4

**Example C.2: Focus Phase #5** (AA1, 1:55:45–1:57:04)

The pair discovers the place in the backend code where the data structure for an API response is built. They identify which existing objects to use to include the additional data. See Figure C.3 for the relevant source code.

- (1) ★ A1: “Ah, now we’re getting somewhere.”
- (2) ▲ A2: < \*puts cursor in line 167 (##reminder setObject forKey isMirror##) \* >
- (3) ★ A1: “Ah, that’s a CMOBJect. That’s actually nice.”
- (4) ◆ A2: “(##reminder setObject forKey isActive##)”

```

155  CMOBJect *object;
156
157  while ((object = [e nextObject]) != nil) {
160      NSMutableDictionary *reminder = [NSMutableDictionary dictionary]
168      [reminder setObject: forKey: @"isMirror"];
169      [reminder setObject: forKey: @"isActive"];
170      [mapping setObject: reminder forKey: [object objectId]];
171      [result addObject: reminder]
172  }

```

**Figure C.3:** Code in the middle of the Focus Phase: A2 wrote two statements to add the keys `isMirror` and `isActive` to the data structure (lines 168–169). This code would not run as the first (anonymous) parameter of the respective calls to the method `setObject` is not set.

- (6) ▲ A2: “Ah, we already have an Object?”
- (7) ▲ A1: “Yes, we have a CMOBJect. We only need to call object `isMirror`.”
- (8) ▲ A2: “But the contents are missing. That’s bad.”
- (9) ● A1: “Ah, you mean the problem is that if we (!..!)”
- (10) ▲ “OK, for mirror it’s unproblematic, right? You can do it properly right away.”
- (11) ▲ A2: “(Right.)”
- (12) ▲ A1: “(Yes) And the active (!..!) ugh”
- (13) ▲ A2: < \*types [object isMirror] in line 169 (i.e., with isActive) \* >
- (14) ★ A1: “Now you’re doing something wrong.”
- (15) ▲ A2: “(Well, in a moment.) < \*continues typing ? @"true" : @"false" \* >
- (16) ● A1: “Wrong line.”
- (17) ◀ A2: < \*formats indentation \* >
- (18) ★ A1: “And another bracket (!..!) ◀ (ah, it’s right.)”
- (19) ▲ A2: < \*switches statements \* >
- (20) ▲ A1: “OK.”

```

168      [reminder setObject: [object isMirror] ? @"true" : @"false"
169                      forKey: @"isMirror"];
170      [reminder setObject:
171                      forKey: @"isActive"];
172      [mapping setObject: reminder forKey: [object objectId]];
173      [result addObject: reminder]

```

**Figure C.4:** Code at the end of the Focus Phase. The value for `isMirror` is now set programmatically; the value for `isActive` is not yet defined. The pair sets the static value `"true"` 16 minutes later.

**Example C.3: Focus Phase #6** (AA1, 1:57:38–2:00:55)

This **Focus Phase** begins shortly after #5. Here, the pair discusses the technical problem around retrieving an object's active status (which A2 noticed in line 8 and A1 understood in line 12 in Example C.2 above).

- (24) ★ A2: “The reminder concerns the object. This means, whether this thing is valid or invalid also depends on the view. <\*looks at A1\*>.”
- (25) ▲ A1: “Yes. (.) It really sucks. <\*looks at A2\*> (...) Bah!”
- (26) ▲ A2: “<\*pushes himself away from desk\*> Shit.”
- (27) ▲ A1: “In such cases we should distinguish the edited and the released and return the isActive for both and let the GUI decide what it wants.”
- (28) ◀ “<\*takes off glasses\*> Bah! That’s expensive, for these two.”
- (29) ▲ A2: “Yes, or the GUI says what it wants.”
- (30) ▲ A1: “(.) You mean a ‘prefer edited’ as an option attached, or what?”
- (31) ◀ “Yes, but even then this query gets crazy complicated, right? Because we have to join a lot of Contents.”
- (32) ▲ A2: “(.) We managed to join it for the other one, too.”
- (33) ▲ A1: “Yes, for the Tasks. For Tasks you now that there is an editedContent eventually. Or a committed.”
- (34) ▲ A2: “(.) Ah, I see.”
- (35) ▲ A1: “<annoyed> Here there can be none, or only a released, or only an edited, or what have you, or (!...!)”
- (36) ▲ A2: “<\*leans back\*> Ah! <laughs>”
- (37) ▲ A1: “That’s really stupid, isn’t it?”
- (38) ▲ A2: “(.) Yes.”
- (39) ▲ A1: “Yes.”
- (40) ★ “Is there an isActive for the Object as a getter for the released or something?”
- (41) ★♦ A2: “Well, let’s see what actually happens. You do a SELECT WHERE (!!M-hm!!) and get the objects (!!Yes!!). What you could do is (!...!) the query already is an Object, an Object WHERE constraint, right? <\*selects statement\*>”
- (42) ▲ A1: “M-hm.”
- (43) ▲ A2: “What you could do is to say, ‘OK, additionally, select all contents where for the Object-IDs *these constraints hold*’ <\*points to screen\*>, right? And also the Content-ID is either committedContent (!...!) nope, is either editedContent or committed or releasedContent of any of these Object-IDs (!!...!!)”
- (44) ▲ A1: “And the thing is inactive.”
- (45) ▲ A2: “Well, you can ignore this for now (!!...!!)”
- (46) ▲ A1: “Nope, because otherwise you would not get those without a content.”
- (47) ◀ “(.) Hold on! (!!...!!)”
- (48) ● A2: “No, listen.”
- (49) ▲ “You only do the fetch. The result is that all these thingies are pushed into the cache with just one SELECT. And then you can ask the Object. That’s fast.”
- (50) ▲ A1: “Even better. (.) You don’t need to pull all the contents (.) Only get the Object-IDs from the content table where the Object-IDs are in these things and it is either edited, released, or committed and valid and (!...!) I mean, active. See, and then you can simply set isActive to ‘Object-ID is in this list’”

## Example C.3 (continued)

- (51) ▲ A2: “‘Object is not a template’ and (!...!).”
- (52) ▲ A1: “Right, yes, and also ‘is not a template’, right.”
- (53) ▲ A2: “And also the Object-IDs from that list.”
- (54) ▲ A1: “Yes. isActive means ‘is template’ or ‘is in that list’.”
- (55) ▲ A2: “Exactly.”
- (56) ▲ A1: “That would probably be the cheapest way.”
- (57) ▲ A2: “Right. You can jot down how you’d imagine that, SELECT-wise. I’m just going to the bathroom for a minute. < \*gets up \* >”
- (58) ▲ A1: “OK”

## C.1.2 Misinterpretations

This is a description from my reconstruction efforts that illustrates the range of possible interpretations.

## Example C.4: Coding Surprises (AA1, 27:20–32:32)

The software uses different types of entries, all represented by subclasses of Node, and there are new as well as legacy implementations of Pages which contain lists of such Nodes. The developers look for the method that is responsible for rendering the label of a list entry.

## First Analysis

Both read in the source code of the current Page class and then almost speak simultaneously. A1 considers the possibility that a Node subclass is responsible, A2 thinks about the Pages:

- A1: “What does it do? < \*reads in method overview \* > Somehow (#prepareForRendering#)”
- A2: “The problem is (!...!) < \*switches to website \* >”
- A1: “Ah, the Nodes have such a render thingy, right?” *propose\_hypothesis*
- A2: “that’s a List.” *explain\_finding*
- A1: “Yes, ok. Ah! That’s still an old list page.” *agree\_finding*
- A2: “Exactly.” *agree\_finding*

For the next four minutes, they read through the code of several Page classes and superclasses, thus following A2’s *finding*. They do not find what they look for and reconsider the Nodes again, thus following A1’s *hypothesis*—which turns out to be correct.

## First Interpretation

They could have saved four minutes of their time if they just followed A1’s hunch. They had two equally good options and just chose the wrong one.

## Second Analysis

Considering the larger context of the session, however, it becomes clear that the pair already worked on a similar label-rendering issue in this session where the respective Node class (not a Page) contained the responsible logic. The pair could easily have checked A1’s hypothesis first.

## Second Interpretation

They should and could have considered both ideas, but instead followed A2’s path without considering A1’s proposal. This episode is therefore an example for low *Togetherness* because the partners did not make sure they understand each other’s ideas. At this time, I coded this episode as *Parallel Production*. This is also the state of understanding reported in Zieris & Prechelt (2016). (For the concepts of *Togetherness* and *Parallel Production*, see Sections 6.4 and 9.5.2, respectively.)

## Example C.4 (continued)

**Third Analysis**

Considering not only the time before the above excerpt, but also the time after reveals that A2 did, in fact, understand A1's proposal. About one minute into following the Pages idea he says:

A2: "Maybe it's the Nodes after all (,) Nope." *agree\_hypothesis + disagree\_hypothesis*

A2 first considers A1 proposal and then rejects it. Either way: He understood it and evaluates it. The German modal particles "ja" and "doch" also indicate that he refers back to A1's idea rather than to a fresh idea of his own (see also page 268 on German modal particles). The second interpretation cannot be true.

**Third Interpretation**

A2 did not simply ignore A1's idea, but just put in 'on hold'. Both developers understood each other's ideas without much communication. The episode is therefore not an example of low but of very *high* *Togetherness*.

**Conclusion**

Pair programmers do not verbalize all they think about. As a conservative approach, I therefore assume full mutual understanding among the pair and then carefully consider contrary evidence.

**C.1.3 Transcripts of AA1 Excerpts**

0:08:35

A2: <\*legt Telefonhörer auf\*> Pass auf: <\*\*\*Product Owner\*\*> geht nicht ran. Aber ich denke, das können wir auch ganz zum Schluss machen, weil ich bin eigentlich der Meinung, hm (!!...!!)

A1: Wir können erstmal das Icon nur machen, als erstes (!!...!!)

A2: Egal, nee. Ich meine (!!da ist es klar!!) Hör' mir doch mal zu Ende zu. Egal, ähm, wo wir's darstellen, wir stellen (!!...!) also beim Icon ist es wirklich klar, weil da heißt das Ding dann 'icon deactivated Punkt PNG'.

A1: M-hm.

0:08:58

(start of Example 7.5)

A2: Um den Text jeweils kann man einfach nen 'span inactive' drumrum machen, und dann kann man das nachher einfach mit nem Schalter durchstreichen oder nicht durchstreichen.

A1: <\*schaut zu A2\*> Ja.

A2: Und das heißt, die Frage können wir einfach am Schluss beantworten.

A1: OK. <\*schaut zum Bildschirm\*> Wie wird denn das momentan in dem da gemacht, das Durchgestrichene?

A2: <\*schaut zum Bildschirm, hebt Augenbrauen, schürzt die Lippen\*>

A1: Ist da auch schon irgendwie span-Tag irgendwiesowas drin? Vielleicht ist das ja schon der Fall. Kannst du mal Source irgendwie angucken? 'Frame source' irgendwie?

A2: <\*öffnet Kontextmenü (,,,) \*> Ist ja grauvoll.

A1: Nur 'Frame öffnen'. Kann der nicht irgendwie 'Source von diesem Frame'?

A2: (,,) <\*öffnet HTML-Code der aktuellen Seite\*>

A1: Hä? Nee, das wollten wir nicht.

A2: Na, dann muss ich doch den Firefox nehmen.

0:09:40

[A2 schließt aktuellen Browser und stellt die gleiche Sitzung im Firefox-Browser wieder her.]

0:10:55

A1: Dann (!!...!)

A2: Achso, 'Source Code', genau.

A1: Genau.

A2: <\*öffnet Kontextmenü\*>

A1: (#Aktueller Frame#) (#anzeigen#) gut.

A2: <\*öffnet Quelltext des aktuellen Frames\*>

A1: M-hm-hm.

A2: <\*sucht "test", selektiert den Treffer: <span class="inactive">test</span\*>

A1: Ah, (#span class inactive#), okay.

A2: Genau.

0:11:08

(end of Example 7.5)

A2: Und genauso müssen wir es eigentlich in allen Listen machen, und wenn man es nicht mehr durchgestrichen haben will, dann nimmt man einfach das (~) raus und die Sache ist erledigt.

A1: Genau, ok.

0:11:19

(start of Example 9.20)

A2: So, das heißt dann (!...!) 'Bearbeitung abschließen', zum Bleistift.

A1: Fangen wir damit an?

A2: Ja, würde ich sagen.

A1: Gut.

A2: <\*wechselt zur IDE, öffnet Dialog "Open Type"\*>

A1: Das ist die (!...!).

A2: <\*wechselt zum Browser\*>

A1: Wie heißt denn die, die Seite?

A2: <\*rechtsklickt im Browser, wählt "Seiteninformationen anzeigen" aus\*>

A1: (#FinishTasksPage#), ok.

A2: <\*wechselt zur IDE, tippt "FTP" in offenen Dialog und wählt FinishTasksPage aus\*>

0:11:45

(end of Example 9.20)

[...]

0:13:10

(start of Example 9.17)

A1: Ja, das heißt wir müssen das getIconPrefix im TaskNode mal überschreiben (.....) das fehlt.

A2: A-ha? Ist das sinnvoll (!!...!!)

A1: Ja, der wird zusammengebaut, der Icon-Name (.....) und der TaskNode im Endeffekt delegiert der nur an den ObjectNode. Und deswegen sind das alles da als static Klassen vorhanden, damit man den Code nicht dupliziert.

A2: Ah! Aber warum baut man das so umständlich zusammen?

A1: Das ist ne andere Frage.

A2: Das würde mich aber mal interessieren! <\*öffnet Aufruf-Hierarchie der aktuell selektierten Methode\*> Guck mal, das gibt's ja hier bei browse.html im renderIcon() <\*öffnet diese Methode\*> und da geht es darum, der macht das über die ViewConfig, 'ja oder nein'.

A1: Jaja, das macht der ja jetzt auch, soll er ja auch. Wir wollen einfach diese Methode überschreiben.

A2: Achso, achso, wegen dem an (!...!) und ein- und wegblenden und alles. Achso.

A1: Genau.

0:14:13

(end of Example 9.17)

[...]

0:16:20

(start of Example 7.2)

A2: Das fetchMiniObject holen wir uns eh, von daher (,,,) <\*fängt an zu tippen "fetchM"\*>

A1: Ja!

A2: <\*Auto-Vervollständigung zeigt zwei Methoden "fetchMiniObject", eine ohne Parameter und eine mit boolean allowMicroObject\*> Gibt's die MicroObjects noch?

A1: Ja. Da noch, bei dir noch. Hab ich (!...!) ist auf'm Arbeitsbranch. true!

A2: Reicht das?

A1: Ja, glaube.  
A2: (,) <\*tippt "true"\*>  
A1: M-hm. (end of Example 7.2)  
0:16:42  
[...]  
0:19:44 (start of Example 8.27)  
A2: Jetzt machen wir aber gleich nen richtiges MiniObject draus <\*entfernt Parameter, ändert Variablentyp\*>  
A1: Nein!  
A2: Ja, doch! Wir haben doch gesehen, dass er uns auch nur isActive fragt und dann berücksichtigt er den editedContent gar nicht.  
A1: Eh, ja! Und? Was passiert wenn der Task auf einem nicht lesbaren Objekt liegt?  
A2: (..) Achso, dann krieg ich ne Exception.  
A1: Jau. Genau.  
A2: Achso, das ist ja Fallback, genau <\*macht Änderungen rückgängig\*> richtig.  
0:20:13 (end of Example 8.27)  
[...]  
0:25:39 (start of Example 6.25)  
A1: So, aber wie wird denn der gerendert?  
A2: <angekelt> Warum hat denn der nen eigenes isActive? <zieht Luft durch die Zähne>  
A1: Keine Ahnung. Will ich jetzt auch nicht (!..!)  
A2: Nee, das will ich aber schon heile machen, weil <lacht> sonst macht der irgendwie was anderes als der andere und dann ist auch wieder Mist.  
0:25:51  
[Beide lesen im Quellcode und führen kurze Entwurfsdiskussion]  
0:27:20  
A2: So.  
A1: Ja, wir müssen jetzt gucken, wie der rausgerendert wird. Weil wir müssen ja sehen, was der (!..!)  
A2: Na da fangen wir mal bei der obersten Seite wieder an. <\*wechselt zum Browser, schaut auf A1s handschriftliche Notizen\*> Die oberste Seite, die du aufgeschrieben hattest, war 'Bearbeitung abschließen', richtig? Nee.  
A1: Wieso machen wir denn jetzt nicht die Version-Seite fertig?  
A2: Weil wir die anderen auch noch nicht fertig gemacht haben.  
A1: Hallo?  
A2: Wir haben die anderen auch noch nicht fertig gemacht. <\*wechselt zur IDE\*>  
A1: Wir waren doch gerade bei der, wieso machen wir die jetzt nicht fertig? (!!Weil ich!!) Du brauchst doch nur das Rendern des Textes ändern.  
A2: <atmet hörbar aus>  
A1: Nicht da <\*zeigt auf Bildschirm\*>, sondern in der ContentVersionsView-Dingsbums. Dann können wir das Ding wenigstens soweit (!..!) (~) funktioniert's dann wenigstens.  
A2: <\*öffnet ContentVersionsViewPage\*>  
0:28:00 (end of Example 6.25, start of Example C.4)  
A1: Was macht der denn? <\*liest in Methoden-Übersicht\*> Irgendwie (#prepareForRendering#)  
A2: Das Problem ist (!..!) <\*wechselt zur Website\*>  
A1: Ach die Nodes haben so nen Render-Kram, oder?  
A2: Das ist ne List.  
A1: Ja, achso. Ah! Das ist noch ne alte Listen-Seite.  
A2: Genau.  
0:28:26 (start of Example 8.21)

A1: Die müsste doch eigentlich umgebaut werden, auf das neue Design (oder sowas, ne?)  
A2: Nee, nee, die sollte so bleiben.  
A1: Die soll so bleiben?  
A2: Ja.  
A1: A-ha.

0:28:33 (end of Example 8.21)  
A2: Und die rendern sich (!...!) (#renderFields#) <\*navigiert zu dieser Methode und liest Quellcode (,,,,,,)\*>  
A1: <unzufrieden> Hm.  
A2: <\*liest weiter, scrollt dann nach oben (,,,,,,)\*>  
A1: Noch weiter oben.  
A2: <\*öffnet Oberklasse (,,)\*> Nee, die sind weiter unten. <\*zurück in vorherige Klasse (,,,,,,)\*> Vielleicht machen's ja wirklich die Nodes. (,) Nee.

0:29:12  
[Beide navigieren mehrmals durch fünf verschiedene Klassen und lesen Quellcode]

0:32:20  
A2: <\*markiert Zeile\*> Hehe, der sagt nämlich (#node.getRenderingClass#)  
A1: Aah!  
A2: Das muss also wiederum die Node können.  
A1: Alles klar.

0:32:32 (end of Example C.4)  
[...]

0:49:53 (start of Example 9.18)  
A1: Alles was er zurückkriegt vom <\*\*\*Backend-System\*\*> sozusagen, packt er da rein.  
A2: Alles?  
A1: Ja. Das heißt, wir können jetzt einfach mal in dem Node so tun als hätten wir da schon was gekriegt.  
A2: Und dieser TaskAccessor wird aber eigentlich auch nur für die TaskOverviewPage benutzt?  
A1: Ja. Wir müssten da nen neuen Key definieren, den müssten wir irgendwie da reinschlumpeln.  
A2: Was wollen wir'n wissen? isActive, ne?  
A1: Joah.  
A2: <\*beginnt zu tippen\*>  
A1: Ich überleg gerade (!...!) ja doch, nee, hm.  
A2: Wen fragt er denn da?  
A1: Na, das ist nen Spezialding.  
A2: Tatsächlich (,,,,) (#CMD\_LIST\_OVERVIEW#) <\*minimiert die IDE und wechselt zum Quellcode des Backends\*>  
A1: Deswegen, das ist wirklich nen Spezial (!...!) warum machst du jetzt wieder woanders weiter?  
A2: Na ich will mal gucken, wie das implementiert ist.  
A1: Ja wieso? Du definierst jetzt einfach nen neuen Key, sagst, den hättest du auch gerne, und äh, dann können wir im GUI schon soweit alles fertig haben, nur dass wir den Key nicht kriegen. Und dann machen wir das zum Schluss in den <\*\*\*Backend-System\*\*> rein. <\*wartet und schaut A2 zu\*>  
A2: <\*navigiert durch den Backend-Quellcode (,,,,,,,,,,,,,,,,,,,,,,)\*> Den interessiert das nämlich (nicht) (!...!)  
A1: Dem gibst du gar nicht an was du haben willst, sondern nur (!...!)  
A2: Der liefert gar keine Keys! <\*wechselt zurück zur IDE\*>  
A1: Nein! Natürlich, der liefert einfach ne Menge zurück. Du fragst ihn aber nichts Bestimmtes. <\*zeigt auf Frontend-Code\*> Das sind die Keys wie sie zurückkommen. Ist nur für's Zurückgeben.

A2: <\*liest im Frontend-Code (,,,,,)\*> Hä? Aber der muss den doch irgendwo fragen? Also, äh (!!...!!)

A1: Nein. Es gibt ein Kommando, das schickt ne feste Liste von, so 'ne feste Map zurück.

A2: Achso!

A1: Und hier wird nur definiert wie die Dinger heißen, damit man sie richtig rausholen kann.

A2: Achso <lacht> jetzt versteh ich (.) dich.

0:52:16 (end of Example 9.18)

[...]

0:59:12 (start of Example 8.19)

A2: <\*setzt erstes von zwei Argumenten eines Aufrufes\*> (##!isActive##) (..) achso (!!...!!)

A1: Es gibt keine Tasks auf Spiegeln.

A2: (...) Nich?

A1: Nö.

A2: <\*setzt zweites Argument auf false\*> Stimmt.

0:59:25 (end of Example 8.19)

[...]

1:05:29 (start of Example 7.3)

A1: <amüsiert> (#TclCode#)?

A2: <grinst> Das willst du nicht wissen.

A1: Ok <schnaubt> (...) dann ist ja alles gut.

1:05:38 (end of Example 7.3)

[...]

1:23:27 (start of Example 9.21)

A2: Ach das ist gar nicht Link, sondern LinkChecker. <\*öffnet die Datei\*> Ah für den gibt's aber nen Test. <\*öffnet auch die Test-Datei\*>

A1: <\*lehnt sich zurück, schaut im Büro umher\*> M-hm.

A2: <\*liest im Testcode\*> (#unreachableUrls#)

A1: <\*schaut zurück auf Bildschirm\*> Hm hm hm hm hm.

A2: (,,,,,,,,,,,,,,,,,,,,,,,,,,,,) (#testFetchUrls#) (#checkUrlsAreUnique#) (#unreachableForUrl-Row#) (,,,,,,,,,)

A1: <\*dreht sich wieder weg\*>

A2: Ahja, hier wird unreachableUrLs getestet.

A1: <\*zurück zum Bildschirm\*> Na das wird noch mehrmals, oder?

A2: Hier nich, hier auch nicht, aber hier <\*liest Testfallnamen\*> (#testFailedUrlsAreReachable#) (#testInternalValidLinkslsNotUnreachable#) (#testLinksToInActiveObjectsFromInactiveAreNot#) (!!...!!)

A1: <\*dreht sich weg\*> M-hm, ja.

1:25:13 (end of Example 9.21)

[...]

1:43:40 (start of Example 9.19)

A2: <\*beginnt TODO-Kommentar zu schreiben, navigiert durch Code\*>

A1: Ah, (#CMReminder#). Gut. Hier <\*zeigt auf Bildschirm\*> da. <\*schaut zu Kollegin, diese spricht ihn an\*>

1:43:57

[A2 liest weiter im Code, A1 spricht mit Kollegin bis diese geht]

1:44:17

A2: Der unterstützt das sogar schon.

A1: Wer?

A2: Na der Reminder. Achso, es gibt tatsächlich welche, wo der erzeugt wird, wo der weiß, dass es ein Mirror ist.

A1: Ach!

1:44:31  
[...]

1:45:11  
[Kollegin kommt wieder, spricht mit A1, A2 liest weiter im Code; A1 wendet sich wieder an A2 ]

1:46:30  
A1: Das ist doch wieder das alte GUI, braucht doch keiner.  
A2: <\*markiert Code-Zeile\*> Äh, bei Reminders?  
A1: Ups.  
A2: Wofür gibt's denn diese propertyClassMap?  
A1: Keine Ahnung.  
A2: Ist vollkommen überflüssig. Wahrscheinlich ruft auch diese propertyClassMap niemand auf.  
<\*sucht nach Verwendungen\*> Doch. (#getClassForProperty#)  
A1: (#jif.RemoteObjectAccessor#), ja schön, und?  
A2: <\*klappt Trefferliste auf\*> Aber der ruft das ja nicht auf dem CMReminderAccessor auf.  
A1: Mmh! Guck mal (#getXmlElementFor#) blubbelblubb (#value#) und sowas, ja weiß ich nicht.  
<\*wendet sich wieder Kollegin zu\*>

1:47:00  
[A1 spricht wieder mit Kollegin bis diese geht, während A2 weiter im Code liest]

1:47:17  
A2: Also irgendwie glaube ich nicht, dass das wirklich verwendet wird. Da hat bloß einer den ReminderAccessor von dem anderen Accessor abgeschrieben und sich gedacht 'Aha, ich muss das hier implementieren, weil ich ja RemoteObjectAccessor bin und da muss ich halt irgendwas reinschreiben'. Ich schreib jetzt unseren neuen Key einfach nicht rein. Sehe ich nicht ein. Ich sehe nicht ein, warum der da drinstehen muss.

A1: Ja.

1:47:42 (end of Example 9.19)  
[...]

1:53:20 (start of Example C.1, Focus Phase #4)  
A1: Jetzt haben wir den anderen noch, der ist noch kaputt.  
A2: <\*springt zum anderen Konstrukt-Aufruf\*>  
A1: Ja und da müssen wir uns jetzt was reinbauen.  
A2: Der muss auf jeden Fall beides.  
A1: Der muss beide da noch irgendwie reinbauen noch.  
A2: Da hatten wir doch schon (!...!)  
A1: Müssen wir in den Accessor jetzt noch was reinklatschen?  
A2: <\*Cursor zum fünften Argument\*> XmlUtils hieß das glaub ich, ne? <\*tippt "XmlUtil.", Autovervollständigung geht auf\*>  
A1: (#parseBoolean#) Ja.  
A2: <\*wählt Eintrag aus\*> (##parseBoolean##) (##data get CMReminderAccessor.KEY\_IS\_ACTIVE##)  
A1: <\*Compilerfehler wird angezeigt\*> <überrascht> Mhm.  
A2: <\*fügt schließende Klammer ein und löscht sie wieder\*> Ist noch nicht ganz korrekt.  
A1: Nee, du musst noch auf String casten.  
A2: <\*kopiert aktuelle Zeile in Zwischenablage\*>  
A1: Und das gleiche nochmal.  
A2: <\*fügt Zeile als sechstes Argument ein, ändert fünftes Argument zu KEY\_IS\_MIRROR, Compilerfehler wird angezeigt\*>  
A1: War isMirror der erste?  
A2: Ja. <\*löscht Rest-Argument von zuvor\*>

A1: OK. FIXME weg und auf String casten

A2: <\*öffnet Tooltip zum Compilerfehler bei KEY\_IS\_MIRROR\*>

A1: Hä? Achso, ja, ok.

A2: <\*wählt "Create constant 'KEY\_IS\_MIRROR' in type 'CMReminder' " was eine neue Konstante anlegt; bewegt die neue Konstante ein paar Zeilen nach oben und löscht den Standard-Wert null\*>

A1: 'Object' ist kreativ.

A2: <\*ändert Wert der Konstanten auf "isMirror"; ändert Typ von Object zu String\*>

A1: OK.

A2: <\*geht zurück zum Konstrukturaufruf, öffnet Tooltip zum anderen Compilerfehler, wählt "Cast argument 1 to 'String' ">

A1: M-hm. Ja.

A2: <\*löscht FIXME-Kommentar\*>

A1: Nicht schlecht.

1:54:56 (end of Example C.1, Focus Phase #4)

[...]

1:55:45 (start of Example C.2, Focus Phase #5)

A1: Ah, da kommen wir der Geschichte näher.

A2: <\*setzt Cursor in Zeile 167\*> (##reminder setObject forKey isMirror##)

A1: Ach das ist CObject. Das ist ja direkt gut.

A2: (##reminder setObject forKey isActive##) Ach wir haben schon nen Object?

A1: Ja, wir haben schon nen CObject. Brauchen wir wirklich nur noch object isMirror abfragen.

A2: Die contents sind aber nicht da. Das ist aber schlecht.

A1: Achso, du willst sagen, das Problem ist, dass wenn wir das so (!...!) OK, bei mirror ist es unproblematisch, ne? Bei mirror kannst du es direkt richtig reinbauen.

A2: (Richtig)

A1: (Ja) Und das active (!...!) ugh

A2: <\*tippt [object isMirror] in Zeile für isActive\*>

A1: Jetzt machst du gerade irgendwas falsch.

A2: (Na, gleich.) <\*tippt weiter ? @"true" : @"false"\*>

A1: Falsche Zeile.

A2: <\*formatiert Statements\*>

A1: Und noch ne Klammer (!...!) (ach nee.)

A2: <\*tauscht Statements\*>

A1: OK.

1:57:04 (end of Example C.2, Focus Phase #5)

A1: Ach da haben wir sonst sowas gemacht irgendwie releasedActive und editedActive (!...!!)

A2: Was jetzt hier passiert in der WHERE-Expression (!...!) achso, genau. Es werden genau die Objekte mit dieser Query geholt.

A1: M-hm (.....) Hm. Und was machen wir jetzt?

1:57:38 (start of Example C.3, Focus Phase #6)

A2: Der reminder betrifft aber das Objekt. Das heißt, ob das Ding jetzt gültig oder ungültig ist, hängt sogar noch von der Ansicht ab <\*schaut zu A1\*>.

A1: Ja. (..) Ist richtig scheiße. <\*schaut zu A2\*> (...) Bah!

A2: <\*schiebt sich vom Tisch weg\*> Scheiße.

A1: In solchen Fällen müsste man eigentlich den vom edited und released getrennt die isActive beide zurückgeben und das GUI entscheiden lassen, was es dann gerne hätte. <\*nimmt Brille ab\*> Bah! Ist aber teuer, für diese beiden.

A2: Ja, oder das GUI sagt an.

A1: (.) So nen *'prefer edited'* noch als Option da ranhängen, oder was?

A1: Ja aber selbst dann wird diese Query saukompliziert jetzt, ne? Weil wir müssen ne Menge Contents ranjoinen.

A2: (.) Wir haben die doch bei dem anderen Teil auch rangejoint gekriegt.

A1: Ja, da waren's Tasks. Bei Tasks weißt du ja, dass es nen `editedContent` im Endeffekt gibt, oder nen `committed`.

A2: (..) Achso.

A1: <genervt> Hier kann es keinen geben, oder nur nen `released`, oder nur nen `edited`, oder weiß ich nicht was, oder (!...!)

A2: <\*lehnt sich zurück\*> Ah! <lacht>

A1: Das ist richtig blöd, oder?

A2: (..) Ja.

A1: Ja. Gibt's das `isActive` noch am Object als Getter für den `released`, oder irgenwas?

A2: Naja, was, guck mal, was ja passiert ist, du machst einmal `SELECT WHERE` (!!M-hm!!) und holst die Objekte (!!Ja!!). Was du natürlich machen könntest, ist (!...!) die Query is ja schon ne Object-, ist ja schon ne `ObjectWHERE`-Einschränkung, ja? <\*selektiert Statement\*>

A1: M-hm.

A2: Was man jetzt machen könnte, ist zu sagen, *'OK, mache außerdem, selecte alle Contents, wo für die Objekt-ID gilt, diese Einschränkung'*, ja? <\*zeigt auf Bildschirm\*> Und außerdem, die Content-ID ist entweder `committedContent`, nee ist entweder `editedContent`, oder `committed`, oder `releasedContent` von einer dieser Objekt-IDs (!!...!!)

A1: Und das Ding ist inaktiv.

A2: Mhm, na das kannst ja erstmal (!!...!!)

A1: Nee, aber dann kriegste immer noch nicht die raus, die gar keinen Content haben. (..) Nee, genau (!!...!!)

A2: Nee, pass auf. Du fährst nur den Fetch ab. Das Ergebnis ist nämlich, dass die Dinger mit einem `SELECT` alle in den Cache reingefahren werden. Und ab da an kannst du einfach das Objekt fragen. Das ist dann schnell.

A1: Noch besser. (.) Du brauchst gar nicht die ganzen Contents ziehen. (.) Du holst dir nur die Objekt-IDs aus der Content-Tabelle, wo die Objekt-IDs sind in diesen Dingen und der ist entweder `edited`, `released` oder `committed` und gültig und (!...!) also aktiv. So, und dann machst du hier einfach nur `isActive` kriegt *'Objekt-ID ist in dieser Liste'*.

A2: *'Object ist kein Template'* und (!...!).

A1: Achja, und *'ist kein Template'* noch, ja.

A2: Und ist noch die Objekt-IDs in dieser Liste.

A1: Ja. `isActive` heißt, *'ist Template'* oder *'ist in dieser Liste'*.

A2: Genau.

A1: Das wäre doch die billigste Variante wahrscheinlich.

A2: Richtig. Du kannst dir ja mal aufschreiben, wie du dir das so `SELECT`-mäßig vorstellst. Ich geh mal kurz für kleine Jungs. <\*steht auf\*>

A1: OK.

2:00:55 (end of Example C.3, Focus Phase #6)

## C.2 Session BA1

Company B develops a social media platform in PHP and JavaScript. The full-stack developers B1 and B2 also deal with MySQL, HTML, and CSS. They have been practicing pair programming between six months and one year.

In session BA1, they take over some code of unknown quality written by outsourced developers. Technically, they want to implement part of a cache. In particular, their logic

should tell whether the requested data has changed since a given timestamp. In their session, they need to understand all existing code for that functionality (a few dozen lines of PHP code), make some additions, and encounter difficulties in specifying what exactly their cache should do. In the first minutes they also struggle with the workstation which is not theirs and not fully configured to their needs.

### C.2.1 Transcripts of BA1 Excerpts

0:01:33 (start of Example 7.13)  
B1: (#Couldn't connect. cat: header.txt#) Curl kann immer noch nicht connecten. <\*wechselt zur IDE\*>  
<\*zurück zur Shell, gibt Skript-Inhalt auf der Shell aus\*> Ja, wir können das ja mal einfach hier so ausprobieren. <\*markiert Zeile mit curl-Kommando\*>  
Weil das doofe ist, wir können's nicht einfach im Browser machen, weil der Mime-type application/json oder javascript ist (!...!)  
<\*Rechtsklick in der Shell, ohne sichtbaren Effekt\*> Wie kann man hier reinpasten?  
B2: Rechts.  
B1: Rechts?  
B2: M-hm  
B1: Habe ich gerade gemacht.  
B2: Nee, mit Links kopierst du, mit rechts (!...!)  
B1: Okay. <\*markiert die curl-Zeile nochmal\*> Mit Links kopier ich? Mit Links klick ich doch einfach nur.  
B2: Aber wenn du was markiert hast und dann linksklickst, dann wird's kopiert.  
B1: Achso. <\*fügt curl-Befehl ein und führt ihn aus\*> (#Couldn't connect#) Gibt's auch CTRL A? <\*ändern Hostnamen im Befehl von localhost auf dev-intern, schlägt wieder fehl\*>  
B2: Aber Curl ist schon noch drauf, oder? Ja, sonst hätten wir keinen Fehler.  
B1: (#Couldn't connect#)  
B2: Versuch mal wget, dass du dir die einfach holst.  
B1: <\*versucht wget auf dev-intern, was fehlschlägt\*> Fehlschlagen.  
B2: Mach mal auf localhost statt dev-intern.  
B1: <\*versucht wget auf localhost, was fehlschlägt\*> Hm, was'n da los?  
B2: Firewall?  
B1: (#Auflösen des Rechnernamens localhost#)  
B2: Vielleicht (~)? Nee, hat er ja aufgelöst.  
0:04:00 (end of Example 7.13)

## C.3 Sessions BB1, BB2, and BB3

The same pair of developers as in session BA1 (see Appendix C.2) implements a new feature from scratch in the course of one afternoon, going through their complete web development stack, starting with template and internationalization in session BB1, continue with controller, model, database layer, and template optics in session BB2, and conclude with making their view more interactive through JavaScript in session BB3.

### C.3.1 Transcripts of BB1 Excerpts

0:16:47 (start of Example 10.8)  
B2: Was überlegst?  
B1: Was eigentlich wo reingeht. <\*liest in Template-Code\*> (#slimcolbox#)  
B2: H2 ist diese hübsche rote Farbe <\*schaut auf Papierausdruck\*>

B1: OK, das ist dann die zweite Überschrift <\*schaut auch auf Papiausdruck\*>  
 Wir müssen eigentlich hier alles reinpacken <\*zeigt auf HTML-Code\*>, wenn es so aussehen soll wie da <\*zeigt auf Ausdruck\*>

B2: Und was ist `slimcolbox`?

B1: `slimcol` ist (!...!) wenn du zwei schmale Spalten nebeneinander hast. Und zwar <\*öffnet Website\*> hier so.

B2: Hm!

B1: Denk ich mal. Kopiert hatten wir <\*\*\*Bereich-Name\*\*>.

