

# Introducing Innovations into Open Source Projects

Dissertation zur Erlangung des Grades  
eines Doktors der Naturwissenschaften (Dr. rer. nat.)  
am Fachbereich Mathematik und Informatik  
der Freien Universität Berlin

von

Sinan Christopher Özbek

Berlin  
August 2010



Gutachter:

Professor Dr. Lutz Prechelt, Freie Universität Berlin  
Professor Kevin Crowston, Syracuse University

Datum der Disputation: 17.12.2010



# Abstract

This thesis presents a qualitative study using Grounded Theory Methodology on the question of how to change development processes in Open Source projects. The mailing list communication of thirteen medium-sized Open Source projects over the year 2007 was analyzed to answer this question. It resulted in eight main concepts revolving around the introduction of innovation, i.e. new processes, services, and tools, into the projects including topics such as the migration to new systems, the question on where to host services, how radical Open Source projects can change their ways, and how compliance to processes and conventions is enforced. These are complemented with (1) the result of five case studies in which innovation introductions were conducted with Open Source projects, and with (2) a theoretical comparison of the results of this thesis to four theories and scientific perspectives from the organizational and social sciences such as Path Dependence, the Garbage Can model, Social-Network analysis, and Actor-Network theory. The results show that innovation introduction is a multifaceted phenomenon, of which this thesis discusses the most salient conceptual aspects. The thesis concludes with practical advice for innovators and specialized hints for the most popular innovations.



# Acknowledgements

I want to thank the following individuals for contributing to the completion of this thesis:

- Lutz Prechelt for advising me over these long five years.
- Stephan Salinger and Florian Thiel for discussion and critique of methodology and results.
- Gesine Milde and Anja Kasseckert for countless hours of proof-reading, comma checking, and browsing the dictionaries to find better words to spice up my poor writing.
- Many thanks go to my family who have always supported me in doing this Ph.D. and always listened when I explained what it is exactly that I am doing.
- Aenslee, Anne, and Ulrike who each in their own style participated in the genesis of this thesis and motivated me to persevere.
- Leonard Dobusch, Steven Evers, and Lina Böcker—in their facility as the Open Source Research Network—for inspirational discussion in the early days of my thesis.
- Karl Beecher, Julia Schenk, Ulrich Stärk, Janine Rohde, Isabella Peukes, and Moritz Minzlaff for proof-reading selected chapters.
- Martin Gruhn and Sebastian Jekutsch for keeping room 008 a great place for working.
- The Saros Team for providing a welcome relief from the sociological work of this thesis.












## Attribution-Noncommercial-Share Alike 3.0 Germany

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Germany License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

### You are free:

-  **to Share**—to copy, distribute, and transmit the work.
-  **to Remix**—to adapt the work.

### Under the following conditions:

-  **Attribution**—You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
-  **Noncommercial**—You may not use this work for commercial purposes.
-  **Share Alike**—If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

### With the understanding that:

- **Waiver**—Any of the above conditions can be waived if you get permission from the copyright holder.
- **Other Rights**—In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights;
  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- **Notice**—For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to <http://creativecommons.org/licenses/by-nc-sa/3.0/de/deed.en>.



# Contents

<b>1</b>	<b>Introduction</b>	<b>19</b>
1.1	Goal and Motivation . . . . .	19
1.2	Context . . . . .	20
1.3	Contributions . . . . .	20
1.4	Outline . . . . .	21
1.5	Note to the Reader . . . . .	21
<b>2</b>	<b>Background</b>	<b>23</b>
2.1	Defining Innovation Introduction . . . . .	23
2.1.1	Introduction vs. Diffusion . . . . .	25
2.2	Types of Innovations . . . . .	25
2.3	Open Source . . . . .	26
2.3.1	Historical Introduction . . . . .	27
2.3.2	Terminology . . . . .	28
2.3.3	Licensing . . . . .	29
2.3.4	Research on Open Source . . . . .	30
2.3.5	The Participants . . . . .	31
2.3.6	Open Source and Organizations . . . . .	33
2.3.7	Community Ideology . . . . .	34
2.3.8	The Open Source Development Process . . . . .	35
2.3.9	Decision Making . . . . .	41
2.3.10	Defining Success . . . . .	42
<b>3</b>	<b>Methodology</b>	<b>45</b>
3.1	Development of the Methodology . . . . .	45
3.1.1	Action research and field experiments . . . . .	45
3.1.2	Surveying the Open Source Community for Change Episodes . . . . .	47
3.1.3	Using Grounded Theory Methodology . . . . .	48
3.2	Grounded Theory Methodology . . . . .	48
3.2.1	Open Coding . . . . .	49
3.2.2	Axial Coding . . . . .	54
3.2.3	Selective Coding . . . . .	58
3.2.4	The Paradigm . . . . .	61
3.3	Use of GTM in Related and Open Source Research . . . . .	62
3.4	GmanDA – Tool Support for Qualitative Data Analysis of E-mail Data . . . . .	63
3.4.1	User Interface . . . . .	65
3.4.2	Software Design . . . . .	68
3.5	Visualization of Temporal Data . . . . .	71
3.6	Data . . . . .	75
3.6.1	Selection Criteria . . . . .	75
3.6.2	Data Collection Site . . . . .	76
3.6.3	List of Projects . . . . .	76

3.7	Threats to Validity . . . . .	79
3.7.1	Credibility and Internal Validity . . . . .	79
3.7.2	Relevance and External Validity . . . . .	80
<b>4</b>	<b>Related Work</b>	<b>83</b>
4.1	Optimizing Open Source Projects . . . . .	83
4.2	Innovation Adoption in Debian . . . . .	85
4.3	Innovation Introduction and Design Discussions . . . . .	88
4.4	Community Building . . . . .	93
4.5	Other Related Work . . . . .	94
<b>5</b>	<b>Results</b>	<b>97</b>
5.1	Quantitative Overview . . . . .	97
5.2	The Introduction Lifecycle . . . . .	98
5.2.1	Failure Reasons . . . . .	101
5.3	Partial Migrations . . . . .	102
5.3.1	Partial Migration at KVM . . . . .	103
5.3.2	Partial Migration at ROX . . . . .	105
5.4	Enactment Scopes of Process Innovation . . . . .	107
5.5	Hosting . . . . .	111
5.5.1	Strategies for Hosting . . . . .	116
5.5.2	Relating Hosting to Innovation Introduction . . . . .	116
5.5.3	Summary . . . . .	118
5.6	Adapter Innovations . . . . .	118
5.7	Forcing, Compliance, and Decisions . . . . .	119
5.7.1	Organizational Innovation Decisions . . . . .	119
5.7.2	Individual Innovation Decisions . . . . .	121
5.7.3	Compliance and its Enforcement . . . . .	123
5.8	Participation Sprints and Time-dependent Behavior . . . . .	126
5.9	Radical vs. Evolutionary Innovation . . . . .	129
5.9.1	Reimplementation in Open Source Development . . . . .	129
5.9.2	Radical Innovation Introductions . . . . .	132
5.10	Tool Independence . . . . .	134
<b>6</b>	<b>Comparison with Classic Models</b>	<b>137</b>
6.1	Path Dependence . . . . .	138
6.1.1	Literature on Path Dependence and Open Source . . . . .	141
6.1.2	Path Dependence and Innovation Episodes . . . . .	142
6.1.3	Implications for the Innovator . . . . .	147
6.2	Garbage Can Model . . . . .	148
6.2.1	The Garbage Can in the Open Source Literature . . . . .	148
6.2.2	The Garbage Can and Innovation Episodes . . . . .	149
6.2.3	Implications for the Innovator . . . . .	155
6.3	Social Network Analysis and Social Network Theory . . . . .	156
6.3.1	SNA in the Open Source Literature . . . . .	156
6.3.2	Social Network Analysis and Innovation Episodes . . . . .	158
6.3.3	Summary and Recommendations for the Innovator . . . . .	162
6.4	Actor-Network Theory . . . . .	163
6.4.1	ANT in the Open Source Literature . . . . .	163
6.4.2	Actor-Network Theory and Innovation Episodes . . . . .	164
6.4.3	Summary . . . . .	166
6.5	Self-Organization and Structuration . . . . .	167
6.6	Heterarchical Organizations . . . . .	167
6.7	Communities of Practice . . . . .	168

6.8	Summary . . . . .	168
<b>7</b>	<b>Case Studies</b>	<b>171</b>
7.1	Information Management in Open Source Projects . . . . .	171
7.1.1	Assessing the Information Manager . . . . .	173
7.1.2	Mass Marketing the Information Manager . . . . .	174
7.1.3	Using Information Management . . . . .	174
7.2	Gift Giving in Open Source Projects . . . . .	177
7.2.1	Socio-Hierarchy and Techno-Centricity . . . . .	177
7.2.2	Idea Splurge and Disconnect . . . . .	179
7.2.3	Idea Ownership . . . . .	180
7.2.4	Gaining Familiarity . . . . .	180
7.2.5	Separation of Discourse Arenas . . . . .	180
7.2.6	Introduction Barrier . . . . .	181
7.2.7	Summary for the Innovator . . . . .	181
7.3	Contact Strategies for Open Source Projects . . . . .	181
7.4	Introducing Automated Testing . . . . .	185
7.4.1	Testing-specific Results . . . . .	188
7.4.2	Summary . . . . .	189
7.5	Increasing Security by Annotation-guided Refactoring . . . . .	189
7.5.1	Summary . . . . .	193
<b>8</b>	<b>Practical Advice</b>	<b>195</b>
8.1	Common Innovations and their Introductions . . . . .	195
8.1.1	Source Code Management . . . . .	195
8.1.2	The Google Summer of Code . . . . .	199
8.1.3	License Switching . . . . .	202
8.1.4	Version Naming . . . . .	204
8.1.5	Bug Tracking Procedures . . . . .	206
8.2	Open Access . . . . .	209
8.3	The Innovator's Guide . . . . .	210
8.3.1	Introduction . . . . .	210
8.3.2	Getting Started . . . . .	210
8.3.3	Getting Your Goal Straight . . . . .	211
8.3.4	Getting a Solution . . . . .	211
8.3.5	Proposing . . . . .	212
8.3.6	Executing . . . . .	213
8.3.7	Adoption and Sustaining . . . . .	214
8.3.8	Write an Episode Recap . . . . .	214
<b>9</b>	<b>Conclusion</b>	<b>215</b>
9.1	Future Work . . . . .	217
9.2	Epilogue . . . . .	218
<b>A</b>	<b>Glossary and Codebook</b>	<b>221</b>
A.1	Episode Directory . . . . .	221
A.2	Innovations . . . . .	233
A.3	Activities . . . . .	239
A.4	Concepts . . . . .	240
A.5	Compliance Enforcement Strategies . . . . .	241
A.6	Episode Outcomes . . . . .	242
A.7	Hosting Types . . . . .	243
A.8	Innovation Types . . . . .	244
A.9	Innovation Decision Types . . . . .	245

A.10 Uncategorized Codes . . . . .	246
<b>B Innovation Questionnaire</b>	<b>249</b>
<b>C Zusammenfassung</b>	<b>255</b>
<b>D Curriculum Vitae</b>	<b>256</b>
<b>Bibliography</b>	<b>259</b>
<b>Additional Non-e-mail Documents</b>	<b>291</b>

# List of Figures

2.1	Open Source project success . . . . .	43
3.1	Coding meta-model used for GTM . . . . .	52
3.2	The Grounded Theory paradigm model . . . . .	61
3.3	An extended paradigm model . . . . .	62
3.4	GmanDA screenshot . . . . .	64
3.5	Code detail view in GmanDA . . . . .	67
3.6	Table representation in GmanDA . . . . .	68
3.7	GmanDA architecture . . . . .	70
3.8	GmanDA visualization . . . . .	71
3.9	Temporal visualization of e-mails . . . . .	73
4.1	Conversation structure in PEP 279 . . . . .	89
4.2	Boundary spanning in PEP 327 . . . . .	91
5.1	Overview of central concepts . . . . .	98
5.2	Simplified phase model of innovation introduction . . . . .	100
5.3	Paradigm for the partial migration in KVM . . . . .	104
5.4	Relationships to the concept of hosting . . . . .	117
5.5	Forcing effects during innovation adoption . . . . .	125
6.1	Organizational theories used in this thesis . . . . .	137
6.2	E-mails written during the Google Summer of Code at ArgoUML . . . . .	146
6.3	The innovator's e-mails over time in the Episode Java 5 @ ArgoUML . . . . .	154
6.4	Social networks of ArgoUML and Bugzilla . . . . .	160
6.5	Social networks of remaining projects . . . . .	161
6.6	Social networks of ArgoUML with communities identified . . . . .	162
7.1	Distribution of information management information types over time . . . . .	175
7.2	Distribution of information management action types over time . . . . .	175
7.3	Results of the up-front investment case study . . . . .	178
7.4	Phase model for the introduction of regression testing . . . . .	186
7.5	Success measures of the introduction at FreeCol . . . . .	187
B.1	Innovation Introduction Survey (short version) . . . . .	250
B.2	Innovation Introduction Survey (long version) . . . . .	251





# List of Tables

2.1	Open Source roles and associated activities and rights . . . . .	37
3.1	Gmane.org mailing list prefixes . . . . .	79
5.1	Episodes per project . . . . .	99



# Chapter 1

## Introduction

### 1.1 Goal and Motivation

How to successfully introduce software engineering innovations such as tools and process improvements into Open Source projects is the question addressed in this thesis.