B2: Die jetzt hier nicht auftauchen.

B1: (~) das Teil was wir gar nicht brauchen. <\*fügt öffnenden Kommentar-Tag ein, scrollt\*> Wo macht er denn den `div` wieder zu?

B2: Hier! <\*zeigt auf Zeile mit zwei schließenden `</div>`-Tags\*>

B1: Ah ja. <\*bewegt Cursor zwischen den beiden `</div>`-Tags\*>

B2: Dieses `clear` brauchen wir glaub ich, damit er unser (~). Müssten wir dann (.) in unseren Sub-Header mit reinnehmen.

B1: <\*kommentiert zweiten `</div>`-Tag aus\*> (~) an der Stelle und danach ein.

B2: Genau.

B1: <\*löscht zweiten `</div>`-Tag\*> (~) verwendet. Er cleart doch nur diesen Inneren. Den anderen haben wir doch eigentlich rausgenommen.

B2: Schmeiß doch den ganzen Block einfach weg.

B1: (.) Achso er macht da unten nen neuen `div`. <\*macht letzte Änderungen rückgängig\*> OK, ja. <\*formatiert Code neu, kommentiert ganzen Block aus\*>

0:19:06 (end of Example 10.8)

## C.4 Session CA1

Developer C1 (4 years of experience) started with implementing a new form in their graphical Java-based application when C2 (9 years of experience) joins him (see Section 4.4.2 for details on the software and the company). In their session, they mostly deal with making their new GUI component in the form toggleable for which they reuse existing GUI logic.

### C.4.1 Transcripts of CA1 Excerpts

0:00:55 (start of Example 9.3)

C1: Also wie gesagt, ich habe schon angefangen, ungefähr ne Stunde programmiert. Da hab ich mit der GUI, mit der GUI hab ich angefangen. Kann ich dir grad mal zeigen. <\*öffnet Übersicht über letzte Änderungen\*> Und zwar hab ich beim (!...!!)

C2: <\*hebt die Hand\*> Erste, ganz kurze Frage. Inwiefern ist die ganze Sache destabilisierend, von wegen der Branch der kommt, und dass da jetzt nur noch Konsolidierungen gemacht werden soll, im Moment, auf'm Hauptast?

0:01:22 (end of Example 9.3)

[...]

0:06:17 (start of Example 9.13)

C2: Gibt's nicht so'n ganzes, gibt's nicht so'n ganzes Panel schon? Hast du das ganze Panel schon vorgefertigt übernommen?

C1: Ja, aber das, äh (!...!!)

C2: Oder nur die einzelnen Teile?

C1: Die beiden Auswahlboxen ist ein Panel. Die Checkbox gehört nicht dazu. Ähm, es gibt noch ne größere Komponente, aber da ist noch 'n Button dabei, und 'n paar andere Sachen, die eigentlich nichts mit diesem Arbeitspunkt zu tun haben.

C2: Mmmmh, würd ich aber trotzdem mal ganz gerne, ganz gern kurz sehen (!!Ok dann!!) einfach mal damit ich mal, hast du, ham wir noch 'n gestartetes <\*\*\*Software-Name\*\*>? <\*schaut durch Taskleiste\*>

C1: Ja, <\*zeigt auf Taskleiste\*> haben wir noch.  
C2: <\*öffnet Anwendung\*>  
C1: Genau, das wär der 'Maßstabs'-Reiter.  
C2: <\*öffnet den Reiter\*>  
C1: Genau, und das wäre praktisch die Komponente <\*zeigt auf Bildschirm\*> mit diesem Button und dem Label oben drin, glaub ich.  
C2: M-hm.  
0:07:05 (end of Example 9.13)  
[...]  
0:12:10 (start of Example 9.4, part 1)  
C2: Dann würde ich mir mal das Ding, FeatureLayerPropertiesPanel, anschauen.  
C1: M-hm.  
C2: <\*schließt Anwendung, wechselt zur IDE, öffnet die Klasse und scrollt sie durch\*>  
C1: Und da gibt es jetzt nen LabelAttributesPanel.  
C2: <\*scrollt wieder hoch\*> Ahja, genau.  
0:12:47 (end of Example 9.4, part 1)  
[...]  
0:13:57 (start of Example 9.4, part 2)  
C2: Dann mal schauen, wo das überall verwendet wird <\*öffnet Liste mit Verwendung der Klassenvariablen, hovert die Einträge nacheinander\*> Okay.  
C1: Ja, also ich hab schon angefangen, die Daten, also das Model zu erweitern.  
0:14:32 (end of Example 9.4, part 2)  
[...]  
0:18:46 (start of Example 11.8)  
C1: Muss dann dieses Panel dann das Interface implementieren, oder?  
C2: [...] Genau, das hier. <\*öffnet entsprechende Klasse, hovert Klassendeklaration mit extends\*> Gott  
C1: <\*diktirt\*> 'implements IEnableableComponentContainer'  
C2: (#AbstractDialogPanel#), das ist ja interessant [...]  
A-ha, was ist denn das schon wieder <\*öffnet die Oberklasse, liest (,,,,,)\*>  
<\*zurück zur vorherigen Klasse, tippt 'implements IEnableableComponentContainer'\*>  
Was hat'n der alles? Mal schauen. <\*öffnet Interface, das hat eine Methode\*>  
C1: (#getComponents#), das ist ja cool. Sieht relativ simpel aus.  
C2: Aha, <\*hovert extends-Deklaration\*> und nen IEnableable <\*öffnet Ober-Interface\*>  
0:19:54 (end of Example 11.8)  
C1: (#setEnabled#), ok.  
0:19:57 (start of Example 9.15)  
C2: Das Problem ist, das passt nicht, mit getComponents. <\*scrollt durch die Datei\*>  
C1: Wieso passt es nicht?  
C2: Denk ich mal, denk ich mal. Ich kann mich natürlich auch täuschen. Aber wobei (!...!) <\*scrollt weiter\*>  
C1: Wir müssen doch eigentlich nur die einzelnen Komponenten dann da rausholen, die das Panel hat, oder (!...!) (ist das schwierig)?  
C2: Ah, das hat sogar nen getContent. Das hat sogar nen getContent, seh ich grad.  
C1: Ok (..) und nen PanelBuilder, kann man da vielleicht das (!...!) die anderen Panels rausholen  
C2: <\*scrollt weiter (,,,,,)\*> ich bin mir nicht sicher ob das alles so tut (,,,,,)  
C1: Gott, dann müssen wir ja noch (..) hoffen, dass es n (!...!) n JPanel alleine, kann man das deaktivieren?  
C2: <\*scrollt weiter (,,,,,)\*> OK, ich würde sagen (,,,,,,,) wollen wir einfach probieren, die Methoden zu implementieren?  
0:21:01 (end of Example 9.15)

## C.5 Session CA2

Experienced developers C2 and C5 continue the implementation which C5 started earlier. They mostly refactor existing code and fix newly introduced bugs in their complex Java-based graphical geo-information system. See Section 4.4.2 for details.

### C.5.1 Transcripts of CA2 Excerpts

- 0:10:07 (start of Examples 8.9 and 10.4)  
 C5: Dann, ist es so, zeig ich dir glaub ich erstmal was ich gemacht hab. <\*opens IFeature AttributeConfiguration\*>  
 C2: OK.
- 0:10:14 (start of Example 6.1)  
 C5: Mhm. Also, was ich gemacht hab, mit Absprache von <\*\*\*Lead Developer\*\*\*>, das ist das I FeatureAttributeConfiguration erweitert um nen IVirtualColumn, in dem ich quasi (.) nen Objekt abholen kann, wo du die IColumns abholen kannst, wo du eventuell noch an nen Provider oder so rankommst.
- 0:10:42 (start of Example 9.12)  
 C2: M-hm. Klar. Zeig mal kurz, wie die aussehen.  
 C5: <\*öffnet die Interface-Datei IVirtualColumn\*>
- 0:10:47 (end of Examples 6.1 and 8.9, start of Example 9.2)  
 Mehr ma(!...!) mehr ist da noch nicht drin, weil (!...!!)  
 C2: O-kaaay.
- 0:10:54 (end of Example 9.12)  
 C5: Weil, ähm <\*wedelt mit den Händen\*> es ist, ich, äh, mehr (!...!!)
- 0:10:58 (start of Example 8.22)  
 C2: Haben wir da überhaupt nen ColumnAttribute? Ist das so?  
 C5: Ich hab da (!...!) wir haben, wir brauchen ja nachher n I, nen ColumnAttribute, um das in diese, äh all (!...!) in diese, wenn du alle abholst, einzufügen zu können.  
 C2: Ist das so? Brauchen wir das überhaupt für die Visualisierung in der Attribut-Tabelle?  
 C5: Wir brauchen für die Visualisierung in der Attribut-Tabelle, wenn wir das über getAll(..) AttributeColumns machen wollen, nen IColumn(..) Attribute.  
 C2: Wenn wir's dadrüber machen wollen, (ja ok.)
- 0:11:26 (end of Example 8.22)  
 C5: Wenn wir's dadrüber machen wollen, das ist richtig. Also so hab ich bis jetzt unsere Absprachen verstanden. Ich hab da noch nicht mehr rein (!...!!)
- 0:11:32 (start of Example 8.18)  
 C2: Wie sind denn die vorhandenen Datenstrukturen, die für die GUI verwendet werden? Ist da nen ColumnAttribute da? Ansonsten würde ich einfach die verwenden?  
 C5: Ich weiß nicht wovon du gerade redest.  
 C2: Von dem was ich gemacht hab, mit dem GUI.  
 C5: Du hattest nur nen VirtualAttribute?  
 C2: (....) Ich dachte eigentlich, dass wir das für die Datenstruktur verwenden, was ich da gemacht habe. (Aber ok. Aber zeig mal.)
- 0:11:55 (end of Example 8.18)  
 C5: Wir können erstmal weitergehen.
- 0:11:58 (end of Examples 9.2 and 10.4)  
 [...]
- 0:16:45 (start of Example 6.12)

- C5: Also momentan hat es keine Auswirkungen, weil es darüber weiter noch nicht funktioniert. Aber in dem Moment, wenn diese Reihenfolge stehen bleibt, und das da oben <\*zeigt auf Zeile\*> scharf gemacht wird, wenn wir das so implementieren wie ich das gedacht habe, würde das momentan in nen Fehler laufen.
- C2: Das passt ja dann aber auch.
- C5: <\*fährt herum, holt tief Luft, sieht C2 an\*>
- C2: Ok, jetzt mal kurz schauen <\*nimmt Maus und liest weiter im Code\*>
- C5: M-hm. <\*schaut zum Bildschirm\*>
- 0:17:05 (end of Example 6.12)
- [...]
- 0:19:30 (start of Example 7.7)
- C5: Ich hab es halt erstmal so gemacht, weil ich möglichst wenig nach `basis` rüberziehen wollte, von den Sachen. Und das Interface, was ich darüber geschoben habe, kennt halt nur Sachen, die in `basis` bekannt sind.
- 0:19:54 (start of Example 6.2)
- C2: Und das `VirtualAttribute`, wo ist das?
- C5: Das hab ich rübergezogen.
- C2: Ja, dann passt's doch.
- 0:20:00 (end of Example 6.2)
- C5: Nicht das `Virtual`, das `IVirtualColumn` <\*zeigt auf Bildschirm\*> habe ich rübergezogen. Das `VirtualAttribute` ist hier in `pro`.
- C2: In `pro` ist das `VirtualAttribute`? OK. <\*opens file VirtualAttribute\*>
- C5: Ja.
- C2: Ja, ok, ja, richtig. Das hatte ich da drin, weil wir es noch nirgends anders gebraucht haben.
- 0:20:37 (end of Example 7.7)
- [...]
- 0:26:11 (start of Example 7.16)
- C2: Wie ist das bei SVN und Verzeichnissen löschen, geht das? (Sollte eigentlich schon gehen)
- C5: Du solltest doch wohl nen Package löschen können.
- C2: Also im CVS geht's ja nicht.
- C5: <\*schaut irritiert zu C2\*>
- C2: Im CVS geht's nicht.
- C5: Ein Package zu löschen?
- C2: Ja, geht in CVS nicht. Das Verzeichnis bleibt immer erhalten.
- C5: Das bleibt im CVS erhalten (!!...!!)
- C2: Und es macht keinen Unterschied ob du es löscht oder nicht löscht, in CVS. <grinst> [...] Es erscheint automatisch nicht, wenn es leer ist
- C5: <\*schaut zurück zum Bildschirm\*>
- C2: unabhängig davon, ob du es gelöscht hast oder nicht. [...] Wie es allerdings bei SVN ist, weiß ich nicht. Da sieht es möglicherweise ganz anders aus.
- C5: M-hm, werden wir ja sehen.
- C2: Würde mich selber interessieren.
- 0:27:01 (end of Example 7.16)
- [...]
- 0:28:16 (start of Examples 6.3 and 6.21)
- C5: Oh, weißt du wie die (!...!) wie ich an die Funktion rankomme, ne Methode zu verändern?
- C2: Das bringt dir doch gar nix.
- 0:28:23 (end of Example 6.21)
- C5: Doch ich kann in der Methode, wenn ich die aufgemacht habe, das `IVirtualColumn` in nen `I` (!!...!!)

- C2: Aber das wird dir nicht viel bringen, aber das geht mit ALT SHIFT C, mach's mit ALT SHIFT C.
- 0:28:33 (end of Example 6.3)  
[...]
- 0:31:17 (start of Example 6.4)  
C5: < \*öffnet Datei mit nächstem Compiler-Fehler, setzt Cursor neben den Fehler\* >  
[Paar wird für 10 Sekunden unterbrochen, beide schauen dann wieder auf den Bildschirm]  
C5: < \*bewegt Cursor in Zeile mit Fehler\* > OK  
C2: < \*schaut im Büro umher\* >  
C5: Oh, ich dachte, die ändert er gleich mit.  
C2: < \*schaut weiter im im Büro umher\* >  
C5: < \*benennt Methode manuell um\* > M-hm < \*macht Umbenennung rückgängig, lässt sich durch IDE Methodenrumpf erzeugen\* >  
C2: < \*schaut wieder auf Monitor\* >
- 0:31:57 (end of Example 6.4)  
[...]
- 0:32:41 (start of Example 6.5)  
C5: < \*öffnet Fehler-Vorkommen und liest Code\* > Ähm (.) ah, ok, die müssen wir ändern. (##VirtualAttributes##)  
C2: Und umbenennen.  
C5: < \*ändert Typ einer Klassenvariablen und benennt sie dann um\* >
- 0:33:07 (end of Example 6.5)  
[...]
- 0:35:21 (start of Example 6.6)  
C5: Das müssten wir auch verschieben.  
C2: Wo bist du? In was für 'ner Klasse? Oder in was für 'nem Modul?  
C5: Das was ich gemacht hab, es gibt 'n (!!...!!)  
C2: In was für 'nem Modul bist du grad?  
C5: Ich habe die Factory, mit der ich diese VirtualColumn erzeuge, oder diese IVirtualColumn habe ich auch nach basis verschoben.  
C2: Ah ja.
- 0:35:29 (end of Example 6.6)  
[...]
- 0:37:15 (start of Example 6.8)  
C5: OK, der braucht dann all, ne? Vermutlich.  
C2: Keine Ahnung (!!Ja!!) was das Ding macht. Ok.  
C5: (..) Das ist damit du 'nen eindeutigen Attributnamen hast.  
C2: M-hm. Ich verstehe. Ok.
- 0:37:30 (end of Example 6.8)  
C5: So. < \*öffnet nächsten Compiler-Fehler\* > Okay. Haben wir die gleichen Änderungen.  
C2: Warte mal kurz.  
C5: Das ist auch wieder die Änderung von VirtualColumn auf VirtualAttributes < \*schaut zu C2\* >  
C2: Äh, Moment.
- 0:37:52 (start of Example 6.7)  
C5: Das ist ne andere Implementierung, die diese Abstracts nicht benutzt.  
C2: (.....) Ne andere Implementierung von der FeatureAttributeConfiguration?  
C5: Ja, ja.  
C2: Die was nicht benutzt?  
C5: Die die abstrakte Klasse, die wir eben angepasst haben, nicht benutzt.

C2: (...) Ok, dann müssen wir's hier noch anpassen.  
C5: <erleichtert> Ja, genau.  
0:38:18 (end of Example 6.7)  
[...]  
0:43:02 (start of Example 6.9)  
C5: Jetzt sind wir wieder fehlerfrei.  
C2: OK.  
C5: So (,,,) was mich dann jetzt interessieren würde, wenn wir uns die GUI mal, (!!Warte mal!!) ob die GUI noch funktioniert.  
C2: Jetzt würd ich gern, noch ne Methode einbauen, mit dem man die, so roh wie sie sind, abholen kann auch.  
C5: M-hm.  
C2: Und dann würde ich mir gern mal die Sache mit dem Dialog und so mal kurz anschauen.  
C5: OK.  
0:43:26 (end of Example 6.9)  
[...]  
0:44:08 (start of Example 7.12)  
C2: Dann probieren wir die GUI aus. Ganz einfach, weil nicht einchecken ist blöd. Ansonsten machen wir jetzt was kaputt, und dann ist das ja auch mit kaputt. Wo war denn das? <\*scrollt durch Paket-Übersicht\*> Wo waren wir denn?  
C5: Du musst die Demo hier unten <\*zeigt auf Bildschirm\*>  
C2: Die hier? <\*klappt eines der vielen demo-Pakete auf\*>  
C5: Da müsste es drin sein, dieses (#EditColumnAttribute#) oder, achso nee (!...!)  
C2: Das hier?  
C5: Warte, du hattest damals die Demo geschrieben, weil du hattest die Attribute reingebracht. Aber das muss (!...!) das ist die Action, die das aufruft. Moment, nee dann muss es doch hier oben sein, oder?  
C2: Meint ich doch <\*startet Demo\*>  
C5: OK, aber, das was wir gemacht haben, hat eher Auswirkungen auf die Action, als auf die GUI. Das heißt, ich würde es gerne aus <\*\*\*Software-Name\*\*> sehen.  
C2: Klar. <\*hovert das "Run"-Menu der IDE\*> Oh Gott, wie läuft denn das noch mal? <\*hovert "Run"-Button\*> Hier muss ich (~) machen?  
C5: Ja, und dann <\*\*\*Name der Run-Konfiguration\*\*>.  
0:45:52 (end of Example 7.12)  
[...]  
0:47:10 (start of Example 8.2)  
C2: <\*schaut auf GUI, stoppt Mausbewegung\*> Ähm  
C5: Einfach 'Attributtabelle'.  
C2: <\*doppel-klickt woanders, anderes Fenster auf\*> Ah, 'Attributtabelle'. Natürlich.  
0:47:21 (end of Example 8.2)  
[...]  
0:52:26 (start of Example 9.10)  
C2: <\*tippt "cad-507"\*> So ging das mit dem Minus, oder?  
C5: Bin mir nicht sicher, Minus (!...!) Ich glaub, das CAD groß oder so? (...) Oder muss das klein sein?  
C2: Ähh, muss ich in meinen E-Mails kurz nachschauen, weiß ich nicht auswendig. <\*verlässt den Arbeitsplatz für eine Minute\*> Ich schreib's groß.  
0:53:44 (end of Example 9.10)  
[...]  
0:55:41 (start of Example 6.10)  
C2: Ähm, das wollten wir eben kurz rausnehmen? Das mit dem gleich Null.

C5: (..)	
C2: Dacht' ich?	
C5: (...)	
C2: <*löscht die Zeilen*>	
0:55:49	(end of Example 6.10)
[...]	
1:03:46	(start of Example 9.7)
C5: (#getVirtualAttributes#), markier die mal (!...!!)	
C2: Wart mal kurz, ich will nur noch ganz kurz schauen (,,,,,,) nur mal ganz kurz noch <*Hände über dem Keyboard*> (..) nee komm <*setzt Breakpoint (,,,,,,)*> dann würd ich einfach das Ding gerne im Debug (!...!)	
1:04:12	(end of Example 9.7)
[...]	
1:05:29	(start of Example 9.9)
C5: Guck mal, ob die Delegates richtig sind.	
C2: Ja, siehste doch.	
C5: Ja, die die, ähm (!...!) Methoden, ob die die (!...!!)	
C2: Schau'n wir gleich.	
C5: Mhm.	
1:05:38	(end of Example 9.9)
[...]	
1:14:07	(start of Example 6.11)
C2: Also, das Problem, was wir haben (.) ist (..) dass wir natürlich keine FeatureProxies im eigentlichen Sinn haben <*schaut zu C5*> (..) für unsere Sachen.	
C5: M-hm. So, ich hab ja mit <***Lead Developer**> gestern halt (!!oder?!) drüber geredet.	
C2: <*schaut zum Monitor*> (.) wobei (!...!)	
C5: <*zieht Zettel zu sich*> Was erklärt, warum ich das mit dem Column gemacht hab	
C2: wobei (,,) ja gut, ich mein (!...!) (....)	
1:14:42	(end of Example 6.11)
[...]	
1:15:59	(start of Example 9.14)
C5: Am FeatureProxy müssen wir nichts ändern.	
C2: An dem wie es funktioniert auf jeden Fall, vielleicht nicht an der Klasse, aber an dem wie es funktioniert.	
C5: Wir müssen irgendnem Provider das FeatureProxyübergeben, um die Werte zu kriegen. Das ist je Änderung im Model.	
C2: Müssen wir schauen. <*beginnt Code zu lesen*>	
C5: Das einzigste ist, dass <***Lead-Developer**> und ich gestern darüber philosophiert haben, dass wir das FeatureProxySet ändern, dann soll es nen getTableModel geben und das man das irgendwie gegen ne Factory oder so austauscht.	
C2: <*liest weiter im Quellcode (,,,,,,)*> Es würde mich jetzt interessieren (,,,) wo das Ding aufgerufen wird.	
1:16:43	(end of Example 9.14)

## C.6 Session CA3

Developers C6 and C7 (less than three and more than ten years of experience, respectively) want to implement a new context menu entry which is only enabled under certain circumstances (see Section 4.4.2 for details on the software and the company). They write test cases for the menu entry to be enabled and disabled, and refactor code along the way. This is a simple task, but their IDE froze many times for over a minute each time which slowed the pair down a lot.

## C.7 Session CA4

Experienced developers C4 and C7 (both ten years of experience) implement a new feature to allow for selection of multiple graphical features while holding down the CTRL key (see Section 4.4.2 for details on the company and the software). They have to adapt many interfaces in the event handling part of the software since their feature is the first to react to the CTRL key. They write tests, do refactorings, and discuss design a lot; quite some time is lost in the second half of the session due to a problem with the team's SVN (Subversion) server. The pair also uses two sets of keyboard and mouse fluently.

### C.7.1 Transcripts of CA4 Excerpts

0:34:25 (start of Example 4.3)

C4: Der `DisplayPoint` ist auch richtig ätzend.

C7: Inwiefern?

C4: Mag' ich nich. Das ist so'n echt Kacke (..) Objekt. < \*öffnet Klasse `DisplayPoint` \* >

C7: < \*liest Quellcode (,,) \* > Urgh.

C4: < \*schließt Klasse `DisplayPoint` \* > Naja, aber egal.

0:34:41 (end of Example 4.3)

## C.8 Session CA5

Experienced developers C3 and C4 start implementing a new feature that allows users to cut existing geometries (such as points, lines, polygons) into parts by drawing arbitrary shapes across them. See Section 4.4.2 for details on the company and the software.

### C.8.1 Focus Phase #1

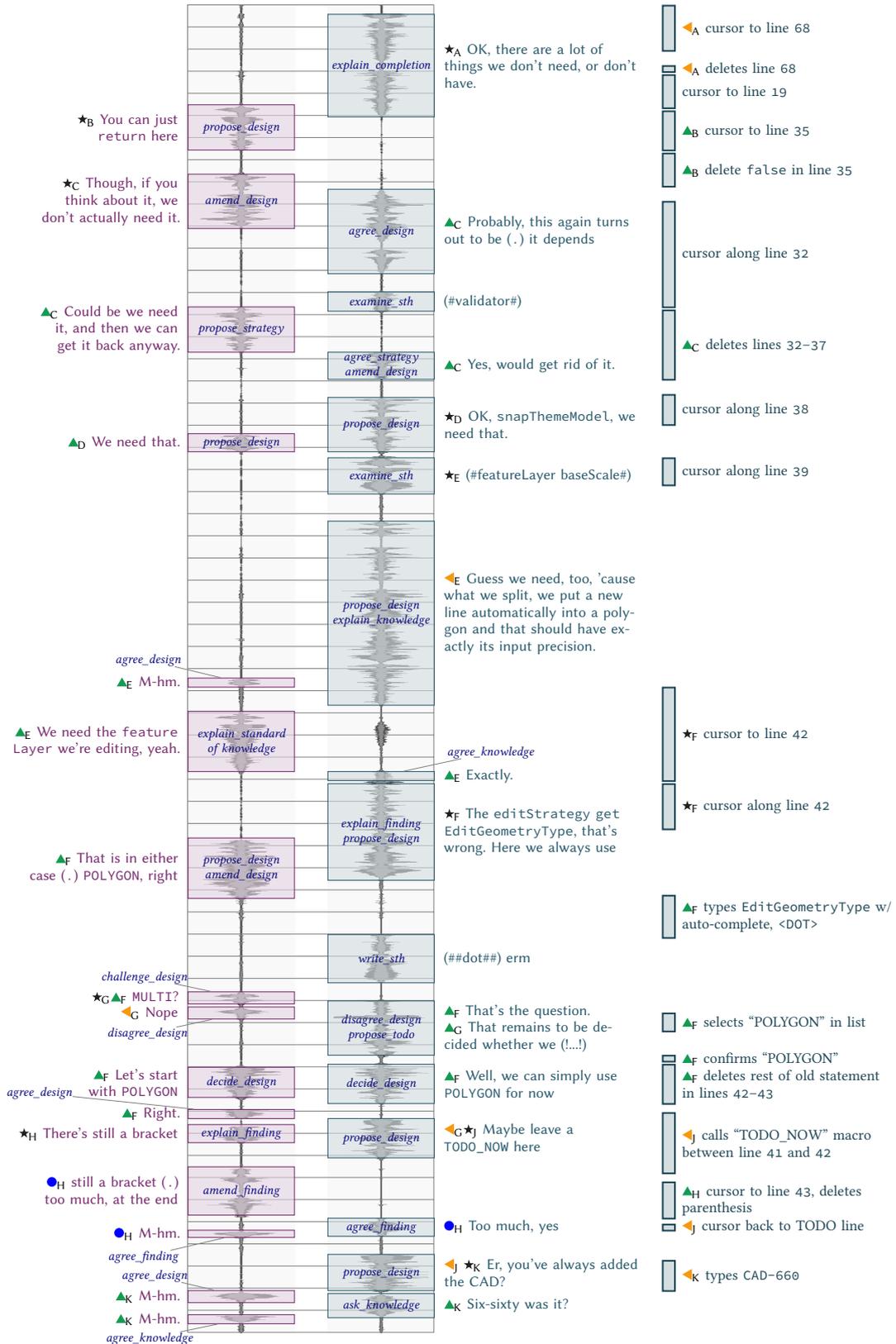
On the next page, I visualize Focus Phase #1 in its entirety as an attempt to make the high degree of concurrency more palpable. See Example 6.15 for the detailed discussion.

```

31 protected void execute(final Component parentComponent) {
32     final Message message = (Message) validator.validate();
33     if (message != null) {
34         SwingMessageIndicator.showMessage(parentComponent, message);
35         return false;
36     }
37
38     final EditOptions editOptions = new EditOptions(new MapModelSelection(snapThemeModel));
39     final IScaleRange scaleRange = new ScaleRange(0, getFeatureLayer().getBaseScale());
40     editOptions.setScaleRange(scaleRange);
41     try {
42         final EditGeometryType editGeometryType = getEditStrategy().getEditGeometryType(
43             getFeatureLayer());
44         editOptions.setGeometryType(editGeometryType);
45     }
46
47     return true;
48 }

```

Figure C.5: Relevant excerpts of the Java code before Focus Phase #1 (see Figure C.6).



**Figure C.6:** Focus Phase #1 of 60 seconds to scale (time goes top-down, vertical intervals are 1 second). Audio waveform of C4 is on the left, C3 is on the right; utterances and base concepts are next to them; C3’s computer interactions are shown as narrow boxes in the right-most column. Line numbers refer to original state of the source code, see Figure C.5.

```

protected void execute(final Component parentComponent) {
    final EditOptions editOptions = new EditOptions(new MapModelSelection(snapThemeModel));
    final IScaleRange scaleRange = new ScaleRange(0, getFeatureLayer().getBaseScale());
    editOptions.setScaleRange(scaleRange);
    try {
        // TODO_NOW (<C4's account name>) 09.05.2008: CAD-660
        final EditGeometryType editGeometryType = EditGeometryType.POLYGON;
        editOptions.setGeometryType(editGeometryType);
    }
}

```

Figure C.7: Relevant excerpts of the Java code after Focus Phase #1 (see Figure C.6).

## C.8.2 Transcripts of CA5 Excerpts

0:17:28 (start of Example 6.20)

C3: Ich bin mir noch nicht sicher, was davon wir wirklich wiederverwenden können.

C4: Ich auch nicht.

C3: Entweder wir versuchen hier ne Methode rauszuziehen und die versuchen zu verwenden. Oder einfach mal, hau drauf, das kopieren, gucken, der Reihe nach, passt das so, ist es das was wir brauchen, wo sind die Unterschiede? Und dann erst gucken, wie fügen wir es wieder zusammen, gibt es Gemeinsamkeiten, die sich verwenden lassen.

C4: Ich such halt eigentlich die ganze Zeit (!...!) <liest im Quellcode\* > hier ist das, bis hierhin irgendwo ist die Dings gestartet. Hier wird der Geometrie-Typ gesetzt, ne? Das hier ist tatsächlich dass er irgendwo den Modus an den Start bringt. Und hier wird es an die Toolbar attacht und der Controller und dann wird er aktiviert. OK, kopieren wir das Ding einfach mal ins execute(). Gucken wir was wir brauchen.

C3: M-hm

0:18:20 (end of Example 6.20)

[...]

0:19:12 (start of Example 6.15, Focus Phase #1)

C3: OK, da gibt's schon mal jede Menge Sachen, die wir nicht brauchen, oder nicht haben. <Cursor zu Zeile 68, löscht Zeile 68, Cursor zu Zeile 19\*>

C4: return kannst hier einfach <zeigt auf Zeile 35\*>

C3: <Cursor zu Zeile 35, löscht false\*>

C4: Wobei wir das, wenn man's genau nimmt, eigentlich nicht brauchen

C3: Wahrscheinlich wird das wieder (.) je nach dem <Cursor entlang Zeile 32\*> (#validator#)

C4: Vielleicht brauchen wir's und dann können wir's uns immer noch wiederholen.

C3: <löscht Zeilen 32-37\*> Ja, würd' ich wegmachen. <in Zeile 38\*> OK, snapThemeModel, das brauchen wir.

C4: Das brauchen wir.

C3: (#featureLayer baseScale#) <Cursor entlang Zeile 39\*> bräucht, brauchen wir eigentlich auch, weil wir das, was wir splitten <Schneidegeste> machen wir ja ne neue Linie automatisch in nen Polygon rein und das soll die genaue Erfassungsgenauigkeit haben, des (!...!)

C4: M-hm.

C3: <Cursor in Zeile 42\*>

C4: Wir brauchen schon den featureLayer den wir editieren, ja.

C3: Genau. <Cursor entlang Zeile 42\*> Die (#editStrategy getEditGeometryType#), das ist falsch. Hier nehmen wir immer (!...!)

C4: Das ist auf jeden Fall (.) POLYGON, genau.

C3: <tippt EditGeometryType mit Auto-Vervollständigung\* > (##Punkt##) ähm

C4: MULTI? Nee.

C3: Das ist die Frage. Das müssten wir noch klären, ob wir (!...!) <wählt "POLYGON" in Liste, bestätigt Eingabe\* > Also, wir können einfach mit POLYGON anfangen <löscht Rest des alten Codes in Zeilen 42-43\*>

- C4: Fangen wir erst mal mit POLYGON an. Genau.
- C3: Ähm, mal nen TODO\_NOW vielleicht lassen? <\*ruft 'TODO\_NOW'-Makro zwischen Zeilen 41 und 42 auf\*>
- C4: Da is noch ne Klammer <\*zeigt auf Bildschirm\*> (!...!) noch ne Klammer (.) zu viel hinten
- C3: <\*Cursor zu Zeile 43, löscht Klammer\*> Zu viel, ahja <\*Cursor zurück zur TODO-Zeile\*>
- C4: M-hm.
- C3: Äh, du hast immer den CAD mit rangeschrieben? <\*tippt CAD-660\*>
- C4: M-hm.
- C3: Sechs-sechzig war's?
- C4: M-hm.
- 0:20:11 (end of Example 6.15)
- [...]
- 0:23:20 (start of Example 6.26, part 1)
- C3: Jetzt weißt du, was bei mir kommt.
- C4: Nein. (!!Ich!!) Nein!
- C3: <\*tippt Ensure\*>
- C4: Bitte bitte nicht.  
<lächelt> Lass uns Tests schreiben. (.) Dann willst du diese Ensures da nicht haben. <\*dreht sich zu C3\*> <lacht>
- C3: Tun sie dir weh, wenn ich sie jetzt hinmach?
- C4: <lachend> Ja, total. Aber gut, wenn's dir nicht wehtut, wenn ich sie beim Testen wieder wegmach.
- C3: Wenn du sie beim Testen (!...!) wenn sie dir beim Testen im Weg sind (!...!) aber so lange sie nicht im Weg sind (!...!) <\*fügt Ensure-Aufrufe ein\*>
- C4: <lächelnd> Argh, die sind im Weg.
- C3: <\*fügt vier Aufrufe von ensureArgumentNotNull ein, für jeden Methodenparameter einen, schaut dann lächelnd zu C4\*>
- C4: Da werden wir uns nie einig. Nie!
- C3: Wenn ne Lösung da ist, die das ganze viel besser macht, z.B. über die AspectJ?
- 0:24:20 (end of Example 6.26, part 1)
- [...]
- 0:43:34 (start of Example 6.26, part 2)
- C3: Du-hu? Schenkst du mir ein Ensure?
- C4: Naargh. <\*fügt Ensure-Statements ein\*> Mein Gott bist du ein Sadist. (,,,,,) Wollen wir da demnächst mal über ein paar Runden gehen? Das mal ausdiskutieren, wenn wir nicht gerade am Programmieren sind?
- C3: Wenn du magst, können wir das gerne machen.
- 0:43:57 (end of Example 6.26, part 2)
- [...]
- 0:47:11 (start of Example 10.7, part 2)
- C3: Würds dir was ausmachen, hier zumindest nen TODO\_NOW ranzumachen? '*Hier Lock releasen, Fragezeichen*'? Einfach um (!...!)
- C4: Nö, überhaupt nicht <\*schreibt TODO-Kommentar\*>
- C3: Dann ist die Information aus'm Kopf bei mir raus.
- 0:47:22 (end of Example 10.7, part 2)
- [...]
- 1:19:58 (start of Example 6.19)
- C4: Dann committen wir das noch.
- C3: <\*klickt auf 'Synchronize'\*> Könnten wir uns noch kurz drüber unterhalten, über Testbarkeit. Jetzt hab ich da drauf geklickt, das wollt' ich nicht.

C4: <scherzend> Warum machst du denn sowas (!!Ja!!) synchronize an den Start bringen.

C3: Das ist (!...!) falsch geklickt (!...!) ich bin's halt gewohnt, dass ich den (!...!) ich bin gewohnt hier links unten nen Button zu klicken, der so aussieht, weil der bei mir hier unten als QuickView ist, ne? Und dann (!...!) war's nen Instinkt.

C4: <scherzend> Ahso.

C3: <lachend> Ja.

1:20:16 (end of Example 6.19)

[...]

1:21:57 (start of Example 10.7, part 3)

C4: Du hinterlässt verhältnismäßig viele TODO\_NOWs. Ich schließe Dinge lieber rund und bin da safe und dann gehe ich zum nächsten.

C3: M-hm. Das ist interessant, weil wenn ich das jetzt gemacht hätte, wenn wir es nicht zu zweit gemacht hätten, hätte ich mehr Sachen gleich gemacht bevor ich weitergegangen wär.

1:22:20 (end of Example 10.7, part 3)

## C.9 Session DA2

Two junior developers D3 and D4 should implement a new toolbar for some module, but encounter problems and decide (after discussion with a senior developer) to perform a preparatory refactoring instead, during which D4 explains many different things to D3 despite being in his very first week at the company. See Section 4.4.3 for more information on the company, the developers, and the session.

### C.9.1 Transcripts of DA2 Excerpts

0:01:54 (start of Examples 7.9, 8.7, and 10.2)

D4: Wie habt ihr denn das jetzt generell gemacht. Weil, äh, irgendwie, habt ihr da jetzt eigentlich mehr ne SWT-Oberfläche als Eclipse, oder so? Und ähm (!...!)

0:02:06 (end of Example 8.7)

D3: Also <Software-Name> an sich ist ja SWT-basiert. Der Kalender, da nutzen wir diese (~) Kalender-Komponente (!!...!!)

D4: Das ist Swing.

D3: Nee, ja, also im Prinzip AWT.

D4: Achso, ok. AWT sogar.