The motivation to pursue this goal—scientific curiosity apart—arises from four current observations and their projections into the future:

First, Open Source software has become a well established and popular alternative to proprietary solutions in many software domains and consequently has attracted an increased number of stakeholders<sup>1</sup>. There are both more users with new bug, feature, and support requests to be satisfied and more developers interested in participating who must coordinate with each other. Also, institutional users such as corporation and government agencies are increasingly using Open Source [145, 358], bringing their own set of expectations regarding a “more predictable and stable development environment” [387], the achievement of quality attributes such as low number of defects and security vulnerabilities [147] and possibly the ability to deploy individuals into the development process [122, p.488].

Second, many Open Source software projects have matured over the years to produce software packets of considerable size [285, 344, 143] and possibly have seen several generations of developers come and go [cf. 172, 461]. The need to rediscover, refactor, and reengineer existing code bases will thus increase over time [75, 345], as will the need to deal with technological changes to development infrastructure, dependencies, and even deployment platforms.

Both these observations indicate that the pressure on Open Source projects is increasing to handle more and better development by a growing number of people with diverging goals. This thesis then argues—following the basic rationale from software development—that the appropriate use of software engineering methods and tools is essential to deal with this pressure and a logical consequence of the “professionalization of Open Source development” [187]. One might hypothesize further that just as projects were forked in the past, when their development progress stalled [348] that in the future we might see projects being forked when they can not modernize their processes to deal with increased popularity and technological change.

---

<sup>1</sup>Precise scientific statistics on the popularity of Open Source software from industrial and end-users is unavailable. Most scientific publications only provide indirect assessments such as the web server market share statistics by Netcraft [347, 326]. Market research firm IDC predicted in 2009 the revenue from Open Source software to grow to 8.1 billion USD by 2013, which would correspond to an annual growth of 20% [174]. Comparing this to the growth estimates for the global software market at 8.5% to reach 457 billion USD in 2013 as well [128], this would put Open Source at less than 2% of global software revenue. Yet, of course comparisons of this kind are difficult to make, since one would expect in particular revenue from services and support to be much higher than the sales figure for Open Source. Red Hat, for instance, alone did post revenues of 653 million USD for 2009 [421]. For a slightly biased and dated assessment of market share of Open Source products in individual software domains see [546].

Third, and related, is the increasing number of institutional actors who release their proprietary software under an Open Source license [518]. Converting such software projects to Open Source development and cultivating a strong community can take years of adjustment and tuning, as the story of the Netscape browser released as Mozilla can attest [231]. Creating processes to accommodate such projects in between structured software development and Open Source is also motivating.

Fourth, studies have found that the use of software engineering methods in Open Source projects is still small [228]. This is motivating as it indicates both potential for improvement and potential for conflicts in the process of introducing such improvements, which must be resolved. Finding a balance between the more structured interests of project leaders and corporate users and the loose coupling preferred by developers, thus avoiding results “too formal or burdensome” or in “conflict with the hacker ethos” [387] yet still effective, appears vital for the continued success of the Open Source development paradigm.

Last, innovation introduction within the special context of Open Source development, where the limitations of hierarchical power and the volunteer nature of the participation restrict top-down action, provides an interesting context in which to consider introduction processes within commercial and classic enterprises. A recent question on Slashdot may illustrate this point:

In a corporate environment more than one hundred engineers were required to adopt a new ‘best practice’ [466], and the collective wisdom of Slashdot readers was tapped to answer the question of how to successfully achieve this. Many participants argued strongly to include the developers in the process to select the improvement and provide mainly education and reasons for why an improvement is necessary and possible. While this would be an applicable approach for Open Source projects, there was a second strong set of suggestions that pointed in the direction of applying discipline, using “pain points”, creating a monetary incentive system or adopting a “comply or be fired” mentality. These types of managerial influence are inapplicable in the context of Open Source, since the volunteers would leave the project if harassed in such a way. Finding ways to achieve change without force in a traditional setting potentially using insights gathered from this line of work thus is the last motivation of this thesis.

## 1.2 Context

For this thesis an Open Source project is an (1) organization of globally distributed participants—a substantial proportion of which are volunteers—who use (2) an open development process to build a piece of software, which (3) is licensed under an Open Source license.

The thesis focuses on volunteer collaboration and excludes (1) software development of Open Source licensed software which uses closed development processes [66] and (2) Inner Source development which uses Open Source development principles but within a closed organizational context such as a corporation [187, 317].

## 1.3 Contributions

This thesis contains the following main contributions:

1. The primary result of the thesis is a conceptualization of phenomena relevant for innovation introduction (see Chapter 5). It was built based on cases from 13 Open Source projects observed in the year 2007. The eight extracted core-concepts constitute the heart of the thesis and are supported by a comparison with four theories and models from the organizational sciences on a theoretical level and by five case studies from an empirical perspective.

2. The detailed description of the research process using Grounded Theory Methodology (GTM) on mailing list data is the second contribution (see Chapter 3). With its focus on both the alternative methodologies attempted and the actual procedures of performing GTM this methodological contribution should be valuable for others who want to study Open Source processes.
3. As part of the thesis, the software GmanDA was developed as a comprehensive, extensible tool-support to aid the research on mailing list data using Grounded Theory Methodology (see Section 3.4). GmanDA is published at <http://gmanda.sf.net> under an Open Source license and is an engineering contribution [227].
4. The thesis provides a practical contribution by combining the theoretical insights into a chapter on advice for the practitioner (see Chapter 8.1), which is focused to help the innovator in the field to achieve introduction both from a generic perspective and for five popular types of innovations.

## 1.4 Outline

This dissertation is structured to fulfill the quality criteria for sound empirical research in Software Engineering in general [280, 266, 156] and Open Source in particular [489]. Accordingly, first the aim of the study is given and the motivation to attain it is outlined (Section 1.1). Next, the central terms such as innovation and introduction are defined (Section 2.1), before an overview of the Open Source development paradigm is given (Section 2.3). Third, the methodology chosen to tackle the research question and analyze data is described and justified in comparison to alternatives which could have been pursued, sampling criteria are explained, and the resulting data set is presented (Chapter 3). The section on methodology closes with an examination of validity threats, and limitations of the chosen approach. Fourth, related work on innovation introduction and software design discussions is presented (Chapter 4). Next, the resulting theory fragments are presented and discussed in detail, which constitutes the primary contribution of this research (Chapter 5). Then, the results are discussed and judged with hindsight to four models and theories from the organizational and social sciences such as the Garbage Can Model, Social Network Theory, Actor-Network Theory, and Path Dependence (Chapter 6). Chapter 7 presents the results from five case studies conducted with students. To counteract the “shortage of practical advice” which pervades much research [434], the results are then condensed into practical advice such as a concise guide for innovators and an overview of the most common innovations introduced with specific advice on their introduction (Chapter 8), before reaching the conclusion and further work in Chapter 9. In the appendix a detailed description of all episodes, innovations and terms is given (Appendix A).

## 1.5 Note to the Reader

This dissertation makes use of hyperlinks in many place: First, links are used to provide easy access to the glossary for definition and terms used in this thesis. Second, and more importantly, all the data from which results are derived are linked from within the text. To make use of these features and evaluate the empirical foundation of this thesis, it is highly recommended to read the dissertation digitally. Information on how to resolve links manually is given in Section 3.2.3).



## Chapter 2

# Background

### 2.1 Defining Innovation Introduction

The term “innovation” was initially adopted for this thesis to find an umbrella to encompass both novel software development *processes* and supporting *tools*. As the analysis proceeded and more *types of innovations* were found to be relevant such as *social innovations* or *conventions*, a comprising definition was sought:

**Definition 1 (Innovation)** *An innovation is a means for changing the development process of a software development project.*

This definition must be distinguished from other definitions assigned to innovations in the management sciences, which emphasize the creation of novel products [240], leading for instance to the question of how innovative the software produced in Open Source projects can be [532, 289]. In this way, the above definition is not about how much novelty can be produced by an Open Source project but how much software engineering innovation it can consume and how the consumption can be achieved.

An “introduction” in this thesis is the consciously performed process of establishing an innovation in an Open Source project, and the innovator<sup>2</sup> is the person who seeks and drives the introduction. Consider the following two examples as illustrations for innovation introductions which occurred in the data analyzed for this thesis: First, in the project *Bugzilla*, the maintainer of the project tried to increase the number of active contributors and among other innovations suggested to the project that new mailing list participants should *introduce themselves*. He gave his reasons such as building a community of trusting peers, and started to convince others of the value of his idea by introducing himself to the list. Even though fifteen people introduced themselves over the next five months, none of these then started to contribute to the project, and the success of the introduction must thus be taken with a grain of salt. Second, in the project *gEDA* two core developers spent more than six months on slowly convincing the project to switch to a *distributed source code management system* by demonstrating its use, tutoring others, enumerating its advantages, and sending links to positive blog-posts to the list. When the project was about to release version 1.0 and began using a *branching scheme*, the innovators had collected sufficient positive opinions to convince the maintainer and the other core developers to switch systems. After the new system was set up, the developers then began adopting it one after the other, until finally it was successfully used project-wide.

*Self  
Introductions  
at Bugzilla*

*Git at gEDA*

With these examples in mind, three other definitions are worth stating. Consider first the following definition for innovations by Rogers:

---

<sup>2</sup>Other popular terms for a person attempting to achieve an introduction are champion [21, 331] and change agent [436].

“An innovation is an idea, practice, or object that is perceived as new by an individual or other unit of adoption.” [436, p.12]

This definition emphasizes two things: (1) There is a subjective dimension to something being innovative, and (2) innovation is tied closely to novelty. The first point is important to enable questions about innovations at different scopes such as the individual, the project, and the Open Source community as a whole. Things which are innovative to one person might not be for another. The aspect of novelty on the other hand provides the opportunity to relate innovation to adoption. Adoption can be seen as the process by which something novel is incorporated into “habits, routines, and other forms of embodied recurrent actions taken without conscious thought” [140, p.48]. Thus, by being adopted and becoming a “daily routine” [240, p.25], the novelty wears off and the innovation process ends.

Denning and Dunham have advocated using the term innovation only once an adoption has occurred. Instead, they use the term invention for novel technologies and tools before the point of adoption to highlight the importance of achieving widespread use for any innovator [140]. For this research, I will not make this distinction because adoption is too gradual a process to allow us to draw a precise line between invention and innovation. If a distinction is possible, I will use the term innovation for ideas, practices, or objects which are in the process of being adopted or prior to it, and the term *adopted innovation* for ideas, practices, or objects which have become part of the processes of the observed unit of adoption. Invention, as the process and result of conceiving new technologies and ideas, is not the focus of this thesis and consequently not used.

Rogers’s definition—as given above—includes all things deemed novel, which in the context of Open Source projects includes new bug reports, new versions of software libraries used in the project, and new versions of the software itself. As these are not of interest for this study, the definition used in this thesis requires innovations to be concerned with the software development process.

This then ties in well with the following two definitions of software process improvement (SPI) given in the literature by the Software Engineering Institute and Fichman and Kemerer:

“The changes implemented to a software process that bring about improvements.” [385, Olson et al. (1989)]

“[C]hanges to an organization’s process for producing software applications—changes in tools, techniques, procedures or methodologies.” [184, Fichman and Kemerer (1994)]

The obvious difference between both definitions is that the definition from Fichman’s study of assimilation and diffusion of software process innovation [183] avoids taking an explicit side on the question whether a change is good or not, but rather assumes the viewpoint of the innovator who has an agenda and intention for it. This perspective is also assumed in this thesis, where the question on what constitutes success in the area of Open Source will be difficult to answer (see Section 2.3.10).

However, the focus of both definitions differs slightly from the one assumed in this thesis because they define the innovation as the change itself rather than the means for making the change. This difference may be regarded as looking preferentially at delta-increments from a given status quo (the changes) rather than looking at the new realities being created comprehensively (the innovation).

Take note that innovations might change the software development process only indirectly. If, for instance, an innovation increases social interaction between project members, which makes the project appear more active, which in turn causes new developers to join the project, this innovation might indirectly lead to increased development progress. It is thus important not to think too focused about traditional means for software process improvement in companies, but include those that might impact the software development process for instance via social and or legal mechanisms. This is particularly important since the understanding of the Open Source development paradigm is still at an early stage.



### 2.1.1 Introduction vs. Diffusion

This thesis assumes an active perspective on how innovation can be spread within a context such as an Open Source project. Hence, the term introduction was chosen in contrast to diffusion which has more passive connotations. Rogers defines the diffusion of an innovation as the “process by which (1) an innovation (2) is communicated through certain channels (3) over time (4) among the members of a social system” [436, p.11]. This definition lends itself to the observation of contexts in which the distance in the communication network is of essential importance to the spreading of the innovation. Yet, assuming the intra-project context of a single Open Source project with communication on a mailing list, the process by which an innovation can be communicated consists of writing a single e-mail reaching the whole project at once. Consequently, it is not necessary to emphasize the spread of discussion about an innovation, but instead one can focus on introduction as the sum of all activities performed to achieve adoption within this particular context.

Many important aspects of innovation diffusion research thus become less crucial when assuming such a tightly-knit sphere of communication. For instance, it is less important to consider the adoption process as a function of the number of members of a social system using an innovation over time (which reveals S-curve functions in diffusion scenarios). Preferably, the focus should be shifted on the actions preceding any adoption, such as project-wide discussion and decision making, which determine and shape whether an innovation is adopted at all.

## 2.2 Types of Innovations

For this thesis it is useful to distinguish the following four top-level categories of innovations:

- *Process Innovations*: Innovations that modify, add, remove, or reorder process steps of the processes used in the project. As an example, consider the use of a *merge window in a fixed interval release scheme*. Instead of releasing the project after a set of features has been completed, a fixed interval release scheme aims for releases at given points in time (two to six months per release are common). To achieve such regularity, a merge window is frequently introduced to restrict the time in which new features can be contributed. After the merge window closes, commits are commonly restricted to bug-fixes and localizations with the intent to encourage testing and debugging of the new features (this intention is not necessarily achieved, as short merge windows make it often necessary to directly start working on new features to be included during the next merge window). The use of this innovation thus primarily changes the process by which software is developed in the project.
- *Tool Innovations*: Innovations that involve the use of software by an individual developer. A typical example of a tool innovation is an *inline documentation tool* such as Doxygen.<sup>3</sup> This type of tool first defines a set of keywords to be used inside of source code comments. Given source code annotated with keywords of this kind, the tool is then able to generate documentation of the application programming interface (API) of the annotated software. Note that the introduction of an *inline documentation tool* is often combined with *rules and conventions* that determine when and how to add such documentation.
- *Service Innovations*: Innovations that involve the shared use of software or data by multiple developers, typically over a network connection such as the Internet. In contrast to tool innovations, the software or the shared data is not run or managed individually, but rather several developers need to access the same running instance of a software or the same set of data. Unlike tools, service innovations thus require hosting on a server which is available to other developers (see Section 5.5).

---

<sup>3</sup><http://www.doxygen.org>

As an example, consider a *continuous integration service* which is being run on a project server to detect failures to build the project and pass an associated test suite [195]. This innovation could be a pure process innovation, if each developer by convention built and tested the project locally before each commit, or a tool innovation, if a developer used tool-support for this. Yet, not sharing a continuous integration system between developers reduces the set-up effort for each developer and, more importantly, increases the reliance on each developer to actually perform the integration (this reliance might not hold, if build and test take considerable time or effort). The price the project has to pay to achieve these advantages of running this innovation is the cost for operating a public server and maintaining the continuous integration software. Note that it is an attribute of this innovation that the decentralized approach to continuous integration is possible at all and that not all service innovations can be transformed into a tool or process innovation.

- *Legal Innovations*: Innovations that affect the legal status of the software being developed or of the project as an organization. This category targets in particular the choice of licenses used for releasing the produced software to the public. As a non-license example consider the example of joining the *Software Freedom Conservancy (SFC)*—a non-profit organization that provides legal, fiscal, and administrative support to member projects. A member project can, for instance, make use of the SFC to receive money when participating in the *Google Summer of Code* or seek legal advice when license violations are committed against the project's source code [386].

As can be seen in the examples, these categories are not exclusive and innovations can be composites of these primary types. For instance, the *source code management system* Subversion as an innovation consists of client applications for use by the individual developer, a server application which is shared by all project participants, and process steps for using it.

In particular, all innovations contain at least a minimal aspect of a process innovation (otherwise they would not be innovations at all). For example, when introducing a tool which can check and correct coding style issues, it must include at least the new process step to use this tool. Nevertheless, there is a dominant type hidden in most innovations which assign it to one of the categories. In the above example Subversion is primarily to be seen as a service innovation when considering its introduction.

Three minor types exist:

- *Documentation*—Innovations that put knowledge, process definitions, rules, guidelines, or instructions into a document form.
- *Conventions*—Innovations that define rules or schemes for the execution of process steps such as how to select names for releases, how to format source code, or which kind of programming language features to use and which to avoid.
- *Social*—Innovations that are based on social mechanisms such as meeting in real life or discussing hobbies and interests beyond Open Source.

## 2.3 Open Source

This section provides the necessary overview of the term Open Source to understand the remainder of this thesis. For readers with sufficient background knowledge it is safe to skip this section, if the following central tenets are understood: (1) Open Source is primarily a term referring to a *licensing scheme* which enables free use, modification, and distribution of software source code by non-copyright holders. (2) From this Open Source licensing scheme a *software development model* of globally distributed participants, who collaborate asynchronously and with little formal structure, arises, which is called the *Open Source development model*. (3) Following this software development model, a *social movement* of participants with diverse motivations and backgrounds has constituted itself with its own norms, jargon, and identity, which is called *the (Open Source) community*.

Outline In the following, this chapter will take a short historical tour into the origins of the Open Source movement

(Section 2.3.1), settle some questions of terminology (Section 2.3.2), give an overview of the research activities on Open Source (Section 2.3.4), discuss the participants in the Open Source movement and their motivations (Section 2.3.5), and present the current knowledge about the development processes and decision making processes inside the Open Source world (Section 2.3.8).

The following sections are structured based on [119, 448, 449, 187].

### 2.3.1 Historical Introduction

The Open Source phenomenon as a large-scale and publicly visible social movement took off in the early nineties with the rise of the Internet, which enabled fast and easy communication and data transfer between hobby programmers worldwide [349]. The legal foundation of this new culture of sharing had been prepared by Richard M. Stallman ten years before in the early eighties, when working on his self-assigned mission to write an operating system which should be free for the general public to use, modify, and share [478]. Stallman's key insight was to use the existing copyright legal regime to design a software license which uses copyright to (1) endow the freedom to use, modify, and distribute software code onto all users [481, 404]—a provision called Open Source<sup>4</sup>—and to (2) forbid any future users to restrict any of these freedoms when producing and distributing modified versions [306, 439, 473]. The clever trick of using copyright to endow these privileges onto users is now called *copyleft* [308] as a play on words and in contrast to the traditional use of copyright as a means to restrict the freedoms of the users to copy, use or modify a work. Equipped with the General Public License (GPL) as a concrete implementation of a copyleft Open Source license, Stallman worked strategically, yet isolated [350, pp.23ff.], on his project he named GNU (GNU's not Unix) to collect all parts necessary for an operating system consisting exclusively of Open Source licensed software [475].

Stallman

In the early nineties this changed when Linus Torvalds, then a student of computer science in Finland, combined the enabling conditions of copyleft licensing and the growing possibilities of the Internet [351] to reach out to other fellow hobbyist programmers to collaborate on his operating system Linux [508]. This broadening of the developer base proved to be a successful change in the development model. Instead of portraying software development as the careful construction of cathedrals by wise and powerful “wizards” in seclusion, the new development model appears more to resemble a “bazaar” [417] full of babbling and opinionated young “hackers” [311] tinkering away on their private hobby projects, sending improvements and suggestions back and forth. While Torvalds used large parts of the GNU project, which at that point was missing little but the core “kernel” part of an operating system, it still turned out that this collaborative model of development of fast release cycles and incorporating feedback from users was superior to the development in isolation, and the idea of *Open Source software development* was born. As a second example arising from such collaboration on the basis of openly licensed code, the Apache project can be named. It originated from text files containing modification of the source code—so called patches—sent between users of the National Center for Supercomputing Applications (NCSA) HTTPd web server, when the development of the software stalled, because the original developer, Robert McCool, had left the NCSA in 1994 [186]. This collaboration was able to produce one of the most popular web servers to date [186].

Torvalds

The Cathedral and the Bazaar

If Stallman is representing the first generation of Open Source participants who worked relatively isolatedly in research institutions, and the hobbyist (among them Linus Torvalds with his operating system) in the early nineties is representing the second, a third generation of Open Source participants sprang to life at the end of the millennium, when the Dotcom boom started to take off. Commercial players discovered the Open Source movement, when the former leader in the web-browser market, Netscape, released its browser suite Mozilla under an Open Source license to the public as a last chance effort to prevent a Microsoft monopoly after having lost the leadership in the browser market to Microsoft [231]. From there, Open Source has become part of the strategy of many IT companies [542] (compare Section 2.3.6 on organizational motivations), transforming the hobbyist movement into a “more mainstream and commercially viable form” [187], which still appears to be growing at an

Netscape  
Mozilla

<sup>4</sup>Or *Free Software* as discussed in Section 2.3.2 below.

exponential rate in 2006 [143]. Studies by market analysts such as Gartner, Forrester or IDC Report confirm the success of Open Source in the IT industry [358, 232, 174] and the underlying idea of copyleft has since moved beyond software, in particular towards cultural production [80] with Creative Commons licensing [308], but also towards Open Hardware [476, 133] or Open Design [19].

For further reading on the history of the Open Source phenomenon, I recommend [176, 350].

### 2.3.2 Terminology

Open Source = Free Software? First, a quick note on the differences between the terms *Open Source* and *Free Software*: This thesis uses the term *Open Source* exclusively because by their legal implication the terms *Free Software* and *Open Source* are virtually identical<sup>5</sup>. At least, if we regard the short explanation of what *Free Software* entails for the Free Software Foundation [481] and which license is on their list of *Free Software Licenses*, and compare it to the *Open Source Definition* by the *Open Source Initiative* (OSI) [404] and their list of compliant licenses<sup>6</sup>, then mismatches arise only in esoteric, outdated, and so far unreviewed licensing cases. Rather, the “philosophies” conveyed alongside the licenses by their sponsoring organizations are different [161, p.24], which led to a heated discourse between their respective founders about the relative merits of using the terms [47]. Stallman, arguing in this debate for the Free Software camp, can be said to take on a position of personal and Kantian ethics [47, p.70] in which the use of the word “free as in free speech” [478] is of primary importance to convey this position. Raymond, arguing for the Open Source initiative, on the other hand maintains a position focused on “technical efficiency and [...] neoliberalism” [47, p.76], in which the political question of achieving freedom becomes subordinate to the advancement of the Open Source movement, which is hindered by the word free being easily misunderstood from a business perspective. Also, Stallman has acknowledged at times the need for this pragmatist view, in which tactical considerations of furthering the Open Source movement are put before the stark moral imperative<sup>7</sup> [139].

While one could make distinctions based on the participants feeling associated to either camp, surveys have found that only a minority of developers feel there is a substantial difference caused by this difference between Free and Open discussed above [211], a tenet which has been attributed to the focus on the technical issues at hand by many participants [47, p.82]. As a last remark to why the term Open Source was chosen as the more appropriate for this thesis, we can quote the FSF in saying that “Open source is a development methodology; free software is a social movement”<sup>8</sup> and then state that this thesis is interested in the development methodology arising from the licensing scheme and not primarily the social movement arising on top of that.

What is an Open Source project? The term *Open Source project* deserves a second remark. This label is surprising in a traditional project management sense [171, pp.111f.], since Open Source projects are not a “temporary endeavor undertaken to achieve a unique service or product” [412, p.4], but rather are (1) open ended and (2) directed at producing source code, most certainly for the use in a software product, leading us here to echo Evers’s definition of an Open Source project as “any group of people developing software and providing their results to the public under an Open Source license” [170]. In this sense a term such as *Open Source collective* would probably be more appropriate, but Open Source project stuck and is likely to stay.

<sup>5</sup>The Spanish term *Libre Software* and the combination as *Free/Libre/Open Source Software* (FLOSS) have found some use as overarching designations, which unfortunately remain restricted to academic circles.

<sup>6</sup>See [http://en.wikipedia.org/wiki/Comparison\\_of\\_free\\_software\\_licences](http://en.wikipedia.org/wiki/Comparison_of_free_software_licences).

<sup>7</sup>See for example also Stallman’s discussion on the relative merit of using the Lesser General Public License as a tactical device to increase the attractiveness of an Open Source platform for proprietary software producers to strengthen the platform as a whole [477].

<sup>8</sup><http://www.fsf.org/philosophy/free-software-for-freedom.html>

### 2.3.3 Licensing

This section expands on the licensing terms used for Open Source development. We already discussed that a license is called an Open Source (or Free Software) license, if it follows the Open Source Definition [404] or the Free Software Definition [481], which both insist on the same requirements of the license: It must permit the users of the software the following four freedoms: (1) run or use the software, (2) modify the software (this requires that the source is public), (3) share or distribute the software, and (4) also share their modified version. If a license allows these things, it is called an Open Source license.

With the increasing success of the Open Source movement, the number of licenses has “proliferated”, so that over 100 licenses are formally recognized by the OSI or FSF [187]. Also, only half of all projects use the GPL or LGPL [306]<sup>9</sup>, which makes it necessary to designate groups of licenses. In this regard, licenses following only the definition above are called “permissive”, because they grant the four freedoms and do not require any restrictions.

From permissive Open Source licenses one can distinguish “restrictive” Open Source licenses. An Open Source license becomes restrictive, if it uses the principle of copyleft [307] to require derivative work to be licensed as Open Source as well. The GNU General Public License is the most well-known representative of this license type [306]. If for example one developer developed code under the GPL and released it for everybody to use, and another developer took this code to build her own software, then the GPL would force her to release her own code under the GPL as well. Generally, one distinguishes between “strong” and “weak” copyleft licenses (strong copyleft licenses are also sometimes called “highly restrictive”). A strong copyleft license such as the GPL will perpetuate the requirement for the source code to be Open Source to all derivative works incorporating this code. Weak copyleft licenses on the other hand only make fewer such restrictions. For instance, the GNU Lesser General Public License (LGPL) requires developers who modify the original version of a software library under the LGPL to make these changes public, but not the source code of the software in which they used this library.