D3: Und, ja, <lacht> da gibt's halt diesen SWT-to-AWT-Container-Schnitzel und dadurch wird der im Prinzip eingebunden. Wie dieser SWT-to-AWT-Dings funktioniert kann ich dir allerdings auch nicht genau sagen. Aber den können wir uns nachher auch mal angucken.

D4: OK. Aber kann man dann da Kontextmenü und sowas auch einbinden? Jetzt nen SWT-Kontextmenü, oder so?

D3: Kann ich dir nicht hundertprozentig sagen. Müssen wir uns mal angucken.

0:02:47 (end of Example 7.9)

[...]

0:04:21 (start of Examples 7.18 and 7.19)

D4: Wie lange machst du das jetzt? Drei Monate?

D3: Bin jetzt seit 1.7. hier. Programmiert habe ich aber schon Jahre vorher. Angefangen halt mit HTML, PHP, und dann rüber zu Delphi.

0:04:34

[...]

0:05:00

D3: Jetzt sitz ich hier bei Java. In zwei Monaten sitze ich wieder in VB.

D4: Ach dann bist du drüben oder was?

- D3: Dann bin ich PS, ja.  
D4: Ach dann in Service dann, nicht Entwicklung?  
D3: Naja, Professional Services, die machen auch ein bisschen Entwicklung.
- 0:05:17  
[...]
- 0:05:40  
D3: Vor allem, wenn ich bedenke, das SVN-Repository liegt auf <\*\*\*Computernamen\*\*\*>. Im Prinzip ist das nur nen Rüberschaufeln von Daten!
- 0:05:47  
[...]
- 0:06:20  
D3: Wir haben ja auch keine Glasfaseranbindung ans Internet. Wir haben ja bloß drei DSL-Leitungen.
- 0:06:26  
[...]
- 0:06:50  
D3: Eigentlich, <\*\*\*Computernamen\*\*\*> ist schon nicht schlecht. Der müsste das eigentlich locker. Mal gucken, was hat denn der hier, zwei Kerne <\*öffnet Systemsteuerung\*>  
D4: Hier, 2.4 [GHz], zwei mal.
- 0:07:29 (end of Examples 7.18 and 7.19)  
[...]
- 0:09:09  
D4: Und das ist jetzt AWT? <\*zeigt auf gestartete Anwendung\*>  
D3: Das ist eine AWT-Komponente, ja.  
D4: Ach du Scheiße.  
D3: Ach was heißt, ach du Scheiße? Ist halt (!...!)  
D4: <lacht> Na dann zeig mir mal kurz die (!!...!!)
- 0:09:23 (start of Examples 7.6 and 8.8)  
D3: Im Prinzip soll jetzt hier oben noch so ne Toolbar hin. Zeig dir dann gleich mal, wie die beim alten Kalender aussah.  
D4: Zeig mir auch dann noch mal, was der bis jetzt alles kann (und kurz was der können soll.)
- 0:09:37 (end of Example 8.8)  
D3: Jaja, ich zeig dir am besten erstmal wo es hin (!...!) wo die Reise hingehen soll <\*öffnet wieder IDE\*> Muss ich grad mal gucken, wo das hier mit den Menüpunkten war, genau, (#Calendar#), ok, doch nicht, (#CalendarWeek#), <\*öffnet Konfiguration eines ExtensionPoints\*> genau (#CalendarTestView#), und dann, das alte hieß, genau (##CalendarView##), genau. <\*ändert Konfiguration eines ExtensionPoints\*>
- 0:10:11 (start of Example 9.6)  
D4: Ist das jetzt noch nen Eclipse-View? Nee?  
D3: <\*zuckt mit den Schultern, schaut lächelnd zu D4\*>  
D4: <\*schnaubt\*>  
D3: <lacht> Ich hab wirklich keine Ahnung von wegen ExtensionPoints und so, da bin ich nicht wirklich konform mit, also da kann ich dir keine große Information geben. <\*startet Anwendung neu\*>  
D4: <\*klickt mit Kugelschreiber, schaut auf Bildschirm\*>
- 0:10:26 (end of Example 9.6)  
[beide warten darauf, dass die Anwendung startet]
- 0:10:59  
D4: Nee, ich hatte hier gerade noch mal in dem Buch gelesen, also, dass du das ja dann auch so ähnlich machen könntest, dass du dann (!...!) zeig noch mal das Eclipse  
D3: <\*navigiert in Anwendung zu Kalenderansicht, Fehlermeldung wird angezeigt, klickt Fehlermeldung weg, wechselt zur IDE\*>

D4: Dass du hier so Contributions hast und wenn du dann den Kalender aktivierst, dass die dann halt auch hier reinkommen. Aber ich weiß halt nicht ob generell überhaupt ne Toolbar geplant ist?

D3: Das kann ich dir auch nicht sagen. Bin ja auch erst seit drei Monaten hier. <\*wechselt zur Anwendung\*>

D4: <lacht> Aber wo soll denn die dann hin, hier oder was?

D3: Eigentlich dachte ich, genau, sollte die hier so hin. <\*hovert schmale Leiste über dem Kalender\*> und eigentlich, weiß auch gar nicht warum sie jetzt nicht <\*klickt herum, Fehlermeldung erscheint\*> (!...!) Was haben wir hier überhaupt für'n Fehler?

D4: (#ClassCastException#)

D3: (#Cannot be cast to CalendarTest#) achso <\*schließt Fehlermeldung\*> wahrscheinlich <\*schließt Anwendung, geht zur IDE zur ExtensionPoint-Konfiguration\*> (#CalendarView#)

D4: Für was sind jetzt hier die Navigation-Sachen, also sind das Actions, oder?

D3: Mhm.

D4: Sind das Actions? Die dann (!...!) wo erscheinen die dann?

D3: Na die werden dann hier durch diese Klasse verarbeitet <\*selektiert Klassennamen einer Action\*> und dann wird halt die entsprechende Methode aufgerufen, also da gibt's ne (!...!) wie heißt die? <\*öffnet Package Explorer\*> nee, nicht hier, sondern (!...!) <\*navigiert zu einer Methode\*> genau, aber ist noch nicht die (!...!) Moment <\*sucht weiter\*>

D4: Na die muss dann irgendeinen Interface erfüllen.

0:12:52

(start of Example 8.4)

D3: Ja, gibt's auch. <\*öffnet Datei\*> (#CalendarTestView#) <\*setzt Cursor in Zeile mit LicenseKey und scrollt nach unten\*>

D4: <lacht> (#LicenseKey#)?

D3: <\*markiert drei Methoden\*> Genau, die werden im Prinzip aufgerufen. Und von da aus, <\*markiert einzelne Zeile\*> geht's dann in die entsprechende Klasse, also bei mir jetzt in den Test-Kalender. <\*scrollt hoch, LicenseKey ist wieder sichtbar\*> (#viewpart#), da isser, der Kalender. OK, jetzt haben wir den natürlich wieder geschlossen.

D4: Ah, LicenseValidator ist von der Komponente, oder was?

D3: Ja, richtig.

0:13:21

(end of Example 8.4)

Die Komponente ist halt gekauft worden, und die haben da so'n setLicenseKey. Den siehst du hier auch noch öfters. Fast jede Komponente, die will noch mal ihren License-Key haben.

D4: Wie? Musst du den kopieren, oder was?

D3: Inwiefern kopieren? Das ist im Prinzip so'n String (!...!!)

D4: Nee, aber, wenn du ne neue Klasse machst, dass du das dann jedes Mal hast, oder (!...!)

D3: Du hast im Prinzip hier (!...!) ja, können wir ja einfach mal nach googeln <lacht> <\*startet Textsuche\*> zum Beispiel hier, die MonthDateArea kriegt den, und (!...!) war's das schon? OK, eigentlich kriegen die noch mehr Komponenten.

D4: Wie, da musst du es für jede Komponente den gleichen LicenseKey setzen?

D3: Ähm, war ich eigentlich der Meinung, aber kann natürlich sein, dass das schon veraltet ist. Muss ich gerade mal eben gucken. <\*scrollt\*> OK, dann hat sich das schon geändert. Gut, dann war es doch nur diese eine Komponente, sehr schön. Hätte mich auch gewundert, wenn jetzt jede Komponente das hat.

Weiß gar nicht, was ist das überhaupt? 'Nen DateAreaBean? <\*hovert Variable, Tooltip erscheint\*> (#DateAreaBean#), ja.

(...) OK, so, wo waren wir stehen geblieben? <\*sieht Stacktrace am unteren Bildschirmrand\*> Die Exception, genau. <\*liest in Stacktrace\*> (#Cannot be cast to CalendarTestView#) achso, (#AbstractOpenCalendarView#), müssen wir hier natürlich auch ändern, in die CalendarView <\*ändert Statement, Fehlermeldung erscheint\*>.

0:15:02

(start of Example 9.8)

Hm? <\*öffnet Details der Fehlermeldung\*> Checkstyle funktioniert nicht, ok. Dann scheißen wir mal auf Checkstyle, würde ich sagen <lacht>.

- D4: <lacht>
- 0:15:15 (end of Example 9.8)
- D3: < \*versucht Anwendung zu starten, Fehlermeldung erscheint\* > Achso jetzt haben wir noch nen ParseError.
- D4: Du hast hier < \*zeigt auf Bildschirm\* > das musst du noch.
- D3: Mhm.
- D4: Hier, da ist doch immer noch 'Test'.
- D3: Ach. < \*ändert Code\* > Daran sollte es aber eigentlich nicht (!...!) < \*Fehlermeldung erscheint\* > doch, cool! Alles klar.
- D4: Mach noch mal 'Organize Imports', dann hast du die weg.
- D3: Macht er eigentlich beim Speichern automatisch. Ja.
- D4: Hast du so eingestellt, oder?
- D3: Also auf meinem Rechner ja. Wie es hier eingestellt ist, weiß ich nicht. Aber ich denke mal genauso.
- D4: Weil standardmäßig ist nichts, nichts eingestellt.
- D3: < \*Fehlermeldung erscheint beim Versuch zu speichern.\* > Hm?
- D4: <genervt> Och, was ist das denn?
- D3: Was hat er denn? Er buildet doch gar nicht. Jetzt haben wir hier schon wieder nen Error. Achso.
- D4: <lacht> OK.
- D3: Ah nee, das tue ich mir jetzt nicht an, dass Ganze umzustricken. Ich zeig's dir woanders.
- D4: Aber was ist denn (!...!)
- D3: < \*macht Code-Änderungen rückgängig\* > Dann schauen wir uns einfach die Toolbar bei der Wiedervorlage an. Weil bis wir das jetzt alles geändert haben.
- D4: Hast du das auf deinem Rechner schon geändert, oder?
- D3: Naja, das ist ja die alte Version, die ich gerade wiederhergestellt habe, der alte Kalender. Und wir machen jetzt hier (##CalendarTestView##), ok. Dann zeige ich dir's in der Wiedervorlage, da ist die glaube ich schon eingebaut die Toolbar.
- D4: Aber das ist jetzt, gut, das ist ja immer noch nen Eclipse-View, ne?
- D3: Mhm.
- D4: Das ist ja abgeleitet von dem ExtensionPoint, oder erfüllt den ExtensionPoint.
- D3: Mhm. Kann's ja auch. Hier ist ja jetzt wieder die CalendarTestView drin.
- D4: Aber deine View ist jetzt der gesamte Kalender, oder?
- D3: Ja.
- D4: Und warum kein Editor? Ach die Editoren habt ihr ganz verworfen, oder?
- D3: Wir arbeiten hier, also ich arbeite hier komplett unabhängig von dem eigentlichen Produkt. < \*öffnet Quellcode\* > Das ist hier im Prinzip von nem Panel und hier ist im Prinzip schon alles AWT. Im Prinzip haben wir angefangen damit (!...!) es gibt Demos von diesem (~) Kalender. Und da habe ich mir die Demo, die am dichtesten an unseren Kalender rankommt, übernommen, hab den hier rein kopiert und dann im Prinzip Stück für Stück angepasst. Weil Problem war ja am Anfang, ich hatte keine Ahnung von Java und mit irgendwas musste ich ja anfangen.
- D4: Achso, hast du vorher noch gar kein Java gemacht?
- D3: Nee, ich war vollkommen unbeleckt in Sachen Java. Deswegen ist der Code auch total für'n Arsch. Wenn du hier siehst, wie viele Zeilen wir inzwischen haben (!...!)
- D4: God-Methods und God-Klassen, oder?
- D3: Das ist (!...!) ja, 1917, müsste man mal refactorn. < \*wechselt zur Anwendung\* > So, Wiedervorlage. Obwohl, bei Positionabsrechnung hat man's auch gesehen.
- 0:18:17 (start of Example 8.13)
- < \*hovert Toolbar im Bereich Wiedervorlage\* > Im Prinzip, so 'ne Toolbar. Die haben sie jetzt also da oben hingemacht.
- D4: OK, und das sind jetzt (!...!) ist das jetzt ne Toolbar oder Coolbar oder sowas und dann die eigenen Widgets drin, oder was ist das?

D3: Ähm, ich vermute, ja. Sicher sagen kann ich es dir auch nicht, weil wie gesagt ich hab das auch noch nicht gemacht bisher. Müssen wir uns einfach mal angucken, wie das in der Wiedervorlage zum Beispiel geschehen ist.

D4: Aber, ähm, soll es dann nicht mal Ziel sein, dass du immer diese Dinger hast?

D3: Ja, genau.

D4: Aber dann ist doch von dem Kalender hier (!...!!)

D3: Ja, das war Quatsch was ich dir vorhin erzählt habe.

D4: Dann soll das ja da auch oben hin, ne?

D3: Wir machen das dann nicht in diese schmale Leiste hin, sondern doch da oben.

0:18:57 (end of Examples 7.6, 8.13, and 10.2)

[...]

0:19:27

D3: Hast du sowas schon mal gemacht, so ne Toolbar-Erstellung?

D4: Ich hab mir gerade hier (!...!) das hab ich mir mal im Urlaub reingezogen das Buch, aber wenn du das natürlich alles nicht nachprogrammierst, dann vergisst du das alles, aber ich hab's mir gerad noch mal durchgelesen. Also zumindest stehen hier die Standardschritte drin.

0:19:49

[...]

0:21:23 (start of Example 7.18)

D4: Wer hat denn das gemacht?

D3: Die ExtensionPoints?

D4: Nee, generell die ToDo-Liste? Also das mit oben einfügen und so.

0:21:30 (end of Example 7.18)

[...]

0:40:03 (start of Example 6.22, part 1)

D3: <lacht> Ähm, okay? Ich sehe, du bist mehr mitgekommen als ich <lacht> Ja, dann probier mal.

0:40:18 (end of Example 6.22, part 1)

[...]

1:14:25 (start of Example 6.18)

D4: <\*benennt Methodenparameter in "lista" um, IDE markiert zwei Compiler-Fehler\*> OK, also da holt er sich noch die (!...!) Objekte dann, wahrscheinlich <\*macht Umbenennung rückgängig\*>

D3: Warum hattest du das gerade umbenannt in lista?

D4: Nee, nur um zu gucken ob der (!...!) was er hiermit macht.

D3: Achso.

1:14:44 (end of Example 6.18, start of Example 7.19)

D4: Ist normalerweise gar nicht meine Vorgehensweise. Normalerweise schreibe ich echt immer nen Test.

D3: Machen wir hier fast gar nicht. Auch wenn wir es eigentlich sollten. Aber, naja.

1:15:04 (end of Example 7.19)

[...]

1:24:24 (start of Example 6.22, part 2)

D4: Ja, willst du dann noch mal machen, oder?

D3: Ich glaube, du bist da mehr involviert in diese ganze Sache. Ich bin da (!...!) für mich ist das schon oberste Wissenskante <lacht>.

1:24:34 (end of Example 6.22, part 2)

[...]

1:29:55 (start of Example 7.15)

D4: Warum steht denn hier AbstractList noch davor? <\*selektiert und löscht den Präfix, die IDE meldet einen Compilerfehler\*> Nee <\*macht Änderung rückgängig\*> Achso, ok. Weil wir in der anonymen Klasse sind.

D3: In 'ner anonymen Klasse? Was ist ne anonyme Klasse?

D4: <\*selektiert das new Statement, das den Anfang der anonymen Klasse markiert\*> Na, sobald du hier, z.B. so nen SelectionAdapter implementierst, dann ist das ja ne anonyme Klasse <\*selektiert den Rumpf der anonymen Klasse\*> weil die Klasse keinen Namen hat.

D3: Achso, ja, richtig.

D4: Deshalb hatte ich mich gerade gewundert, weil ich das nicht gesehen hatte. Und wenn ich jetzt natürlich this mache <\*entfernt Präfix nochmal\*>, dann ist das der SelectionAdapter, also bzw. die Implementierung davon. <\*fügt Präfix wieder ein\*>

1:30:25

(end of Example 7.15)

D3: Ja, ist richtig.

D4: Gut, warum das jetzt ne ListPreview hat, müssen wir ja nicht verstehen <lacht>. <\*öffnet nächsten Compiler-Fehler, hovert Fehler-Meldung\*>

D3: 'Import' einfach.

D4: Jetzt müsste ich ja nen Import machen können <\*wendet vorgeschlagene Autokorrektur an\*> ja.

1:30:49

(start of Example 7.17)

Kennst du das denn, mit dem OSGi-Class-Loading?

D3: Class was? Nicht wirklich, nee.

D4: Soll ich kurz sagen, oder?

D3: Na, ja klar.

D4: [...] Jedes Bundle [...] hat nen eigenen ClassLoader und der kann nur Klassen aus anderen Bundles laden, wenn [...] das andere Bundle, wo du die Klasse haben willst, davon das Package exportiert und wenn dein eigenes Bundle das explizit importiert. Und dann hast du hier halt immer in den Manifest-Dateien [...] da kannst du einmal Import-Package machen [...] dann sagst du, dass dieses Package haben willst.

D3: M-hm.

D4: [...] und normalerweise sollte man immer Import-Package nehmen, <lacht> hier ist es immer Require-Bundle. So bestimmst du halt den logischen Namen von dem Bundle was du importieren willst, so bist du explizit von diesem Bundle abhängig. Also du kannst jetzt nicht sagen, ich nehme das Bundle weg und nehme dann ein anderes Bundle, das auch dieses Package exportiert, dann würde sich OSGi dieses Bundle nehmen. [...] Deswegen ist Import-Package eigentlich immer schöner. Weil mit Require-Bundle importierst du meistens auch mehr Packages als du brauchst. [...] Dann kannst halt zur Laufzeit nichts mehr austauschen. Naja, aber, egal.

1:35:15

(end of Example 7.17)

Äh, so <\*öffnet nächsten Compiler-Fehler\*> ach jetzt musste BusinessAction (!..!) nagut <\*öffnet nächsten Compiler-Fehler, lässt automatisch Methodenstummel erstellen\*>

1:35:42

(start of Example 6.22, part 3)

D4: Also, sag, wenn du jetzt machen willst, ne?

D3: Nee, mach mal erst mal weiter. Ich sag, wenn ich wieder voll drin bin. Dann schrei ich schon.

1:35:52

(end of Example 6.22, part 3)

[...]

1:36:35

(start of Example 7.14)

D4: Kennst du denn das Pattern? [...] Weil das ist ja so ne Art Template Method. Also ich hab das hier halt ausgelagert, die gemeinsame Logik. <\*navigiert zu abstrakter Klasse, selektiert Aufruf einer abstrakten Methode\*>

D3: H-hm.

D4: [...] internalExecute ist ja praktisch meine Template Method. [...] Und dann kannst du halt sehr schön immer Sachen, die allgemein sind, auslagern und machst dann für die Sachen, die du noch nicht weißt, dann die abstrakte Methode, und die implementieren dann einfach deine Oberklassen.

D3: H-hm.

D4: Ist halt sehr schön, weil du kein Copy-Paste machen musst.

- D3: Richtig. (end of Example 7.14)  
 1:37:58  
 [...] (start of Example 6.22, part 4)  
 1:41:17  
 D4: Ja aber, wenn du willst, kannst du jetzt auch (!!...!!)  
 D3: Ja [...] lass uns mal kurz die Seiten tauschen. (end of Example 6.22, part 4)  
 1:41:22

## C.10 Session DA5

I introduced the base layer and the base concepts in Section 3.4 and illustrated their application with an excerpt from session DA5 which Plonka (2012, pp. 185–186) characterized as “*nudging*”, as a “*strategy [...] to provide a subtle learning opportunity*”. I did not analyze the whole session but just about 100 seconds. I shortened the pair’s exchange in Example 3.1. Here, I provide the full dialog with comments (see next subsection for the original German transcript).

### Example C.5: Coding with Base Concepts (DA5, 22:50–24:30)

The two developers are in the process of writing a test case. They already introduced about 20 lines of test setup and execution and now are about to write the first assertion.

- |   |   |
|---|---|
| (1) D8: “(##for##) <*selects <i>foreach</i> template from autocompletion*> Well, I go through and if I find one, what do I do with it?” | D8 presents a partial design proposal and asks her partner to complete it. (An <i>ask_design</i> would not contain any proposal.) |
| <i>propose_design</i> <sub>OE</sub>   |   |
| (2) D2: “Actually, it should have found exactly one. Normally, it should be only one activity.”   | D2 makes clear he disagrees with D8’s proposal without making one of his own.   |
| <i>disagree_design</i> + <i>explain_knowledge</i>   |   |
| (3) D8: “I see. Then I say ‘ <i>assert</i> ’?”  | D8 agrees with D2 and revises her proposal.   |
| <i>agree_knowledge</i> + <i>amend_design</i> <sub>OE</sub>  |   |
| (4) D2: “Exactly.”  | D2 is content.  |
| <i>agree_design</i>   |   |
| (5) D8: “<*deletes for statement*> So, only check for this?”  | D8 now doubts the proposal.   |
| <i>disagree_design</i>  |   |
| (6) D2: “Actually, yes, only that (!...!) actually, it would be enough that there is exactly one activity.”                             | D2 reassures D8 and refines the proposal.   |
| <i>amend_design</i>   |   |
| (7) D8: “Ah, during the test, I can see what’s in there. I’m still thinking with my old <i>sys.out</i> ”                                | D8 addresses her own mental model to explain why she was not sure about the original proposal.                                    |
| <i>explain_standard_of_knowledge</i>  |   |
| (8) D2: “<laughs> No, we can debug this.”   | D2 announces to use the debugger at some later point to achieve what D8 had in mind.  |
| <i>propose_todo</i> <sub>PI</sub>   |   |
| (9) D8: “(##Assert##), that I still know”   | D8 states she only remembers how to begin an assert statement and writes it down.   |
| <i>explain_standard_of_knowledge</i>  |   |
| (10) D2: “There is ‘ <i>assertTrue</i> ’ (!...!) ‘ <i>assertEquals</i> ’ you have to do”  | D2 tells D8 which assert method to select.  |
| <i>propose_step</i> <sub>PI</sub>   |   |
| (11) D8: “(##assertEquals##) (#boolean expected#)”  | D8 speaks while typing and reads out loud the autocompletion overlay.   |
| <i>write_sth</i>  |   |
| (12) D2: “‘ <i>true</i> ’”  | D2 tells D8 which value to insert.  |
| <i>amend_design</i> <sub>PI</sub>   |   |

## Example C.5 (continued)

- (13) D8: “(##true##) and now it’s expecting a boolean? No.” *explain\_finding<sub>D</sub>* D8 notices the auto-completion does not match with her expectations.
- (14) D2: “*activities.size() equals one*.” *amend\_design<sub>PI</sub>* D2 tells D8 which expression to insert.
- (15) D8: “Ah, so it’s boolean indeed.” *disagree\_finding* D8 acknowledges the auto-completion is not wrong after all.
- (16) D8: “<\*wiggles mouse\*> Get lost!” *say\_off\_topic* D8 tries to get rid of the auto-completion overlay.
- (17) D2: “Nope, I’ll stay here.” *say\_off\_topic* D2 makes a joke.
- (18) D8: “I don’t know what it’s called! (##activities.size()##)” *explain\_standard\_of\_knowledge* D8 cannot read the variable name behind the overlay, but the auto-completion helps her remember it.
- (19) D2: “*equals one*.” *amend\_design<sub>PI</sub>* D2 tells D8 how to complete the expression.
- (20) D8: “<\*types ==1\*> Greater than zero? Nope, equals one. Must be only one.” *challenge\_design<sub>OE</sub>* D8 briefly considers another expression.
- (21) D2: “I hope, there is only one. ‘*There can be only one.*’ That’s the Highlander principle.” *explain\_knowledge* D2 explains some reservations (again, there is no concept more specific than *knowledge*) and makes another joke.
- (22) D8: “It took me a while until I understood what y’all are talking about. Highlander. <chuckles>” *say\_off\_topic* D8 refers to the joke. Possibly it is a running gag in the team.
- (23) D8: “OK. This is done now. And now <\*inserts two empty lines\*> we go on.” *explain\_completion* D8 does not make a concrete proposal (neither *design* nor *step*) but evaluates the result: The assert statement is finished.
- (24) D2: “We can also test this first?” *propose\_step<sub>OE</sub>* D2, in contrast, makes a concrete proposal.
- (25) D8: “Yes? OK. <\*deletes empty lines\*>” *agree\_step* D8 seems surprised by D2’s proposal, as if she expected the session to take a different route (which she does not verbalize), but agrees to it.
- (26) D2: “You know, I have to admit, I’m a bit nervous.” *explain\_knowledge* D2 appears to justify his proposal to run the test case before adding new logic. If his disagreement with D8’s assessment of the situation was clearer, one could annotate *disagree\_completion* instead.

**Technical note:**

The ‘expert’ D2 dictates to write the statement `assertEquals(true, activities.size()==1)`, which is quite verbose and does not indicate his proficiency with the JUnit framework. To the very least, he could have proposed `assertTrue(activities.size()==1)`. A comparison of integers would have been more straightforward and also produces more readable error messages: `assertEquals(1, activities.size())`. In fact, in turn (13), his ‘novice’ colleague D8 appears to be confused by D2’s proposal to write an assertion with a boolean value where integers would be more fitting.

### C.10.1 Transcripts of DA5 Excerpts

- 0:22:50 (start of Examples 3.1 and C.5)
- (1) D8: (`##for##`) *<\*wählt "foreach" Vorlage aus Autovervollständigung aus\*>* So, ich geh einmal durch und wenn ich eine finde, was mach ich damit?
  - (2) D2: Eigentlich müsste er ja nur genau einen gefunden haben. Im Normalfall dürfte er nur eine Aktivität haben.
  - (3) D8: Achso. Dann sage ich `'assert'`?
  - (4) D2: Genau.
  - (5) D8: *<\*löscht for-Statement\*>* Also nur danach prüfen?
  - (6) D2: Eigentlich ja, nur dass es (!...!) Eigentlich würde es sogar reichen, dass es genau eine `activity` gibt.
  - (7) D8: Achso beim Test kann ich sehen, hier sehen, bei Introspection oder so, was da drin steht. Ich bin immer noch bei meiner alte `Sys.out`.
  - (8) D2: *<lacht>* Nee, wir können das debuggen.
  - (9) D8: (`##Assert##`), das hab ich noch
  - (10) D2: `'assertTrue'` gibt's (!...!) `'assertEquals'` musste machen.
  - (11) D8: (`##assertEquals##`) (`#boolean expected#`)
  - (12) D2: `'true'`.
  - (13) D8: (`##true##`) und jetzt soll hier ein boolean? Nee.
  - (14) D2: `'activities.size() gleich eins'`.
  - (15) D8: Ach doch ein boolean.
  - (16) *<\*bewegt die Maus über Autocomplete-Popup\*>* Geh weg!
  - (17) D2: Nee, ich bleib hier sitzen
  - (18) D8: Ich weiß nicht wie das heißt! (`##activities.size()##`)
  - (19) D2: `'gleich eins'`.
  - (20) D8: *<\*tippt ==1\*>* Größer Null? Nee, gleich eins. Muss nur einer sein.
  - (21) D2: Ich hoffe, dass es nur einen davon gibt. Es kann nur einen geben. Das Highlander-Prinzip ist das.
  - (22) D8: Ich habe lange gebraucht, bis ich das verstanden haben, wovon ihr sprecht. Highlander. *<lacht>*
  - (23) OK. Das haben wir jetzt. Und jetzt *<\*fügt zwei Zeilenumbrüche ein\*>* gehen wir weiter.
  - (24) D2: Wir können das auch erstmal testen?
  - (25) D8: Ja? OK. *<\*löscht Leerzeile\*>*
  - (26) D2: Ich muss ja gestehen, ich hab ein bisschen Angst.
- 0:24:30 (end of Examples 3.1 and C.5)

### C.11 Session EA1

Company E develops a graphical desktop application for different logistics related tasks in the C++ programming language. Prior to session EA1, experienced developer E2 (ten years of experience) already tried to debug a display error that leads to routes of ferries being displayed with an extra segment. In their PP session, he and junior developer E1 (five months of development and PP experience) go through the unfamiliar source code step by step with a debugger. They would mostly set up a certain state, inspect variables, develop and test hypotheses, but they do not change any code. They end their session after 80 minutes because of a team meeting without having really circled in on the error.

### C.11.1 Transcripts of EA1 Excerpts

0:04:23

(start of Example 10.5)

E2: Jetzt zeig ich dir mal wie ich das gemacht hab. <\*klickt Fehlermeldung weg\*> OK, das ist ganz normal. <\*startet Anwendung\*> So, dann hab ich jetzt (#Routing#) (#Stationen laden#) da hatten wir jetzt gehabt die <\*"Routename"\*> genau. So, wenn er die jetzt dann routet (!...!) ah hier unten ist sie. Ah nee, was ist jetzt das? Falsche Route. <\*öffnet andere Route\*> Genau. Und jetzt ist der Fehler quasi das hier. Also wenn ich jetzt die Fähren aktiviere, zum Beispiel jetzt so <\*klickt im Menü\*> dann sieht man dass das letzte Segment hat noch so'n Extra-Punkt.

E1: Jo.

E2: Und der letzte Punkt sollte eigentlich der hier sein <\*zeigt auf zwei Punkte in der GUI\*> und er nimmt aber diesen hier.

0:05:35

[E2 erklärt weiter, E1 nickt immer zu]

0:09:54

E2: result ist die Anzahl der Punkte. Das wird eigentlich genauso ausgerechnet. Das wird jetzt nochmal hier erhöht. <\*öffnet Inspector\*> Und jetzt hat er 131. Vorher hat er einen gehabt, das ist glaube ich einfach der Startpunkt.

E1: M-hm.

E2: Und dann hat er jetzt noch mal 130 draufgekriegt. Und jetzt steht ja hier in diesen pPoints, da stehen ja dann die (...) stehen die Punkte da drin.

E1: Und die Polygon-Punkte, über die die Route geht, die haste noch drin?

E2: Ja genau, dieses TraceFerry, das hat nen Ausgabeparameter, wo jetzt die Punkte reinkopiert werden.

E1: M-hm.

E2: Und der ruft das immer hintereinander auf, weißte der macht sich nen großes Array (!!Ja!!) und da tut er erst für den einen Stummel, und dann sagt er dem anderen, 'bitte ab hier reinkopieren', so macht er das dann.

E1: M-hm.

E2: Und in dem TraceFerry werden die dann da reinkopiert.

E1: OK.

E2: Genau. Und ähm, jetzt tut er in dieses pPoints-Array, das beinhaltet jetzt dann quasi die Punkte. Und da müsste jetzt der Endpunkt eventuell richtig sein, oder auch nicht, das müssen wir jetzt halt noch sehen. Ähm, jetzt ist wieder die Frage, wie viele das dann sind. Kann ich da einfach 132 eingeben. So, also der 131er ist Null, das heißt, es sind tatsächlich schon 131 belegt, ja? Und jetzt siehst du, dass der gleich dem ist.

E1: Ja.

E2: Das heißt, da ist der Fehler eigentlich schon drin.

E1: M-hm. Aber warum?

E2: Also ich glaube wir könnten jetzt auch, zum Beispiel wenn wir uns den mal kurz notieren. Ich schreib mal kurz auf. Kannste mal diktieren?

[E1 diktiert mehrere Punkt-Koordinaten, E2 schreibt sie auf Papier]

E2: Wenn wir jetzt wieder laufen lassen <\*setzt Programmausführung ab Breakpoint fort\*> dann können wir jetzt nämlich hier sehen, dass das wahrscheinlich genau der da ist. <\*vergleicht Werte\*> Ja, das ist der, und der ist da schon doppelt. Der Fehler legt dann also bei diesem TraceFerry irgendwo drin. Jetzt müssen wir gucken, dass wir da wieder hinkommen. Ich lass einfach nochmal laufen.

E1: Also sprich, der wird eigentlich (!...!) wir unterbrechen die Route insofern (!...!) also der geht halt hier hin, geht hier hin, geht wieder zurück, und setzt den als Endpunkt dann auch, oder was?

E2: Genau, also er hat quasi von den Punkten die ganzen da vorne. Dann hat er den, dann hat er den, und anstatt dass er den, setzt er den falsch.

E1: Ja, und dann geht er wieder zurück. OK.

E2: OK, alles klar. Ähm, ach nee, Quatsch. Also jetzt route ich noch mal. Jetzt haben wir natürlich dummerweise nen Haufen Breakpoints. Jetzt suchen wir mal die Stelle, wo er die Polygone zusammensammelt. So, hier sammelt er jetzt die Polygone. Und dann geht er in dieses Trace Ferry hier rein. [...] Jetzt warten wir auf diesen Punkt hier. Jetzt hab ich irgendwie gedacht, dass es an dem lag, aber daran liegt's nicht, sondern es liegt an dem TraceFerry, da kommt's falsch raus.

0:13:45

(end of Example 10.5)

## C.12 Session JA1

Domain expert J2 had designed and implemented a plugin-based architecture in Java to monitor and download remote files from the servers of different radio stations about a year earlier. He invites experienced consultant J1 to a distributed pair programming session to review and clean-up the code together in order to ease the implementation of a new feature later on (see Section 4.4.4 for more details on the context, the developers, and their session). In session JA1, they review but one class, try and fail to refactor it by extracting local methods, and ultimately decide to rewrite the whole system from scratch.

### C.12.1 Transcripts of JA1 Excerpts

0:01:47

J2: Wir haben heute hier einen Task, hier am NewsPlugin etwas zu machen. Im großen Ganzen geht's um Refactoring von den ganzen Klassen, weil die schon relativ alt sind. Du weißt ja, wie bei mir normalerweise alte Klassen aussehen (!!...!!)

J1: <lacht> richtig ja.

J2: <lacht> die etwas älter sind. Es geht zum einen um Refactoring zum anderen würde ich gerne diese Samba-File-Geschichte loskriegen, das heißt Zugriff auf das DFS via Samba.

J1: M-hm.

J2: Und ersetzen durch Zugriffe auf's lokale Dateisystem mit Linux-Freigaben.

J1: M-hm.

J2: Weil das einfach die Stabilität deutlich verbessert.

J1: Ja klar.

0:02:29

(start of Examples 5.1 and 7.1)

(1) J2: Kennst du das NewsPlugin, oder kennst du das nicht?

(2) J1: <atmet hörbar aus> zeig's mir einfach noch mal.

(3) J2: Das mit dem Nachrichten-Mitschnitt, ich glaube wir sharen mal.

0:02:38

[Das Paar startet eine Saros-Sitzung.]

0:04:09

(start of Examples 6.13 and 10.3)

(4) J2: OK, aber ich kann dir ja schon mal sagen, was dieses Plugin im Großen und Ganzen tut.

(5) J1: Ja

0:04:15

(start of Examples 8.1 and 9.1)

(6) J2: Also hinten raus ploppt der Nachrichten-Mitschnitt von jeder Stunde.

(7) J1: M-hm.

(8) J2: Die Vorgehensweise ist so, es gibt mehrere (.) Processors, also es gibt das zentrale Plugin, dann gibt es mehrere Processors, die sich alle um eine Welle jeweils kümmern.

(9) J1: <\*nickt\*>

(10) J2: Ähm. (.) Bei den meisten ist es so, dass kurz nach der vollen Stunde geprüft wird, ob auf dem Share ein, oder, ob es auf dem entfernten Share eben eine neue Datei vorliegt.

- (11) Wenn ja, (.) äh wird die letzte Datei ausgewählt und es wird angefangen zu prüfen, wie die sich in ihrer Größe sich noch verändert.
- (12) J1: <\*hört auf zu nicken, schaut nach rechts oben\*>
- (13) J2: Das heißt, es wird so lange geguckt, bis die Datei nicht mehr größer wird, dann ist sie wohl fertig.
- (14) J1: M-hm
- (15) J2: Und dann wird sie abgeholt und zur Transkodierung gegeben.
- 0:05:00 (start of Example 8.10)
- (16) J1: In was für nem Zeitfenster wird dann geguckt?
- 0:05:03 (start of Example 7.20)
- (17) J2: Ich fange an zu gucken um zwei Minuten nach der vollen Stunde, weil da garantiert ist, dass dann Nachrichtendateien vorliegen wenn welche vorliegen.
- (18) J1: OK
- (19) J2: Und monitor' dann eben so lange diese Datei bis sie fertig ist. Das kann bis zu sieben Minuten dauern, je nach Welle.
- 0:05:19 (end of Example 7.20)
- (20) J1: Hm genau, aber mh, also das Zeitfenster für die Veränderung?
- (21) J2: Ja genau, das ist, äh, Zeitfenster für die Veränderung, das ist variabel, je nachdem wie die Nachrichten gehen.
- (22) Das weiß ich ja nicht. Also es ist so, dass, die legen automatisch immer ne neue Datei an. Wenn die Nachrichten zu Ende sind, wird wieder ne Datei angelegt. Das heißt, ich hab quasi nie mehr als die Nachrichten.
- (23) J1: Ja, nee, ich mein jetzt nur weil du sagst, du guckst halt so lange, ähm, äh, bis die Größe aufhört sich zu ändern, ja?
- 0:05:48 (end of Example 8.10)
- (24) J2: M-hm
- (25) J1: Musst du ja noch 'nen gewisses Zeitfenster noch einplanen, in der immer noch 'ne Veränderung stattfinden könnte.
- (26) J2: Ja gut, bis maximal fünf vor der neuen Stunde. Also, ich warte wirklich lange.
- 0:06:00 (start of Example 7.23)
- (27) J1: <lacht> Nee, ich mein tatsächlich die Größe jetzt, die Größe des Zeitfensters, also (.) du wartest 10 Sekunden, dann nach 10 Sekunden entscheidest du, in den 10 Sekunden hat sich jetzt nichts mehr verändert, dann ist die Datei wohl fertig.
- (28) J2: Achso, das meinst du, nee 30 Sekunden.
- (29) J1: 30 Sekunden, das wollt ich wissen.
- 0:06:12 (end of Example 7.23)
- (30) J2: Das ist 30 Sekunden lang das Zeitfenster. Jetzt hab ich dich verstanden.
- 0:06:15 (end of Examples 5.1, 6.13, 7.1, 8.1, and 9.1)
- [...]
- J2: Aber kann ich dir gleich zeigen.
- J1: Ja. Und das NewsPlugin macht jetzt in dieser ganzen Sache was davon? Also, genau dieses Monitoring und dann die Delegation an die einzelnen Wellen-Plugins, oder?
- J2: Nee. Das NewsPlugin kümmert sich eigentlich nur darum (!...!) das wird eben periodisch aufgerufen vom Cron-Server [...]
- 0:06:33 (end of Example 10.3)
- [...]
- 0:08:27 (start of Example 8.3)
- J2: Die Funktion ist extrem lang, aber die lässt sich eigentlich ganz gut auch aufspalten, denke ich. Also da sollten wir nachher im Zuge des (!...!) oder jetzt im Zuge des Refactorings vielleicht mal drübergehen. Weil ich denke, dass vor allem nachher, wenn wir die Dateien austauschen wollen, oder den Zugriff auf die Dateien austauschen wollen, wir (!...!) das in kleinen Bröckchen irgendwie besser geht, also wenn wir da ne bessere Übersicht haben.

J1: Ja.  
J2: Teilst du meine Ansicht?  
J1: Ja. Ähm, gibts, können, gnah.  
J2: Wenn können auch gerne durchgehen, wenn du magst.  
J1: (.) Ja? (... ) Ähm, wie isses denn (!...!) ja, lass' uns mal durchgehen.  
J2: Also, wir haben hier [...]  
0:09:19 (end of Example 8.3)  
[...]  
0:09:32 (start of Example 8.14)  
J1: Wir gehen rein mit der `currentTime`.  
J2: Genau, `currentTime`, das ist der aktuelle Zeitstempel beim Aufruf.  
J1: Beim Aufruf, wozu braucht er die (von außen)?  
J2: Brauch ich noch öfters zum Beispiel hier bei `getLastFile` <\*selektiert Statement\* >, wenn er sich die (!...!) damit er ne Vergleichszeit hat, ne?  
J1: Ja, nee, ich meine, warum muss das von außen reingegeben werden? Also, die Funktion weiß doch eigentlich auch selber wie viel Uhr wir haben.  
J2: Dass jeder Processor die gleiche Zeit kriegt.  
J1: OK (.) ja.  
0:10:01 (end of Example 8.14)  
J2: <\*öffnet Datei\* > Also wir haben hier, siehste ja, Zeile 85 <\*markiert diese Zeile\* >, dass wir die `currentTime` hier speichern und dann kriegt jeder dieselbe. Also das ist die Zeit beim Aufruf. Deswegen habe ich die von außen reingegeben. Aber du hast Recht, wir können nachher mal prüfen, inwiefern wir die da dringend brauchen, also dass jeder dieselbe Zeit hat. Weil dann könnten wir das auch nach innen verlagern, weil es ist natürlich schon ein bisschen komisch, warum man die `currentTime` hier braucht.  
J1: Das wäre jetzt halt meine nächste Frage gewesen, warum die halt (!...!)  
0:10:31 (see Example 8.6)  
J1: Gibt's nen zwingenden Grund, dass alle `NewsProcessors` die gleiche (Zeit)?  
0:10:39  
[...]  
0:13:15 (start of Example 10.6)  
J2: Und setzt aber dafür das `remoteNewsFile`. Und zwar kriegt er das aus der Funktion `getLastFile`. Das wäre ne Funktion, die es nachher auch zu ersetzen gilt. [...] Sollen wir in die Funktion reingehen, oder?  
J1: Nee, erstmal nicht bitte.  
J2: Erstmal nicht, ok.  
0:13:43 (end of Example 10.6)  
[...]  
0:14:15 (see Example 8.6)  
J1: Kann das (!...!) Ist das nen erwarteter Fall, also kann das passieren?  
J2: Also das ist der Fall wenn er nichts gefunden hat.  
J1: Aber sollte das passieren? Das sollte eigentlich nicht passieren, gell?  
J2: So, und wenn die Minute nicht (!...!) wenn es nicht zwei nach der vollen Stunde ist, dann NOOP, macht er halt nix.  
J1: < \*\*J2s Name\*\* >? Hörst du mich?  
J2: Ja.  
J1: OK.  
0:14:35 (start of Example 8.23)  
J1: In diesen Fall hier mit dem Error sollte er eigentlich niemals reinrutschen, wenn alles glatt läuft, richtig?  
J2: Wenn alles glatt läuft, nie reinrutschen, genau.

J1: (OK)

J2: Also wenn er ne Datei gefunden hat, dann rutscht er da auch nicht rein. Ja? <\*selektiert return-Statement im default-Fall\*> Weil dann ist er ja in diesem FileTracking-Modus. (...) Ja? (.) Oder nein? Warum sagst du nix? <lacht>

0:14:58 (end of Example 8.23)

[...]

0:19:13 (start of Example 9.16)

J2: Ok, dann haben wir hier (!...!) öh, ja, es wird halt kopiert. Siehst du ja hier, Zeile 101 <\*setzt Cursor in Zeile 101\*>. Und zwar in die Datei localNewsFile <\*selektiert this.localNewsFile in Zeile 101\*>. (...) <\*setzt Cursor in Zeile 102\*> kopiert er das hier (!!Moment!!) ähm

J1: Momentmomentmoment

J2: Hm?

J1: Kurz gucken.

J2: (...) Was meinst?

J1: OK. Ne, ich hab die Zeile nur nicht gelesen gehabt.

J2: OK.

J1: (...) OK, ja.

0:19:39 (end of Example 9.16)

[...]

0:22:59 (start of Example 9.11)

J1: Weil du die in ner Klassenvariable gespeichert hast [...] Warum muss remoteNewsFile in der Klasse selbst definiert sein? Warum reicht das nicht, das in der Methode zu machen?

J2: Ähm, das remoteNewsFile? Du meinst, warum es hier nicht reicht?

J1: Ja, das remoteNewsFile.

J2: Lass mal überlegen, ob der <\*\*\*J1s Name\*\*> da recht hat. <\*beginnt zu scrollen\*>

J1: <\*liest im Code\*> Es gibt ein Auftreten [...]

0:23:48 (end of Example 9.11)

[...]

0:24:16 (start of Example 8.24)

J2: Ja, also diese (!...!) man sollte es vielleicht so sagen: Die muss ich mir merken, denn diese execute-Methode, <\*\*\*J1s Name\*\*>, die wird alle 30 Sekunden aufgerufen.

J1: Ja?

J2: Das heißt, ich muss mir ja den Status speichern. Wenn ich mir die Datei von remote geholt habe und dann das FileTracking mache, muss ich mir ja ne Referenz darauf speichern. (...) Ich will's ja nicht jedes Mal (...) neu anlegen. Kann ich ja gar nicht.

0:24:44 (end of Example 8.24)

[...]

0:27:47 (start of Example 8.17)

J1: Das heißt, die gesamte Funktionalität hier oben ist eigentlich über alle anderen News-Plugins, also die Unter-News-Plugins, gleich. Sehe ich das richtig, oder?

J2: Nee, das ist nicht korrekt. Denn, die sind alle so'n bisschen unterschiedlich.

J1: 'So'n bisschen unterschiedlich', ok?

J2: Ja, es ist etwas problematisch hier [...]

J1: Das heißt, jede einzelne Klasse mit '\_News' am Ende macht was eigenes in der execute-Methode.

J2: Ja, die wird überschrieben. Siehst du ja auch hier (!...!!)

J1: Jaja, das sehe ich. Aber, äh, das, womit es überschrieben wird, ist halt wirklich, also, wenn du zwei beliebige Dateien nimmst und die vergleichst, sind die immer irgendwie unterschiedlich.

J2: Die sind unterschiedlich, ja!

0:28:44 (end of Example 8.17, start of Example 8.5)

J2: Das ist gleich für <\*\*\*Alpha\*\*> und <\*\*\*Beta\*\*>. [...]

J1: <herausfordernd> Und deswegen ist es in AlphaNews.  
J2: Warte mal, wenn du jetzt (!!...!!)  
J1: Also wenn ich jetzt Alpha\_News und Beta\_News vergleichen würde, dann wäre es nicht unterschiedlich.

0:28:57 (start of Example 8.15)  
J2: Ja, dann guck dir bitte mal Beta\_News an. <\*öffnet Beta\_News, scrollt in die Mitte der Datei\*  
Ich hab sie gerade offen. Du siehst, hier gibt es keine execute-Methode.  
J1: <\*springt zu J2s Position\* > Da gibt es tatsächlich keine execute-Methode.  
J2: Nein, weil für die beiden ist es gleich.  
J1: Und das heißt?  
J2: Wie, 'das heißt'?

0:29:18 (end of Example 8.5)  
J1: Das heißt, äh, wo ist denn die execute-Methode für <\*>Beta\*>?  
J2: Ja die execute-Methode von <\*>Beta\*> entspricht der von <\*>Alpha\*>.  
J1: Ja, klar. Aber da muss ja irgendwo eine Verbindung hergestellt werden. [...]  
J2: Ich verstehe die Frage nicht. Es tut mir leid. [...]  
J1: [...] ach (#extends Alpha\_News#)

0:29:47 (end of Example 8.15)  
[...]

0:40:29 (start of Example 9.5)  
J1: <\*hat Code-Block selektiert, ruft Refactoring "Extract Method" auf\* > Was macht das alles?  
J2: Also es guckt eigentlich nur nach der Dateigröße [...] und lädt sie runter.  
J1: <\*tippt den Methodennamen "downloadRemoteFile", klickt "Extract"\* >  
J2: Deswegen wollt ich nämlich hier oben dann <\*selektiert einige Zeilen\* >, damit wir das irgendwie unterteilen könnten (!!...!!)  
J1: <\*Fehlermeldung geht auf\* > Funktioniert so noch nicht.  
J2: Also hier (!!...!!)  
J1: <\*schließt Fehlermeldung, liest Quellcode\* > Stimmt, ja der macht ja nen return, genau.  
J2: Also (!!...!!)  
J1: Das funktioniert noch nicht.

0:40:42 (end of Example 9.5)  
[...]

0:53:56 (start of Example 6.24)  
J1: Ahja, das heißt diesen <\*selektiert Zeilen 88 bis 105\* > (!!...!) das heißt diesen Fall können wir eigentlich mal in das try [Zeilen 76 bis 79] reinziehen, oder?  
J2: Das können wir (,,) das, nein! Nein-nein-nein-nein-nein-nein. Können wir nicht, weil <\*Cursor zu Zeile 51, selektiert if-Schlüsselwort für J1\* > bedenke bitte in Zeile einundfünfzig <\*selektiert ganzes if-Statement für J1\* > das localNewsFile  
J1: Zeile einundfünfzig, was?  
J2: Das können wir können wir nicht machen.  
J1: Achso, du meinst <\*scrollt nach oben\* > dass wir (!!...!)  
J2: Was wir aber machen können (!!...!!)  
J1: Aargh-ha-ha-ha urgh <kapitulierend>, oh Gott. Ja, du hast recht, du hast recht.

0:54:34 (end of Example 6.24)  
[...]

0:58:23 (start of Example 10.9)  
J2: Wo fang ich an zu erklären? <lacht> Das ist nämlich etwas komplizierter wie du denkst.  
J1: Sicherlich.

J2: Weil wenn `<Welle Alpha>` oder `<Welle Beta>` keine eigenen Nachrichten haben, wenn das bei denen irgendwie ausfallen sollte, dann übernehmen sie diese `<fallback>`-Nachrichten. Und je nach dem, wenn es ihre eigenen sind, also wenn das Konstrukt hier hinten `true` ist `<zeigt auf Code>`, dann sind es ihre eigenen Nachrichten-Dateien, dann soll er sie löschen. Wenn es die `<fallback>`-News sind, soll er sie natürlich nicht löschen, weil es ja noch eventuell noch andere gibt, die die auch noch benötigen. Ja? Aber das kann man in Zukunft alles etwas anders gestalten (!!...!!)

J1: `<schüttelt den Kopf>` Andere Baustelle, andere Baustelle, das ist (!!...!!)

J2: Ja, das ist ist noch ne andere Baustelle. Das habe ich nur deswegen gemacht, weil der Download immer so lange gedauert hat, von so ner blöden Nachrichten-Datei. Da habe ich mir gedacht, da muss es nicht noch länger dauern, dann verwende ich für alle dieselbe. Wenn das nachher auf dem lokalen Dateisystem läuft, dann geht das natürlich bedeutend schneller und dann kann man die auch zweimal runterladen.

0:59:32

(end of Example 10.9)

## C.13 Session JA2

The same pair as in session JA1 (see Section 4.4.4 for more details on the context, the developers, and their session) starts with implementing J2's system from scratch about two weeks later. In the first part of the session, J2 shows J1 a number of helper implementations he wrote in the meantime and J1 criticizes them. Afterwards, they discuss and collect requirements together in a plain text file.

### C.13.1 Transcripts of JA2 Excerpts

0:13:41

(start of Example 8.11)

J2: Im Grunde genommen funktioniert es so, du hast halt einen `TranscodeJob`, den du anlegen möchtest. Da kann man beliebige `InputFiles` eben setzen. Ja? Du kannst beliebig viele setzen. Ich habs deswegen gemacht, dass man nicht gleich mehrere setzen kann, weil du das ja auch schon letztes Mal richtig erkannt hattest, das blockweise funktioniert. (!!Genau!!) Also du hast ein `InputFile`, dann kommt alles was damit gemacht werden soll, dann `OutputFile`. Dann kommt wieder das `InputFile`, alles was gemacht werden soll, `OutputFile`. (!!Ja!!) Also so ist das aufgebaut. (!!Ja!!) OK, ähm (!!...!!)

J1: Ja, momentmomentmoment, du hast doch gerade gesagt, du kannst beliebig viele setzen.

J2: Ja, in dem ich sag, `(##job.setInputFile##)` noch ein `InputFile`.

0:14:25

(end of Example 8.11)

[...]

0:17:53

(see Example 8.6)

J1: Wieso ist der Encoder als String und nicht als Konstante hinterlegt?

J2: Weil es da sehr sehr viele gibt. Und ich weiß nicht (!!...!) und das kann sich mit den ffmpeg-Versionen immer wieder ändern.

J1: Ja genau, und dann will ich es ja nicht überall mit String-Suche, wenn sich da was ändert, irgendwie (!!...!!)

J2: Sicher. Nee, aber was da auf jeden Fall noch reinkommt, ist: Ich hab ne Klasse Encoder. Und du kannst dir auch alle Encoder listen lassen, die es so gibt. Das heißt eigentlich wäre der richtige Weg, dass man sich die listet und dann den entsprechenden Encoder raussucht.

J1: Würde ich so nicht sehen, nee. Ich würd' nen Enum machen. Ich würd nen ganz einfaches Enum machen.

J2: Nee, finde ich nicht gut.

J1: Wieso findest du das nicht gut?

J2: Finde ich nicht gut (!!...!!)

J1: Anhand welcher Dings willst du denn den Encoder raussuchen?  
0:18:38 (start of Example 9.22)  
J1: Anhand welchen Parameters möchtest du es jetzt (!...!) also wenn du hier schreibst 'set Encoder' von was? 'Encoder.getEncoderBy' und dann kommt was?  
J2: Nee, pass auf. Es geht folgendermaßen. Du kannst dir ne Liste von allen zur Verfügung stehenden Audio-Encoder geben lassen.  
J1: <zufrieden, erwartungsvoll> Ja. Und wie finde ich raus, welches der richtige ist?  
J2: (, , ,) Äh, ja gut. Ist jetzt (!...!) ich mein du kannst ja nen beliebigen wählen. Es gibt ja nicht 'den richtigen'.  
J1: Ja, klar, aber nehm ich denn nen 'beliebigen', oder nehm ich den ersten, oder den fünften, oder was? Ich will mir ja wahrscheinlich einen auswählen der WMAs generiert.  
J2: Genau. (!!...!!)  
J1: Das ist dann der WMA-Encoder, wmv2.  
J2: Ja, genau, der heißt halt so (!!...!!)  
J1: Genau, 'der heißt halt so' und du sagst jetzt, dieser Name kann sich in späteren Versionen von ffmpeg möglicherweise ändern, richtig?  
J2: Ja, man weiß es nicht.  
J1: 'Man weiß es nicht', so das heißt man möchte es nicht an dem Namen festmachen, sondern ich möchte es daran festmachen, dass es der WMA-Encoder ist. Da würde sich doch anbieten (!!...!!)  
J2: Nen Enum mit nem String, im Konstruktor nen String, ja, hm.  
0:19:43 (end of Example 9.22)  
[...]  
0:25:21 (see Example 8.6)  
J1: Warum ist denn das Timeout da?  
0:25:23

## C.14 Session KA1

Company K develops and operates a large real-estate online platform. Junior developers K1 and K2 come together to work out an API between their respective teams' subsystems: K1 is responsible for a mobile app for which K2 writes the endpoint with Java Spring web framework.<sup>1</sup> Before they can start with their actual task of session KA1, they first need to change the target URL of a single link which takes them more than 45 minutes and the help of two colleagues, because their development environment was not properly set up. Afterwards, K1 explains the data he needs with some dummy JSON file he prepared and K2 considers which internal microservices are able to provide which kind of data.

### C.14.1 Transcripts of KA1 Excerpts

0:51:17 (start of Example 7.10)  
K2: Na, ich kann dir erstmal zeigen, was wir jetzt schon haben, und dann können wir vergleichen, was das Finanzexposee hat.  
K1: <\*kündigt an, dass er noch kurz etwas auf seinem Telefon fertig machen möchte; das dauert ca. 90 Sekunden\*>  
K2: Also, wir haben ExposeApiClient <\*öffnet diese Klasse\*>. Der hat so ne getAsJson und damit können wir uns generell erstmal das Exposé-JSON, was von der API kommt, holen. Das ist aber nicht schön, weil du hast da hundert Sachen, die du nicht brauchst, und so verschachtelt und alles.

---

<sup>1</sup>Project homepage: <https://spring.io/>

- K1: Genau, und es gibt auch irgendwie tausend Fallunterscheidungen und es ist absolut beschissen dokumentiert, habe ich festgestellt. Es ist irgendwie nur für einen Objekttyp dokumentiert, es gibt aber ungefähr 20.
- K2: Genau, richtig. Ja, genau, es gibt 20, ja. Äh, genau, dann haben wir den `ExposeApiService`, der nutzt jetzt so'n Teil `<*öffnet diese Klasse*>`, also den `ExposeApiClient`, holt sich den JSON und packt das bei uns in nen Mapper rein. Und der Mapper bildet das dann auf, ähnlich wie beim Finanzexposee, auf die Objekte, die wir brauchen. `<*wählt eine Reihe von Datenklasse in der Übersicht aus*>`.
- 0:53:37 (end of Example 7.10)  
 Bisher haben wir dann nur das Exposee selber `<*öffnet diese Klasse*>`, das ist für mich dieses Hauptelement (!!...!!)
- K1: M-hm. Da haste die ID (!!...!!)
- K2: Da haben wir nur die ID jetzt drin (!!...!!).
- K1: Und das hat dann halt nen Sub-Objekt.
- K2: Genau und das ist dann das `RealEstate`. Und das `RealEstate` hat eigentlich auch noch (!!...!) was wichtig zurückkommt wäre auch laut Schema irgendwie `ContactDetails`, oder sowas, aber das brauchen wir glaub ich nicht `<*schaut zu K1*>`. Weiß nicht, das Exposee braucht die glaub ich auch nicht, äh Finanzexposee
- 0:54:00 (start of Example 7.21)  
 In dem `RealEstate` da haben wir jetzt auch erst mal nur die Daten, die wir brauchen `<*öffnet diese Klasse*>`. Das sind bei uns Titel, ne Adresse, der Preis, `livingSpace`, `plotArea` (!!...!!)
- K1: Was ist denn `plotArea`?
- K2: Ähm, `siteArea`, also Grundstückspreis (!!OK!!) nee nicht Grundstückspreis, Grundstücksfläche. (!!OK!!) `livingArea` ist halt Wohnfläche.
- K1: Ja
- K2: Und dann haben wir noch den `marketValue`, das ist auch so ne Art Preis. Und wir haben `constructionYear` und `modernizationYear`.
- 0:54:29 (end of Example 7.21)  
 [...]
- 0:59:23 (start of Example 7.11)  
 K1: Und dann brauch ich halt diese ganzen, so ein paar Sachen, da dieser `floorSpace` und so weiter, und wie viele Zimmer das Teil hat, und dann halt diese ganzen `taxValues`.
- K2: Das haben wir schon `<*selektiert nächste JSON-Zeile*>` genau, das müssten wir halt noch gucken `<*selektiert nächste JSON-Zeile*>` da ist halt nur die Frage, wo die alle herkommen, bzw. wie wir die finden und ob das Unterschiede zwischen den Typen sind. Ich glaube bei dem einen Meeting haben die gesagt, bei irgendeinem gab es noch Probleme. Ich weiß aber nicht, ob die das schon gefixt haben. Weil eigentlich haben sie gesagt, sie haben keine Bugs mehr offen.
- 1:00:01 (end of Example 7.11)  
 Wir können ja einfach mal bei denen noch mit reingucken. Das habe ich auch noch auf.
- 1:00:05 (start of Example 7.22)  
 Hier haben wir auch so nen Holder `<*öffnet Klasse ExposeHolder*>` das ist schon das Data-Element. [...] Hier müsste alles dabei sein. `courtage`, brauchst du das?
- K1: Was ist denn `courtage`?
- K2: Keine Ahnung, `<*öffnet Online-Wörterbuch*>` es ist auf jeden Fall `<*gibt 'courtage' ein*>` (`#broker's commission`, `broker's fee#`)
- K1: Ja, (`#broker's commission#`). Ja, die wär cool. [...] Ja, `broker's commission`, so taucht's dann später glaube ich auf (!!M-hm!!) also das passt schon, also das bräuchte ich im Prinzip.
- 1:01:24 (end of Example 7.22)

## C.15 Session KB1

Session KB1 was recorded two months after session KA1. Developers K2 (who is now more experienced in the domain) and K3 (who is more of a database expert) amend their data model: First they introduce a new model class and discuss which fields to include. In the second half they write and debug a database migration to adapt the database schema.

### C.15.1 Transcripts of KB1 Excerpts

0:02:39 (start of Example 8.20)  
K3: Das heißt, wir machen jetzt erstmal nur das Refactoring, oder?  
K2: Würd' ich sagen.  
K3: Ja, ne? Und dann machen wir weiter.  
K2: Wenn wir das <\*hovert Übersicht lokal geänderter Dateien\*> nebenbei mit anfassen, ist jetzt nicht schlimm, das kann ruhig hier mit rein.  
K3: Was ist das?  
K2: Also einmal <\*öffnet eine der geänderten Dateien\*> ist das der Controller, da hab ich jetzt das Datumfeld in zwei Datumfelder, so'n Datum-Zeit-Feld gesplittet.  
K3: Ja, ok.  
0:02:58 (end of Example 8.20)  
[...]  
0:15:54 (start of Example 7.8)  
K3: Oder wollen wir einfach nen Preis ranschreiben?  
K2: Ähm, naja, die Idee war (!...!) dieser Preis da wollen die ja noch viel experimentieren. Deswegen gibt's halt jetzt gerade zwei Konfigurationsvariablen, einmal für LOW und für HIGH, im Preis. Und je nachdem du halt hier für'n (.) Enum hast (.) wird das dann entsprechend rausgezogen. (...) Deswegen wollten wir den Preis nicht direkt schreiben.  
K3: OK. Na gut. Ja man könnte auch einfach den Preis direkt als Zahl reinschreiben, und dann kannst du so viele Varianten haben wie du willst.  
0:16:27 (end of Example 7.8)

## C.16 Sessions KC1 and KC2

Sessions KC1 and KC2 were recorded six months after session KB1. The team switched its technology stack from Java to CoffeeScript.<sup>2</sup> Now, developers K2 and K3 are in the process of getting to know the jQuery JavaScript library because they want to write an integration test of an auto-completion feature, for which they want to programmatically enter characters into an input field. In session KC1, they set up their test environment and discuss different test approaches. After a lunch break, they try out these approaches which does not work out as intended and struggle with their debugger.

### C.16.1 Transcripts of KC2 Excerpts

0:14:12 (start of Example 6.23, part 1)  
K2: <\*öffnet Doku\*> (,,,,,) Das ist Event-Binding, aber ich will's ausführen <\*scrollt runter\*> (,,,,,) Ah, <\*selektiert Text unten auf dem Bildschirm\*> (#keypress#) und dann <\*scrollt selektierten Text in Bildschirmmitte\*> wird das ausgelöst.  
K3: <\*liest Text hinter der Selektion\*> (#without an argument#), okay.

---

<sup>2</sup>Project homepage: <https://coffeescript.org/>

K2: <\*scrollt hoch\*> Aber, wie kann ich jetzt 'n (!...!) <\*bewegt Cursor auf code mit eventData\*>  
 K3: <\*liest Text bei Cursor\*> (#EventData#)  
 K2: Wir müssen dem doch irgendwie sagen, was er machen soll. <\*scrollt runter\*> Vielleicht müssen wir's einfach mal googlen. Google weiß sowas bestimmt.  
 K3: Oder wir gucken einfach bei (~) wie der das macht, im Code.  
 K2: Im Code <\*wechselt zur IDE\*>  
 0:15:04 (end of Example 6.23, part 1)  
 [...]
 0:53:13  
 K2: Jetzt mach ich einfach so nen keypress, oder was (,,,) <\*tippt keypress-Aufruf ohne Parameter\*>  
 K3: Eigentlich müssten wir das ja auch irgendwo sehen, oder, das Ding? Auf der Seite. Oder?  
 K2: (,,) Das können wir machen, indem wir mal 'n (,) (..) Breakpoint machen. Hier machen wir mal 'n keydown (,) <\*dupliziert Zeile\*>. Oder, nee, testen wir's einfach (,) <\*löscht Zeile wieder, wechselt zum Browser, klickt auf Zeile\*> Also eigentlich sollte es hier schon da sein.  
 K3: Ja.  
 K2: <\*lädt die Seite neu\*>  
 K3: Aber was du da jetzt aufgeschrieben hast (!!...!!)  
 K2: Oh, da isses.  
 K3: Naja, immerhin.  
 K2: <\*klickt im Browser herum\*> steht da jetzt (!!...!!) nichts drinne, aber (!...!) was?  
 0:53:54 (start of Example 6.23, part 2)  
 K3: Aber, ja, ich glaube nicht, dass wir damit nen Event auslösen, sondern das ist eher ein Event-Handler, den wir da (.) nicht implementieren.  
 K2: Wenn's leer (!...!) nee wenn's leer ist, das hatten wir doch hier geguckt <\*öffnet Doku\*> (#to trigger the event manually#)  
 K3: Achso, (#without an argument#) (,,) aber (,,,)  
 0:54:24 (end of Example 6.23, part 2)

## C.17 Session MA1

Company M develops software for multiple clients in the energy and logistics sector. In session MA1, developer M2 goes through a number of database tables and asks M1 many questions about their and their columns' purpose. Since M2 had prepared a list of SQL SELECT queries on the tables and columns in question their session is efficient and only lasts 25 minutes.

### C.17.1 Transcripts of MA1 Excerpts

0:05:13 (start of Example 7.4)  
 M1: We have a lookup table, which is this one. If you could open it?  
 M2: Actually, I have it here. <\*highlights and executes next SELECT query in file\*>  
 M1: So here we have, based on the componentName, we have these three columns.  
 M2: OK, got it.  
 0:05:30  
 [M1 explains different details and M2 listens]  
 0:09:43  
 M2: <\*writes comment 'take values from the lookup table'\*>  
 M1: By the way: This is one lookup table, but there will be one or two more, similar like that, but for other websites. Because this lookup table is for one website.  
 M2: Ah <\*slow nodding\*> okay, got it.

0:10:13

[M1 explains more details and M2 listens.]

0:11:01

M2: <\*adds 'Lookup to be done based on componentName and fk\_website'\*>

M1: M-hm.

M2: Ok, cool. Understood.

0:11:16

(end of Example 7.4)

[...]

0:13:26

(start of Example 8.25)

M1: Everything that is not in the Vessel table can be changed along the time.

M2: OK.

M1: But it's more likely that the Machinery and the Equipment won't change, as you know, adding a crane to a ship is not something you do every day.

0:13:41

(end of Example 8.25)

## C.18 Sessions OA1 and OA2

Company O develops a web-based project planning tool. Junior developers O3 and O4 are tasked with writing a test case for some new functionality. Even though they get some help from a colleague along the way, they do not make progress in sessions OA1 and OA2 (separated by a lunch break). There are multiple reasons for this: They neither know that part of the production code nor the underlying technology (React and Redux<sup>3</sup>) nor their development environment so they resort to 'console.log'-debugging for which they have to rebuild the software in three-minute cycles. The pair speaks English throughout the session, which is neither developer's first language (see also Section 4.4.5 for more details on this session).

### C.18.1 Transcripts of OA1 Excerpts

0:59:36

(start of Example 6.17)

O4: I want to propose that we will have a closer look at this TeamSelect which is used in the test and (!..!) (,,,) In the test, we have this function createTeamSelect. And the result of that, we want to make some checks, whether (..) it works.

O3: Yeah, so TeamSelect comes from TeamSelect.coffee

O4: And we use Team (!..!) TeamSelect

O3: Yeah, so it comes from TeamSelect.coffee which is a (..) (React component).

O4: (.....) So it's a, the class TeamSelect inside (..) this file.

O3: Exactly.

O4: TeamSelect is exported.

O3: Exactly. Which is a, which is also a React component.

O4: Yeah, where is it? <\*looks at O3's screen\*>

O3: We just call it (!..!!)

O4: This is the class? <\*points right half of O3's screen\*>

O3: Exactly. So we just call it in the test.

O4: (..) There is a render function, and (!..!) (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,) Yes, in the tests, we define props (,,,) (#createTeamSelect#), these are the props.

---

<sup>3</sup>Project homepages: <https://reactjs.org/> and <https://redux.js.org/>

- O3: M-hm. But (!!And!!) we don't have, we don't want to force any state, we just want to (!...!) So what we want to do, is to, erm (...) so we can force some some props in the state. But we can just (.) call (..) the render function of the React component and check if the props are correct. (...) We don't want (!...!) in here (!...!) I mean, because if here we would add something here that says initial value (!!Yeah!!) because we then would be testing the mock, it doesn't make any sense.
- O4: Yeah, this is why I would say, we should think about whether this makes sense.
- 1:02:51 (start of transcribed part of Example 6.17)
- O3: If we test against the real React component, I think, we can (!...!) so we just have to render the React component, the real one instead of the mocked one and check if the prop is there or not.
- O4: (.....) This initial value, it is here. TeamSelectForm is used TranslatedTeamSelect. TranslatedTeamSelect is in the TeamSelectContainer, which is also exported, but which is not a part of TeamSelect.
- O3: (...) Hm. <\*selects definition of TeamSelectContainer\*> Yeah? (.....) Yeah, but this is the (!!...!!)
- O4: So maybe we have to test TeamSelectContainer.
- O3: (.....) I don't know.
- O4: (...) Or is TeamSelectContainer used somewhere else? <\*turns to his machine\*>
- O3: So, TeamSelectContainer is the wrapper to make it a stateful component, so it connects to Redux. But I don't know if we need to call the (.....) if we need Redux here anyway.
- O4: <\*continues searching (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)\*>
- O3: <\*looks at source code at the same time (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)\*>  
Ah, what about this filterSelectedTeam? <\*selects definition of said method\*>
- O4: <\*looks to O3\*> Sorry?
- O3: What about this filterSelectedTeam? So it gets the formValue <\*hovers code\*>, it's exactly what we want. (#props.fields.teamSelect.value#) <\*open debugger in web browser (,, ,,,)\*> Ah, it's not. <\*goes back to editor\*>
- O4: <\*continues searching (,,,,,,,,,)\*>
- O3: <\*puts cursor in test code\*> But let's try getting the, instead of rendering (...) the TeamSelect, we can try with the TeamSelectContainer.
- O4: I'm just looking if there's already a test (..) for it, but I don't see it.
- O3: <\*searches for occurrences of roleSelect in TeamSelect code (,,,,,,,,,,,,,,,,,,,,,,,,,,,, ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)(.....) <\*selects three lines in TeamSelect code\*> I think it's this guy that we want.\*>
- O4: Sorry?
- O3: <meekly> Sorry to interrupt, erm
- O4: No, it's okay.
- O3: I think it is that what we want, 'cause you were saying that we are looking at the wrong thing. So we were looking at the TeamSelect section, where you define the name and you have this button, and there is this part <\*points to selected lines\*>. So we have the className which is the roleSelect and selectProps with (#props.fields.roleSelect#) and if you go here <\*goes to browser\*> it's this.
- O4: (#roleSelect#)
- O3: <\*back to code\*> I think it's the selectProps. So selectProps <\*to browser\*> select Props. And at some point, we have the initial value that is coming (..) from somewhere.
- O4: But, yeah, I think it's coming from the container.
- O3: Right.
- O4: It's defined in a component which is covering, which wraps this TeamSelect component. (... ..) This is my impression, I'm not sure.
- O3: OK, so let's try to render instead of the TeamSelect, to render the TeamSelectContainer.
- 1:08:53 (end of Example 6.17)

## C.19 Session OA5

Experienced developer O1 and junior developer O3 work on a defect. They amend test cases, refactor the production code, and eventually fix the bug. Along the way, O1 explains general software development principles. The pair speaks English throughout the session, which is neither developer's first language (see also Section 4.4.5 for more background information).

## C.20 Session OA8

Again, see Section 4.4.5 for details on the company, the software, and the team. In the larger context of a bug, junior developers O3 and O4 investigate a test case that began to fail after they started their bugfix. In session OA8, they investigate both production and test code to understand the reason (which is: they changed some implementation but did not adapt the mock objects used in the tests accordingly). They eventually adapt the mock objects and write new test cases to reproduce the bug. For the first 17 minutes they are accompanied by O1. The group speaks English throughout the session, which is neither developer's first language.

### C.20.1 Transcripts of OA8 Excerpts

0:10:35 (start of Example 8.16)  
 O4: `offsetFraction` means in the middle `<hovers offsetFraction: 0.5 in test case*>`  
 O3: The `offsetFraction`, we didn't change that. So, I'm assuming this is okay. But we did change the `offsetDays` (!...!!)  
 O4: I want to know what the meaning of `offsetFraction` is in this test.  
 0:10:49 (end of Example 8.16)  
 [...]  
 0:13:42 (start of Example 6.16)  
 O4: `(#setDragProperties#)` (,,) the `offsetDays` has `<selects part of the console output*>` (, not-a-number. (..) Ok?  
 O3: Because, because it doesn't have any arguments anymore. We changed the function. So, it says that it should be, it was expecting some arguments there and didn't receive any arguments. I would, can you go to the code, to what we changed?  
 O4: `<puzzled>` We just changed the calculation.  
 O3: `<nods*>`  
 O4: Maybe it's not defined, the result is not defined, maybe. (,,,,,) `<moves cursor to test code*>`  
 O1: Not-a-number is the result here of `offsetDays`, so it's not-a-number is the result (!...!!)  
 O4: `<hovers definition of duration mock*>` Maybe, maybe, we have to give the start and end point for the `dataModel`.  
 O1: `<annoyed>` Can you look here? `<points to screen*>` `offsetDay` is NaN, not-a-number `<looks at O4*>` (!Yeah!!) A number is expected and we don't return a number. `<looks at O4*>` It might just be like we missed the brackets and pass a function or something. That might already be (!...!)  
 O4: My idea is that (!...!!)  
 O3: Yeah, I don't know which arguments we are passing right now.  
 O1: So let's have a look at the function `setDragProperties`.  
 0:15:03  
 [They search for and then look at the code under test. O1 leaves the group at 0:17:00.]  
 0:17:54  
 O4: `<hovers changed production code*>` `dataModel.getEnd()` is not defined in the test, I guess.  
 O3: But if it calls this function (!...!)