How can licenses be grouped?

The GPL as the most common strong copyleft license does not only require the perpetuation of the Open Source freedoms, but also mandates that the whole resulting derivative work is licensed under the GPL, an attribute which in general is called “viral” [438]. Such viral aspects of copyleft licenses might clash with provisions by other copyleft licenses—so-called license incompatibilities—thus preventing the combination in a derivative work of source code under both licenses [209].

What is a *viral* license?

The term “business-friendly” is used for licenses which permit the inclusion of code into a proprietary software product [191, p.6], which applies to all “permissive” licenses and some weak copyleft licenses such as the LGPL or the GPL with Classpath exception.<sup>10</sup>

What is a *business-friendly* license?

Some insights have been gathered by analyzing the correlation of license choice with project characteristics or project success measures such as team size. Lerner and Tirole found Open Source software targeting an end-user audience more likely to use a highly restrictive license when compared to software for developers [306], a finding which was replicated by Comino et al. based on the same data source [100]. Restrictive licenses have been found both positively [96] and negatively [485] correlated with project success, suggesting that other factors are likely responsible.

Since Open Source licensing allows anybody to take Open Source code and develop a derivative work, the question arises how Open Source projects protect against such “forking” [348] into separately developed versions or more generally against “hijacking” [386] of work products by third parties. This is not to say that forking is a bad thing. The history of famous forks contains examples where forking was beneficial for overcoming stalled development progress (GCC vs. EGCS, NCSA httpd vs. Apache), as a temporary solution (glibc vs. Linux libc) or when incompatible goals emerged (FreeBSD, OpenBSD, NetBSD) [348].

How to protect against hijacking or forking?

<sup>9</sup>Daily statistics of license use in Open Source projects can be obtained from <http://www.blackducksoftware.com/oss/licenses/>.

<sup>10</sup>See for instance <http://codehaus.org/customs/licenses.html>.

Research shows that many of the more successful Open Source projects use several tactics to protect against forking and hijacking of code by commercial entities [386]. The most important is the use of the GPL as a copyleft license, which ensures that code will be publicly accessible and can be reintegrated into the project which was forked [348]. Other defenses are the building of a well-known brand name, the incorporation of the project under a legal umbrella organization for litigation purposes, or the use of trademarks [386].

For a primer into Open Source legal affairs for the practitioner see [191], for an in-depth tutorial consult [439], for reference purposes [473], and for legal commentaries [33, 338, 248].

As a last point: The clever trick of subverting copyright to establish a regime of freedom and sharing, based on the zero marginal cost of copying, is frequently seen as Richard Stallman's genius stroke, but Eben Moglen as one of the authors of the GPL Version 3 and a legal counsel to the FSF draws our attention to another implication of the digital age [349]: Since all information in the digital age is essentially nothing but bitstreams, the legal universe of copyright, patents, and trade secrets would be more and more based on distinctions which no longer match the physical world and thus become arbitrary. From history Moglen then argues that such a legal regime must likely fade and be replaced by something which more closely resembles reality [349].

### 2.3.4 Research on Open Source

Before we step through the most important insights regarding Open Source, here a quick overview of how the research unfolded. The starting point for interest in Open Source as a research phenomenon can likely be traced back to Raymond's Cathedral and the Bazaar [415] published in 1998. Credited also to have triggered Netscape to open source their browser [231], Raymond catapulted Open Source into the public interest. In 1999, we then see the first publications on Open Source, yet mostly from participants in the community [144], such as Roy Fielding from Apache [186] or Eben Moglen from the FSF [349], or as reactions to Raymond's ideas [50]. In these early days First Monday journal was the most important publication venue [415, 68, 349, 50, 351, 295, 511]. The potential for research due to the public nature of Open Source development was realized early on [237], and after the first scientific studies, such as [554, 346, 170], were published,<sup>11</sup> there was soon an explosion in publications [119].

Most important to open the research area were a number of case studies such as on Apache [346], Mozilla [347, 422], FreeBSD [273], and Linux [351, 295],<sup>12</sup> which generated the first empirical insights about Open Source, and two publications from Lerner and Tirole [305] and Feller and Fitzgerald [176], which formed the base on which much of the further work built. From there, the focus of research first moved to uncover the motivation [238, 211, 252, 555, 234, 298, 53, 552, 172] and demographics [211, 132] of the participants. Also, a workshop series was started co-located with the International Conference on Software Engineering by Feller et al. [181, 177, 178, 179], which was complemented beginning in 2005 by the International Conference on Open Source Systems (OSS) first organized by Scotto and Succi [458]. Already at this early point it also became visible that Open Source is not only interesting for the software engineers and computer scientists (for instance [326, 347, 271]), but also for economists and management science (early work includes [530, 305, 532, 297, 270]) or from a sociological or literary perspective (for instance [44, 47, 316]) [119].

Methodologically, case studies and surveys dominate early research [119] until tools for "crawling" project infrastructure and analyzing larger number of projects became more widespread, enabling a large number of "mining" studies (see [276] for an overview and [326, 259, 558, 321, 401, 228, 327, 563, 208, 100, 76, 286, 143] for notable results) and some using Social Network Analysis (e.g. [326, 321, 113, 469, 223]). Research and discussion on methodology has increased and now includes for instance discussion on research frameworks [442], on process modeling [318, 268], on visualization of communication [381], or on the notion of success in Open Source development [166, 484, 114, 112, 286].

<sup>11</sup>Bowman et al. beat others to the first paper on Linux at ICSE'99, but only discuss the kernel from a technical perspective [61].

<sup>12</sup>Later and less important case studies were published on Gnome [207] and NetBSD [149].

We can distinguish the following five major research topics beside those already mentioned: (1) Software Development Process [170, 271, 171], focusing on decision making [313, 116], bug fixing [118], reuse [226], architecture and design [23, 26, 18, 21, 24, 22], quality assurance [483, 502, 561, 290], release management [168], innovation adoption [377, 293], documentation [354], knowledge management [454, 441], tool usage [554, 462, 430, 300], role migration [269], requirements management [447], usability engineering [363, 514, 14], (2) licensing [486, 338, 306, 487, 33, 137, 96], (3) firms and Open Source, focusing on business models [38, 241], interaction between firms and Open Source projects [542, 122, 543, 353, 43, 388, 544, 247], and comparisons to closed source development [347, 401], (4) socialization and sociology [44, 117, 160, 386, 161, 47, 162, 153, 387, 115, 92, 461, 413, 437, 139, 159, 163], (5) optimizing the Open Source process [147, 454, 392, 390, 379, 255, 501].

Three literature reviews provide good overviews of the growing body of research [119, 448, 440]. For a discussion of the research methods used and projects under study refer to [119, 490, 489].

### 2.3.5 The Participants

The best current assessment of the Open Source participant population is given in the survey results of the FLOSS project [211]. The data indicates that the Open Source participants are exclusively male (99%), young (70% between 20 and 30 years of age), mostly affiliated with computer science from their school or job background (33% software engineers, 16% students, 11% programmers, 9% consultants), [211] and live in North America (41%) and Europe (40%) [217]. 70% of the respondents of the survey answered that they spend less than 10 hours per week working on Open Source projects, 14% spend 11–20 hours and 9% can be considered working more than half-time on Open Source with spent hours per week between 21 and 40 [211].

Who is participating?

One of the biggest research topics in the area of Open Source has been the question why participants would spend their time on collaborating on Free Software projects [298, 210, 238, 252, 172, 461]. Answers to this question were first sought using theoretical means [176] and surveys [238, 252, 298, 210], and more recently using interviews [461] and case study research [172]. They resulted in the following reasons, whose relative importance both for joining and staying is still under discussion [552]:

What motivates individuals?

- Learning and sharing skills [210]—Knowledge-based motives have been found to be the most commonly named motivation of joining and staying in the OSS community.
- Developers are frequently users of their own software [211, p.21]—This motivation is most commonly known as “Scratching an Itch” [415] or less frequently so as “user-programmers” [53], in which software developers contribute to or start Open Source development to satisfy a particular personal need, just as Linus Torvalds did with developing Linux [508]. While user-programmer motivations have been given by a third of all participants when surveyed [211, p.45], more recent research by Shah emphasizes that personal need is mostly an initial motivation and that a shift to other motives needs to occur for the participant to stay in the community [461].
- Making money—Lakhani and Wolf found 40% of survey respondents to be paid [298] and Ghosh et al. report making money as a motivation for 12% of participants [211]. Yet, the actual numbers are likely to be highly project-specific. For instance, [235] found only 10%–14% of participants in various Apache projects had a job which involves contributing to Apache. Berdou reports on two surveys among developers of Gnome and KDE in which 51% (101 of 199) and 45% (29 of 64) of formal project members respectively stated that they were being paid to work on Open Source in general and 37% (73 of 199) on Gnome and 38% (24 of 64) KDE in particular [43].
- Career advancement, improved job opportunities, signaling to possible future employers [305]—These motives recast the previous motivation of making money into the future. It is interesting to observe that while learning has ranked highly in surveys, improving job opportunities has not [211, 552].

How many developers are being paid for contributing?

- “Just for Fun” [508, 323]—Such hedonistic “homo ludens payoff” in economic terms [53] might, for instance, result from becoming immersed in the “flow” [121] of programming or the joy of mastering the technical challenges of a problem [323].
- Gift-Culture (reciprocity) [44]—This motivation derives from the insight that giving away a gift can cause the recipient to feel obliged to give something in return [333]. One possible interpretation for Open Source participants appears to be that participation might be given away freely to gain social status and reputation [53] (see below). Moglen has criticized this interpretation for just being a reformulation of the “proprietary institution” of “market barter”: “Free software, at the risk of repetition, is a commons: no reciprocity ritual is enacted there. A few people give away code that others sell, use, change, or borrow wholesale to lift out parts for something else” [349]. Still, some “moral economy” [79] might be at work for those 55% surveyed by Ghosh et al., who state that they take more than they give, to close the gap to those 9% who feel that they give more than they take [211, p.50].
- Altruistic motives—In particular stressed by Richard Stallman as a strong motive to participate in Open Source is the belief in the ethical imperative of keeping software free and Open Source to anybody [478, pp.89ff.] as a core tenet of Open Source ideology. Raymond on the other hand derides such pure altruistic motives as just “ego satisfaction for the altruist” [417, p.53].
- Social identity and social movement [252, 159]—When regarding Open Source as a social or computerization movement [159] which attracts those with a “Hacker” social identity [311, 419, 97] and believing in Open Source ideology [487] (see next section), the desire to participate in this movement and new form of collaboration has been given by 30%–37% of survey respondents in [211].
- Self-determination and ownership, reputation and status—Other motivations have been mentioned for the participation in Open Source development, such as positive feeling gathered from a sense of ownership over a software development project or the desire to gain reputation among other software developers for one’s contributions. Hars and Ou for instance found 80% of developers stating self-determination as an important motivational factor [238], which unfortunately was not explored by other studies. While reputation is a common extrinsic motivation found in the literature [119], when asked directly, reputation was given as a motivation to join and stay in the community by only 9–12% [211].

To further delve into the question what motivates (and makes) the hacker in every Open Source participant, I can recommend Coleman’s Ph.D. thesis on the social construction of hacker culture [97].

Grouping into technical, social, and economic factors [176, 57], or into norm-oriented, pragmatic, social/political, and hedonistic motives, as well as the distinction between extrinsic and intrinsic motivations [53] have been proposed to structure this set of results. Further, one interesting question arises from the motive of making money: How can money be made from Open Source software development, if users can redistribute the software to others for no charge? And given part of the answer has been found to result from individuals being paid by companies<sup>13</sup>, which should assume a utility-oriented perspective: What motivates companies to devote capital to the development of Open Source?

For individuals, Hann et al. found in their study of economic returns from Open Source participation in an Apache project that moving from the rank of committer to being a member of a Project Management Committee or the Board of Directors was positively correlated with wages, while the number of contributions and more generally open source development experience was not [234].

What motivates companies? For companies, the following motivations, business models, and strategies have been found or postulated [418, 38, 241, 122, 472]:

<sup>13</sup>See [247] for a discussion on the attitudes of paid developers compared to volunteer participants and principal-agent problems which arise due to relative independence of the developers.



- Companies can be users too—Companies in particular might be tempted to cut costs by using/reusing software and source code from Open Source projects to satisfy corporate needs [122]. A common example is the use of Linux as an operating system for routers, which is adapted by manufacturers to their particular hardware, and who are then obligated under the GPL to reveal the changes made [258]. More generally, companies might adapt most of the motivations for individuals, such as gathering reputation, which might then be called marketing, or learning, which might be labeled training. This line of analysis is elaborated below in Section 2.3.6.
- Crowdsourcing [256] of development—This is the prospective counterpart of the previous motivation: A company releases software as Open Source because they hope to benefit from future collaboration with the Open Source community.
- Opening parts—Under this strategy, a company waves control of layers of a platform stack while maintaining control over more important ones [542, 38]. A manufacturer of hardware might for instance release the operating system or drivers as Open Source to make the platform more attractive while maintaining profits via hardware sales [504].
- Partly open—Instead of opening a particular layer in an architecture stack, an alternative strategy can be to open a software product in a way which prevents commercially interesting uses by competitors [542], for instance by dual licensing schemes [527].
- Complementary products and services and loss leaders—Companies might release software openly and offer products such as proprietary server software to an Open Source client [418] or services such as support, consulting, training, documentation, maintenance, administration, integration, merchandise, or hosting of the software as complementary offerings to the software [288, 418] by moving down the value chain of software development [472].