O4: <\*(.....) switches to test code, hovers mock definition (,,,,,,), (,,,)(I have an idea.) <switches to production code, inserts console.log statement (,,,,,,), >\*>

O3: You can also use, like, the debug (~) for the test.

0:19:22

[They struggle with using the debugger, and eventually return to the code.]

0:25:26

O3: So the way I see it now, this is getting an object, but it was expecting a number.

O4: <\*hovers assertion\*> The expectation is that offsetDays is 1. And offsetDays is not-a-number. This is the value. <\*hovers test code (,,,)\*> And (,,) (#emit#) (,,,,) (#to.have.been.calledWith expectedDragProperties#) (,,,)

O3: <\*points at screen\*> But it's outputting an object. So, an object is not a number.

O4: (~) <\*looks puzzled at O3\*>

O3: That's what I'm understanding of this. (..) Because it was expecting this guy (#to.have.been.calledWith#) with a number. But expectedDragProperties is not a number, it's an object! It's a key-value pair. <\*looks at O4\*>

O4: Erm (...) <\*hovers call of setDragProperties in test code\*> this is the function which (!...!) <\*reads in test code\*> (,,,,,,)

O3: <\*points to screen\*> <slightly helpless> The expectedDragProperties, the value, it's an object, so (!...!)

O4: <\*moves cursor around the screen (,,,,,,)\*>

O3: What are you thinking?

O4: What I first wanted to do was, how I started was this console output <\*hovers console output\*> (#dataModel#), which is like that <\*points at the output\*>. And I made this code at the <\*production class\*> in this new function. (#console.log#). <\*points at screen\*>

O3: The thing is, in the test we don't use the real data model, we stub, we fake one.

O4: Yes, and the problem is, maybe in the fake one, this end and start is not defined. <\*looks at O3\*> So we maybe have to define them, so this function can be used.

O3: Yeah, maybe we have to add it to the stub, yeah.

O4: This was my idea.

O3: M-hm.

0:29:18

[They run the test cases again]

0:31:11

O4: <\*reads the output\*> (#undefined#) ok, but not (..) not not-a-number.

O3: (...) Yeah, because what is not a number is the argument, not the function call.

O4: (...) Erm.

O3: Because the test is, it should have been called with these arguments, and it's the arguments that is not a number.

O4: <\*reads in production code\*> Here we make the output (..) (#durationInDays#) (..) ok, it makes this calculation afterwards. <\*hovers line containing call of Math.floor\*> (..) So, I would like to know what (!...!) let me try please

O3: Sure!

0:32:09

[They add a debugging statement and run the tests.]

0:33:04

O3: (#Not-a-number#)

O4: OK, this is what I thought.

O3: Which is weird, because the Math function should be (!...!) <\*turns to her own machine\*>

O4: In the test, maybe we can (!...!) <\*opens test code\*>

O3: So we are using Math dot? <\*turns to O4\*>



O3: We just need (!...!)

O4: < \*changes mock definition to dataModel.getEnd().returns\*> Is it a moment (..) a moment object? < \*looks at O3\*> It's a Date (..) a Date object.

O3: So it returns, and then it's like, the (~) and dot stub and moment, I think. Can we look, how we stub models? Somewhere else in the code?

O4: I'm about to look whether (!...!) < \*starts search for getEnd, another mock definition is the first match\*> OK

O3: Yeah, < \*points to screen\*> (#returns moment#)

O4: < \*selects two lines of mock code from the first match\*> this is exactly what we need

O3: Yep. < \*leaves the desk\*>

0:39:51

O4: < \*copies the lines, navigates to test case, pastes the mock code, changes the end date to three days after the start date, comments out the duration mock code, runs test case, test succeeds\*>

0:41:42 (end of Example 6.16)

[..]

0:49:10 (start of Example 6.14)

O4: < \*moves cursor to line 108\*> And the offset is?

O3: Erm, offsetDays would be (..) a lot. Because, like the difference between these two guys < \*points to start and end dates\*>, right?

O4: What was the meaning of offsetDays?

O3: It's the distance < \*holds up two index fingers\*>. (..) So, how big is the bar < \*points at screen with two fingers\*>.

O4: No, it's an offset, not a duration or width.

O3: Offset is distance, so it's like distance < \*mimics growing and shrinking distance between her two index fingers\*> (..) so the distance that goes from the beginning of the bar to the end of the (..) bar, I think.

O4: < \*hovers lines 94 and 95 in the previous test case's setup\*> No, it must be three or five days here in this. But it's 1! < \*selects the assertion in line 99\*>

O3: (.....) (Erm, has it to do something with the weekend?)

O4: < \*hovers line 96\*> (#zero point five#) (..) < \*hovers lines 94 and 95\*> three days

O3: But we can check that if we console-log these. < \*looks at O4\*>

O4: Hm?

O3: We can check that, if you console-log, erm, so this, the real variable value, here in the < \*\*production code file\*\*>, we can like console-log the (..)

O4: < \*switches to production code\*>

O3: that object that has the offsetDays (..)

O4: < \*scrolls down\*>

O3: < \*yeah, we can console-log this. < \*looks at O4\*>\*>

O4: (..) In the test, we have to think about what the right value is before we start the test. < \*looks at O3\*>

O3: Right, but like, what's the meaning in the real life of offsetDays? < \*looks at O4\*>

O4: This is what I want, I'm (thinking about).

O3: <nods> So, in order to do that I would say, let's check these values. Let's console-log this.

O4: Ah, you mean on the < \*hovers the calendar view with cursor\*>, when I make a manual test, we log it out.

O3: Yeah, exactly. < \*looks at O4\*>

O4: Ok (..) to have feeling what it, what the meaning of this is.

O3: Exactly.

0:51:19 (end of Example 6.14)

## C.21 Sessions PA1 and PA2

Company P develops and operates a platform for car part retailers and buyers written in PHP. In session PA1, experienced developers P1 and P2 (both with five years of experience) go through a database migration written by P1 and later discuss the requirements that led to the schema change in the first place. They continue after their lunch break with session PA2 where they test and debug the migration and end up refactoring their test cases.

## C.22 Sessions PA3 and PA4

Experienced developers P1 (more backend proficient) and P3 (more frontend) continue the implementation of a new API endpoint which P3 already started. In PA3, P3 shows his existing implementation for which they write tests; P1 explains some backend-related software development best practices. On the next day, in session PA4, they continue with implementing the database access which causes them problems because of some idiosyncrasy of their object-relational (OR) mapper.

### C.22.1 Transcripts of PA3 Excerpts

0:29:53

(start of Example 9.23)

P1: Wichtig ist halt, dass du kenntlich machst, dass nicht die letzten beiden 0.01 miteinander in Beziehung stehen, weil die vielleicht doch nicht in Beziehung stehen, und jemand geht hin und sagt 'So, guck mal hier steht 001' (!!....!!)

P3: Welche letzten beiden?

P1: Zum Beispiel die letzten beiden in der Zeile 31 und 32. Angenommen die würden, die beiden Zahlen würden, nicht in Bezug zueinander stehen, (.) und derjenige, der die Implementierung mit reinen Zahlen sieht, denkt sich 'Oh, die haben doch nen Bezug zueinander, dann geb ich den doch mal die gleiche Konstante'. Und dann kommt jemand hin und führt die noch überall anders ein. (!!Joah!!) Jetzt haben die alle den gleichen Bezug. Jetzt weißt du, dass die explizit tatsächlich umgerechnet werden sollen

P3: So lange sie den gleichen Bezug haben, kann man sie auch so behandeln. In dem Moment wo sich das ändert, muss man's dann anpassen.

P1: Ja, aber das weiß derjenige, der den Code sieht ja nicht, wenn du da reine Zahlen zu stehen hast. Wenn du da nur Zahlen zu stehen hast, die alle den gleichen Wert haben. Was ist denn bei 3660?

P3: Wann haben wir jemals 3660 als Prozentanzahl?

P1: Oder bei 3600. Nee, bei 3600 ist so'n Beispiel. Das ist ne Umrechnung von Stunden und Minuten, es kann aber auch Sekunden und Minuten sein. Also je nach dem welchen Zusammenhang du hast, es können zwei gleiche Zahlen sein, die können aber zwei völlig unterschiedliche Dinge sein.

P3: Aber auf unseren Fall angewendet hat das doch jetzt keine Relevanz.

P1: Doch, hat es. Weil es ne Magic Number ist und Magic Number heißt (!!....!!)

P3: Aber sie ist ja nicht mehr Magic, wir haben sie gerade hier bezeichnet.

P1: <genervt> Ja, wir haben sie so bezeichnet, weil sie jetzt einen Bezug zwischen diesen einzelnen Zahlen herstellt. Vorher war nicht klar (!!....!!)

P3: Ich verstehe nicht, was du jetzt willst, gerade.

P1: Ich wollte dir erklären warum wir das machen (!!....!!)

P3: <genervt> Das habe ich verstanden.

P1: Gut. Ist doch okay.

P3: <nervöses Lachen> Ich hab gerade versucht zu verstehen, was du jetzt noch ändern willst.

P1: Ich wollte nichts ändern. Ich wollte gar nichts ändern.

P3: <erleichtert> Ok.

P1: Ich wollte dir nur klarmachen, dass es wichtig ist (!!...!!)

P3: <genervt> Hab ich verstanden.

P1: mit dieser Umbenennung den Bezug herzustellen.

P3: <genervt, auf den Bildschirm gerichtet> So.

P1: Nicht nur die Variable umzubenennen.

P3: <genervt> Is ok.

0:31:37

(end of Example 9.23)

## C.23 Data Mapping

So far, three other researchers have analyzed pair programming sessions from the same repository as I did: Plonka (2012), Salinger (2013), and Schenk (2018). Here, I provide a mapping of which companies, sessions, and developers they refer to in their writing.

Plonka (2012) does not identify individual sessions or developers, but only the companies. Combining the information from her Tables 4.1 and 4.3, company numbers 1 to 4 appear to correspond to companies C, E, F, and D, respectively. As stated in Example 3.1, I was also able to reconstruct the origin of one of her five transcripts.

Plonka	Global ID
Company 1	Company C
Company 2	Company E
Company 3	Company F
Company 4	Company D
Transcript 4: Expert & Novice	DA5 (22:32–27:04): D2 & D8

**Table C.1:** Mapping of Plonka’s data

Salinger (2013) mentions six industrial PP sessions, but does not uniquely name the developers. There are also two mistakes: In his Table 4.8, session “PR2.2” (=CA1) is listed with a length of 01:57 hours although its actual length is only 01:18 hours; on his page 325, the participants of session “PR2.1” (=CA2) and “PR2.2” (=CA1) are said to be disjoint, but C2 actually takes part in both.

Salinger	Global ID
Session PR1.1: developers P1 & P2	Session BA1: developers B1 & B2
Session PR2.1: developers P1 & P2	Session CA2: developers C5 & C2
Session PR2.2: developers P1 & P2	Session CA1: developers C2 & C1
Session PR2.3: developers P1 & P2	Session CA3: developers C7 & C6
Session PR2.4: developers P1 & P2	Session CA5: developers C4 & C3
Session PR3.1: developers P1 & P2	Session DA2: developers D3 & D4

**Table C.2:** Mapping of Salinger’s data

Schenk (2018, p. 137) analyzed eight recordings: sessions JA2 to JA9. Her developers “Dom” and “Arc” (domain and architect) are developers J2 and J1, respectively.

The current extent and known usages of the PP session repository are maintained in a technical report that is regularly updated:

Franz Zieris & Lutz Prechelt (2020b). *PP-ind: A Repository of Industrial Pair Programming Session Recordings*. arXiv: 2002.03121v3 [cs.SE].

## Appendix D Meta-Analyses

---

As part of my literature review (Chapter 2), I performed a number of meta-analyses of reported pair programming effects. Here, I report the details of these analyses.

### D.1 Technical Information

I use two types of effect sizes in my meta-analyses (Borenstein et al., 2009, Ch. 4). First, I use an unbiased standardized effect size, **Hedges's  $g$** , which is based on the absolute mean difference  $D$  (more precisely,  $g$  is  $D$  divided by the pooled standard deviation corrected by factor for small sample sizes, see Figure 2.4). For ratio scales with a natural zero point, I additionally use the **means' ratio  $R$**  to calculate the **relative mean difference  $D\%$**  (with  $D\% = (R - 1) \cdot 100$ ).

To see the difference in these approaches, consider a hypothetical study with the mean scores of  $\bar{X}_1 = 80$  and  $\bar{X}_2 = 90$  and another with  $\bar{X}_1 = 30$  and  $\bar{X}_2 = 40$ . Both have the same absolute mean difference of  $D = 10$  and, given the same pooled standard deviation, also the same standardized effect size  $g$ . Comparing the studies' ratios, however, makes clear the intuitive difference of their outcomes. The first study has a ratio of  $R = 90/80 = 1.125$  (or a higher score by  $D\% = 12.5\%$ ), while the second has a ratio of  $R = 40/30 \approx 1.333$  (or  $D\% \approx 33.3\%$ ).

Meta-analyses for standardized effect sizes and for mean ratios follow the same procedure, except that mean ratios are (a) converted to a log scale first and (b) lead to summary effects and standard errors that are on a log scale as well and thus need to be exponentiated in the end (ibid., Ch. 4). The resulting confidence intervals are therefore not symmetric.

I use fixed-effect and random-effects models which calculate the summary effect as a weighted mean of the primary studies' effect sizes, with the weights being inverse to the respective study's effect size variance (ibid., Ch. 11 & 12). I report the heterogeneity with the  $I^2$  statistic which estimates the "proportion of the observed variance that reflects real differences in the effect size" (ibid., Ch. 16).

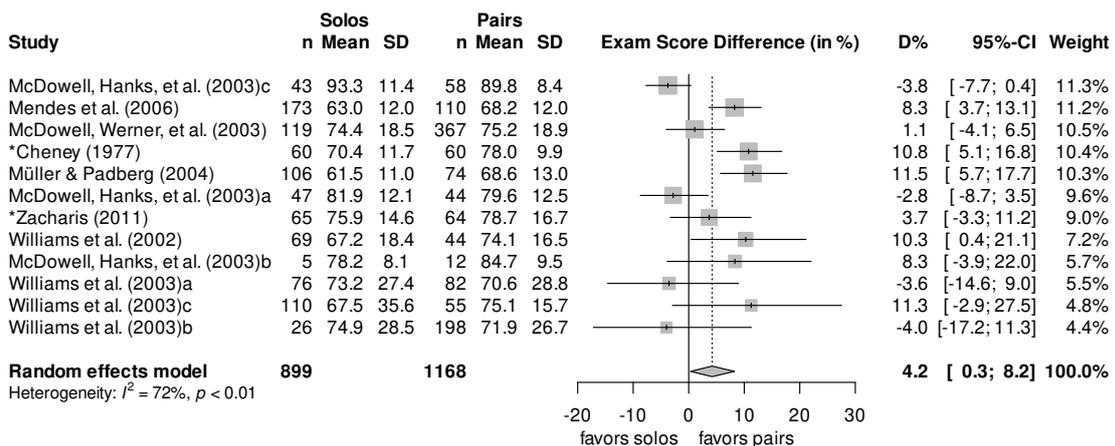
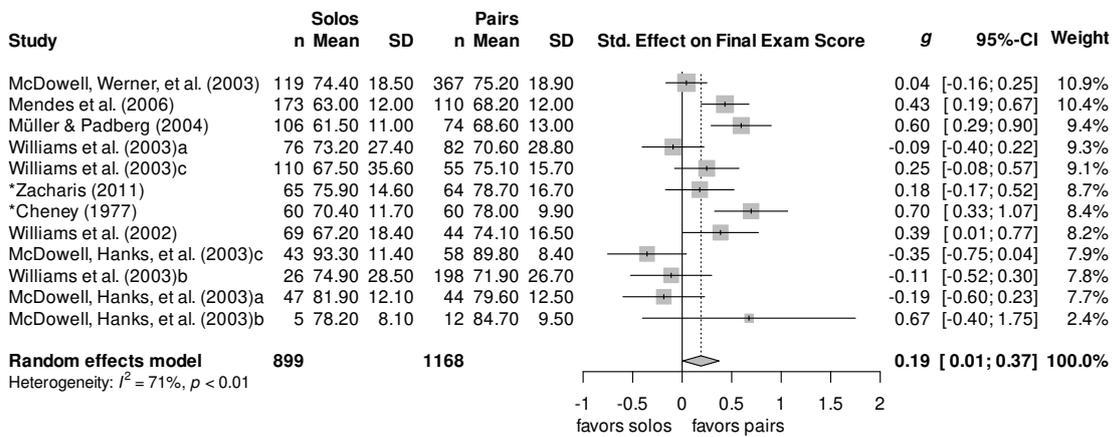
The forest plots below show the primary studies as boxes sized proportional to their weight, positioned at their effect size point estimates, and with whiskers indicating the corresponding confidence intervals (ibid., Ch. 41). The studies are sorted in decreasing order by their weight. The summary effect is shown as a diamond spanning the full confidence interval. All confidence intervals are given for 95%.

## D.2 Pair Programming Effect on Students' Exam Scores

Salleh et al. (2011, Sec. 3.4) performed a fixed-effect meta-analysis of six studies on the effect of students working in pairs or alone throughout the semester on their respective exam scores. Since some primary studies report on more than one experiment, Salleh et al. summarize the effects of a total of 10 experiments. They report a small summary effect size (Hedges's  $g = 0.16$ , 95%-CI: [0.06, 0.26]). I extend their meta-analysis as follows:

- I include the statistical results of two additional educational studies I discuss on page 55: Cheney (1977) and Zacharis (2011).
- Unlike Salleh et al., I use a random-effects model rather than a fixed-effect model because I do not expect all these different studies to point to the same effect.
- In addition to calculating the standardized effect size which is a measure of the means' absolute differences, I also perform a meta-analysis based on the means' ratios.

**Result:** Pair programming during the semester has a small positive effect on students' exam scores, a plus of about 4%.



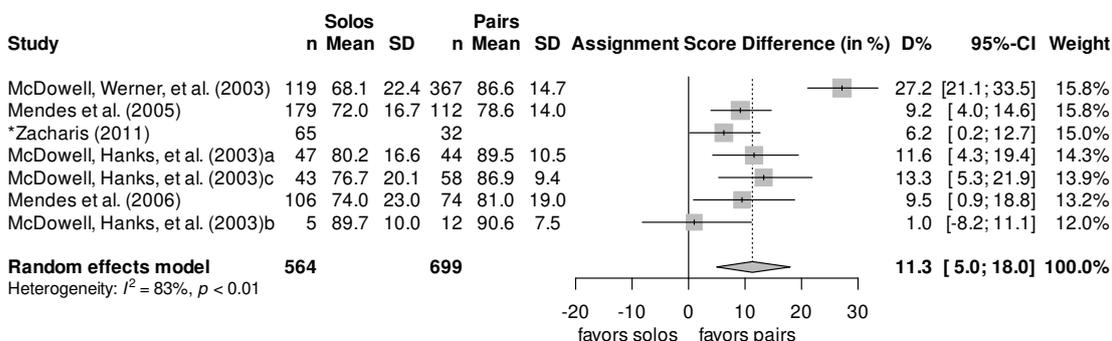
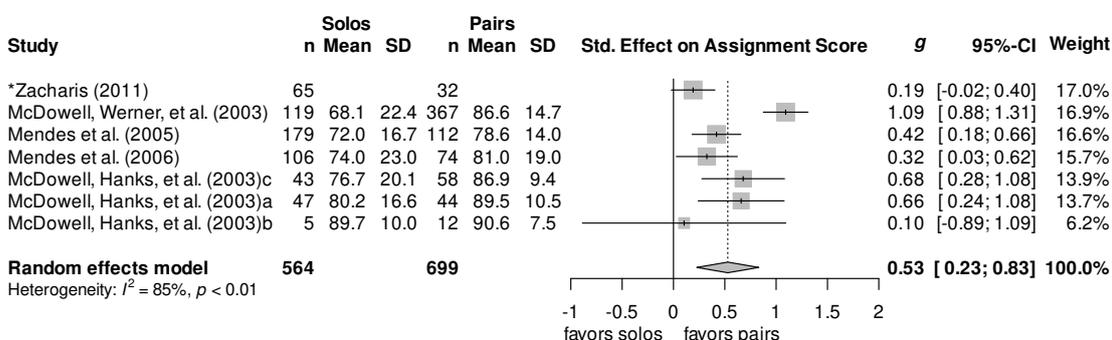
**Figure D.1:** Meta-analysis of PP effects on exam scores. The first forest plot shows the standardized effect sizes (with an overall small positive effect), the second one shows relative differences. Overall, pairers have 4.2% higher exam scores compared to non-pairers. Data for all studies (except those with a \*) comes from Salleh et al. (2011, Fig. 4); see there for full citation. Heterogeneity between the studies is high and significant.

### D.3 Pair Programming Effect on Students' Assignment Scores

Salleh et al. (2011, Sec. 3.4) performed a meta-analysis of four studies (a total of 6 experiments) on the effect of students working in pairs on their assignment scores. They report a medium summary effect size (Hedges's  $g = 0.67$ , 95%-CI: [0.54, 0.80]). I extend their meta-analysis in the following ways:

- I use a random-effects model because I consider the studies to be functionally different.
- I include the statistical results of Zacharis (2011) which I discussed on page 55. He reports the data of different homework assignments on which the same students worked either alone or with a partner. I used a fixed-effect model to calculate a summary effect (see Appendix D.3.1) which I then include in the random-effects analysis of all studies.
- In addition to calculating the standardized effect size, I also perform a meta-analysis based on the means' ratios.

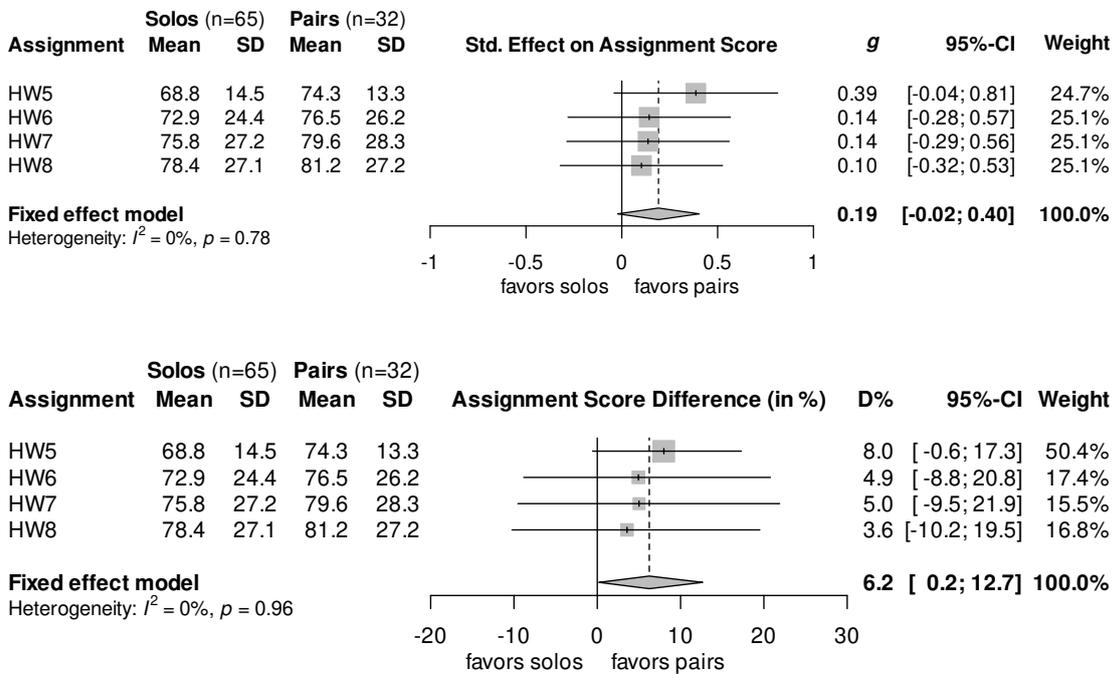
**Result:** The assignment scores of pair-programming students are about 11% higher.



**Figure D.2:** Meta-analysis of PP effects on assignment scores. The first forest plot shows the standardized effect sizes (with an overall medium positive effect), the second one shows relative differences. Overall, pairers have 11.3% higher assignment scores compared to non-pairers. Data for all studies (except those with a \*) comes from Salleh et al. (2011, Fig. 5); see there for full citation. The data for Zacharis (2011) is in itself the result of a meta-analysis (see Appendix D.3.1). Heterogeneity is high and significant.

### D.3.1 Meta-Analysis of Assignment Score Data by Zacharis (2011)

Zacharis (2011, Table V) reports statistical information on four consecutive homework assignments. Since the students worked on these in the same pair/solo constellations, I assume a common underlying effect. I calculated a standardized effect size and a ratio-based effect size for inclusion in the respective random-effects analyses of Figure D.2.



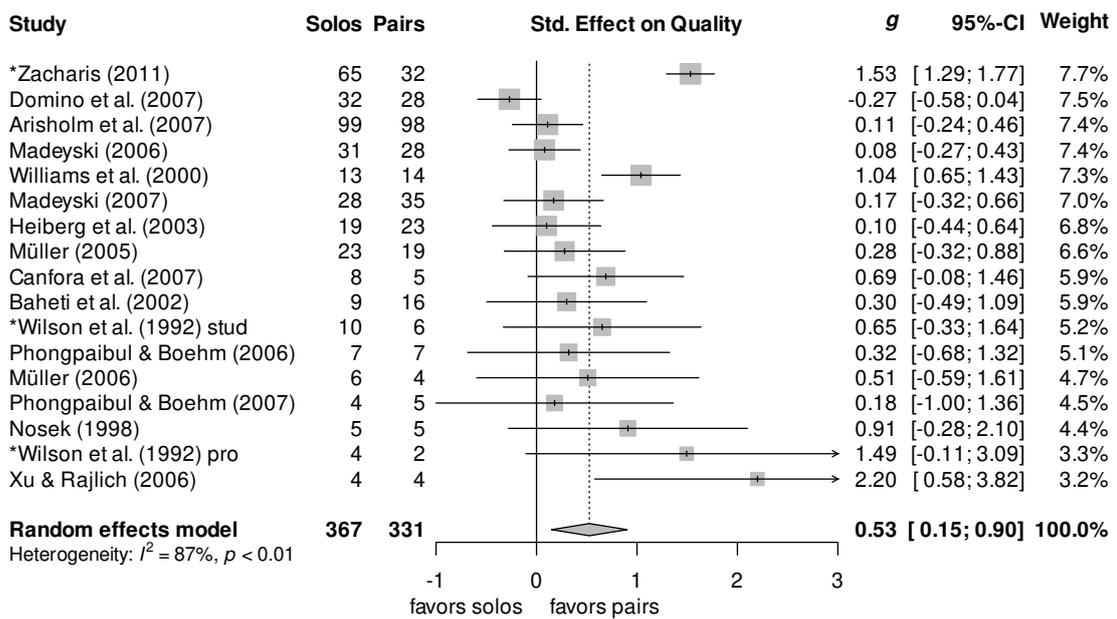
**Figure D.3:** Meta-analysis of PP effects on assignment scores of Zacharis (2011). The first forest plot shows the standardized effect sizes, the second one shows relative differences. Heterogeneity is insignificant which supports the assumption of a fixed-effect model.

## D.4 Pair Programming Effect on Quality

Hannay et al. (2009) performed a meta-analysis of 14 studies on the effect of pair programming on quality. They report a small summary effect size (Hedges's  $g = 0.33$ , 95%-CI: [0.07, 0.60]). I extend their meta-analysis as follows:

- I include the statistical results of Zacharis (2011), who reports the defect density for four consecutive homework assignments. I summarize these partial results with a fixed-effect meta-analysis in Appendix D.4.1 before I include it in the random-effects model.
- I include the statistical results of Wilson et al. (1992), who report on two similar experiments, one with students and one with professionals. I do not assume a common effect and thus include both as individual studies in the random-effects model.

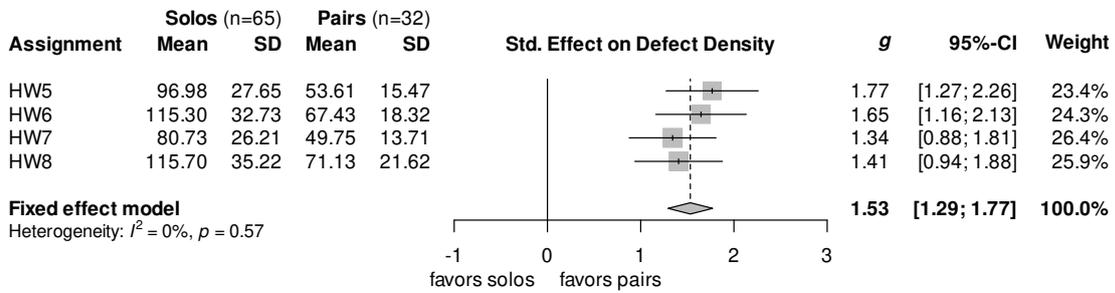
**Results:** Pair programming has a positive effect on quality. The size of that effect is unclear.



**Figure D.4:** Meta-analysis of PP effects on quality. The forest plot shows the standardized effect sizes. Overall, pair programming has a medium positive effect on quality, but the confidence interval is wide. Data for all studies (except those with \*) comes from Hannay et al. (2009, Fig. 1); see there for full citation. The data for Zacharis (2011) is itself the result of a meta-analysis (see Appendix D.4.1). Heterogeneity is high and significant.

**D.4.1 Meta-Analysis of Quality Data of Zacharis (2011)**

Zacharis (2011, Table IV) reports the defect densities of four consecutive homework assignments (see discussion on page 63). Since the same students worked on these in the same pair/solo constellations, I assume a common underlying effect and calculated the standardized effect size for inclusion in Figure D.4.



**Figure D.5:** Meta-analysis of PP effects on quality of Zacharis (2011). The forest plot shows standardized effect sizes. Heterogeneity is insignificant.

# Index

---

*The man who publishes a book without an index ought to be damned 10 miles beyond hell, where the Devil himself cannot get for stinging nettles.*

– John Baynes (1758–1787)

## Symbols

- ★ (*initiative activity*), 190, 191, **192**,  
192–209, 212–218, 220–222, 224,  
227, 228, 234, 235, 285, 291, 350,  
373–376, 395
  - expecting, 191, **192**, 192, 197, 204, 207,  
209, 224, 227, 234, 235
  - non-expecting, 191, **192**, 192, 198, 209,  
224, 235
- ▲ (*pair-referential activity*), 190, 191, **192**,  
192–209, 212–217, 220–222, 224,  
227, 228, 234, 235, 242, 350,  
373–377, 395
- ◀ (*self-referential activity*), 190–192, **193**,  
193, 194, 196–198, 201, 203, 204,  
206–209, 212–216, 218, 220, 221,  
224, 234, 350, 374–376, 395
- (*corrective activity*), 190, 191, **194**,  
194–199, 201, 203–205, 208, 209,  
211, 212, 214–218, 220, 221, 234,  
235, 347, 350, 367, 374–376, 395
- ◆ (*conversational defect*), 190, 191, 194, **196**,  
196–198, 201, 203, 204, 208,  
212–218, 220, 221, 224, 234, 242,  
263, 350, 367, 374–376

## A

- adjacency pairs, 112, 192, 234
- AGSE, 128, 140, 144–146, 151, 369
- Appropriateness, 196
  - appropriate, 191, **196**, 196, 224, 234
  - misled, 191, **196**, 196, 201, 224, 235,  
242, 276, 317, 322

## B

- back channel, 112, 267
- base activities, 87, **132**
- base coding, 167, **169–170**
- base concepts, 87, **132–135**
  - constructive vs. unconstructive verbs,  
**134**, 195
  - initiative vs. reactive verbs, **134**, 190,  
191
- objects
  - \*\_activity, 134
  - \*\_completion, 133
  - \*\_design, 133
  - \*\_finding, 134
  - \*\_gap in knowledge, 134
  - \*\_hypothesis, 134
  - \*\_knowledge, 134
  - \*\_requirement, 133
  - \*\_standard of knowledge, 134
  - \*\_state, 133
  - \*\_step, 133
  - \*\_sth, 133
  - \*\_strategy, 133
  - \*\_todo, 133
- verbs
  - agree\_\*, 135
  - amend\_\*, 135
  - ask\_\*, 135
  - challenge\_\*, 135
  - decide\_\*, 135
  - disagree\_\*, 135
  - explain\_\*, 135
  - propose\_\*, 134

- remember\_\**, 135
- base layer, 87, **130–137**
- Branching Wildly, 28, 186, **304**, 309, 313, 314, 324, 325, 337, 345, 364
- Breakdown, 27, 28, 72, 183, 184, 190, 198, **200**, 200, 201, 204, 210, 211, 214, 218, 222, 223, 225, 226, 229, 234, 235, 243, 270, 316, 332, 335, 336, 359, 361, 362, 364, *see also* Normal Programming and Focus Phase
- C**
- camera angle, 155, 156, 271
- catalyzed, *see* Episode
- Clarification Cascade, 261, **264**, 264, 269, 270, 272, 277, 289, 337, 343, 367
- Co-Production, 28, 185, 230, **280**, 280, 282, 283, 287–289, 291, 293, 294, **295**, 295, 296, 299–301, 304, 306–310, 312, 321, 324, 326–332, 336, 337, 343, 345, 346, 350, 361, 363, 364
- coding
- axial coding, **119–121**, 126, 174–175
  - focused coding, **126**
  - initial coding, **126**
  - open coding, **119**, 126, 174
  - selective coding, **122–123**, 126
  - theoretical coding, 126
- common ground, 78–80, 85, 88, 93, **110–111**, 270, 288, 289, *see also* Refer to Common Ground
- Complementary Gaps, 320, 322, **323**, 323, 327, **329**, 329, 350, 351, 353, 363, 368
- constructivism, 115, 125–126, 166, 360
- conversational defect, *see* Symbol: **◆**
- conversational role, **191**, 191, 234, 347, 350, 359, *see also* Symbols: **★**, **▲**, **▼**, **●**
- conversational turns, 112
- corrective activity, *see* Symbol: **●**
- D**
- D knowledge, 353
- data collection, 144–160
- data segmentation, 119, 129, 131, 132, 174
- Developers
- A1, 142, **160**, 160, 161, 209, 210, 232, 241–243, 254, 273, 274, 277, 284, 293–297, 303, 310, 317, 319, 326, 327, 373–385
  - A2, 142, **160**, 160, 161, 209, 210, 232, 241–243, 273, 277, 293–297, 310, 317, 319, 326, 327, 373–385
  - B1, 137, 161, 174, 248, 249, 295, 312, 317, 322, **385**, 385–387, 426
  - B2, 161, 170, 248, 249, 295, 312, 317, 322, **385**, 385–387, 426
  - C1, 161, 227, 243, 284, 290, 292, 296, 317, 327–329, 331, **387**, 387, 388, 426
  - C2, 161, **162**, 162, 170, 171, 192–198, 223, 225, 227, 228, 236, 242, 243, 245, 246, 248, 250, 251, 262, 266, 271, 274, 283, 284, 286–288, 290, 292, 296, 299, 308, 309, 314, 317, 324–329, 331, 387–393, 426
  - C3, 161, 205–210, 224, 226, 233, 243, 311, 314, 327, 368, 394–398, 426
  - C4, 157, 161, 173, 205–210, 224, 226, 233, 311, 314, 327, 368, **394**, 394–398, 426
  - C5, 18, 161, **162**, 162, 170, 171, 192–198, 223, 225, 227, 228, 242, 245, 246, 248, 250, 251, 262, 266, 271, 274, 275, 283, 286–288, 290, 292, 298, 299, 308, 309, 314, 317, 324–326, 389–393, 426
  - C6, 161, **393**, 393, 426
  - C7, 161, 173, 331, **393**, 393, **394**, 394, 426
  - C8, 157
  - D2, 136, 137, 170, 404–406, 426
  - D3, 161, **163**, 163, 174, 224, 225, 228, 229, 243, 245–247, 249–252, 263, 266, 268, 286, 288, 298, 304–307, 314, 325, 329, 330, 398–404, 426
  - D4, 156, 160, 161, **163**, 163, 224–226, 228, 229, 243, 245–247, 249–252, 263, 264, 266, 268, 270, 284, 286, 288, 297, 298, 303–307, 314, 318, 325, 329–331, 334, 398–404, 426
  - D6, 161, 163, 228, 229
  - D7, 161, 163, 228
  - D8, 136, 137, 170, 404–406, 426
  - E1, 161, 309, 310, 324, **406**, 406–408
  - E2, 161, 309, 310, 324, **406**, 406–408