Given these motivations, the amount of influence companies wield via paid developers usually is categorized somewhere between the extremes of “sponsored Open Source projects”, in which the sponsoring organization maintains full control, and “community controlled” projects, which are independent from influence of, for instance, a company or university [544, 543]. Another categorization of influence has been proposed by Dahlander and Magnusson, who distinguish symbiotic, commensalistic<sup>14</sup>, and parasitic approaches and list their relative advantages and managerial challenges [122].

Whatever the motivation of companies, Open Source is no “magic pixie dust” [559]; on the contrary, the creation of a community is hard work, as the story of Mozilla can exemplify [231].

### 2.3.6 Open Source and Organizations

The goal of this section is to explain in further detail what we know about users of Open Source software. The section focuses on organizations such as government agencies, companies, non-profit organizations, and academic institutions [455], because individual users have not received much attention beyond their potential to be developers and field testers. This discussion is included here because the number of users who do not contribute to projects is a magnitude larger than the one of those who do [347].<sup>15</sup>

Such adopters of Open Source software must focus on economic utility and consider the total cost of ownership (TCO, such as the cost of training or maintenance) more important than license costs alone [139, 522]. Ven et al. interviewed 10 Belgian organizations about their adoption of Open Source software and found 5 relevant factors: (1) cost, (2) source code availability, (3) maturity, (4) vendor lock-in, (5) support [522]. Yet, organizations make contradictory claims for all factors, for instance citing Open Source software both as reliable and unreliable. Some of these contradictions can be resolved

Why would organizations adopt Open Source?

<sup>14</sup>In a commensalistic approach the company tries to gain benefits while avoiding inflicting harm on the community.

<sup>15</sup>Rossi et al. surveyed 146 Italian companies with an Open Source business model and found that only 54% of even these participated in at least one OSS project [55]. Ven et al. interviewed 10 Belgian organizations using Open Source software and found that none had contributed to the projects despite two of them building applications on top of Open source solutions [522, pp.56f.].

when acknowledging that attributes such as software quality and maturity are not per se deductible from the Open Source mode of production, but rather are dependent on each individual project and the software it produces [522].

Particularly interesting is the contradiction regarding source code availability, which has been called the “Berkeley Conundrum” [176], asking the question what is the big deal about source code, if nobody uses it. Ven et al. find that indeed the barriers of skill and time for modification of source code make many organizations perceive Open Source software equally as a black box as proprietary software. In addition though, a second group of organizations derived trust in the software from the availability, despite not making any use of it. Only 3 of the 10 Belgian organizations reported that they used the availability to improve their understanding of the software for building applications on top or customizing the software [522].

Beside these 5 decision factors, the Open Source literature identifies additional soft, managerial [218] or institutional factors which are important for Open Source adoption. A structural perspective can be exemplified by the migration of the administration of the city of Munich to Linux that required considerable path breaking to move away from the existing Microsoft software ecosystem [150]. Dalle and Jullien as a second example discuss the importance of local networks of Linux usage to counteract the network externalities arising in the dominating network of Microsoft Windows usage [125]. Lin and Zini discuss the adoption of Open Source in schools and find that beside cost also a reduction in piracy and the possibility to use Open Source for constructive learning were of importance [315].

Whatever the reasons for adopting Open Source, organizations face new challenges once they use Open Source. In comparison to commercial software providers, Open Source projects rarely operate on schedules and can guarantee new features or fixes to be available in a certain release [147, p.84]. To overcome such problems, organizations can either acquire support from Open Source vendors or pay project participants directly [522, 455]. O'Mahony also points to incorporation of Open Source projects as a direct result of the emergence of organizational and corporate interests towards a project [387].

### 2.3.7 Community Ideology

What is believed? From the *who* and *why* on our way to the *how* of Open Source software development, we should take an intermediate step and consider ideology in this section. Following Trice and Beyer, ideologies are “shared, relatively coherently interrelated sets of emotionally charged beliefs, values, and norms that bind some people together and help them make sense of their worlds” [509]. More forcefully put, one could say that ideology is one of the main causal inputs for explaining the nature of Open Source development [161].

The work by Elliot and Scacchi [160, 161, 162, 159, 163] puts Open Source ideology into a context from four related perspectives: First, communities of practice as loosely coupled groups of voluntarily participating members [301] with their shared goals, interests, and beliefs deriving from working in a common domain provide a suitable model in which ideology can arise [161]. Second, taking a view of Open Source projects as virtual organizations, ideology might be derived via their respective organizational culture [160]. Third, by focusing on the development of software as the central activity, the theoretical frame of an occupational community and its culture can be adopted [160]. These three perspectives include an integrative view on shared goals, interests, beliefs, and activities, but differ in the relative importance they put on the individual aspects [109]. Last, the authors consider Open Source a computerization movement, i.e. a social movement whose adherents attempt to affect social change by computer usage, and trace back its creation through the stages of social unrest, popular excitement, formalization, and institutionalization to highlight the importance of a shared ideology and a shared opposition [159, 163].

Stewart and Gosain [486, 487] provide an ideology based on the Open Source seminal and academic literature using the virtual organization perspective as a backdrop. For this introduction, we include additional aspects from Elliot's and Scacchi's work who employed Grounded Theory Methodology [492]

on data from and on the Open Source community, such as IRC chat transcripts, websites, interviews, and OSS literature to build a similar OSS ideology based on the occupational community context, but which is less detailed than Stewart's and Gosain's.

Following the definition put forth by Trice and Beyer, ideology is seen as the composition of beliefs (assumptions of causal mechanisms), values (preferences regarding behaviors and outcomes), and norms (behavioral expectations) [487, p.292]:

- The authors find strong beliefs in the superiority of outcomes resulting from (1) the Open Source development model, (2) source code being licensed under an Open Source license, (3) transparent and open processes, as well as the beliefs (4) that more people will improve the quality of a software product, (5) that practical work such as writing code trumps theoretical discussion<sup>16</sup>, and (6) that status is gained by the recognition of others.<sup>17</sup> To this we might add the concept of a "Rational Culture" [554, p.335], i.e. (7) the belief in the superiority of rational, emotionally uncharged discussion.
- They next deduce (1) sharing, (2) helping, (3) learning, and (4) cooperation as valued behavior, and (5) technical expertise and (6) reputation as valued outcomes of activities of the Open Source community. One might want to add (7) having fun [508] to straighten the balance between mastering technical challenges and volunteers plainly enjoying what they are doing, or as Moglen puts it: "Homo ludens, meet Homo faber" [349].
- Last, they discuss Open Source norms, in which they include (1) to not fork, (2) to contribute through established channels such as the maintainer, and (3) credit people for their contributions. From Elliot and Scacchi we can add (4) to immediately accept outsiders and (5) to use informal means of management.

### 2.3.8 The Open Source Development Process

To understand the way Open Source software is produced, it first needs to be stressed that there is no such thing as *the* Open Source development process or model [171, 316]. Open Source Software development is too diverse a phenomenon to capture due to three factors [200]:

How is Open Source developed?

- While there is a clear definition of what constitutes software as Open Source licensed, the way that software is developed is the choice of those who produce it. Thus, an Open Source project, following the definition given above as a group of people producing Open Source software, can choose any development process it seems fit. For instance, Raymond noticed that the Open Source movement at large started initially in a "cathedral" style of developing with "individual wizards or small bands of mages working in splendid isolation" [417, p.21], concentrating on a masterful cathedral, and only later moved to what he calls the "bazaar", babbling and chaotic, leaderless and ever-changing, as the common paradigm for development [417]. Worse, project leaders and participants are free to change the way they work [76] and might do so as the project matures. Stewart and Gosain, for instance, found that projects in early stages of development are less likely to profit from a large developer pool for finishing tasks in the issue trackers [488] and thus, for instance, might want to restrict participation for "closed prototyping" until the project's architecture has sufficiently settled [271]. Concluding, care must be taken to not confuse observed behavior (one might even find a project using formal specification for instance) with process elements which are part of Open Source development.
- As discussed in the previous sections, there are different types of participants with diverse motivations, goals and ideologies, different types of licenses, and there are also different types of projects: From the plethora of small individualistic hobby projects [561, 85] to the well-known and established foundations such as Gnome and Apache [387], to the projects sponsored and

<sup>16</sup>Moon and Sproull cite the Linux Kernel FAQ with "A line of code is worth a thousand words. If you think of a new feature, implement it first, then post to the list for comments" [351].

<sup>17</sup>The authors group belief 1 and beliefs 4–6 into *process beliefs* and beliefs 2 and 3 into *freedom beliefs*.

controlled by companies [543, 544] looking for crowdsourcing incentives [256, 426], all these development efforts are to be brought together under a single roof.

This causes differences in the processes used by these projects, for instance in role and role migration [269] and release management [168].

- Last, the methods for capturing and presenting software development processes appear to have difficulties with dealing with the many dimensions such as behavior, social, technical, and legal aspects, which come together to define what Open Source development is.

While these points highlight the difficulty of a unified perspective, there is still a set of commonalities besides the publishing of software under Open licensing terms, which gives rise to something worth labeling “the” Open Source development model. The most central two are [66]: (1) Open Process—Project members communicate using asynchronous, low-bandwidth, “lean” Internet-based media such as mailing lists and forums [554], which are publicly visible to all interested. (2) Open Collaboration—The project is open to new members joining, and uses infrastructure tools and technologies to facilitate collaboration, such as source code management systems, bug trackers, project websites, mailing lists, IRC channels, wikis, etc. [190, cf.].

Other commonalities such as *open deployment* on platforms consisting of OSS software, *open environment* using Open Source tools such as compilers or revision control systems [462], and *open releases* in short intervals have been proposed [66], but are less important for the Open Source development model.

What causes the development process? In the following, I consider Open Source development as emergent from the combination of multiple factors such as licensing, infrastructure and media use, or ideology which I will discuss in turn. I believe this is the more natural approach, even if today these factors are often consciously set in a way to follow the Open Source development model.

The most important factor giving rise to the Open Source development model is the use of Open Source licensing [137]. Given that software is published under an Open Source license, third parties can freely download and modify the software at will, possibly sending their changes back to the project. This is likely to be the main reason why participation in Open Source projects is loosely coupled with participants joining and leaving at any time, but has more implications: Mockus et al. found that for successful software the freedom to modify and contribute changes was utilized by a group of people which was a magnitude larger than the core team producing the software, and that a group another magnitude larger reported problems without contributing code [347]. Thus, by being used and allowing the user to modify the software, a hierarchy of participation naturally arises between those who just use the software, those who spend time to report problems, those who invest time in fixing the problems they experience and those who lead the project, which is called the Onion Model of Open Source participation [555]. This hierarchy arises from the actions of the individual participants who over time can advance into the center of influence for the project. While many such role changes can be performed by participants on their own by volunteering for the roles, other changes can only be made in interaction with other project members. In particular, attaining those in the center of the project must be granted or assigned [269]. For instance, project leaders might have to grant commit access for a participant to become a developer [113].<sup>18</sup> Many different paths for role migration have been found [269], but advancement based on merits deriving from contributing to the project is believed to be the most important one [555, 186]. Commonly, the following roles are being used to express the relative position of an individual in a particular project in the onion model [200, 269, 113, 555], ordered here from the outside in:

- *(Passive) user*—A user is a person who runs the software provided by the project.
- *Lurker or observer*—Lurkers follow the communication of the project passively without writing messages themselves [366, 411]. Studies on the naturalization of participants into an Open Source project have found lurking to be an important part of successful joining [153, 533].

<sup>18</sup>See also the discussion on equal and open access 8.2.

Activity	Role					
	User	Lurker	Active User	Contributor	Developer	Core Developer
Uses Program	✓	✓	✓	✓	✓	✓
Reads Mailing list		✓	✓	✓	✓	✓
Discusses			✓	✓	✓	✓
Contributes				✓	✓	✓
Has Commit Access					✓	✓
Has Large Influence						✓

Table 2.1: Different roles and their commonly associated activities or rights [200].

- *Active user or mailing list participant*—An active user reads and writes on the mailing list of the project. Participants in this role might seek assistance to problems they experience or provide such user-to-user assistance to others [297], report bugs [561, 118] or request features to be implemented [22].
- *Co-Developer or Contributor*—A co-developer or contributor is a developer who contributes to the project any kind of artifact, but does not have direct access to the repository of the project. The term *periphery* is commonly used for co-developers and less engaged users [555]. It carries connotations to Lave and Wenger's theory of legitimate peripheral participation [301], which has been proposed as a model for how role advancement could be explained [555, 153].
- *Developer or Committer*—A developer is a person who has write-access to the repository to contribute source code to the project. The term committer is used as a synonymously to highlight this fact. In high profile projects such as Apache, where obtaining commits privileges is more difficult, the term developer might be used to refer to co-developers [10].
- *Core Developer*—A core developer is a person who is well respected within the project and has large influence on the direction the project takes. Commonly, core developers can be defined as those who contribute the majority of source code lines to the project [347].
- *Maintainer, Project Leader, Project Owner*—These are commonly core developers who have administrative control over the infrastructure of the project [229] and wield the most influence in the project. The project's initiator commonly belongs into this group as well.