- J1, 161, **163**, 163, 164, 186, 187, 199, 200, 225, 227, 231, 239, 240, 244, 254, 258–260, 262–265, 267–271, 274–277, 281–283, 285, 287–289, 293, 296–298, 303, 307, 308, 310–314, 316–319, 329–331, 408–414, 426
- J2, 161, **163**, 163, 164, 186, 187, 199, 200, 225, 231, 239, 240, 244, 253, 254, 258–260, 262, 263, 265, 267, 269–271, 274–277, 281–283, 285, 287, 289, 293, 296, 298, 303, 307, 308, 310–314, 316, 317, 319, 329–331, 408–414, 426
- K1, 158, 161, 247, 253, 254, 267, 297, **414**, 414, 415
- K2, 150, 158, 161, 229, 230, 246, 247, 253, 254, 267, 273, 296, 318, 331, 332, 360, **414**, 414, 415, **416**, 416, 417
- K3, 150, 158, 161, 229, 230, 246, 273, 296, 318, 331, 332, 360, **416**, 416, 417
- K4, 158, 161
- M1, 161, 242, 276, 324, 325, 417, 418
- M2, 161, 242, 276, 324, 325, 417, 418
- O1, 161, 210–213, 235, **420**, 420
- O3, 150, 161, **164**, 164, 202–204, 210–222, 225–227, 231, 235, 242, 243, 264, 270, 276, 318, 332, 346, 418–423
- O4, 150, 160, 161, **164**, 164, 202–204, 210–222, 225–227, 242, 243, 264, 270, 276, 318, 332, 346, 418–423
- O6, 161, 243
- P1, 161, 235, 299, 300, 327, 328, 353–355, **424**, 424, 425
- P2, 161, 354, 355, **424**, 424
- P3, 161, 235, 299, 300, 327, 328, 353, 354, **424**, 424, 425
- Direct Asking, 258–261, 264, **265**, 265–267, 269, 270, 273, 274, 277, 285, *see also* Explanation Elicitor
- distributed pair programming, 26, 51, 52, 54–56, 84–85, 144–146, 163–164, 187, 226, 231–232, 265, 285, 293, 408
- DPP, *see* distributed pair programming
- driver and navigator, 41, 72–73, 76, 80, 83–85, 88, 93, 142, 145, 229, 234, 359, 367
- ## E
- emergent research design, **114**, 140, 180, 341–348
- Entice to Simple Step, 258–261, 264, 268, 269, 272, 274–277, 284, 297, 298, 300, *see also* Explanation Elicitor and Explanation
- to failure, 272, 275–277, 298
- to success, 272, 275–277, 298
- Episode, 28, 176, 185, 186, 255, 277, **280**, 280–301, 303–310, 312–314, 321, 324–326, 328, 337, 341–343, 345–347, 350, 353, 354, 357, 361, 364, 367
- Catalyzed Episode, 186, **303**, 303, 304, 306–310, 312–314, 325, 330
- ignored, **283**, **285**, **290**, 291, **292**, **301**
- needs investigation, 283, **285**, 287
- partial success, 283, **285**, 288, 289, 294, 306
- postponed, 283, **285**, 286, 287, 306, 311
- resignation, 283, **285**, 285, 286, 306, 309
- Sub-Episode, 28, 186, 297, 301, **303**, 303–310, 312–314, 324–326, 330, 337, 342, 345
- successful, **283**, **285**, **288**, **289**, **306**, 306, **343**
- unnecessary, 283, **285**, 286, 289, 306
- evaluation, *see* member reflection
- Evaluativeness, **195**
- evaluative, 191, 195, 201, 204, 227, 234, 317
- non-evaluative, 191, 196, 198, 201, 224, 225, 234, 242, 318
- expecting, *see* ★ (*initiative activity*)
- Explanation, 28, 185, 255, 260, 266, 267, 272, 274, 275, 277, 280, 283, 285, 289, 297, 298, 317, 318, 343, 350, 367, *see also* Present New Fact, Refer to Common Ground, and Entice to Simple Step
- Explanation Elicitor, 28, 185, 255, 260, **261**, 261–267, 269, 270, 272, 275–277, 280, 283, 285, 289, 343, 350, 357,

367, *see also* Improper Asking, Direct Asking, Refer to Common Ground, Entice to Simple Step, and Make Proposition  
 frustrating, 261–263  
 ignored, 261–263, 265, 271  
 insufficient, 261–267, 269  
 successful, 261–264, 269  
 explanation trigger, *see* Explanation Elicitor

## F

finding, *see* Improper Asking  
 Fluency, 27, 93, 183, 184, 190, 193, 200, 200, 201, 204, 209, 210, 222, 223, 226, 228, 234, 289, 335, 336, 346, 347, 350, 359, 364, *see also* Togetherness, Normal Pair Programming, Breakdown, and Focus Phase  
 Focus Phase, 27, 183, 184, 190, 198, 200, 200, 201, 204–206, 209, 210, 222, 223, 226, 227, 234, 311, 314, 327, 335, 336, 347, 359, 361, 362, 364, 368, 373, 375, 376, 383–385, 394–396, *see also* Normal Programming and Breakdown  
 frustrating, *see* Explanation Elicitor

## G

G knowledge, 28, 39, 91, 93, 185, 186, 211, 244, 249, 249, 251, 252, 255, 291, 298, 300, 316–337, 342, 346, 347, 350–356, 358, 361, 364, 368  
 G Need, 315–317, 318, 318–323, 328–332, 335–337, 350, 351, 354, *see also* Knowledge Need and S Need  
 G Opportunity, 28, 186, 316, 321, 322, 323, 327–331, 333, 336, 337, 347, 351, 353–355  
 Grice's maxims, 111  
 Maxim of Quantity, 299  
 Maxim of Relation, 299

## H

Hypothetical Target Content, 254, 254, 259, 261, 270, *see also* Target Content

## I

ignored, *see* Explanation Elicitor and Episode  
 Improper Asking, 259–263, 264, 264, 265, 267, 269, 273, 277, 285, *see also* Explanation Elicitor  
 informed consent, 146  
 Initial Constellation, 28, 93, 186, 316, 319, 319–323, 350, 351, 353, 354, 356, 358, *see also* No Relevant Gaps, One-Sided S Gap, Two-Sided S Gap, One-Sided G Gap, Two-Sided G Gap, Complementary Gaps, and Too-Big Two-Fold Gap  
 initiative activity, *see* Symbol:, ★  
 insufficient, *see* Explanation Elicitor  
 interviews vs. observation, 49, 51, 113, 143

## K

knowledge, 39  
 explicit and tacit knowledge, 36  
 in philosophy, 35  
 in software engineering, 36–39  
 in the cognitive sciences, 35–36  
 knowledge silo, 23, 24, 44, 45, 105  
 knowledge transfer, 40  
 knowledge types, 244, *see also* S knowledge and G knowledge  
 meta-knowledge, 134, 172, 260  
 Knowledge Need, 28, 92, 93, 170, 186, 240, 246, 314, 316, 317, 319–321, 323, 329, 335, 336, 343, 345, 347, 350–352, 354–356, 358, 361, 363, 364, *see also* S Need and G Need  
 Knowledge Want, 27, 28, 170, 184–186, 237–239, 240, 240–244, 246, 251, 253–255, 257, 258, 260–265, 267, 269, 272, 276, 277, 280–285, 287–291, 293–297, 299, 300, 303–305, 307, 310–314, 317, 320, 330, 331, 337, 342, 343, 346, 354, 355, 368  
 collective, 185, 240, 240, 243, 254, 280, 282, 283, 288, 295, 296, 300, 343  
 external, 185, 239, 240, 240, 242, 244, 255, 260, 272, 276, 280, 282–284, 288, 297, 299, 300, 304, 314, 317, 331, 343, 346, 354, 368

- internal, 185, 239, **240**, 240, 241, 243, 244, 246, 254, 255, 258, 260, 261, 263–265, 276, 280, 282, 283, 288–291, 293, 295, 299, 300, 303, 304, 310, 314, 317, 343, 346, 354, 368
- L**  
language barrier, 190, 228, 230, 234, 264, 265, 327, 335, 359, 361, 364, *see also* Togetherness
- M**  
magic of source code, 85, 231  
Maintaining Togetherness, 27, 184, 190, 224, 225, 228–230, 232–236, 291, 295–297, 311, 314, 322, 327, 335, 345, 346, 359, 362, 364, 368  
Make Proposition, 254, 259–261, 263–265, **270**, 270, 271, 273, 275, 277, 297, *see also* Explanation Elicitor  
indifferent, 270  
optimistic, 261, 270, 271, 275  
pessimistic, 261, 270, 271, 275  
member reflection, **116**, 147, 150, 153–154, 347, 350–355, 358, 361  
memos, **124**, 175  
code notes, **124**  
operational notes, **124**, 344  
theoretical notes, **124**, 344  
misled, *see* Appropriateness  
mob programming, 143, 158  
Mode, 28, 170, 176, 185, **280**, 280, 282, 283, 287, **288**, 288–290, 297–301, 303, 304, 306–310, 321, 326, 342, 343, 350, 351, 353, 354, 356, 358, 361, 363, 364, 367, *see also* Push, Pull, Pioneering Production, and Co-Production
- N**  
No Relevant Gaps, 320, 322, **323**, 323, 351  
non-evaluative, *see* Evaluativeness  
non-expecting, *see* ★ (*initiative activity*)  
Normal Pair Programming, 183, 190, 198, **200**, 200, 201, 204, 205, 210, 211, 218, 222, 223, 227, 234, 364, *see also* Breakdown and Focus Phase
- O**  
one shared plan, 190, 228–230, 232, 234, 264, 291, 292, 297, 309, 311, 314, 327, 335, 359, 361, 364, *see also* Togetherness  
One-Sided Gap, **319**, 319, 323, 363  
One-Sided G Gap, 320, 322, **323**, 323, 327, 329, 351, 354, *see also* G Opportunity  
One-Sided S Gap, 320–322, **323**, 323, 327, 351, 353, 355, 368, *see also* Primary Gap  
opinions, 46, 49, 51, 78, 92, 173, 233, 272–274
- P**  
P&P concepts, **133**, 137, 141  
pair programming, **143–144**  
as research instrument, 38, 77, 80, 82, 86, 95, 145  
effect on assignment scores, 55–56, 429  
effect on code quality, 63–64, 431  
effect on effort, 63–64  
effect on exam scores, 55–56, 428  
practice vs. work mode, 19, 21, 25, **40**, 42–44, 47, 144, 359  
pair-referential activity, *see* Symbol: ▲  
Parallel Production, 230, **280**, 288, 289, **296**, 297, 301, 345, 346, 361, 377  
Pioneering Production, 28, 185, **280**, 280, 282–284, 286, 287, 289, 290, **291**, 291–296, 299–301, 304–309, 312, 321, 324–330, 332, 336, 337, 343, 345, 350, 359, 363  
Silent Pioneer, **280**, 280, 286, 288, 289, 291–293, 295, 326, 337, 345, 361, 363  
Talking Pioneer, **280**, 280, 284, 286, 288, 289, 291–295, 326, 337, 345, 361, 363  
PP, *see* pair programming  
Present New Fact, 258–260, 266, 272–277, 297, 299, 300, *see also* Explanation  
Primary Gap, 28, 84, 186, 316, **321**, 321–333, 335–337, 345, 347, 351, 353–355  
Production, *see* Pioneering Production and Co-Production

- Propellor, 28, 185, **280**, 280–283, 285,  
288–291, 297–299, 303, 306–309,  
313, 342, 343, 364, 367
- Proposition, *see* Make Proposition
- Pull, 28, 39, 176, 185, **280**, 280–288, **289**,  
289–291, 293, 297–301, 303–310,  
321, 324–330, 332, 336, 337, 342,  
343, 345, 350, 354, 359, 361, 363
- Push, 28, 39, 176, 185, **280**, 280–285,  
288–294, **297**, 297–301, 303–310,  
321, 324–332, 336, 337, 342, 343,  
345, 350, 353, 354, 361, 363, 364
- Q**
- QDA, 174–175
- R**
- Refer to Common Ground, 258–261, 264,  
**266**, 266–268, 272, 274, 276, 277,  
297–300, *see also* Explanation and  
Explanation Elicitor
- reflective interview, 146, **149–150**, 154,  
157, 179, 210, 351, 354–355
- Return Explicitly, 28, 186, 304, **307**,  
307–310, 313, 314, 324, 337, 342,  
345, 362, 364, 367
- rich vs. exhaustive codes, 75, 132
- S**
- S knowledge, 28, 39, 91, 93, 185, 186, 211,  
**244**, 244, 249, 252, 255, 262, 296,  
316–337, 342, 346, 347, 350–356,  
358, 359, 361, 364, 368
- S Need, 315, 316, **317**, 317–326, 329–332,  
334–337, 350, 353, 354, 367, *see*  
*also* Knowledge Need and G Need
- Saros, 84, 163, 187, 226, 408
- Scope Limiting, 28, 186, 304, 307, **310**,  
310–314, 322, 327, 330, 345, 362,  
364, 367
- Secondary Gap, 28, 186, 316, 321, **322**,  
322–324, 326–328, 330–333, 336,  
337, 347, 353–355
- self-referential activity, *see* Symbol: ◀
- session recordings, 147–148, 161
- Sessions
- AA1, 142, 156, 158, 161, **160–162**, 185,  
205, 226, 232, 241, 243, 273, 277,  
293–297, 317, 319, 326, 327, 351,  
**373**, 373, 375–377
- BA1, 155, 158, 161, 170, 248, 317, **385**,  
385, 386, 426
- BB1, 158, 161, 312, 320, 322, 351, **386**,  
386
- BB2, 161, 322, **386**, 386
- BB3, 158, 161, 322, **386**, 386
- CA1, 149, 158, 161, 227, 243, 284, 290,  
292, 296, 317, 325, 327, 328, 330,  
331, 367, **387**, 426
- CA2, 148, 149, 155, 156, 161, **162**, 170,  
171, 175, 192–198, 226, 227, 245,  
248, 250, 262, 266, 271, 274, 283,  
286, 287, 290, 292, 298, 308, 317,  
320, 322, 324, 325, 327, 342, 351,  
**389**, 426
- CA3, 149, 155, 161, 330, 331, **393**, 426
- CA4, 149, 161, 173, 324, **394**
- CA5, 149, 158, 161, 189, 205, 210, 224,  
226, 233, 311, 327, 373, **394**, 426
- CB1, 149, 157, 158
- DA1, 153, 158
- DA2, 155, 156, 158, 161, **163**, 224, 225,  
228, 243, 245, 246, 249–252, 263,  
266, 268, 286, 304–307, 318, 320,  
325, 327, 329–331, 334, 351, **398**,  
426
- DA5, 136, 147, **404**, 404, 426
- DA6, 147
- EA1, 155, 156, 158, 161, 309, 320, 324,  
**406**, 406
- JA1, 158, 161, **163–164**, 175, 177, 186,  
199–201, 226, 231, 239, 253, 254,  
257, 258, 263, 265, 267–271, 274,  
275, 277, 281, 285, 287, 293, 296,  
303, 307, 308, 310, 312, 316, 318,  
319, 324, 327, 329–331, 343, 357,  
**408**, 408, 413
- JA2, 158, 161, 265, 267, 277, 298, 317,  
318, 330, **413**, 426
- KA1, 141, 156, 158, 161, 247, 253, **414**,  
414, 416
- KB1, 158, 161, 246, 273, 329, 360, **416**,  
416
- KC1, 158, 161, 331, 360, **416**, 416
- KC2, 158, 161, 229, 318, 331, 332, 345,  
360, **416**, 416

- MA1, 147, 159, 161, 227, 231, 242, 276, 324, 325, **417**, 417
- OA1, 159, 161, **164**, 189, 198, 218, 219, 222, 226, 227, 231, 243, 318, 331, 332, 346, 351, 352, **418**, 418
- OA2, 158, 159, 161, 222, 231, 243, 318, 331, 332, 346, 418
- OA3, 158
- OA4, 158
- OA5, 159, 161, 227, 231, 235, 327, **420**
- OA6, 158
- OA7, 158
- OA8, 158, 159, 161, 175, 198, 201, 210, 218, 226, 227, 231, 270, 276, 318, 346, **420**, 420
- OA9, 158
- OA10, 158, 352
- PA1, 159, 161, 324, 352–355, **424**, 424
- PA2, 161, 353–355, **424**, 424
- PA3, 161, 235, 299, 327–329, 353, 354, **424**, 424
- PA4, 159, 161, 327, 328, 352, 354, 355, **424**, 424
- shared understanding of software development, 190, 228, 230, 234, 264, 291, 335, 359, 361, 364, *see also* **Togetherness**
- shared understanding of the system, 190, 228, 230, 234, 264, 265, 297, 335, 359, 361, 364, *see also* **Togetherness**
- Simple Step, *see* Entice to Simple Step
- speech acts, **112–113**, 131, 132, 134, 135, 261
- successful, *see* Explanation Elicitor and Episode
- T**
- Target Constellation, 28, 92, 93, 186, 316, **319**, 319, 320, 323, 347, 350, 351, 354
- Target Content, 28, 185, 238, 239, 241–243, **244**, 244, 245, 247–255, 257–259, 261, 262, 264, 269, 270, 280, 282, 283, 285–296, 298, 306, 310, 312, 313, 325, 342, 343, 367, *see also* **Hypothetical Target Content**
- theoretical sampling, 117, **118**, 147, 152, 158–160
- theoretical saturation, **118**, 361
- theoretical sensitivity, **123**, 135, 149, 169
- Timeliness**, **195**
- delayed, 191, 195, 196, 198, 201, 224, 242
- prompt, 191, 195, 196, 201, 204, 209, 224, 228, 234
- Togetherness**, 27, 93, 184, 190, **222**, 222–224, 226, 228–231, 234–236, 243, 261, 262, 264, 265, 267, 269, 270, 277, 282, 289, 291–293, 295, 296, 309, 314, 319, 322, 327, 335–337, 346, 347, 350, 354, 359, 361, 362, 364, 377, 378
- Too-Big Two-Fold Gap**, 320, **323**, 323, 331, 332, 351, 353
- Topic**, 27, 28, 185, 186, 238, 239, 242, 243, **244**, 244–255, 257–259, 261–265, 267, 269, 270, 277, 280–289, 291–296, 301, 303–314, 321, 324, 326, 327, 337, 342, 343, 361, 367
- clarification, 244
- transactive memory system, **98**, 319
- Two-Sided Gap**, **319**, 323, 363
- Two-Sided G Gap**, 322, **331**, 332
- Two-Sided S Gap**, 320, 322, **323**, 323, 326, 327, 351, 355, 368, *see also* **Secondary Gap**
- U**
- universal concepts, **134**, 137, 141
- W**
- workspace awareness, 190, 228, 230, 234, 264, 265, 291, 292, 327, 335, 359, 361, 364, *see also* **Togetherness**



# Name Index

---

## A

Abrahamsson, Pekka, 43, 44, 70, 71, 255  
Adamidis, Panagiotis, 58, 61  
Ahlsweide, Tom, 58  
Altman, Douglas G., 50  
Anastas, Jeane, 140  
Anderson, Ann, 21  
Anderson, John R., 76  
Andres, Cynthia, 22, 40–42, 44  
Angelis, Lefteris, 58, 61  
Argyle, Michael, 198  
Arisholm, Erik, 24, 51, 52, 54, 63–67, 73, 74,  
94, 236, 347, 431  
Armour, Phillip G., 23, 36, 38  
Austin, John L., 112, 113, 192

## B

Babbage, Charles, 20  
Bahrami, Bahador, 94, 95, 103  
Baker, Paul, 254  
Balik, Suzanne, 60  
Baltes, Sebastian, 37, 38  
Barke, Helena, 7  
Beck, Kent, 22, 23, 40–42, 44, 64, 67, 99, 190,  
333, 334, 350, 367  
Beckwith, Laura, 301  
Begel, Andrew, 23, 43–46  
Bellini, Emilio, 55, 61  
Belshee, Arlo, 46, 47, 68, 69, 83, 84, 204, 234  
Bergel, Alexandre, 301  
Bicker, Mary, 130  
Bonar, Jeffrey G., 38  
BonJour, Laurence, 26, 35  
Borenstein, Michael, 50, 58, 427  
Bossche, Piet van den, 97–103  
Boudigou, Françoise, 46  
Boulay, Benedict du, 24, 72, 73, 75, 76, 83,  
131, 184, 234, 342, 359  
Boyer, Kristy Elizabeth, 62  
Brekenfeld, Victor, 7  
Bross, Fabian, 268

Brougham, Jennifer C., 57, 61  
Bryan-Kinns, Nick, 234  
Bryant, Sallyann, 24, 57, 72–76, 83, 87, 131,  
132, 184, 234, 342, 359, *see also*  
Freudenberg, Sallyann  
Bullock, Heather E., 24  
Burnett, Margaret, 301  
Böhm, Andreas, 117, 122

## C

Canfora, Gerardo, 55, 59–61, 92  
Cannon-Bowers, Janis A., 98–103, 234  
Cao, Lan, 58, 77, 78, 85, 86, 234  
Chan, Keith C. C., 66  
Chaparro, Edgar Acosta, 57  
Charmaz, Kathy, 117, 118, 124–128, 152,  
169, 178, 179, 360  
Cheney, Paul H., 21, 55, 61, 428  
Chisholm, Paul, 21  
Chong, Jan, 69, 80, 81, 83–86, 92, 156, 184,  
204, 234, 294, 342  
Cimitile, Aniello, 55, 59, 60, 92  
Cliburn, Daniel C., 58  
Cockburn, Alistair, 24  
Cohen, Cynthia F., 72, 74–76, 234  
Collins, Rosann Webb, 72, 74–76, 234  
Coman, Irina D., 43, 69, 70  
Conboy, Kieran, 46  
Constantine, Larry L., 21  
Coplien, James, 20–22  
Corbin, Juliet, 117–129, 135, 152, 167, 168,  
178, 179, 333, 344  
Crandall, Bill, 43  
Crutcher, Robert J., 77  
Cunningham, Ward, 21, 184, 204, 236  
Curtis, Bill, 36  
Cusumano, Michael, 43

## D

Davenport, Thomas H., 34  
Deeks, Jonathan J., 50

Deligiannis, Ignatios, 58, 61  
DeMarco, Tom, 43  
Dey, Ian, 152  
di Bella, Enrico, 69  
Diehl, Stephan, 37, 38  
Dimsdale, Bernie, 20  
Dittrich, Yvonne, 24, 44, 81, 83, 85–87, 369  
Domino, Madeline Ann, 72, 74–76, 234  
Dybå, Tore, 24, 50–52, 54, 58, 63–67, 73,  
236, 347, 431

**E**

Eapen, Abraham, 46  
Edwards, Richard L., 57  
Ehrhardt, Claus, 110, 111, 172  
Ehrlich, Kate, 37, 38, 333, 334  
Ellece, Sibonile, 254  
Ellis, Judi, 78  
Engvik, Harald, 66, 73, 74, 94  
Eris, Ozgur, 86  
Estácio, Bernardo José da Silva, 51, 52, 54,  
56

**F**

Ferati, Mexhid, 57  
Fernald, Julian, 24  
Fiehler, Reinhard, 79  
Fiore, Stephen M., 98–103  
Fitzgerald, Brian, 25, 46, 49, 125, 126  
Fleming, Scott D., 83–86, 92, 129, 255, 290  
Flick, Uwe, 113, 114, 117, 341  
Flor, Nick V., 78–80, 85, 86, 92, 100, 131,  
235, 301  
Francis, Jo, 20  
Freeman, Susan F., 57, 61  
Freudenberg, Sallyann, 72, 73, 75, *see also*  
Bryant, Sallyann  
Friedell, Morris F., 110  
Frith, Chris D., 94, 95, 103  
Fritz, Thomas, 38, 334  
Fronza, Ilenia, 69, 70  
Fuegi, John, 20  
Furnham, Adrian, 198

**G**

Gabelica, Catherine, 98–103  
Gallis, Hans, 65–67, 73  
Gamma, Erich, 163, 229, 250  
García, Félix, 55, 59–61, 92

Garfinkel, Harold, 110  
Gehringer, Edward, 60  
Gerard, Harold Benjamin, 198  
Gettier, Edmund L., 35  
Gevaert, Hudson, 59, 60  
Gijsselaers, Wim H., 97–103  
Glaser, Barney G., 117, 118, 124–126, 152  
Glaß, Kelvin, 7  
Gleick, James, 20  
Goodwin, Gerald F., 100–103, 234  
Graham, Jean Ann, 198  
Grice, H. Paul, 111, 112  
Grot, Claudia, 46  
Grundy, John C., 51, 52, 54–59, 61, 428, 429  
Gumperz, John J., 371

**H**

Hackman, J. Richard, 99  
Hanks, Brian, 56, 57  
Hannay, Jo E., 24, 50–52, 54, 58, 63–67,  
73–76, 94, 236, 301, 347, 431  
Harms, Thomas, 233  
Harrington, Kieran, 112  
Hartnett, Gerard, 46  
Healey, Patrick G. T., 234  
Hedges, Larry V., 50, 58, 427  
Heffner, Tonia S., 100–103, 234  
Helm, Richard, 163, 229, 250  
Henderson, Austin, 81  
Hendrickson, Chet, 21  
Heringer, Hans Jürgen, 110, 111, 172  
Herlihy, Elizabeth, 130  
Hevner, Alan R., 72, 74–76, 234  
Higgins, Julian P. T., 50, 58, 427  
Hildenbrand, Bruno, 117, 123  
Holt, Richard C., 37, 334  
Hoskin, Nathan, 60, 64, 66, 431  
Howard, Elizabeth V., 57  
Hughes, Janet, 82, 85, 86  
Hulkko, Hanna, 43, 44, 70, 71, 255  
Hurlbutt, Tom, 80, 81, 83–86, 92, 184, 204,  
234, 342  
Hutchins, Edwin, 78–80, 85, 86, 100, 235  
Höst, Martin, 140

**I**

Idris, Sufian, 57

**J**

Jaeger, Beverly K., 57, 61

Janes, Andrea, 57  
 Jeffries, Ron, 21, 184, 204, 236  
 Jensen, Randall W., 21  
 Johnson, Ralph, 163, 229, 250  
 Jones, Danielle L., 83–86, 92, 129, 255, 290  
 Jones, Edward Ellsworth, 198  
 Jordan, Brigitte, 81  
 Jucker, Andreas H., 263  
 Juristo, Natalia, 48

**K**

Kahlert, Björn, 7  
 Kameli, Nader, 46  
 Kampenes, Vigdis By, 50, 58  
 Kardorff, Ernst von, 113, 114, 117, 341  
 Katira, Neha, 59, 60  
 Kelle, Udo, 120  
 Kemerer, Chris F., 43  
 Kerr, Jessica, 334  
 Kessler, Robert R., 21, 40, 41, 44, 57, 142, 184, 204, 236  
 Kirschner, Paul A., 97–103  
 Kissinger, Cory, 301  
 Klausen, Tove, 78  
 Kleb, William L., 46  
 Klemmer, Scott R., 86  
 Ko, Amy J., 38  
 Korpi, Harri, 43, 70, 71, 92, 255  
 Kropp, Edna, 7  
 Kubelka, Juraj, 301

**L**

Lassenius, Casper, 46, 56, 71  
 Latham, Peter E., 94, 95, 103  
 Layman, Lucas, 59, 60  
 Leach, Joe, 234  
 Lee, Roger, 58  
 Leifer, Larry, 86  
 Lethbridge, Timothy C., 49  
 Lewis, Scott, 61, 62  
 Li, Paul Luo, 38  
 Linden, Janet van der, 23, 24, 44, 73–76, 81, 83–87, 156, 234, 369  
 Linnert, Barry, 7  
 Liou, Lin L., 60, 64, 66, 431  
 Lister, Timothy, 43  
 Lofland, John, 113  
 Lofland, Lyn H., 113  
 Lovelace, Ada, 20

Lui, Kim Man, 66

**M**

MacCormack, Alan, 43  
 Madeyski, Lech, 59, 60  
 Marcus, Andrian, 77, 80, 86, 132, 301  
 Martin, David, 80, 85, 86  
 Martin, Robert C., 77  
 Mathieu, John E., 100–103, 234  
 Mcdermott, Wesley, 46  
 McDowell, Charlie, 24  
 Mendes, Emilia, 51, 52, 54–59, 61, 428, 429  
 Miklos, Corey, 46  
 Miller, Carol, 60  
 Misirli, Ayse Tosun, 48  
 Miyake, Naomi, 95–97, 103  
 Mockus, Audris, 37, 38, 334  
 Muhr, Thomas, 124  
 Murphy, Gail C., 38, 80, 82, 86, 92, 255, 334  
 Murphy-Hill, Emerson, 38, 334  
 Mäntylä, Mika V., 51–54, 62  
 Müller, Matthias M., 59, 60

**N**

Nagappan, Nachiappan, 23, 43–46  
 Niessen, Cornelia, 37, 40  
 Nosek, John T., 60, 63, 64, 66, 431

**O**

Ohlsson, Magnus C., 140  
 Okada, Takeshi, 96, 97, 103, 196, 301  
 Olsen, Karsten, 94, 95, 103  
 Osborne, Jason A., 59, 60  
 Ou, Jingwen, 38, 334

**P**

Padberg, Frank, 59, 60  
 Palmieri, David W., 24  
 Pandey, Ajay, 46  
 Patton, Michael Quinn, 109, 113–115, 117, 118, 341, 346  
 Paul, Manoj, 46  
 Paulhus, Delroy L., 354  
 Phaphoom, Nattakarn, 69  
 Piattini, Mario, 55, 59–61, 92  
 Plauger, P.J., 21  
 Plonka, Laura, 7, 23, 24, 27, 44, 71, 73–76, 81, 83–87, 92, 128–131, 135, 143, 145, 147, 148, 150, 151, 153,

- 155–157, 165, 168, 204, 234, 342,  
369, 373, 404, 426
- Plummer, Robert, 86
- Prechelt, Lutz, 7, 27, 84–88, 128–135, 137,  
148, 151, 157, 165, 169, 170, 190,  
191, 193, 195, 196, 204, 230, 234,  
240, 241, 254, 261, 264, 265, 301,  
342, 343, 345–347, 359, 364, 369,  
377
- Price, Kimberly Michelle, 62
- Prikladnicki, Rafael, 51, 52, 54, 56
- Prusak, Laurence, 34
- Przyborski, Aglaja, 109, 117, 120, 122–124,  
126
- Pyhäjärvi, Maaret, 235
- Péraire, Cécile, 23, 44
- R**
- Rajlich, Václav, 57, 61, 77, 80, 86, 92, 132,  
301
- Ralph, Paul, 7, 23, 44, 125, 126, 245
- Ramesh, Balasubramaniam, 58
- Rasmusson, Jonathan, 46, 255
- Reenskaug, Trygve, 21
- Rees, Geraint, 94, 95, 103
- Reeves, Laretta, 35, 36
- Regnell, Björn, 140
- Reichertz, Jo, 114, 123
- Richards, John, 82, 85, 86
- Robbes, Romain, 301
- Robillard, Pierre N., 23, 26, 36–38, 69, 70,  
333, 334
- Robson, Colin, 117, 123
- Rodríguez, Fernando J, 62
- Roepstorff, Andreas, 94, 95, 103
- Rogers, Yvonne, 78
- Romero, Pablo, 24, 57, 72, 73, 75, 76, 83, 131,  
184, 234, 342, 359
- Rooksby, John, 80, 85, 86
- Rosenberg, Doug, 42
- Rosson, Mary Beth, 301
- Rothstein, Hannah R., 50, 58, 427
- Rouncefield, Mark, 80, 85, 86
- Runeson, Per, 140
- Russo, Barbara, 57
- S**
- Sacks, Harvey, 112
- Salas, Eduardo, 98–103, 234
- Salinger, Stephan, 7, 27, 42, 71, 77, 84–88,  
128–135, 137, 141, 143, 145, 147,  
150, 151, 153, 155, 156, 165,  
168–170, 173, 176, 190, 191, 193,  
195, 196, 204, 234, 241, 254, 265,  
342, 345, 359, 364, 369, 371, 373,  
426
- Salleh, Norsaremah, 51, 52, 54–61, 428, 429
- Salman, Iflaah, 48
- Salo, Outi, 43
- Schegloff, Emanuel A., 112
- Schenk, Julia, 7, 26, 27, 84–86, 145, 149, 151,  
153, 156, 231, 369, 373, 426
- Schindler, Christian, 43–45
- Schmeisky, Holger, 7
- Schön, Donald A., 333, 334
- Schütz, Alfred, 108, 109
- Searle, John R., 112, 113, 198
- Sedano, Todd, 23, 44
- Segal, Judith, 73–76, 156, 369
- Segers, Mien, 97–103
- Sfetsos, Panagiotis, 58, 61
- Sharifabdi, Kamran, 46
- Sharp, Helen, 23, 24, 44, 73–76, 81, 83–87,  
156, 234, 369
- Shaw, Marvin E., 97
- Shneiderman, Ben, 37
- Siino, Rosanne, 69, 156, 294
- Sillito, Jonathan, 38, 80, 82, 86, 92, 255
- Sillitti, Alberto, 43, 69, 70
- Sim, Susan Elliott, 37, 49, 334
- Simon, Herbert A., 96, 97, 103, 196, 301, 333
- Singer, Janice, 49
- Sjøberg, Dag I.K., 24, 50–52, 54, 58, 63–67,  
73, 74, 94, 236, 347, 431
- Skaar, Anne Lise, 21
- Slunecko, Thomas, 109
- Sobo, Elisa J., 130
- Soloway, Elliot, 37, 38, 333, 334
- Sommerville, Ian, 26
- Sonntag, Sabine, 37, 40
- StackOverflow, 43
- Stalnaker, Robert C., 110
- Stamelos, Ioannis, 58, 61
- Steiner, Ivan D., 97
- Steinke, Ines, 113, 114, 117, 341
- Stephens, Matt, 42
- Stewart, Jennifer K., 57

- Stol, Klaas-Jan, 25, 49, 125, 126  
 Strasser, Garold, 98  
 Strauss, Anselm, 117–129, 135, 152, 167, 168, 178, 179, 333, 344  
 Stumpf, Simone, 301  
 Subrahmaniyan, Neeraja, 301  
 Subramaniam, Nantha Kumar, 57  
 Succi, Giancarlo, 43, 57, 69, 70  
 Sutedjo, Imelda, 46
- T**  
 Tessem, Bjørnar, 46  
 Thompson, Simon G., 50  
 Toole, Betty Alexandra, 20  
 Toye, George, 86  
 Tracy, Sarah J., 28, 115, 127, 141, 150, 152, 178, 349, 350, 356–358  
 Tschan, Franziska, 98
- V**  
 Van Toll III, Theodore, 58  
 VanDeGrift, Tammy, 57, 61  
 Vanhanen, Jari, 43, 46, 51–54, 56, 62, 70, 71, 92, 255  
 Vaughan, Sandra I., 98  
 Vijay, Vivek, 46  
 Visaggio, Corrado Aaron, 55, 59–61, 92  
 Vlasenko, Jelena, 69, 70  
 Vlissides, John, 163, 229, 250  
 Volder, Kris De, 38, 80, 82, 86, 92, 255  
 Volmer, Judith, 37, 40  
 von Neumann, John, 20
- W**  
 Wake, William C., 75  
 Walle, Thorbjørn, 73–76, 301  
 Webb, Noreen M., 61, 62  
 Weber, Max, 108  
 Wegner, Daniel M., 98  
 Weinberg, Gerald M., 20, 21, 108  
 Weisberg, Robert W., 35, 36  
 Werner, Linda, 24  
 Wesslén, Anders, 140  
 Wiebe, Eric, 60  
 Williams, Kipling D., 94  
 Williams, Laurie, 21, 24, 40, 41, 44, 57, 59, 60, 142, 184, 204, 236  
 Wilson, Judith D., 60, 64, 66, 431  
 Wittenbaum, Gwen M., 98  
 Wohlin, Claes, 140  
 Wohlrab-Sahr, Monika, 117, 120, 122–124, 126  
 Wood, William A., 46
- X**  
 Xu, Peng, 77, 78, 85, 86, 234  
 Xu, Shaochun, 57, 61, 77, 80, 86, 92, 132, 301
- Y**  
 Yang, Sherry, 301  
 Yin, Robert K., 51  
 Yngve, Victor, 112  
 Yuksel, Aybala, 57
- Z**  
 Zacharis, Nick Z., 55–57, 63, 64, 428–432  
 Zarb, Mark, 82, 85, 86  
 Zerbe, Wilfred J., 354  
 Zhou, Minghui, 37, 38, 334  
 Zhu, Jiamin, 38  
 Zieris, Franz, 85, 86, 145, 148, 151, 157, 230, 240, 261, 264, 301, 343, 345–347, 369, 377  
 Zin, Abdullah Mohd, 57  
 Ziv, Yael, 263  
 Zuliani, Paolo, 57