An overview of the activities associated with each role is given in Table 2.1, which also highlights that roles closer to the core are believed to include the activities of outer roles [200]. These are the roles common to Open Source development, yet projects also frequently assign build engineers, release managers, feature requesters, or smoke test coordinators [269].

It should be stressed that the onion model represents the magnitude of contributions and their type as well as the influence and control over the project, but not project communication (see Section 6.3 on Social Network analysis). Also, roles are local and situational [521, p.191] rather than global, i.e. a person who contributes to the project during one release cycle might refrain from doing so in another.

A second important aspect of the Open Source development process which we can derive at least in part from the loose participation caused by the license use is the preference for *informal* processes [447]. In traditional software engineering it is hard to imagine how development could be structured if not by some formalisms for at least requirements engineering. Yet again, Open Source licensing grants its users the right to do what they want rather than to stick to formal rules, which is well summed up by Bergquist and Ljungberg:

“[...] the community is primarily a loosely coupled network of individuals with no organizational forces in terms of economy or management that can force other individuals to behave in a certain way.” [44, p.310]

How formal is development?

For requirements engineering for instance, a discussion on the mailing list or enumeration of desired features in the project wiki appears to be preferred [447]. Still, Open Source development knows some formal process elements such as coding guidelines, hacker guides plus how-tos [190], request for comments (RFCs) [62], extension protocols such as from the Python or XMPP communities [24], and (open) standards [268] such as POSIX and the Linux Standard Base [502], but their use is driven by pragmatism and becomes more common as projects grow institutionalized [187, 422].

One might argue that if traditional software engineering emphasizes formal ways to structure software development, yet recognizes how important the informal meetings “around the water-cooler” [249, p.86] are, then Open Source development attempts to maximize the use of such informal means and strives to remove all hurdles to informal communication [249, cf.p.90]. This is done primarily by reducing the cost of initiating contact via the creation of a place for casual, everyday contact on the mailing list and using Open-Communication as a core principle [66]. Another way to put it: You can develop software without formal means, but you cannot do so without informal ones.

What impact  
does  
infrastructure  
have?

A second mechanism by which Open Source development might be explained to emerge is the use of common project infrastructure tools, such as source code management systems, bug trackers [339], project websites, mailing lists, IRC channels, wikis, etc. [190, cf.]. Unlike a top-down process where in reality often “thick binders describing the organization’s software development method (SDM) go unused” [430], the emergence of process from tool usage is an alternative strategy combining informalisms and structure. Adopting Lessig’s *code is law* perspective, i.e. the insight that IT systems can enforce certain behavior just as laws do [307], it can be argued that the use of these tools brings about certain ways of developing [300, 430]. For example, bug trackers such as Bugzilla prescribe certain workflows based on the bug statuses they support [339, 430].

The most striking example of technologically-implicated software process is the hierarchy of control which arises from the use of version control systems [300]. This hierarchy can be interpreted to give the onion model’s inner core its structural power and also drives the emergence of heterarchies when scaling projects to large number of participants such as in Linux [260] (this is further expanded in Section 5.7 on forcing effects).

A third way to explain the rise of the Open Source development model can be linked to the software architecture. Following Conway’s law in that the architecture of a software application will follow the organizational structure [102], it is no surprise that the independence of volunteer participation should make loosely coupled module and plug-in-architectures popular in Open Source projects [507, 41]. Conversely, one might argue that structuring the software appropriately serves to coordinate and organize the development process [395] and thus has been thought to be one of the central concerns for the chief architect [395], project founder or benevolent dictator [420] in charge of it.<sup>19</sup> An example for structural influence is the Mozilla project, which consciously increased modularization to open the project [324] after its initial attempt to attract contributors failed due to the complexity of its code base [559].

What is an  
OSS  
‘ecosystem’?

The argument that architecture has much influence on the structure of a project’s development process can also be viewed at a larger perspective: Given that Open Source participants usually spend only limited amounts of time on developing [211] and that it is important to achieve a “plausible promise” [417, pp.47f.] quickly to bootstrap development, there is widespread reuse of code, Open Source components, and libraries [226]. This can be well exemplified with how Linus Torvalds made use of the GNU project to bootstrap his Linux project with the necessary tools and command-line utilities to be run on his operating system kernel [508, Chap.8]. The importance to leverage such an Open Source “ecosystem” of related projects has been discussed in particular by Jensen and Scacchi [268]. The authors describe how Open Source projects such as Mozilla, Apache, and NetBeans are interdependent via boundary objects such as their community and product infrastructure, resulting in patterns of integration such as NetBeans and Mozilla developing a spell-checker module together, and conflict, such as NetBeans developers clashing with limitations of the IssueZilla bug tracker which was originally developed by Mozilla as Bugzilla, but forked to be included in the web portal software offered by Collab.Net. As

<sup>19</sup>Baldwin and Clark offer a game-theoretical discussion of these options for the architect and potential module developers [18].

a second example, Stahl notes the importance of an existing ecosystem of tools and applications by comparing the lively Open Source community in bioinformatics to the difficulties the Open Source chemistry movement has to get off the ground [474]. As a last example, Barcellini et al. mention the ecosystem around the programming language Python containing in its center the programming language core and the standard libraries, which is surrounded by the module space of auxiliary components and libraries and then applications domains such as web development and bioinformatics, each with a sizable community of thematically related projects [26, 22, 21].

A last origin for the nature of Open Source development has been proposed in the notion of *social capital*, i.e. the intangible social connections that individuals build by module ownership, voting, shared peer review, or gift-exchange [448]. In the Linux project, for instance, there is no central repository to which all contributors commit their code, but rather all developers use independent decentralized repository clones. It is only by a social convention that Linus Torvalds's personal repository contains what might be called the "official" Linux code [528].

Given these legal, technical, and social factors which can be seen as giving rise to the Open Source development model, we can now follow the structure proposed by Crowston et al. and discuss how individual phases in the software development lifecycle are handled under the Open Source model:

What is known about the development lifecycle?

- Requirement engineering and planning—As discussed above, requirements engineering is frequently done in an informal way by relying on discussion, to-do lists, and road-maps on mailing lists as well as wikis [447, 379, 554]. Planning similarly is rather agile and flexible with only 20% of projects using a document to plan releases according to a survey among 232 developers [561]. It emerges from the activities and intentions of the individuals in an adhocracy [343] rather than being dictated from above. It is based on "what is done is done" and "if you want it fixed, do it yourself" [213]. While benevolent dictators and project leaders such as Linus Torvalds of Linux or Guido van Rossum reserve themselves the right for unilateral decisions [92, 420] and project direction, they will have to "speak softly" [416] to prevent alienating their project members.
- Design—When Bowman et al. looked at the Linux kernel code in 1999, they did not find any documentation on its design [61]. Bezroukov—known for his snide remarks on Open Source in its early days [50, 49]—cites the approach on design in Linux as "having a tail-light to follow".<sup>20</sup> He alleges that Linux is only following well-established standards such as POSIX, with well-known implementations such as FreeBSD and Solaris [50, 335]. Even the spat between Linus Torvalds and Andrew Tannenbaum over the better architecture for designing an operating system appears to indicate that much: Torvalds—then still a student—favored the pragmatic and proven monolithic design, Tannenbaum—the professor—the experimental but promising micro-kernel approach [500].

Little is known on how design works in Open Source projects, and the time when the Open Source world could still be described as following the Unix philosophy of small tools combined with filters and pipes [203, 204] has passed. Zhao and Elbaum found design to be largely informal and undocumented with only 32% of projects using some form of design document. In Section 4.3 the work by Barcellini et al. is discussed, who provide the largest body of research on design discussion by studying the process for extensions to the Python programming language.

When moving closer to the implementation level, Hahsler found in a study of 519 projects from SourceForge.net that 12% of the 761 developers who added or modified more than 1,000 lines of code used the name of a design pattern [202] at least once in their commits.

- Implementation—Mockus et al. were first to discover the long tail distribution of code contributions to Open Source projects in their study of Apache and Mozilla, in which a small fraction of the project participants produce the majority of the source code [347], which has since been found

<sup>20</sup>The idea that Open Source projects only follow the lead established by proprietary software vendors was first made in the internal Microsoft memos on Open Source [517] leaked in October 1998 to Eric S. Raymond, who released them after editing over the Halloween weekend, which gave them their name as "the Halloween documents" (<http://catb.org/~esr/halloween/>).

to hold in other intensely studied individual cases such as FreeBSD [149] and Gnome [287], and when mining repositories of many projects [228].

In the same study the authors also discussed how code ownership is used in Open Source projects to coordinate development [347]. They report that code ownership was based on the “recognition of expertise” in a particular part of the source code rather than a formally or strictly assigned exclusive area of work. Spaeth found in a social network study of 29 medium-sized projects that the average number of authors per file was rather low at 2.4, indicating that files can be said to be owned by one or two developers [469]. Similarly, Krogh et al. found the average number of modules edited per developer at 4.6 modules in a 16 module project, indicating specialization [533]. De Souza et al. have discussed how such ownership is affected by the waxing and waning participation causing the transformation of ownership structures such as core/periphery shifts or the project moving to or from domination by a single author [138, p.202].

Last, Crowston et al. have studied how tasks get assigned in Open Source projects and found self-assignment to be the most important mode [116].

How is quality  
assured?

- Quality assurance (QA) and maintenance—QA in Open Source projects rests strongly on the assumption that finding and correcting defects can be parallelized efficiently, a notion called Linus’s Law and paraphrased as “Given enough eyeballs, all bugs are shallow” [417]. In this quality assurance model of field testing or peer review in the wild [526], users of the software can report software failures they experience to the project as bug reports or use the availability of source code to investigate software failures on their own. Crowston and Scozzi looked at the bug trackers of four Open Source projects and found that users fixed on average one fifth of all occurring bugs in this manner [118]. Zhao and Elbaum report 44% of respondents in a quality assurance survey thought that users found “hard bugs” unlikely to be found by developers [561]. Mockus et al. concluded that only 1% of all Apache users file bug reports [347], but this appears sufficient for users to uncover 20% or more of all bugs for more than half of all projects [561]. As with implementation, coordination in quality assurance is done implicitly, with less than 10% of bugs having been assigned during the bug-fixing cycle [118], but including considerable communication: Sandusky and Gasser report that 61% of bug reports in a randomly drawn sample of 385 bugs undergo negotiation between reporter and project participant [445], in particular to make sense of the bug [394, cf.] and discuss viable design options.

Code reviews<sup>21</sup> [483] performed by fellow project members before committing (*Review-Then-Commit*, RTC) to the repository or based on notifications after a commit (*Commit-Then-Review*, CTR) [10] are common, yet most projects do not have formalized processes for this [483, 186]. Unit and regression testing is sometimes used, but suffers from the lack of specification [502]. Yet, depending on the problem domain testing and code review usage may vary widely: Koru et al. report that 78% of 106 biomedical Open Source projects stated they used unit testing and only 36% peer reviewed code more than half the time before a release [290].

Østerlie and Wang have explored how maintenance in the Open Source distribution Gentoo works and found it to be a cyclic process of negotiating and sensemaking between users and developers who strive to resolve ambiguity about what has caused the defects based on existing infrastructure and its limitations [393, 394].

Research has also highlighted the role of users helping each other to resolve those kinds of failures which are caused by misconfiguration rather than defects [297].

- Release Management—Approaches may vary from creating a snapshot copy of a particular version of the project repository every now and then and uploading it to the project’s web server to “complex combination of subprocesses and tools” [168]. For well-established projects such as Linux, Subversion, and Apache httpd, Erenkrantz lists how these projects assign release authority, choose version numbers, perform pre-release tests, and approve, distribute, and package releases [168].

<sup>21</sup>The terminology here is a little bit confusing with peer review sometimes being used for code reviews as well.



Raymond has stressed the importance of frequent releases to give users a chance to participate in Open Source quality assurance [417]. Together with the continuous work done at Open Source projects and the increasing importance for regular releases for commercial consumers of Open Source software, this has led to the increased adoption of *time-based releases* over feature-based releases [43, p.88] (see Section 8.1.4).<sup>22</sup>

More formal attempts to capture the development processes of Open Source projects have been made by Jensen and Scacchi [268], Evers [171], Dietze [147], and Lonchamp [318], using means such as rich hypermedia, process flow modeling and process ontologies [268, 269], activity and class diagrams from the Unified Modeling Language [371, 147], multi-perspective modeling [171] or the Software Process Engineering Meta-Model [370, 318].

For further reading: Fogel's guide to Open Source development is written from the practitioner's perspective [190], and [119, 187] cater to the scientific mind.