# Bibliography

---

- Jeane Anastas (2000). *Research Design for Social Work and the Human Services*. Columbia University Press. ISBN: 978-0-231-52928-0 (cited on page 140).
- John R. Anderson (1987). “Methodologies for studying human knowledge.” In: *Behavioral and Brain Sciences* 10 (3), pp. 467–477. DOI: [10.1017/s0140525x00023554](https://doi.org/10.1017/s0140525x00023554) (cited on page 76).
- Michael Argyle, Adrian Furnham, & Jean Ann Graham (1981). *Social Situations*. Cambridge University Press. ISBN: 0-521-29881-4 (cited on page 198).
- Erik Arisholm, Hans Gallis, Tore Dybå, & Dag I.K. Sjøberg (2007). “Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise.” In: *IEEE Transactions on Software Engineering* 33 (2), pp. 65–86. DOI: [10.1109/TSE.2007.17](https://doi.org/10.1109/TSE.2007.17) (cited on pages 65–67, 73).
- Phillip G. Armour (2000). “The five orders of ignorance.” In: *Communications of the ACM* 43 (10), pp. 17–20. DOI: [10.1145/352183.352194](https://doi.org/10.1145/352183.352194) (cited on pages 23, 36, 38).
- John L. Austin (1962). *How To Do Things With Words. The William James Lectures delivered at Harvard University in 1955*. Oxford University Press. ISBN: 0-674-41152-8 (cited on pages 112–113, 192).
- Charles Babbage (1889). *Babbage’s Calculating Engines: Being a Collection of Papers Relating to Them; Their History, and Construction*. Ed. by Henry P. Babbage. London: E. & F.N. Spon. ISBN: 1108000967 (cited on page 20).
- Bahador Bahrami, Karsten Olsen, Peter E. Latham, Andreas Roepstorff, Geraint Rees, & Chris D. Frith (2010). “Optimally Interacting Minds.” In: *Science* 329 (5995), pp. 1081–1085. DOI: [10.1126/science.1185718](https://doi.org/10.1126/science.1185718) (cited on pages 94–95, 103).
- Paul Baker & Sibonile Ellece (2011). *Key Terms in Discourse Analysis*. Continuum International Publishing Group. ISBN: 978-1-8470-6320-5 (cited on page 254).
- Sebastian Baltes & Stephan Diehl (2018). “Towards a theory of software development expertise.” In: *Proc. 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ESEC/FSE ’18*. ACM Press, pp. 187–200. DOI: [10.1145/3236024.3236061](https://doi.org/10.1145/3236024.3236061) (cited on pages 37–38).
- Kent Beck (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley. ISBN: 0201616416 (cited on pages 22–23, 40–42, 44, 64, 67, 99, 190, 333–334, 350, 367).
- Kent Beck & Cynthia Andres (2004). *Extreme Programming Explained: Embrace Change*. 2nd ed. Addison Wesley Professional. ISBN: 0-321-27865-8 (cited on pages 22, 40–42, 44).
- Andrew Begel & Nachiappan Nagappan (2007). “Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study.” In: *First Int’l. Symp. on Empirical Software Engineering and Measurement. ESEM ’07*. IEEE, pp. 255–264. DOI: [10.1109/esem.2007.12](https://doi.org/10.1109/esem.2007.12) (cited on page 43).
- Andrew Begel & Nachiappan Nagappan (2008). “Pair Programming: What’s in it for Me?” In: *Proc. Second ACM-IEEE Int’l. Symp. on Empirical Software Engineering and Measurement. ESEM ’08*. ACM, pp. 120–128. DOI: [10.1145/1414004.1414026](https://doi.org/10.1145/1414004.1414026) (cited on pages 23, 44–46).
- Emilio Bellini, Gerardo Canfora, Félix García, Mario Piattini, & Corrado Aaron Visaggio (2005). “Pair designing as practice for enforcing and diffusing design knowledge.” In: *Journal of*

- Software Maintenance and Evolution: Research and Practice* 17 (6), pp. 401–423. DOI: [10.1002/smr.322](https://doi.org/10.1002/smr.322) (cited on pages 55, 61).
- Arlo Belshee (2005). “Promiscuous Pairing and Beginner’s Mind: Embrace Inexperience.” In: *Agile Development Conf. ADC 2005*. IEEE, pp. 125–131. DOI: [10.1109/ADC.2005.37](https://doi.org/10.1109/ADC.2005.37) (cited on pages 46–47, 68–69, 83–84, 204, 234).
- Andreas Böhm (2004). “Theoretical Coding: Text Analysis in Grounded Theory.” In: *A Companion to Qualitative Research*. Ed. by Uwe Flick, Ernst von Kardorff, & Ines Steinke. Sage Publications. Chap. 5.13, pp. 270–275. ISBN: 0-7619-7374-5 (cited on pages 117, 122).
- Michael Bolton (2011). *Jerry Weinberg Interview (from 2008)*. URL: <https://www.developsense.com/blog/2011/01/jerry-weinberg-interview-from-2008/> (visited on 2018-06-17) (cited on page 20).
- Laurence BonJour (2010). *Epistemology. Classic Problems and Contemporary Responses*. 2nd ed. Rowman & Littlefield Publishers, Inc. ISBN: 978-0-7425-6418-3 (cited on pages 26, 35).
- Michael Borenstein, Larry V. Hedges, Julian P. T. Higgins, & Hannah R. Rothstein (2009). *Introduction to Meta-Analysis*. Wiley John + Sons. ISBN: 978-0-470-05724-7 (cited on pages 50, 58, 427).
- Piet van den Bossche, Wim H. Gijssels, Mien Segers, & Paul A. Kirschner (2006). “Social and Cognitive Factors Driving Teamwork in Collaborative Learning Environments: Team Learning Beliefs and Behaviors.” In: *Small Group Research* 37 (5), pp. 490–521. DOI: [10.1177/1046496406292938](https://doi.org/10.1177/1046496406292938) (cited on pages 97–103).
- Fabian Bross (2012). “German modal particles and the common ground.” In: *Helikon. A Multidisciplinary Online Journal* 2, pp. 182–209. URL: [http://helikon-online.de/2012/Bross\\_Particles.pdf](http://helikon-online.de/2012/Bross_Particles.pdf) (visited on 2019-10-02) (cited on page 268).
- Nick Bryan-Kinns, Patrick G. T. Healey, & Joe Leach (2007). “Exploring mutual engagement in creative collaborations.” In: *Proc. 6th ACM SIGCHI Conf. on Creativity & Cognition*. C&C ’07. ACM, pp. 223–232. DOI: [10.1145/1254960.1254991](https://doi.org/10.1145/1254960.1254991) (cited on page 234).
- Sallyann Bryant (2004). “Double Trouble: Mixing Qualitative and Quantitative Methods in the Study of eXtreme Programmers.” In: *IEEE Symp. on Visual Languages and Human Centric Computing*. VL/HCC ’04. IEEE, pp. 55–61. DOI: [10.1109/VLHCC.2004.20](https://doi.org/10.1109/VLHCC.2004.20) (cited on pages 72–76, 87, 131–132).
- Sallyann Bryant, Pablo Romero, & Benedict du Boulay (2006). “The Collaborative Nature of Pair Programming.” In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Pekka Abrahamsson, Michele Marchesi, & Giancarlo Succi. Vol. 4044. Lecture Notes in Computer Science. Springer, pp. 53–64. DOI: [10.1007/11774129\\_6](https://doi.org/10.1007/11774129_6) (cited on pages 72, 74–76, 83).
- Sallyann Bryant, Pablo Romero, & Benedict du Boulay (2008). “Pair Programming and the Mysterious Role of the Navigator.” In: *International Journal of Human-Computer Studies* 66 (7), pp. 519–529. DOI: [10.1016/j.ijhcs.2007.03.005](https://doi.org/10.1016/j.ijhcs.2007.03.005) (cited on pages 24, 72–76, 83, 131, 184, 234, 342, 359).
- Gerardo Canfora, Aniello Cimitile, & Corrado Aaron Visaggio (2004). “Working in pairs as a means for design knowledge building: an empirical study.” In: *Proc. 12th IEEE Int’l. Workshop on Program Comprehension*. IWPC ’04. IEEE, pp. 62–68. DOI: [10.1109/WPC.2004.1311048](https://doi.org/10.1109/WPC.2004.1311048) (cited on page 55).
- Gerardo Canfora, Aniello Cimitile, Félix García, Mario Piattini, & Corrado Aaron Visaggio (2005). “Confirming the influence of educational background in pair-design knowledge through experiments.” In: *Proc. 2005 ACM Symp. on Applied Computing*. SAC ’05. ACM Press, pp. 1478–1484. DOI: [10.1145/1066677.1067013](https://doi.org/10.1145/1066677.1067013) (cited on pages 55, 59–60, 92).
- Janis A. Cannon-Bowers & Eduardo Salas (2001). “Reflections on Shared Cognition.” In: *Journal of Organizational Behavior* 22 (2), pp. 195–202. DOI: [10.1002/job.82](https://doi.org/10.1002/job.82) (cited on pages 98–99).

- Lan Cao & Balasubramaniam Ramesh (2004). “An exploratory study on the effects of pair programming.” In: *Proc. 8th Int’l. Conf. on Empirical Assessment in Software Engineering*. EASE ’04. IET, pp. 21–28. DOI: [10.1049/ic:20040395](https://doi.org/10.1049/ic:20040395) (cited on page 58).
- Lan Cao & Peng Xu (2005). “Activity Patterns of Pair Programming.” In: *Proc. 38th Annual Hawaii Int’l. Conf. on System Sciences*. HICSS ’05. IEEE. DOI: [10.1109/HICSS.2005.66](https://doi.org/10.1109/HICSS.2005.66) (cited on pages 77–78, 85–86, 234).
- Edgar Acosta Chaparro, Aybala Yuksel, Pablo Romero, & Sallyann Bryant (2005). “Factors Affecting the Perceived Effectiveness of Pair Programming in Higher Education.” In: *Proc. 17th Workshop of the Psychology of Programming Interest Group*. PPIG ’05, pp. 5–18. URL: <https://ppig.org/files/2005-PPIG-17th-chaparro.pdf> (cited on page 57).
- Kathy Charmaz (2006). *Constructing Grounded Theory. A Practical Guide Through Qualitative Analysis*. SAGE Publications. ISBN: 978-0-7619-7352-2 (cited on pages 117–118, 124–128, 152, 169, 178–179, 360).
- Paul H. Cheney (1977). “Teaching Computer Programming in an Environment Where Collaboration Is Required.” In: *AEDS Journal (Association for Educational Data Systems)* 11 (1), pp. 1–5 (cited on pages 21, 55, 61, 428).
- Jan Chong, Robert Plummer, Larry Leifer, Scott R. Klemmer, Ozgur Eris, & George Toye (2005). “Pair Programming: When and Why it Works.” In: *Proc. 17th Workshop of the Psychology of Programming Interest Group*. PPIG ’05, pp. 43–48. URL: <https://ppig.org/files/2005-PPIG-17th-chong.pdf> (cited on page 86).
- Jan Chong & Rosanne Siino (2006). “Interruptions on Software Teams: A Comparison of Paired and Solo Programmers.” In: *Proc. 20th Anniversary Conf. on Computer Supported Cooperative Work*. CSCW ’06. ACM, pp. 29–38. DOI: [10.1145/1180875.1180882](https://doi.org/10.1145/1180875.1180882) (cited on pages 69, 80, 156, 294).
- Jan Chong & Tom Hurlbutt (2007). “The Social Dynamics of Pair Programming.” In: *Proc. 29th Int’l. Conf. on Software Engineering*. ICSE ’07. IEEE, pp. 354–363. DOI: [10.1109/ICSE.2007.87](https://doi.org/10.1109/ICSE.2007.87) (cited on pages 80–81, 83–86, 92, 184, 204, 234, 342).
- Daniel C. Cliburn (2003). “Experiences with pair programming at a small college.” In: *Journal of Computing Sciences in Colleges* 19 (1), pp. 20–29. URL: <https://dl.acm.org/doi/10.5555/948737.948741> (cited on page 58).
- Alistair Cockburn & Laurie Williams (2001). “The Costs and Benefits of Pair Programming.” In: *Extreme Programming Examined*. Ed. by Giancarlo Succi & Michele Marchesi. Addison-Wesley, pp. 223–243. ISBN: 0-201-71040-4 (cited on page 24). Repr. of “The Costs and Benefits of Pair Programming.” In: *Proc. eXtreme Programming and Flexible Processes in Software Engineering – XP 2000*. Cagliari, Sardinia, Italy, 2000.
- Irina D. Coman, Alberto Sillitti, & Giancarlo Succi (2008). “Investigating the Usefulness of Pair-Programming in a Mature Agile Team.” In: *Agile Processes in Software Engineering and Extreme Programming*. Vol. 9. Lecture Notes in Business Information Processing. Springer, pp. 127–136. DOI: [10.1007/978-3-540-68255-4\\_13](https://doi.org/10.1007/978-3-540-68255-4_13) (cited on pages 43, 69–70).
- Irina D. Coman, Pierre N. Robillard, Alberto Sillitti, & Giancarlo Succi (2014). “Cooperation, Collaboration and Pair-Programming: Field Studies on Backup Behavior.” In: *Journal of Systems and Software* 91, pp. 124–134. DOI: [10.1016/j.jss.2013.12.037](https://doi.org/10.1016/j.jss.2013.12.037) (cited on pages 69–70).
- Larry L. Constantine (1995). *Constantine on Peopleware*. Yourdon Press. ISBN: 0-13-331976-8 (cited on page 21).
- Larry L. Constantine (2011). *Comment on blog post “Improving Code Quality with Pair Programming”*. URL: <https://www.benlinders.com/2011/improving-code-quality-with-pair-programming/#comment-467> (visited on 2019-10-02) (cited on page 21).

- James Coplien (1998). "A Generative Development-Process Pattern Language." In: *The Patterns Handbook: Techniques, Strategies, and Applications*. Ed. by Linda Rising. Cambridge University Press, pp. 243–300. ISBN: 0-521-64818-1 (cited on page 22).
- James Coplien (2015). *Two Heads are Better than One*. Computing Now, IEEE Computer Society; archived at <https://web.archive.org/web/20170711190543/https://computingnow.computer.org/web/agile-careers/content?g=8504655&type=article&urlTitle=two-heads-are-better-than-one> (cited on pages 20–21).
- Juliet Corbin & Anselm Strauss (1990). "Grounded Theory Research: Procedures, Canons, and Evaluative Criteria." In: *Qualitative Sociology* 13 (1), pp. 3–21. DOI: [10.1007/BF00988593](https://doi.org/10.1007/BF00988593) (cited on page 121).
- Robert J Crutcher (1994). "Telling What We Know: The Use of Verbal Report Methodologies in Psychological Research." In: *Psychological Science* 5 (5), pp. 241–244. DOI: [10.1111/j.1467-9280.1994.tb00619.x](https://doi.org/10.1111/j.1467-9280.1994.tb00619.x) (cited on page 77).
- Bill Curtis (1984). "Fifteen Years of Psychology in Software Engineering: Individual Differences and Cognitive Science." In: *Proc. 7th Int'l. Conf. on Software Engineering*. ICSE '84. IEEE Press, pp. 97–106. URL: <https://dl.acm.org/doi/10.5555/800054.801956> (cited on page 36).
- Michael Cusumano, Alan MacCormack, Chris F. Kemerer, & Bill Crandall (2003). "Software development worldwide: The state of the practice." In: *IEEE Software* 20 (6), pp. 28–34. DOI: [10.1109/ms.2003.1241363](https://doi.org/10.1109/ms.2003.1241363) (cited on page 43).
- Thomas H. Davenport & Laurence Prusak (2000). *Working Knowledge: How Organizations Manage What They Know*. Harvard Business School Press. ISBN: 978-1578513017 (cited on page 34).
- Tom DeMarco & Timothy Lister (2013). *Peopleware: Productive Projects and Teams*. 3rd ed. Addison Wesley. ISBN: 978-0-321-93411-6 (cited on page 43).
- Ian Dey (1999). *Grounding Grounded Theory: Guidelines for Qualitative Inquiry*. Emerald Group Publishing Limited. ISBN: 978-0122146404 (cited on page 152).
- Enrico di Bella, Ilenia Fronza, Nattakarn Phaphoom, Alberto Sillitti, Giancarlo Succi, & Jelena Vlasenko (2013). "Pair Programming and Software Defects – A Large, Industrial Case Study." In: *IEEE Transactions on Software Engineering* 39 (7), pp. 930–953. DOI: [10.1109/TSE.2012.68](https://doi.org/10.1109/TSE.2012.68) (cited on page 69).
- Madeline Ann Domino, Rosann Webb Collins, Alan R. Hevner, & Cynthia F. Cohen (2003). "Conflict in Collaborative Software Development." In: *Proc. 2003 SIGMIS Conf. on Computer Personnel Research*. SIGMIS CPR '03. ACM, pp. 44–51. DOI: [10.1145/761849.761856](https://doi.org/10.1145/761849.761856) (cited on pages 72, 74–76, 234).
- Richard L. Edwards, Jennifer K. Stewart, & Mexhid Ferati (2010). "Assessing the effectiveness of distributed pair programming for an online informatics curriculum." In: *ACM Inroads* 1 (1), pp. 48–54. DOI: [10.1145/1721933.1721951](https://doi.org/10.1145/1721933.1721951) (cited on page 57).
- Claus Ehrhardt & Hans Jürgen Heringer (2011). *Pragmatik*. (lit. "Pragmatics"). Paderborn: Wilhelm Fink. ISBN: 978-3825234805 (cited on pages 110–111, 172).
- Bernardo José da Silva Estácio & Rafael Prikladnicki (2015). "Distributed Pair Programming: A Systematic Literature Review." In: *Information and Software Technology* 63, pp. 1–10. DOI: [10.1016/j.infsof.2015.02.011](https://doi.org/10.1016/j.infsof.2015.02.011) (cited on pages 51–52, 54, 56).
- Reinhard Fiehler (2005). "Gesprochene Sprache." In: *Duden*. Vol. 4: *Die Grammatik*. Ed. by Dudenredaktion. 7th ed. (lit. "Speech"). Bibliographisches Institut & FA Brockhaus AG, pp. 1175–1256. ISBN: 978-3-411-04047-6 (cited on page 79).

- Brian Fitzgerald, Gerard Hartnett, & Kieran Conboy (2006). "Customising agile methods to software practices at Intel Shannon." In: *European Journal of Information Systems* 15 (2), pp. 200–213. DOI: [10.1057/palgrave.ejis.3000605](https://doi.org/10.1057/palgrave.ejis.3000605) (cited on page 46).
- Uwe Flick, Ernst von Kardorff, & Ines Steinke, eds. (2004). *A Companion to Qualitative Research*. SAGE Publications. ISBN: 0-7619-7374-5 (cited on pages 113–114, 117, 341).
- Nick V. Flor & Edwin Hutchins (1991). "Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance." In: *Empirical studies of programmers: Fourth workshop*. Ablex Publishing Corp., pp. 36–64 (cited on pages 78–80, 85–86, 100, 235).
- Nick V. Flor (1998). "Side-by-side collaboration: a case study." In: *International Journal of Human-Computer Studies* 49, pp. 201–222. DOI: [10.1006/ijhc.1998.0203](https://doi.org/10.1006/ijhc.1998.0203) (cited on pages 78–80, 85–86, 92, 131, 301).
- Susan F. Freeman, Beverly K. Jaeger, & Jennifer C. Brougham (2003). "Pair Programming: More Learning And Less Anxiety In A First Programming Course." In: *Proc. ASEE Annual Conf.* Pp. 8.912.1–8.912.9. URL: <https://peer.asee.org/11728> (cited on pages 57, 61).
- Sallyann Freudenberg (2006). "The 'Tag Team': Tools, tasks and roles in collaborative software development." PhD thesis. University of Sussex. URL: <https://salfreudenberg.files.wordpress.com/2017/03/finalthesis.pdf> (cited on pages 72, 75).
- Sallyann Freudenberg, Pablo Romero, & Benedict du Boulay (2007). "'Talking the talk': Is intermediate-level conversation the key to the pair programming success story?" In: *Proc. AGILE 2007 Conf.* IEEE, pp. 84–91. DOI: [10.1109/AGILE.2007.1](https://doi.org/10.1109/AGILE.2007.1) (cited on page 73).
- Morris F. Friedell (1969). "On the structure of shared awareness." In: *Behavioral Science* 14 (1), pp. 28–39. ISSN: 1099-1743. DOI: [10.1002/bs.3830140105](https://doi.org/10.1002/bs.3830140105) (cited on page 110).
- Thomas Fritz, Jingwen Ou, Gail C. Murphy, & Emerson Murphy-Hill (2010). "A Degree-of-Knowledge Model to Capture Source Code Familiarity." In: *Proc. 32nd Int'l. Conf. on Software Engineering*. ICSE '10. New York, NY, USA: ACM, pp. 385–394. DOI: [10.1145/1806799.1806856](https://doi.org/10.1145/1806799.1806856) (cited on pages 38, 334).
- Ilenia Fronza, Alberto Sillitti, & Giancarlo Succi (2009). "An Interpretation of the Results of the Analysis of Pair Programming during Novices Integration in a Team." In: *Proceedings of the Third Int'l. Symp. on Empirical Software Engineering and Measurement*. ESEM '09. IEEE, pp. 225–235. DOI: [10.1109/ESEM.2009.5315998](https://doi.org/10.1109/ESEM.2009.5315998) (cited on pages 69–70).
- John Fuegi & Jo Francis (2003). "Lovelace & Babbage and the Creation of the 1843 'Notes.'" In: *IEEE Annals of the History of Computing* 25 (4), pp. 16–26. DOI: [10.1109/MAHC.2003.1253887](https://doi.org/10.1109/MAHC.2003.1253887) (cited on page 20).
- Catherine Gabelica, Piet van den Bossche, Stephen M. Fiore, Mien Segers, & Wim H. Gijsselaers (2016). "Establishing team knowledge coordination from a learning perspective." In: *Human Performance* 29 (1), pp. 33–53. DOI: [10.1080/08959285.2015.1120304](https://doi.org/10.1080/08959285.2015.1120304) (cited on pages 98–103).
- Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN: 0-201-63361-2 (cited on pages 163, 229, 250).
- Harold Garfinkel (1967). *Studies in Ethnomethodology*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc. ISBN: 0-7456-0005-0 (cited on page 110).
- Markus Gärtner (2016). *Interview with Jerry Weinberg*. URL: <http://www.shino.de/2016/01/10/interview-with-jerry-weinberg/> (visited on 2018-06-17) (cited on page 20).
- Edmund L. Gettier (1963). "Is Justified True Belief Knowledge?" In: *Analysis* 23 (6), pp. 121–123. DOI: [10.2307/3326922](https://doi.org/10.2307/3326922) (cited on page 35).

- Hudson Gevaert (2006). "Pair Programming Unearthed." MA thesis. University of Manitoba. URL: <http://hdl.handle.net/1993/20460> (cited on pages 59–60).
- Barney G. Glaser & Anselm Strauss (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. AldineTransaction. ISBN: 0-202-30260-1 (cited on pages 117–118, 124–125, 152).
- Barney G. Glaser (1978). *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory*. The Sociology Press. ISBN: 978-1884156014 (cited on page 126).
- Barney G. Glaser (2007). "Remodeling Grounded Theory." In: *Historical Social Research, Supplement* 19, pp. 47–68. URL: <https://nbn-resolving.org/urn:nbn:de:0168-ssoar-288341> (cited on page 118).
- James Gleick (2011). *The Information: A History, A Theory, A Flood*. Pantheon Books. ISBN: 978-0-307-37957-3 (cited on page 20).
- H. Paul Grice (1975). "Logic and Conversation." In: *Syntax and Semantics. 3: Speech Acts*. Ed. by Peter Cole & Jerry L. Morgan. New York: Academic Press, pp. 41–58. ISBN: 978-0127854236 (cited on pages 111–112).
- John J. Gumperz (1982). *Discourse Strategies*. Studies in Interactional Sociolinguistics 1. Cambridge University Press. ISBN: 0-521-24691-1 (cited on page 371).
- J. Richard Hackman, ed. (1990). *Groups that work (and those that don't): Creating Conditions for Effective Teamwork*. Jossey-Bass. ISBN: 1-55542-187-3 (cited on page 99).
- Brian Hanks (2006). "Student attitudes toward pair programming." In: *Proc. 11th Annual SIGCSE Conf. on Innovation and Technology in Computer Science Education*. ITiCSE. ACM, pp. 113–117. DOI: [10.1145/1140124.1140156](https://doi.org/10.1145/1140124.1140156) (cited on page 57).
- Brian Hanks (2008). "Empirical evaluation of distributed pair programming." In: *International Journal of Human-Computer Studies* 66 (7), pp. 530–544. DOI: [10.1016/j.ijhcs.2007.10.003](https://doi.org/10.1016/j.ijhcs.2007.10.003) (cited on page 56).
- Jo E. Hannay, Tore Dybå, Erik Arisholm, & Dag I.K. Sjøberg (2009). "The effectiveness of pair programming: A meta-analysis." In: *Information and Software Technology* 51 (7), pp. 1110–1122. DOI: [10.1016/j.infsof.2009.02.001](https://doi.org/10.1016/j.infsof.2009.02.001) (cited on pages 24, 51–52, 54, 63–67, 236, 347, 431).
- Jo E. Hannay, Erik Arisholm, Harald Engvik, & Dag I.K. Sjøberg (2010). "Effects of Personality on Pair Programming." In: *IEEE Transactions on Software Engineering* 36 (1), pp. 61–80. DOI: [10.1109/TSE.2009.41](https://doi.org/10.1109/TSE.2009.41) (cited on pages 66, 73–74, 94).
- Thomas Harms (2017). "Untersuchung von Fokus-Phasen in Paar-Programmierungssitzungen." (lit. "Investigation of Focus Phases in Pair Programming Sessions"). BA thesis. Freie Universität Berlin. URL: <https://www.inf.fu-berlin.de/inst/ag-se/theses/Harms17-pp-fokus.pdf> (cited on page 233).
- Kieran Harrington (2018). *The Role of Corpus Linguistics in the Ethnography of a Closed Community: Survival Communication*. Routledge. ISBN: 978-1-138-71442-7 (cited on page 112).
- Julian P. T. Higgins, Simon G. Thompson, Jonathan J. Deeks, & Douglas G. Altman (2003). "Measuring inconsistency in meta-analyses." In: *BMJ* 327 (7414), pp. 557–560. DOI: [10.1136/bmj.327.7414.557](https://doi.org/10.1136/bmj.327.7414.557) (cited on page 50).
- Bruno Hildenbrand (2004). "Anselm Strauss." In: *A Companion to Qualitative Research*. Ed. by Uwe Flick, Ernst von Kardorff, & Ines Steinke. Sage Publications. Chap. 2.1, pp. 17–23. ISBN: 0-7619-7374-5 (cited on pages 117, 123).
- Elizabeth V. Howard (2006). "Attitudes on Using Pair-Programming." In: *Journal of Educational Technology Systems* 35 (1), pp. 89–103. DOI: [10.2190/5k87-58w8-g07m-2811](https://doi.org/10.2190/5k87-58w8-g07m-2811) (cited on page 57).

- Hanna **Hulkko** & Pekka Abrahamsson (2005). "A Multiple Case Study on the Impact of Pair Programming on Product Quality." In: *Proc. 27th Int'l. Conf. on Software Engineering*. ICSE '05. ACM, pp. 495–504. DOI: [10.1145/1062455.1062545](https://doi.org/10.1145/1062455.1062545) (cited on pages 43–44, 70–71, 255).
- Edwin **Hutchins** (1989). "The Technology of Team Navigation." In: *Intellectual Teamwork: Social and Technical Bases of Collaborative Work*. Ed. by Jolene Galegher, Robert E. Kraut, & Carmen Egido. Lawrence Erlbaum Associates. Chap. 8, pp. 191–220. ISBN: 0-8058-0534-6 (cited on page 78).
- Edwin **Hutchins** & Tove Klausen (1998). "Distributed cognition in an airline cockpit." In: *Cognition and Communication at Work*. Ed. by Yrjö Engeström & David Middleton. Cambridge University Press. Chap. 2, pp. 15–34. ISBN: 0-521-64566-2 (cited on page 78).
- Andrea **Janes**, Barbara Russo, Paolo Zuliani, & Giancarlo Succi (2003). "An Empirical Analysis on the Discontinuous Use of Pair Programming." In: *Proc. 4th. Intl. Conf. Extreme Programming and Agile Processes in Software Engineering*. XP '03. Springer, pp. 205–214. DOI: [10.1007/3-540-44870-5\\_26](https://doi.org/10.1007/3-540-44870-5_26) (cited on page 57).
- Ron **Jeffries**, Ann Anderson, & Chet Hendrickson (2001). *Extreme Programming Installed*. Addison-Wesley. ISBN: 978-0201708424 (cited on page 21).
- Randall W. **Jensen** (2003). "A Pair Programming Experience." In: *Crosstalk. The Journal of Defense Software Engineering* 16 (3), pp. 22–24. URL: <https://web.archive.org/web/20041027082334/http://www.stsc.hill.af.mil/crosstalk/2003/03/jensen.html> (cited on page 21).
- Danielle L. **Jones** & Scott D. Fleming (2013). "What use is a backseat driver? A qualitative investigation of pair programming." In: *Proc. IEEE Symp. on Visual Languages and Human Centric Computing*. IEEE, pp. 103–110. DOI: [10.1109/VLHCC.2013.6645252](https://doi.org/10.1109/VLHCC.2013.6645252) (cited on pages 83–86, 92, 129, 255, 290).
- Edward Ellsworth **Jones** & Harold Benjamin Gerard (1967). *Foundations of social psychology*. Oxford, England: Wiley. ISBN: 978-0471449065 (cited on page 198).
- Brigitte **Jordan** & Austin Henderson (1995). "Interaction Analysis: Foundations and Practice." In: *The Journal of the Learning Sciences* 4 (1), pp. 39–103. DOI: [10.1207/s15327809jls0401\\_2](https://doi.org/10.1207/s15327809jls0401_2) (cited on page 81).
- Andreas H. **Jucker** & Yael Ziv, eds. (1998). *Discourse Markers. Descriptions and Theory*. John Benjamins Publishing Company. ISBN: 9027250715 (cited on page 263).
- Vigdis By **Kampenes**, Tore Dybå, Jo E. Hannay, & Dag I.K. Sjøberg (2007). "A systematic review of effect size in software engineering experiments." In: *Information and Software Technology* 49 (11–12), pp. 1073–1086. DOI: [10.1016/j.infsof.2007.02.015](https://doi.org/10.1016/j.infsof.2007.02.015) (cited on pages 50, 58).
- Neha **Katira**, Laurie Williams, Eric Wiebe, Carol Miller, Suzanne Balik, & Edward Gehringer (2004). "On Understanding Compatibility of Student Pair Programmers." In: *Proc. 35th SIGCSE Technical Symp. on Computer Science Education*. SIGCSE '04. ACM, pp. 7–11. DOI: [10.1145/971300.971307](https://doi.org/10.1145/971300.971307) (cited on page 60).
- Neha **Katira**, Laurie Williams, & Jason A. Osborne (2005). "Towards increasing the compatibility of student pair programmers." In: *Proc. 27th Int'l. Conf. on Software Engineering*. ICSE '05. ACM, pp. 625–626. DOI: [10.1145/1062455.1062572](https://doi.org/10.1145/1062455.1062572) (cited on page 60).
- Udo **Kelle** (2007). "'Emergence' vs. 'Forcing' of Empirical Data? A Crucial Problem of 'Grounded Theory' Reconsidered." In: *Grounded Theory Reader*. Ed. by Günter May & Katja Mruck. Köln: Zentrum für Historische Sozialforschung, pp. 133–156 (cited on page 120).
- Jessica **Kerr** (2017). *Hyperproductive development*. URL: <https://jessitron.com/2017/06/24/the-most-productive-circumstances-for> (visited on 2017-08-28) (cited on page 334).

- Cory **Kissinger**, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, & Mary Beth Rosson (2006). "Supporting End-User Debugging: What Do Users Want to Know?" In: *Proc. Working Conf. on Advanced Visual Interfaces*. AVI '06. ACM Press, pp. 135–142. DOI: [10.1145/1133265.1133293](https://doi.org/10.1145/1133265.1133293) (cited on page 301).
- Juraj **Kubelka**, Romain Robbes, & Alexandre Bergel (2018). "The Road to Live Programming: Insights From the Practice." In: *Proc. 40th Int'l. Conf. on Software Engineering*. ICSE '18. ACM, pp. 1090–1101. DOI: [10.1145/3180155.3180200](https://doi.org/10.1145/3180155.3180200) (cited on page 301).
- Timothy C. **Lethbridge**, Susan Elliott Sim, & Janice Singer (2005). "Studying Software Engineers: Data Collection Techniques for Software Field Studies." In: *Empirical Software Engineering* 10 (3), pp. 311–341. DOI: [10.1007/s10664-005-1290-x](https://doi.org/10.1007/s10664-005-1290-x) (cited on page 49).
- Paul Luo **Li**, Amy J. Ko, & Jiamin Zhu (2015). "What Makes a Great Software Engineer?" In: *Proc. 37th Int'l. Conf. on Software Engineering*. ICSE '15. IEEE, pp. 700–710. DOI: [10.1109/icse.2015.335](https://doi.org/10.1109/icse.2015.335) (cited on page 38).
- John **Lofland** & Lyn H. Lofland (1995). *Analyzing Social Settings. A Guide to Qualitative Observation and Analysis*. 3rd ed. Wadsworth Publishing Company. ISBN: 9780534247805 (cited on page 113).
- Kim Man **Lui** & Keith C. C. Chan (2003). "When Does a Pair Outperform Two Individuals?" In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Michele Marchesi & Giancarlo Succi. Vol. 2675. Lecture Notes in Computer Science. Springer, pp. 225–233. DOI: [10.1007/3-540-44870-5\\_28](https://doi.org/10.1007/3-540-44870-5_28) (cited on page 66).
- Kim Man **Lui** & Keith C. C. Chan (2004). "A Cognitive Model for Solo Programming and Pair Programming." In: *Proc. Third IEEE Int'l. Conf. on Cognitive Informatics*. ICCI '04. IEEE, pp. 94–102. DOI: [10.1109/COGINF.2004.1327463](https://doi.org/10.1109/COGINF.2004.1327463) (cited on page 66).
- Kim Man **Lui** & Keith C. C. Chan (2006). "Pair programming productivity: Novice–novice vs. expert–expert." In: *International Journal of Human-Computer Studies* 64 (9), pp. 915–925. DOI: [10.1016/j.ijhcs.2006.04.010](https://doi.org/10.1016/j.ijhcs.2006.04.010) (cited on page 66).
- Lech **Madeyski** (2006). "Is External Code Quality Correlated with Programming Experience or Feelgood Factor?" In: *Extreme Programming and Agile Processes in Software Engineering. XP 2006*. Springer Berlin Heidelberg, pp. 65–74. DOI: [10.1007/11774129\\_7](https://doi.org/10.1007/11774129_7) (cited on pages 59–60).
- Robert C. **Martin** (2002). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall. ISBN: 978-0-13-597444-5 (cited on page 77).
- John E. **Mathieu**, Tonia S. Heffner, Gerald F. Goodwin, Eduardo Salas, & Janis A. Cannon-Bowers (2000). "The Influence of Shared Mental Models on Team Process and Performance." In: *J. of Applied Psychology* 85 (2), pp. 273–283. DOI: [10.1037/0021-9010.85.2.273](https://doi.org/10.1037/0021-9010.85.2.273) (cited on pages 100–103, 234).
- Charlie **McDowell**, Linda Werner, Heather E. Bullock, & Julian Fernald (2003). "The Impact of Pair Programming on Student Performance, Perception, and Persistence." In: *Proc. 25th Int'l. Conf. on Software Engineering*. ICSE '03. IEEE, pp. 602–607. DOI: [10.1109/ICSE.2003.1201243](https://doi.org/10.1109/ICSE.2003.1201243) (cited on page 24).
- Naomi **Miyake** (1986). "Constructive Interaction and the Iterative Process of Understanding." In: *Cognitive Science* 10 (2), pp. 151–177. DOI: [10.1207/s15516709cog1002\\_2](https://doi.org/10.1207/s15516709cog1002_2) (cited on pages 95–97, 103).
- Thomas **Muhr** (1994). "ATLAS/ti: ein Werkzeug für die Textinterpretation." In: *Texte verstehen: Konzepte, Methoden, Werkzeuge*. Ed. by Andreas Boehm, Andreas Mengel, & Thomas Muhr. Vol. 14. (lit. "ATLAS/ti: A tool for text interpretation"). UVK Univ.-Verl. Konstanz, pp. 317–324. ISBN: 3-87940-503-4. URL: <https://nbn-resolving.org/urn:nbn:de:0168-ssoar-14628> (cited on page 124).

- Matthias M. Müller & Frank Padberg (2004). “An Empirical Study about the Feelgood Factor in Pair Programming.” In: *Proc. 10th IEEE Int’l. Software Metrics Symp.* METRICS ’04. IEEE, pp. 151–158. DOI: [10.1109/METRIC.2004.1357899](https://doi.org/10.1109/METRIC.2004.1357899) (cited on pages 59–60).
- John T. Nosek (1998). “The case for collaborative programming.” In: *Communications of the ACM* 41 (3), pp. 105–108. DOI: [10.1145/272287.272333](https://doi.org/10.1145/272287.272333) (cited on page 63).
- Takeshi Okada & Herbert A. Simon (1997). “Collaborative Discovery in a Scientific Domain.” In: *Cognitive Science* 21 (2), pp. 109–146. DOI: [10.1207/s15516709cog2102\\_1](https://doi.org/10.1207/s15516709cog2102_1) (cited on pages 96–97, 103, 196, 301).
- David W. Palmieri (2002). “Knowledge Management Through Pair Programming.” MA thesis. North Carolina State University. URL: <http://www.lib.ncsu.edu/resolver/1840.16/1429> (cited on page 24).
- Ajay Pandey, Nader Kameli, Abraham Eapen, Corey Miklos, Francoise Boudigou, Imelda Sutedjo, Manoj Paul, Vivek Vijay, & Wesley Mcdermott (2003). “Application of Tightly Coupled Engineering Team for Development of Test Automation Software – A Real World Experience.” In: *Proc. 27th Annual Int’l. Computer Software and Applications Conf.* COMPAC ’03. IEEE, pp. 56–63. DOI: [10.1109/cmpsac.2003.1245322](https://doi.org/10.1109/cmpsac.2003.1245322) (cited on page 46).
- Michael Quinn Patton (2002). *Qualitative Research and Evaluation Methods*. 3rd ed. Sage Publications. ISBN: 0761919716 (cited on pages 109, 113–115, 117–118, 341, 346).
- Nattakarn Phaphoom, Alberto Sillitti, & Giancarlo Succi (2011). “Pair Programming and Software Defects – An Industrial Case Study.” In: *Agile Processes in Software Engineering and Extreme Programming. XP 2011*. Ed. by Alberto Sillitti, Orit Hazzan, Emily Bache, & Xavier Albaladejo. Vol. 77. Lecture Notes in Business Information Processing. Springer, pp. 208–222. DOI: [10.1007/978-3-642-20677-1\\_15](https://doi.org/10.1007/978-3-642-20677-1_15) (cited on page 69).
- Laura Plonka (2009). “Bericht Workshop Paar Programmierung bei [Firma C].” reprint as ancillary file of Zieris & Prechelt (2020b). (lit. “Report Pair Programming Workshop at [Company C]”). URL: <https://arxiv.org/src/2002.03121v3/anc/workshop-report-company-c.pdf> (cited on pages 145, 148, 153, 155, 157).
- Laura Plonka, Judith Segal, Helen Sharp, & Janet van der Linden (2011). “Collaboration in Pair Programming: Driving and Switching.” In: *Agile Processes in Software Engineering and Extreme Programming. XP 2011*. Ed. by Alberto Sillitti, Orit Hazzan, Emily Bache, & Xavier Albaladejo. Vol. 77. Lecture Notes in Business Information Processing. Springer, pp. 43–59. DOI: [10.1007/978-3-642-20677-1\\_4](https://doi.org/10.1007/978-3-642-20677-1_4) (cited on pages 73–76, 86, 156, 369).
- Laura Plonka, Helen Sharp, & Janet van der Linden (2012a). “Disengagement in pair programming: Does it matter?” In: *Proc. 34th Int’l. Conf. on Software Engineering.* ICSE ’12. IEEE, pp. 496–506. DOI: [10.1109/ICSE.2012.6227166](https://doi.org/10.1109/ICSE.2012.6227166) (cited on pages 23, 73, 84–86, 234, 369).
- Laura Plonka, Judith Segal, Helen Sharp, & Janet van der Linden (2012b). “Investigating Equity of Participation in Pair Programming.” In: *Proc. Agile India 2012*. IEEE. DOI: [10.1109/AgileIndia.2012.16](https://doi.org/10.1109/AgileIndia.2012.16) (cited on page 369).
- Laura Plonka (2012). “Unpacking Collaboration in Pair Programming in Industrial Settings.” PhD thesis. Open University. URL: <https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.577973> (cited on pages 27, 71, 81, 92, 128, 130–131, 135, 143, 145, 147–148, 150–151, 153, 156–157, 168, 369, 373, 404, 426).
- Laura Plonka, Helen Sharp, Janet van der Linden, & Yvonne Dittrich (2015). “Knowledge transfer in pair programming: An in-depth analysis.” In: *International Journal of Human-Computer Studies* 73, pp. 66–78. DOI: [10.1016/j.ijhcs.2014.09.001](https://doi.org/10.1016/j.ijhcs.2014.09.001) (cited on pages 24, 44, 81, 83, 85–87, 369).

- Lutz **Prechelt**, Franz Zieris, & Holger Schmeisky (2015). “Difficulty Factors of Obtaining Access for Empirical Studies in Industry.” In: *2015 IEEE/ACM 3rd Int’l. Workshop on Conducting Empirical Studies in Industry*. CESI ’15. IEEE, pp. 19–25. DOI: [10.1109/cesi.2015.11](https://doi.org/10.1109/cesi.2015.11) (cited on page 367).
- Lutz **Prechelt**, Holger Schmeisky, & Franz Zieris (2016). “Quality Experience: A Grounded Theory of Successful Agile Projects without Dedicated Testers.” In: *Proc. 38th Int’l. Conf. on Software Engineering*. ICSE ’16. ACM, pp. 1017–1027. DOI: [10.1145/2884781.2884789](https://doi.org/10.1145/2884781.2884789) (cited on page 368).
- Aglaja **Przyborski** & Thomas Slunecko (2009). “Against Reification! Praxeological Methodology and its Benefits.” In: *Dynamic Process Methodology in the Social and Developmental Sciences*. Ed. by Jaan Valsiner, Peter C.M. Molenaar, Maria C.D.P. Lyra, & Nandita Chaudhary. Chap. 7, pp. 141–170. DOI: [10.1007/978-0-387-95922-1\\_7](https://doi.org/10.1007/978-0-387-95922-1_7) (cited on page 109).
- Aglaja **Przyborski** & Monika Wohlrab-Sahr (2014). *Qualitative Sozialforschung. Ein Arbeitsbuch*. 4th ed. (lit. “Qualitative Social Research. A Textbook”). Oldenbourg Verlag. ISBN: 978-3-486-70892-9 (cited on pages 117, 120, 122–124, 126).
- Maaret **Pyhäjärvi** (2018). *Power dynamics in pairs and mobs*. URL: <https://visible-quality.blogspot.com/2018/05/power-dynamics-in-pairs-and-mobs.html> (visited on 2018-09-13) (cited on page 235).
- Paul **Ralph** (2013). “The illusion of requirements in software development.” In: *Requirements Engineering* 18 (3), pp. 293–296. DOI: [10.1007/s00766-012-0161-4](https://doi.org/10.1007/s00766-012-0161-4) (cited on page 245).
- Jonathan **Rasmusson** (2003). “Introducing XP into Greenfield Projects: Lessons Learned.” In: *IEEE Software* 20 (3), pp. 21–28. DOI: [10.1109/ms.2003.1196316](https://doi.org/10.1109/ms.2003.1196316) (cited on pages 46, 255).
- Trygve **Reenskaug** & Anne Lise Skaar (1989). “An environment for literate Smalltalk programming.” In: *Conf. Proc. on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA’89. ACM Press, pp. 337–345. DOI: [10.1145/74877.74912](https://doi.org/10.1145/74877.74912) (cited on page 21).
- Jo **Reichertz** (2004). “Abduction, Deduction and Induction in Qualitative Research.” In: *A Companion to Qualitative Research*. Ed. by Uwe Flick, Ernst von Kardorff, & Ines Steinke. Sage Publications. Chap. 4.3, pp. 159–164. ISBN: 0-7619-7374-5 (cited on pages 114, 123).
- Pierre N. **Robillard** (1999). “The Role of Knowledge in Software Development.” In: *Communications of the ACM* 42 (1), pp. 87–92. DOI: [10.1145/291469.291476](https://doi.org/10.1145/291469.291476) (cited on pages 23, 26, 36–38, 333–334).
- Colin **Robson** (2002). *Real World Research*. 2nd ed. Blackwell Publishing. ISBN: 978-0-631-21305-5 (cited on pages 117, 123).
- Fernando J **Rodríguez**, Kimberly Michelle Price, & Kristy Elizabeth Boyer (2017). “Exploring the Pair Programming Process: Characteristics of Effective Collaboration.” In: *Proc. 2017 ACM SIGCSE Technical Symp. on Computer Science Education*. SIGCSE ’17. ACM, pp. 507–512. DOI: [10.1145/3017680.3017748](https://doi.org/10.1145/3017680.3017748) (cited on page 62).
- Yvonne **Rogers** & Judi Ellis (1994). “Distributed Cognition: An Alternative Framework for Analysing and Explaining Collaborative Work.” In: *Journal of Information Technology* 9 (2), pp. 119–128. DOI: [10.1057/jit.1994.12](https://doi.org/10.1057/jit.1994.12) (cited on page 78).
- John **Rooksby**, David Martin, & Mark Rouncefield (2006). “Reading as Part of Computer Programming. An Ethnomethodological Enquiry.” In: *Proc. 18th Workshop of the Psychology of Programming Interest Group*. PPIG ’06, pp. 198–212. URL: <https://ppig.org/files/2006-PPIG-18th-rooksby.pdf> (cited on pages 80, 85–86).
- Stephan **Salinger**, Laura Plonka, & Lutz Prechelt (2008). “A Coding Scheme Development Methodology using Grounded Theory for Qualitative Analysis of Pair Programming.” In:

- Human Technology: An Interdisciplinary Journal on Humans in ICT Environments* 4 (1), pp. 9–25. DOI: [10.17011/ht/urn.200804151350](https://doi.org/10.17011/ht/urn.200804151350) (cited on pages 87, 128–130, 165, 204, 234, 342, 369).
- Stephan Salinger (2013). “Ein Rahmenwerk für die qualitative Analyse der Paarprogrammierung.” (lit. “A framework for the qualitative analysis of pair programming”). PhD thesis. Freie Universität Berlin. DOI: [10.17169/refubium-14050](https://doi.org/10.17169/refubium-14050) (cited on pages 27, 42, 71, 77, 87–88, 128–132, 137, 141, 143, 145, 147, 150–151, 153, 155–156, 165, 168–169, 173, 176, 369, 371, 373, 426).
- Stephan Salinger, Franz Zieris, & Lutz Prechelt (2013). “Liberating Pair Programming Research from the Oppressive Driver/Observer Regime.” In: *Proc. 35th Int’l. Conf. on Software Engineering. ICSE ’13 (NIER)*. IEEE, pp. 1201–1204. DOI: [10.1109/ICSE.2013.6606678](https://doi.org/10.1109/ICSE.2013.6606678) (cited on pages 83, 85–86, 345, 367, 369).
- Stephan Salinger & Lutz Prechelt (2013). *Understanding Pair Programming: The Base Layer*. Norderstedt: Books on Demand. ISBN: 978-3-7322-8193-0. URL: <http://www.inf.fu-berlin.de/inst/ag-se/pubs/SalPre13-baseconbook.pdf> (cited on pages 27, 88, 128, 131–135, 137, 151, 165, 169–170, 190–191, 193, 195–196, 204, 241, 254, 265, 342, 359, 364, 369).
- Norsaremah Salleh, Emilia Mendes, & John C. Grundy (2011). “Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review.” In: *IEEE Transactions on Software Engineering* 37 (4), pp. 509–525. DOI: [10.1109/TSE.2010.59](https://doi.org/10.1109/TSE.2010.59) (cited on pages 51–52, 54–61, 428–429).
- Norsaremah Salleh, Emilia Mendes, & John C. Grundy (2014). “Investigating the effects of personality traits on pair programming in a higher education setting through a family of experiments.” In: *Empirical Software Engineering* 19 (3), pp. 714–752. DOI: [10.1007/s10664-012-9238-4](https://doi.org/10.1007/s10664-012-9238-4) (cited on pages 57–58).
- Iflaah Salman, Ayse Tosun Misirli, & Natalia Juristo (2015). “Are Students Representatives of Professionals in Software Engineering Experiments?” In: *Proc. 37th Int’l. Conf. on Software Engineering. ICSE ’15*. IEEE, pp. 666–676. DOI: [10.1109/icse.2015.82](https://doi.org/10.1109/icse.2015.82) (cited on page 48).
- Outi Salo & Pekka Abrahamsson (2008). “Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of Extreme Programming and Scrum.” In: *IET Software* 2 (1), pp. 58–64. DOI: [10.1049/iet-sen:20070038](https://doi.org/10.1049/iet-sen:20070038) (cited on page 43).
- Emanuel A. Schegloff (1968). “Sequencing in Conversational Openings.” In: *American Anthropologist* 70 (6), pp. 1075–1095. DOI: [10.1525/aa.1968.70.6.02a00030](https://doi.org/10.1525/aa.1968.70.6.02a00030) (cited on page 112).
- Emanuel A. Schegloff & Harvey Sacks (1973). “Opening up Closings.” In: *Semiotica* 8 (4), pp. 289–327. DOI: [10.1515/semi.1973.8.4.289](https://doi.org/10.1515/semi.1973.8.4.289) (cited on page 112).
- Julia Schenk, Lutz Prechelt, & Stephan Salinger (2014). “Distributed-Pair Programming can work well and is not just Distributed Pair-Programming.” In: *Proc. 36th Int’l. Conf. on Software Engineering. ICSE ’14*. ACM Press, pp. 74–83. DOI: [10.1145/2591062.2591188](https://doi.org/10.1145/2591062.2591188) (cited on pages 84, 151, 369).
- Julia Schenk (2018). “Industrially Usuable Distributed Pair Programming.” PhD thesis. Freie Universität Berlin. DOI: [10.17169/refubium-938](https://doi.org/10.17169/refubium-938) (cited on pages 26–27, 84–86, 145, 149, 151, 153, 156, 231, 369, 373, 426).
- Christian Schindler (2008). “Agile Software Development Methods and Practices in Austrian IT-Industry: Results of an Empirical Study.” In: *Proc. Int’l. Conf. on Computational Intelligence for Modelling, Control and Automation (CIMCA), Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC), Innovation in Software Engineering (ISE)*. IEEE, pp. 321–326. DOI: [10.1109/CIMCA.2008.100](https://doi.org/10.1109/CIMCA.2008.100) (cited on pages 43–45).
- Donald A. Schön (1983). *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, Inc. ISBN: 978-0-465-06878-4 (cited on pages 333–334).

- Alfred Schütz (1932). *Der Sinnhafte Aufbau der sozialen Welt. Eine Einleitung in die verstehende Soziologie.* (lit. “The meaningful construction of the social world. An introduction to the comprehending sociology”). Vienna: Julius Springer. ISBN: 3709131081 (cited on pages 108, 458).
- Alfred Schuetz (1953). “Common-Sense and Scientific Interpretation of Human Action.” In: *Philosophy and Phenomenological Research* 14 (1), p. 1. DOI: 10.2307/2104013 (cited on page 109).
- Alfred Schutz (1954). “Concept and Theory Formation in the Social Sciences.” In: *The Journal of Philosophy* 51 (9), pp. 257–273. DOI: 10.1007/978-94-010-2851-6\_2 (cited on pages 108–109).
- Alfred Schutz (1967). *The Phenomenology of the Social World.* English translation of Schütz (1932). Northwestern University Press. ISBN: 978-0810103900 (cited on page 108).
- John R. Searle (1969). *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press. ISBN: 978-0521071840 (cited on pages 112–113, 198).
- Todd Sedano, Paul Ralph, & Cécile Péraire (2016). “Sustainable Software Development through Overlapping Pair Rotation.” In: *Proc. 10th ACM/IEEE Int’l. Symp. on Empirical Software Engineering and Measurement. ESEM ’16.* ACM Press, 19:1–19:10. DOI: 10.1145/2961111.2962590 (cited on pages 23, 44).
- Panagiotis Sfetsos, Panagiotis Adamidis, Lefteris Angelis, Ioannis Stamelos, & Ignatios Deligiannis (2013). “Heterogeneous Personalities Perform Better in Pair Programming: The Results of a Replication Study.” In: *Software Quality Professional Magazine* 15 (4), pp. 4–15 (cited on pages 58, 61).
- Kamran Sharifabdi & Claudia Grot (2002). “Team Development and Pair Programming - Tasks and Challenges of the XP Coach.” In: *Proc. 3rd Int’l. Conf. on Extreme Programming and Flexible Processes in Software Engineering. XP 2002,* pp. 166–169 (cited on page 46).
- Marvin E. Shaw (1981). *Group Dynamics: The Psychology of Small Group Behavior.* 3rd ed. McGraw-Hill. ISBN: 0-07-056504-X (cited on page 97).
- Ben Shneiderman (1976). “Exploratory experiments in programmer behavior.” In: *International Journal of Computer & Information Sciences* 5 (2), pp. 123–143. DOI: 10.1007/bf00975629 (cited on page 37).
- Jonathan Sillito (2006). “Asking and Answering Questions During a Programming Change Task.” PhD thesis. University of British Columbia. URL: <http://pages.cpsc.ucalgary.ca/~sillito/work/dissertation.pdf> (cited on page 82).
- Jonathan Sillito, Gail C. Murphy, & Kris De Volder (2008). “Asking and Answering Questions during a Programming Change Task.” In: *IEEE Transactions on Software Engineering* 34 (4), pp. 434–451. DOI: 10.1109/TSE.2008.26 (cited on pages 38, 80, 82, 86, 92, 255).
- Alberto Sillitti, Giancarlo Succi, & Jelena Vlasenko (2012). “Understanding the Impact of Pair Programming on Developers Attention: A Case Study on a Large Industrial Experimentation.” In: *Proc. 34th Int’l. Conf. on Software Engineering. ICSE ’12,* pp. 1094–1101. DOI: 10.1109/ICSE.2012.6227110 (cited on pages 69–70).
- Susan Elliott Sim & Richard C. Holt (1998). “The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize.” In: *Proc. 20th Int’l. Conf. on Software Engineering. ICSE ’98.* Washington, DC, USA: IEEE Computer Society, pp. 361–370. DOI: 10.1109/ICSE.1998.671389 (cited on pages 37, 334).
- Herbert A. Simon (1996). *The Sciences of the Artificial.* 3rd ed. The MIT Press. ISBN: 9780262193740 (cited on page 333).
- Elisa J. Sobo, Elizabeth Herlihy, & Mary Bicker (2011). “Selling medical travel to US patient-consumers: the cultural appeal of website marketing messages.” In: *Anthropology & Medicine* 18 (1), pp. 119–136. DOI: 10.1080/13648470.2010.525877 (cited on page 130).

- David Socha & Kevin Sutanto (2015). "The "Pair" as a Problematic Unit of Analysis for Pair Programming." In: *Proc. 8th Int'l. Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE '15, pp. 64–70. DOI: [10.1109/CHASE.2015.16](https://doi.org/10.1109/CHASE.2015.16) (cited on pages 144, 364).
- David Socha, Robin Adams, Kelly Franznick, Wolff-Michael Roth, Kevin Sullivan, Josh Tenenber, & Skip Walter (2016). "Wide-Field Ethnography: Studying Software Engineering in 2025 and Beyond." In: *Proc. 38th Int'l. Conf. on Software Engineering Companion*. ICSE '16. ACM Press, pp. 797–802. DOI: [10.1145/2889160.2889214](https://doi.org/10.1145/2889160.2889214) (cited on pages 144, 364).
- Elliot Soloway, Kate Ehrlich, & Jeffrey G. Bonar (1982). "Tapping into Tacit Programming Knowledge." In: *Proc. 1982 Conf. on Human Factors in Computing Systems*. CHI '82. ACM, pp. 52–57. DOI: [10.1145/800049.801754](https://doi.org/10.1145/800049.801754) (cited on page 38).
- Elliot Soloway & Kate Ehrlich (1984). "Empirical studies of programming knowledge." In: *IEEE Transactions on Software Engineering* 10 (5), pp. 595–609. DOI: [10.1109/TSE.1984.5010283](https://doi.org/10.1109/TSE.1984.5010283) (cited on pages 37, 333–334).
- Ian Sommerville (2007). *Software Engineering*. 8th ed. Addison-Wesley. ISBN: 978-0-321-31379-9 (cited on page 26).
- Sabine Sonnentag, Cornelia Niessen, & Judith Volmer (2006). "Expertise in Software Design." In: *The Cambridge Handbook of Expertise and Expert Performance*. Ed. by K. Anders Ericsson, Neil Charness, Paul J. Feltovitch, & Robert R. Hoffmann. Cambridge University Press, pp. 373–387. DOI: [10.1017/CBO9780511816796.021](https://doi.org/10.1017/CBO9780511816796.021) (cited on pages 37, 40).
- StackOverflow (2018). *2018 Stack Overflow Developer Survey*. URL: <https://insights.stackoverflow.com/survey/2018> (cited on page 43).
- Robert C. Stalaker (2002). "Assertion." In: *Formal Semantics: The Essential Readings*. Ed. by Paul H. Portner & Barbara H. Partee. Blackwell Publishers Ltd, pp. 147–161. ISBN: 978-0-631-21542-4. DOI: [10.1002/9780470758335.ch5](https://doi.org/10.1002/9780470758335.ch5) (cited on page 110).
- Ivan D. Steiner (1972). *Group Process and Productivity*. Academic Press. ISBN: 0-12-665350-X (cited on page 97).
- Matt Stephens & Doug Rosenberg (2003). "Pair Programming (Dear Uncle Joe, My Pair Programmer Has Halitosis)." In: *Extreme Programming Refactored: The Case Against XP*. Apress. Chap. 6, pp. 135–160. ISBN: 978-1-59059-096-6 (cited on page 42).
- Klaas-Jan Stol, Paul Ralph, & Brian Fitzgerald (2016). "Grounded theory in software engineering research." In: *Proc. Int'l. 38th Conf. on Software Engineering*. ICSE '16. ACM Press, pp. 120–131. DOI: [10.1145/2884781.2884833](https://doi.org/10.1145/2884781.2884833) (cited on pages 125–126).
- Klaas-Jan Stol & Brian Fitzgerald (2018). "The ABC of Software Engineering Research." In: *ACM Transactions on Software Engineering and Methodology* 27 (3), 11:1–11:51. DOI: [10.1145/3241743](https://doi.org/10.1145/3241743) (cited on pages 25, 49).
- Anselm Strauss & Juliet Corbin (1990). *Basics of Qualitative Research. Grounded Theory Procedure and Techniques*. Sage Publications. ISBN: 978-0803932500 (cited on pages 117–129, 135, 152, 167–168, 178–179, 333, 344).
- Bjørnar Tessem (2003). "Experiences in Learning XP Practices: A Qualitative Study." In: *Extreme Programming and Agile Processes in Software Engineering*. Springer Berlin Heidelberg, pp. 131–137. DOI: [10.1007/3-540-44870-5\\_17](https://doi.org/10.1007/3-540-44870-5_17) (cited on page 46).
- Betty Alexandra Toole (1996). "Ada Byron, Lady Lovelace, An Analyst and Metaphysician." In: *IEEE Annals of the History of Computing* 18 (3), pp. 4–12. DOI: [10.1109/85.511939](https://doi.org/10.1109/85.511939) (cited on page 20).
- Sarah J. Tracy (2010). "Qualitative Quality: Eight "Big-Tent" Criteria for Excellent Qualitative Research." In: *Qualitative Inquiry* 16 (10), pp. 837–851. DOI: [10.1177/1077800410383121](https://doi.org/10.1177/1077800410383121) (cited on pages 28, 115, 127, 141, 150, 152, 178, 349–350, 356–358).

- Franziska Tschan (2000). *Produktivität in Kleingruppen: Was machen produktive Gruppen anders und besser?* (lit. "Productivity in Small Groups: What do productive groups do different and better?") Verlag Hans Huber. ISBN: 9783456834467 (cited on page 98).
- Theodore Van Toll III, Roger Lee, & Tom Ahlswede (2007). "Evaluating the Usefulness of Pair Programming in a Classroom Setting." In: *6th IEEE/ACIS Int'l. Conf. on Computer and Information Science*. ICIS '07. IEEE, pp. 302–308. DOI: [10.1109/icis.2007.96](https://doi.org/10.1109/icis.2007.96) (cited on page 58).
- Tammy VanDeGrift (2004). "Coupling pair programming and writing." In: *Proc. 35th SIGCSE technical symposium on Computer science education*. SIGCSE '04. ACM Press, pp. 2–6. DOI: [10.1145/971300.971306](https://doi.org/10.1145/971300.971306) (cited on pages 57, 61).
- Jari Vanhanen & Casper Lassenius (2005). "Effects of Pair Programming at the Development Team Level: An Experiment." In: *Proc. 2005 Int'l. Symp. on Empirical Software Engineering (ISESE)*. IEEE, pp. 336–345. DOI: [10.1109/ISESE.2005.1541842](https://doi.org/10.1109/ISESE.2005.1541842) (cited on pages 56, 71).
- Jari Vanhanen & Harri Korpi (2007). "Experiences of Using Pair Programming in an Agile Project." In: *Proc. 40th Annual Hawaii Int'l. Conf. on System Sciences*. HICSS'07. DOI: [10.1109/HICSS.2007.218](https://doi.org/10.1109/HICSS.2007.218) (cited on pages 43, 70–71, 92, 255).
- Jari Vanhanen & Casper Lassenius (2007). "Perceived Effects of Pair Programming in an Industrial Context." In: *Proc. 33rd EUROMICRO Conf. on Software Engineering and Advanced Applications*. IEEE, pp. 211–218. DOI: [10.1109/EUROMICRO.2007.47](https://doi.org/10.1109/EUROMICRO.2007.47) (cited on page 46).
- Jari Vanhanen & Mika V. Mäntylä (2013). "A systematic mapping study of empirical studies on the use of pair programming in industry." In: *International Journal of Software Engineering and Knowledge Engineering* 23 (09), pp. 1221–1267. DOI: [10.1142/S0218194013500381](https://doi.org/10.1142/S0218194013500381) (cited on pages 51–54, 62).
- William C. Wake (2002). *Extreme Programming Explored*. Addison-Wesley. ISBN: 978-0-201-73397-6 (cited on page 75).
- Thorbjørn Walle & Jo E. Hannay (2009). "Personality and the Nature of Collaboration in Pair Programming." In: *Proc. 3rd Int'l. Symp. on Empirical Software Engineering and Measurement*. ESEM '09. IEEE, pp. 203–213. DOI: [10.1109/ESEM.2009.5315996](https://doi.org/10.1109/ESEM.2009.5315996) (cited on pages 73–76, 301).
- Noreen M. Webb & Scott Lewis (1988). "The Social Context of Learning Computer Programming." In: *Training and Learning Computer Programming: Multiple Research Perspectives*. Ed. by Richard E. Mayer. Mallory International. Chap. 8, pp. 179–206. ISBN: 978-0805800739 (cited on pages 61–62).
- Max Weber (1922). "Wirtschaft und Gesellschaft." In: *Grundriss der verstehenden Soziologie*. (lit. "Economy and Society"). Tübingen: J. C. B. Mohr (Paul Siebeck) (cited on page 108).
- Max Weber (1978). *Max Weber: Selections in Translation*. Ed. by W. G. Runciman. Trans. by Matthews, E. Cambridge University Press. ISBN: 0-521-29268-9 (cited on page 108).
- Daniel M. Wegner (1987). "Transactive Memory: A Contemporary Analysis of the Group Mind." In: *Theories of Group Behavior*. Ed. by Brian Mullen & George R. Goethals. New York: Springer. Chap. 9, pp. 185–208. DOI: [10.1007/978-1-4612-4634-3\\_9](https://doi.org/10.1007/978-1-4612-4634-3_9) (cited on page 98).
- Gerald M. Weinberg (1971). *The Psychology of Computer Programming*. Van Nostrand Reinhold Company. ISBN: 978-0-442-29264-5 (cited on pages 20–21, 108).
- Robert W. Weisberg & Lauretta Reeves (2013). *Cognition. From Memory to Creativity*. Wiley. ISBN: 978-0-470-22628-5 (cited on pages 35–36).
- Kipling D. Williams (2010). "Dyads Can Be Groups (and Often Are)." In: *Small Group Research* 41 (2), pp. 268–274. DOI: [10.1177/1046496409358619](https://doi.org/10.1177/1046496409358619) (cited on page 94).
- Laurie Williams, Robert R. Kessler, Ward Cunningham, & Ron Jeffries (2000). "Strengthening the Case for Pair Programming." In: *IEEE Software* 17 (4), pp. 19–25. DOI: [10.1109/52.854064](https://doi.org/10.1109/52.854064) (cited on pages 184, 204, 236).

- Laurie Williams (2000). "The Collaborative Software Process." PhD thesis. Department of Computer Science, The University of Utah. URL: <https://collaboration.csc.ncsu.edu/laurie/Papers/dissertation.pdf> (cited on pages 41, 236).
- Laurie Williams & Robert R. Kessler (2000). "The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education." In: *Proc. 3th Conf. on Software Engineering Education & Training*. IEEE, pp. 59–65. DOI: [10.1109/CSEE.2000.827023](https://doi.org/10.1109/CSEE.2000.827023) (cited on page 57).
- Laurie Williams & Robert R. Kessler (2001). "Experiments with Industry's "Pair-Programming" Model in the Computer Science Classroom." In: *Computer Science Education* 11 (1), pp. 7–20. DOI: [10.1076/csed.11.1.7.3846](https://doi.org/10.1076/csed.11.1.7.3846) (cited on page 236).
- Laurie Williams (2001). "Integrating Pair Programming into a Software Development Process." In: *Proc. 14th Conf. on Software Engineering Education and Training*. CSEET'01. IEEE, pp. 27–36. DOI: [10.1109/CSEE.2001.913816](https://doi.org/10.1109/CSEE.2001.913816) (cited on page 41).
- Laurie Williams & Robert R. Kessler (2002). *Pair Programming Illuminated*. Addison-Wesley Professional. ISBN: 978-0-201-74576-4 (cited on pages 21, 40–41, 44, 142).
- Laurie Williams, Lucas Layman, Jason A. Osborne, & Neha Katira (2006). "Examining the Compatibility of Student Pair Programmers." In: *AGILE 2006*. IEEE, pp. 411–420. DOI: [10.1109/AGILE.2006.25](https://doi.org/10.1109/AGILE.2006.25) (cited on pages 59–60).
- Judith D. Wilson, John T. Nosek, Nathan Hoskin, & Lin L. Liou (1992). "The Effect of Collaboration on Problem-Solving Performance Among Programmers." In: *Algorithms, Software, Architecture. Information Processing 92: Proceedings of the IFIP 12th World Computer Congress*. Vol. 1. North-Holland Publishing Co., pp. 86–93. ISBN: 0-444-89747-X (cited on pages 60, 63–64, 66, 431).
- Gwen M. Wittenbaum, Sandra I. Vaughan, & Garold Strasser (1998). "Coordination in Task-Performing Groups." In: *Theory and Research on Small Groups*. Ed. by R. Scott Tindale, Linda Heath, John Edwards, Emil J. Posavac, Fred B. Bryant, Yolanda Suarez-Balcazar, Eaaron Henderson-King, & Judith Myers. Plenum Press. Chap. 9, pp. 177–204. DOI: [10.1007/0-306-47144-2\\_9](https://doi.org/10.1007/0-306-47144-2_9) (cited on page 98).
- Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, & Anders Wesslén (2012). "Empirical Strategies." In: *Experimentation in Software Engineering*. Springer Berlin Heidelberg, pp. 9–36. DOI: [10.1007/978-3-642-29044-2\\_2](https://doi.org/10.1007/978-3-642-29044-2_2) (cited on page 140).
- William A. Wood & William L. Kleb (2003). "Exploring XP for scientific research." In: *IEEE Software* 20 (3), pp. 30–36. DOI: [10.1109/ms.2003.1196317](https://doi.org/10.1109/ms.2003.1196317) (cited on page 46).
- Shaochun Xu, Václav Rajlich, & Andrian Marcus (2005). "An Empirical Study of Programmer Learning during Incremental Software Development." In: *Proc. Fourth IEEE Conf. on Cognitive Informatics*. ICCI 2005, pp. 340–349. DOI: [10.1109/COGINF.2005.1532650](https://doi.org/10.1109/COGINF.2005.1532650) (cited on pages 77, 80, 86, 132, 301).
- Shaochun Xu & Václav Rajlich (2005). "Pair Programming in Graduate Software Engineering Course Projects." In: *Proc. Frontiers in Education 35th Annual Conf.* DOI: [10.1109/fie.2005.1612027](https://doi.org/10.1109/fie.2005.1612027) (cited on pages 57, 61, 92).
- Robert K. Yin (2014). *Case Study Research. Design and Methods*. 5th ed. Sage Publications. ISBN: 978-1-4522-4256-9 (cited on page 51).
- Victor Yngve (1970). "On getting a word in edgewise." In: *Chicago Linguistics Society, 6th Meeting*, pp. 567–578 (cited on page 112).
- Nick Z. Zacharis (2011). "Measuring the Effects of Virtual Pair Programming in an Introductory Programming Java Course." In: *IEEE Transactions on Education* 54 (1), pp. 168–170. DOI: [10.1109/te.2010.2048328](https://doi.org/10.1109/te.2010.2048328) (cited on pages 55–57, 63–64, 428–432).

- Mark Zarb, Janet Hughes, & John Richards (2012). "Analysing Communication Trends in Pair Programming Videos using Grounded Theory." In: *Proc. 26th BCS Conf. on Human Computer Interaction (HCI)*. DOI: [10.14236/ewic/HCI2012.106](https://doi.org/10.14236/ewic/HCI2012.106) (cited on pages 82, 86).
- Mark Zarb, Janet Hughes, & John Richards (2013). "Industry-Inspired Guidelines Improve Students' Pair Programming Communication." In: *Proc. 18th ACM Conf. on Innovation and Technology in Computer Science Education*. ITiCSE '13, pp. 135–140. DOI: [10.1145/2462476.2462504](https://doi.org/10.1145/2462476.2462504) (cited on pages 82, 85–86).
- Mark Zarb, Janet Hughes, & John Richards (2014). "Evaluating Industry-Inspired Pair Programming Communication Guidelines with Undergraduate Students." In: *Proc. 45th ACM Technical Symp. on Computer Science Education*. SIGCSE '14. ACM Press. DOI: [10.1145/2538862.2538980](https://doi.org/10.1145/2538862.2538980) (cited on pages 82, 86).
- Wilfred J. Zerbe & Delroy L. Paulhus (1987). "Socially Desirable Responding in Organizational Behavior: A Reconciliation." In: *The Academy of Management Review* 12 (2), pp. 250–264. ISSN: 03637425. URL: <http://www.jstor.org/stable/258533> (cited on page 354).
- Minghui Zhou & Audris Mockus (2010). "Developer Fluency: Achieving True Mastery in Software Projects." In: *Proc. 18th ACM SIGSOFT Int'l. Symp. on Foundations of Software Engineering*. FSE '10. ACM, pp. 137–146. DOI: [10.1145/1882291.1882313](https://doi.org/10.1145/1882291.1882313) (cited on pages 37–38, 334).
- Franz Zieris & Stephan Salinger (2013). "Doing Scrum Rather Than Being Agile: A Case Study on Actual Nearshoring Practices." In: *Proc. 2013 IEEE 8th Int'l. Conf. on Global Software Engineering*. ICGSE '13. IEEE, pp. 144–153. DOI: [10.1109/ICGSE.2013.26](https://doi.org/10.1109/ICGSE.2013.26) (cited on page 145).
- Franz Zieris & Lutz Prechelt (2014). "On Knowledge Transfer Skill in Pair Programming." In: *Proc. 8th ACM/IEEE Int'l. Symp. on Empirical Software Engineering and Measurement*. ESEM '14. ACM. DOI: [10.1145/2652524.2652529](https://doi.org/10.1145/2652524.2652529) (cited on pages 151, 261, 264, 343, 367, 369).
- Franz Zieris (2015). "Qualitative Analysis of Knowledge Transfer in Pair Programming." In: *2015 IEEE/ACM 37th IEEE Int'l. Conf. on Software Engineering*. ICSE '15 (Doctoral Symposium). IEEE, pp. 855–858. DOI: [10.1109/icse.2015.277](https://doi.org/10.1109/icse.2015.277) (cited on page 367).
- Franz Zieris & Lutz Prechelt (2016). "Observations on Knowledge Transfer of Professional Software Developers During Pair Programming." In: *Proc. 38th Int'l. Conf. on Software Engineering Companion*. ICSE '16 (SEIP). ACM, pp. 242–250. DOI: [10.1145/2889160.2889249](https://doi.org/10.1145/2889160.2889249) (cited on pages 151, 230, 240, 301, 346, 368–369, 377).
- Franz Zieris & Lutz Prechelt (2019). "Does Pair Programming Pay Off?" In: *Rethinking Productivity in Software Engineering*. Ed. by Caitlin Sadowksi & Thomas Zimmermann. Apress. Chap. 21. DOI: [10.1007/978-1-4842-4221-6\\_21](https://doi.org/10.1007/978-1-4842-4221-6_21) (cited on page 368).
- Franz Zieris & Lutz Prechelt (2020a). "Explaining Pair Programming Session Dynamics from Knowledge Gaps." In: *Proc. 42nd Int'l. Conf. on Software Engineering*. ICSE '20. ACM. DOI: [10.1145/3377811.3380925](https://doi.org/10.1145/3377811.3380925) (cited on pages 151, 347, 368–369).
- Franz Zieris & Lutz Prechelt (2020b). *PP-ind: A Repository of Industrial Pair Programming Session Recordings*. arXiv: [2002.03121v3 \[cs.SE\]](https://arxiv.org/abs/2002.03121v3) (cited on pages 148, 151, 157, 368, 426, 455).
- Franz Zieris (2020). "When Grounded Theory is Not Enough: Additions for Video-Based Analyses of Software Engineering Process Phenomena." In: *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik, 24.–28. Februar 2020, Innsbruck, Österreich*. Ed. by Michael Felderer, Wilhelm Hasselbring, Rick Rabiser, & Reiner Jung. SE '20, pp. 153–154. DOI: [10.18420/SE2020\\_47](https://doi.org/10.18420/SE2020_47) (cited on page 368).
- Abdullah Mohd Zin, Sufian Idris, & Nantha Kumar Subramaniam (2006). "Improving Learning of Programming Through E-Learning by Using Asynchronous Virtual Pair Programming." In: *Turkish Online Journal of Distance Education* 7, pp. 162–173 (cited on page 57).

## Selbstständigkeitserklärung

Name: Zieris  
Vorname: Franz

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Dissertation selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht. Diese Dissertation wurde in gleicher oder ähnlicher Form noch in keinem früheren Promotionsverfahren eingereicht.

Mit einer Prüfung meiner Arbeit durch ein Plagiatsprüfungsprogramm erkläre ich mich einverstanden.

Berlin, 25. August 2020

.....  
Unterschrift