Last, the Open Source development process is changing and will further change as the movement matures. Lin even argues that any static view must miss essential aspects of the dynamic and loose interaction in the Open Source development model [316]. Fitzgerald argues that *Open Source 2.0* is an appropriate term when comparing the processes used by early hobbyist projects to those caused by the influx of developers paid by companies and organizations [187].<sup>23</sup>

Open Source  
2.0

### 2.3.9 Decision Making

Some aspects of how Open Source projects make decisions have been discussed in Section 2.3.8 on the development process above: (1) Open Source projects usually do have a hierarchical or heterarchical power structure resulting from control over project servers [229] (in particular via version control [300, 260, 269]) and from meritocracy based on the relative contributions (both technical and discursive [442, p.237]) of participants [186]. (2) Project leaders often assume the role of a benevolent dictator reserving themselves the right for unilateral decision [92, 420]. Yet, leaders have to use this privilege rarely for fear of alienating their project community [462, 416]. In fact, one could argue from an anthropological perspective that the power of the project leader is rather weak because project members are independent as volunteers, leading to a *reverse dominance hierarchy* [54] where project members hold significant power via their contributions, time invested or expertise.

How are  
decisions  
made?

It seems that Open Source projects do have a "bias for action" rather than a bias for coordination [554], implying that the use of meritocratic decision making is most common. In other words, whoever is volunteering for the work implicated by a decision opportunity is likely to decide indirectly. Li et al. for instance studied 258 software modification decisions in OSS projects and found that only 9% of all identified issues failed to be decided and that 58% were decided without evaluating the options raised in discussion [313]. Only 31% of decision episodes which were raised on the mailing list were decided collaboratively [313]. Similarly, Schuster interviewed Open Source core developers in the project GNU Classpath for an innovation introduction and found that formalizing decisions for general situations was deemed inefficient in many situations. Rather, the project members were expected to use common sense to make better decisions based on actual circumstances [379]. We might conclude that just as tasks are frequently self-assigned by project members [116], the number of decision situations in which a conflict could arise is minimized in the Open Source development model. In this sense, Open Source decision making and leadership could be described as ranging from the consensual and meritocratic to the hierarchical, yet participatory approaches [521].

Bias for Action

Fitzgerald hypothesizes that with increased maturity and size, projects will adopt more formal decision making processes [187], such as voting in Apache [186], and establish meta-processes for creating such

<sup>22</sup>Yet, the Debian project still prescribes to the "when it's done" feature-based schedule [293] despite large delays with the release of Debian 3.0 (woody) and 3.1 (sarge) [340, 4] and the success of Ubuntu—the largest distribution derived from Debian [293, pp.22f.]—with regular releases targeted twice a year [529].

<sup>23</sup>This thesis includes a substantial set of projects in both categories.

formal structures, such as the steering board in Gnome [207].

### 2.3.10 Defining Success

The central motivation for this research is to enable an innovator to succeed in introducing an innovation into a community-managed Open Source project. This thesis does not want to restrict on the goals of the innovator with an introduction, but rather assumes that it is targeted at increasing the project's success in one way or the other<sup>24</sup>. As there are different motivations, participants, and variations in the development process, there must be different notions of success [114, p.145]. Thus, at this point we want to review the success measures commonly used for assessing Open Source projects.

On this question there are two separate research areas which unfortunately have not been integrated. First, there is the literature on Open Source success (e.g. [114, 485]), which originated in the question of how to measure success as a target variable in studies about the influence of certain project characteristics. Lerner and Tirole, for instance, wanted to know how license choice affected project success [306], and Koch sought a correlation between using mailing lists and source code management systems and project success [286]. Second, there is the literature on Open Source quality assessment (e.g. [89, 471, 459, 405]), which originated in the problem which of several Open Source products to choose as an organization or individual user. While the goals of both camps are different, the questions they ask and metrics they use are similar and focus on the current and prospective attributes of the product and the developer community. The resulting measures differ from those for commercial development such as, for instance, "being on-time, on-budget, and meeting specifications" [485]. This is because these commercial measures only make sense from a tailored perspective of an individual stakeholder who, for instance, has a budget to pay a developer or needs a particular feature at a particular time. The success of a project as a whole is much harder to assess because developers are volunteers<sup>25</sup> and management and specification is informal [447].

Open Source  
Success  
Measures

In the first camp of Open Source success, many different measures have been proposed (an overview can be found in [114]): Grewal et al. used the number of commits as a proxy for technical success and number of downloads as a measure of commercial success [223]. Comino et al. used self-reported development stage (pre-alpha, alpha, beta, stable, mature) [100]. Stewart et al. used an increasing number of subscribers at the project announcement site Freshmeat as a measure of user popularity and the number of releases as a measure for community vitality [485]. Weiss evaluated measuring project popularity using web search engine results and finds the number of links to the project homepage and the number of pages on the project's homepage to be meaningful indicators [540].

Two papers include interviews with developers on what makes a successful project for them. Schweik and English interviewed eight developers from projects of varying sizes and found a lively developer community (1), widely used software (3), valuable software (2), achievement of personal goals (1), and reuse in other projects (1) mentioned as primary factors associated with success [166]. Failure on the other hand was unanimously reported to be correlated with a low number of users leading the developers to abandon the project [166]. Crowston et al. asked on the community news-site Slashdot and identified 170 thematic units in 91 responses: user and developer satisfaction and involvement (44%), product attributes such as quality or features (14%), process attributes (13%), recognition and influence (13%), and downloads and users (5%) [114].

Koch noticed the problems which arise due to these different measures and suggested the use of data envelopment analysis (DEA) to consolidate them into a single efficiency score [286]. Unfortunately, he found a negative correlation between mailing list and revision control usage to the resulting score [286, p.412], casting his results into doubt.

<sup>24</sup>It is also likely that the innovator is looking for mid-term or long-term success using the innovation introduction, as those need time to become adopted and have an effect [436].

<sup>25</sup>While some developers might be paid [298, 211, 235, 43], from the perspective of the project their sponsoring organizations still participate voluntarily.



Figure 2.1: One possible way for the innovator to look at Open Source project success as based on both community and product success.

In the second camp of Open Source quality measures, many evaluation frameworks have emerged such as Open Business Readiness Rating (OpenBRR) [536, 471], Qualification and Selection Open Source (QSOS) [459], Quality of Open Source Software (QualOSS) [89], two Open Source Maturity Models (OSMM), one by consulting firm CapGemini [154], and one by systems integrator Navica [216, 215], one OpenSource Maturity Model (OMM) [405], and one Goal-Question-Metric-based<sup>26</sup> unnamed one [188]. For instance, QSOS provides an assessment methodology in which each project and its software to be analyzed is evaluated according to a hierarchical list of criteria whose top-level categories are intrinsic durability, industrial solution, technical adaptability, and community [459]. By measuring the leaf-nodes in the hierarchy (for instance, the age of the project or the developer turnover) and then assigning weights to each criteria, the methodology intends to help users choose a suitable Open Source product. Similarly, OpenBRR also uses such hierarchical criteria (the eleven top-level items here are usability, quality, security, performance, scalability, architecture, support, documentation, adoption, community, professionalism), but also includes a pre-screening based on eight items such as licensing and programming language [471].<sup>27</sup> QualOSS as a third example uses product and community as a top-level distinction and then splits into the product attributes of maintainability, reliability, portability, usability, performance, functionality, security, and compatibility and into the community attributes maintenance capacity, sustainability, and process maturity [89].

Open Source  
Quality  
Measures

Several authors have criticized the proposed measures or cautioned on the way to derive them. For instance, number of downloads from the project's website is the most commonly used proxy for use, but may be (1) meaningless for many projects since the software might be distributed as a standard package in Linux distributions making manual downloads unnecessary, (2) highly dependent on product domain [257], and (3) skewed by the number of releases as the same users download a new version [548][468, p.42]. The number of unique committers to the project repository—a popular proxy on the size of the active community—can also be easily skewed: Project leaders might commit patches themselves in one project or grant commit access readily in another [468, p.38]. Similarly, the number of releases over a fixed period is most likely also dependent on the development style of the project [468, p.43]. Furthermore, some attributes such as development stage are self-reported by each project and might be influenced by the interpretation of values.<sup>28</sup>

Criticism

Returning to our initial question of how an innovator could assess the success of an Open Source project (in case the innovator does not have a particular success measure already in mind) and considering the

<sup>26</sup>See [30].

<sup>27</sup>For a comparison of OpenBRR and QSOS see [141].

<sup>28</sup>Consider for instance that only 1.6% of projects reported themselves as “mature” on SourceForge [100], while there is no definition given in the selection dialog on how to distinguish from being a “stable” project.

Product Success results from [114], I would propose to take a two-sided approach: On the one side, product success as measured by the satisfaction and interest of users and the quality of the software should be regarded. On the other side, process or community success should be measured using developer satisfaction and involvement as well as the ability to publish a release on regular basis and keep the number of open issues at bay by achieving low average bug lifetime. Both sides are important to sustain the success of an Open Source project because they support each other: A vital project with a successful community should produce attractive software, increasing popularity or product success. Vice versa, successful software with many users should provide input, motivation, and a recruiting base for building a successful community (see Figure 2.1) [484, 347]. Last, no matter which criteria for success the innovator selects, it is unlikely that they will be compatible with all individual goals of each participant. Motivations for participants are too diverse and in many cases do not directly include the success measures mentioned above (cf. Section 2.3.5). For example, learning and sharing skills can certainly be pursued in another project, if the current project fails. Or, when looking to satisfy a personal itch, the number of downloads by users, participation of other developers and a particularly high level of quality might be quite unimportant. Even reputation might be achieved without having many releases, a healthy community, or well-established processes.

## Chapter 3

# Methodology

The methodology chosen to answer the research question of how innovations are introduced into Open Source projects is a retrospective e-mail analysis conducted using Grounded Theory Methodology (GTM) following Strauss and Corbin [493]. A qualitative method—as represented by the GTM—in combination with data from observation [342] was chosen because (1) the hitherto novelty of the research field required a holistic approach to assure an understanding of both the insights to be gathered in this field and the questions necessary to be asked to attain them, (2) Open Source software development is a social phenomenon which emerges from decentralized, loosely coupled, distributed collaboration over extended periods of time and is thus not generally reproducible in the laboratory [389, pp.13f.],<sup>29</sup> and (3) the basic constituents of an innovation introduction process—discussion, execution, and adoption—are not well represented in data sources for automated analysis such as software repositories [6].

### 3.1 Development of the Methodology

The realization that a passive observational, qualitative method was necessary to attain insights in this research area came after two previous attempts with different methods proved less successful: First, I conducted four case studies of innovation introduction as action research and field experiments. Second, I tried to conduct a survey among Open Source developers to elicit innovation episodes. Both will be now discussed in more detail to set the stage for the use of Grounded Theory.

#### 3.1.1 Action research and field experiments

The first method used to tackle the research question was action research (AR), a collaborative, iterative research method in which researcher and participants in the studied field work together to attain a common goal [312, 498, 11, 134]. Using action research<sup>30</sup> and under my supervision, Robert Schuster introduced a self-devised process innovation focused on knowledge management into the project GNU Classpath (details about the case are given in Section 7.1). He did so as a member of the project after involving the project leadership and gathering positive feedback from the community as a whole.

---

<sup>29</sup>Academic software projects published under an Open Source license could potentially provide a borderline case, if Open Source development processes are being followed strictly. While it has been argued that academic software production can benefit from adopting Open Source development processes [3] and surveys of Open Source developers show that 7% percent are employed at academic institutions [298, p.9], there are two shortcomings of this approach: (1) Academic software projects have a given power structure from the principal investigator over Ph.D. students down to Masters and Bachelor students. Such a given power hierarchy will in particular affect the decision processes when adopting innovation and skew results. (2) Starting and growing an academic software project to attain size and relevance necessary for study is out of proportion for a Ph.D. thesis.

<sup>30</sup>At that time we did not know that our actions could be subsumed under the label of this research method.

Over the course of three months Schuster guided and led the process improvement effort, iteratively improving the design of the innovation and addressing the needs of the community [379]. While this introduction effort was considered a success by the community, there were three troublesome aspects:

1. The amount of effort necessary to conduct an introduction should not be underestimated and was justified only because Schuster had a secondary motivation as a member of the project to further his influence and understand the processes used by GNU Classpath to operate. This manifests itself both as problematic for students who have a restricted time frame for their thesis as well as for myself who was bound to the academic calendar and thus could not sustain an on-going engagement stretching into the lecture period. These insights are seconded by Jaccheri and Østerlie who proposed using Action Research as a methodology for software engineering education and research. They concluded that goals need to be revisited often in the face of the considerable effort necessary to achieve them using this methodology [265, p.4].
2. Schuster had already been a project member when he forged the agreement first with the project leadership and then with the community. For an outsider getting in contact with the project in contrast becomes more important. Consequently, the research questions the next two studies focused on became skewed towards the early stages of being a participant: First, Luis Quintela-Garcia focused on efficacy of out-of-the-blue proposals accompanied by code gifts (see Section 7.2) and Alexander Rossner used three different contact strategies for innovation proposals (see Section 7.3). This consequence of skewed research questions can be similarly observed with the students who performed Action Research under Letitiza Jaccheri's supervision and whose topics to a large degree revolve around becoming members of their targeted projects: Tjøstheim and Tokle worked on understanding how to become accepted as new members in Open Source projects [505], Mork (among other questions) studied how he as an external developer could influence a commercially-dominated Open Source project such as Netbeans [354], and finally Holum and Løvland investigated how to alleviate the problems of becoming a project member [255].

As a secondary implication of not being a project member and also from having a research question focused on getting in contact with a project, it becomes more difficult to establish the researcher client agreement (RCA). To fulfill the relevant criteria for an RCA such as (1) understanding Action Research and its consequences, (2) getting a clear commitment from the Open Source project to perform Action Research, and (3) defining an explicit assignment of roles and responsibilities [134, p.70] requires a trusting relationship and necessarily to already have been in contact. When again comparing our approach to other researchers which have used Action Research in the Open Source context [265], two other problems of the RCA in the Open Source context emerge:

- (a) Since an Open Source project is an emergent social phenomenon arising from fluctuating participation around a more or less stable set of core developers, the question arises of who actually is the "client" for the RCA. Mork, in his action research studies of leadership and documentation practice in Open Source projects, solved this problem in the following ways: In the case of studying leadership, he acquired approval from the commercial owners of the project under study only and otherwise remained "covert" to the other participants [353, p.28–30]. In the case of studying documentation practices, he accepted the social contract that the non-commercial project used [354, p.16]. In other words, an explicit RCA with the community was skipped in both cases and replaced by indirect means.

Holum and Løvland also note the difficulty to obtain an RCA and hint at the problems arising from a missing RCA such as lack of research focus when reflecting about their action research study in the project Apache Derby [255, p.66]. They suggest that establishing an RCA in the Open Source context should be done in accordance to the organizational structure of the Open Source project and to the existing ways control is exerted: Projects in which mechanisms for reaching consensus exist such as the Apache project [186] should receive the proposed agreement directly for discussion, while in projects steered by benevolent

dictators such as Python [420] the researcher should take heed to a possible veto by the maintainer [255].

- (b) How can we talk about an explicit assignment of role and responsibilities, if avoiding to take such explicit notions is one core characteristic of the Open Source development paradigm? [379]
- 3. We entered the project with a predetermined choice of an innovation to introduce. While we allowed for input from the project and followed sound ethical conduct [373, 48], we were focused on our own given solution and did not consider to discuss other potential problems which the community could have deemed more important.

It is in particular this last point which deviates so much from the canonical definition of AR that for remaining studies we actually preferred the term of field experiment [236].

Both experiments resulted in relevant insights (see Section 7.2 and 7.3), yet we did not feel that we had really pushed through to an innovation being successfully introduced in the sense of achieving process change or attaining a goal, but rather got stuck in contacting the project and enabling the use of the innovation.

As a final attempt to stage a successful introduction, I tried to introduce automated regression testing into an Open Source Java project. I invested a four week period to implement and then another one to fix a test suite of seventy test cases, but the innovation did not catch on immediately inside the project. At the time, my efforts seemed to be unsuccessful<sup>31</sup> and the introduction overly difficult for the amount of insight generated. Using an active, experimental approach was abandoned at this point.

### 3.1.2 Surveying the Open Source Community for Change Episodes

As a second method, a survey for Open Source developers was designed (see Appendix B). This survey tried to elicit change episodes from the participants and poll their goals, methods, hurdles, and insights related to innovation introduction they conducted or experienced. This survey was circulated on lists frequented by Open Source developers<sup>32</sup>. Yet, even though these lists had more than 500 subscribers each, only six replies could be gathered.

Results proved our assumption wrong that an innovation introduction is a conscious entity in the conceptual world of an Open Source participant. For instance, while many projects switched their *source code management system* from CVS to *Subversion* over the years preceding the survey, they did not abstract this to the level of a new tool being introduced and possibly their processes being changed. Krafft confirms this in his study on innovation diffusion in the Debian project: Even members in a preselected group of Debian contributors on the topic of diffusion “had not previously thought much about the issues” [293, p.78].

Rather than reporting on how changes to the development process were achieved, three of the replies focused on changes to the code base without any abstraction to process. Only one reply attained the level of abstraction the study was meant to achieve, for instance in a reply to the question what had been helpful to reach a goal set for an innovation:

“(1) Offering the change as a parallel alternative rather than an either/or choice (this was CRITICALLY important) (2) Patience and not being too pushy (3) Doing it, not just talking about it” [Resp.3]

<sup>31</sup>As it turned out the introduction was actually a success—albeit at a much slower pace than expected—eventually resulting in 270 test cases (see Section 7.4).

<sup>32</sup>In particular, user lists of tools and APIs commonly used for Open Source development were targeted such as *Boost*, a C++ library for common application development tasks, *AutoConf*, a package for generating platform-specific scripts to be used in the build-process of applications, *Subversion*—a software for centralized source code management, *Simple Directmedia Layer* (SDL), a multimedia library for accessing input and graphics devices, and *Bugzilla*, a software for bug tracking.

Facing only meager outcomes and receiving hostile e-mails in reply to targeting the mailing list (for instance “(1) This is lame. (2) You get paid for this and we don’t.” [Resp.5]), the survey method was abandoned.

### 3.1.3 Using Grounded Theory Methodology

After the use of a survey had thus failed, the need to validate the basic terms of our research question became more striking: Do innovation introductions actually occur in Open Source projects? Why is innovation introduction not a consciously perceived part of the life of an Open Source project? To answer these *existence questions* of the research area [157, p.288], publicly available e-mail lists used for communication by Open Source projects were downloaded from the mailing list archive Gmane.org (see Section 3.6) and analyzed. This quickly resolved the issue: Introductions do occur and result in visible discussion on the mailing-list, but in many different shapes and types. They often occur *en passant* to the day-to-day business of developing a software product and only rarely do developers reflect on the implications of their doing on the software development process such as the maintainer of *ArgoUML* in the following quote:

“[...] I want to constantly try to find the best way to do things, especially when it comes to communicating ideas and solutions among the developers.” [argouml:4788]

While reading through the mailing lists of these initial projects and seeing the wealth of innovation-specific behavior, it became increasingly clear that a consistent methodological framework and tool support for managing both data and analysis is necessary. This need led to the adoption of Grounded Theory Methodology as a framework for theory building from qualitative data (see the following Section) and to the development of GmanDA as a tool to facilitate the analysis (see Section 3.4).

## 3.2 Grounded Theory Methodology

GTM is a method from the social sciences to perform qualitative data analysis with the goal to generate theory. Its particular focus is to keep in touch with reality by providing a set of interrelated coding practices and general advice on working with qualitative data. At the center of GTM is always the work on data, with different types of emphasis depending on whether data should be explored (Open Coding), conceptual relationships be established (Axial Coding), or a scientific narrative be built (Selective Coding) [493].<sup>33</sup> Each of these will be introduced in turn:

**Open Coding** Open Coding is the first step when building a theory from a set of data. As the name suggests, the emphasis is not bound to a particular phenomenon occurring in the data, but rather open to all types of events, occurrences, and entities of interest. The task of the researcher in this step is to attach codes, i.e. conceptual labels or names, to the occurrences of phenomena. In the easiest case, this involves repeating the name given by the participants involved in the incident (so-called “in vivo coding”). In other cases, if the entity remains tacit to the participants, the researcher herself needs to come up with an appropriate name. As a natural step in Open Coding, the researcher will strive to define the conceptual label and delineate it from others. In the process of this, the label can outgrow the initial stage of being a “code”, i.e. a name for something which occurred in data, to become a concept, an independent theoretical entity, which can be thought, mulled, and reasoned about as an entity separate from data. A detailed outline of how Open Coding was performed for this thesis is given in Section 3.2.1.

**Axial Coding** If Open Coding can be summed up as the practice of keeping an open mind and uncovering the plethora of conceptual entities hidden in the code, then Axial Coding is the focused discussion on a single such entity in relationship to the conceptual world surrounding it. The goal at this point in the research

<sup>33</sup>This thesis started out following the more methodical GTM according to Strauss and Corbin [493] in contrast to Glaser’s GTM focusing on emergence of conceptual insights [214]. Over time this began to change and the approach finally used is squarely between both position, best characterized following Charmaz as constructivist GTM [78].



process is to increase the conceptual “thickness” of concepts, i.e. giving them more substance and defining their meaning precisely, enlarge the set of occurrences known in data, characterize the properties of the concept, and establish relationships to other concepts. Concepts which have undergone Open Coding are defined and grouped but usually “thin” in their implications and associations. Concepts after Axial Coding in contrast should be integrated into a conceptual domain and be “rich” in detail.

GTM following Strauss and Corbin includes as a last step in the research process the practice of Selective Coding as a method of focusing the research towards a resulting theory which can be presented to a reader. Selection is necessary because the result of Axial Coding is an unstructured network of well integrated concepts, yet not necessarily a narrative linear structure of explanation suitable for a written manuscript.

Selective  
Coding

Orthogonal to these coding practices are several other strategies for dealing with the task of analyzing data, the most important of which are memoing—the activity of taking note of conceptual ideas, explorations, and intermediate steps —, the paradigm—a general structure for the analysis of actions by participants—, and diagramming—a strategy for exploring the conceptual relationship between codes.

At this point it should be stressed again what actually constitutes a result of a Grounded Theory analysis. Corbin and Strauss state three aims of doing qualitative research in general: (1) Description, (2) conceptual ordering, and (3) theory, which are distinct, but still intertwined [105, p.53]. All analysis has to start with description, because “there could be no scientific hypotheses and theoretical [...] activity without prior or accompanying description” [105, p.54]. This becomes more apparent when considering “description” to be more than just textual rendering of raw data, but rather as the mapping of this data into the domain of things understood by the researcher. Only what is understood and grasped by the researcher can be used to construct a conceptualization of the data. Conceptualization for computer scientists is probably most closely related to object-oriented modeling, where we group sets of related objects under the label of a certain class and then put up hierarchical subtype relationships between classes to form a taxonomy of their associated objects. Well-developed categories then become the basis for theory, i.e. a set of statements about the relationship between conceptual entities and what must be the ultimate goal of research using GTM.

What kind of  
results to  
expect?

Be aware that while GTM was designed specifically to include mechanisms of constant comparison with data to keep the research and the resulting theory firmly “grounded” and ensure some levels of internal and external validity [155, p.137], GTM should and cannot replace methods primarily aimed at validating the resulting theory [105, pp.316f.].

The remainder of this section will discuss the details of how GTM was used in this thesis. Readers are safe to skip this discussion if they have a basic understanding of GTM or are primarily interested in the results and accept that GTM is a rigorous method to produce results of relevance and sufficient validity. GTM, however, is not an easy methodology, and I suggest in particular “What GTM is not” [496] to the interested reader.

### 3.2.1 Open Coding

For a more detailed description of the practices associated with GTM and how these were used in this research, we first turn to Open Coding. This practice is associated in particular with the early stages of analyzing data and has the goal of uncovering and assigning conceptual labels to the phenomena occurring in data. In this section we discuss the following questions: (1) What is the unit of coding or granularity in which data was segmented for coding? (2) How did coding proceed in the face of large amounts of data as represented by thousands of e-mails in a mailing list? (3) How were codes assigned to data represented in the research process?

### 3.2.1.1 Segmentation of Data

GTM does not pay much attention to the question of how to perform segmentation of data into units of coding, since as a purely qualitative approach without any intentions to move into quantitative analysis it only needs mappings between excerpts of data ('quotations') and concepts to be usable for operations internal to the research process. The researcher will ask (1) "where did this concept occur in data?" and wants to be pointed to quotations which contain or make reference to the concept, and (2) "to which concepts does this excerpt refer?", which maps the other way. Yet, when looking at other qualitative methods such as verbal analysis which include segmentation as an explicit step as part of the methodology, it becomes more obvious that the granularity of the segments has an important influence on the aspects in the data which become more pronounced [81]:

Smaller segment sizes and high granularity put focus on the micro-structure of communication and can uncover more detail in data, but might "miss[...] the idea that is embedded in the macro interference" [81, p.285]. Larger segment sizes and low granularity cause less work, but might reduce the amount of information derived from data [81, p.286].

The researcher must balance this trade-off carefully to attain correspondence between the unit of coding and the research question. Since we are interested in the project-level processes of innovation introduction, it was deemed sensible to avoid high granularity segmentation down to words and sentences, but rather use single e-mails as the primary unit of coding for each innovation introduction.<sup>34 35</sup>

### 3.2.1.2 How Coding Proceeded in Data

Given the large number of e-mails in each mailing list, the question of how coding proceeded inside each list becomes relevant. Only a small sub-set of all e-mails on a mailing list contain innovation-associated phenomena and information, and not every e-mail can be read. To put this into perspective (see Section 3.6 for details): Of the 33,027 e-mails in 9,419 threads included in this analysis, I have read 5,083 e-mails (15%) in 2,125 threads (23%) and coded 1,387 e-mails (4%) in 788 threads (8%).

Two modes, explained below, have been found to be of central importance while Open Coding in this manner: scanning and searching. Both modes were employed either in a single pass, i.e. identifying potentially interesting e-mails and directly coding them, or using two passes, i.e. first identifying all messages containing innovation-relevant content and then as a second pass coding them. The trade-off between one and two passes is that doing two passes requires less concentration but more time. Regardless whether one or two passes are used for initial Open Coding, messages are often revisited to modify existing codes, for instance if boundaries between codes have changed or additional aspects have become of interest (this is also true for later Axial Coding, when the number of codes attached to an e-mail increases again because of in-depth study).

Scanning vs.  
Searching

- Primarily, messages were scanned by date, i.e. by looking at the title of threads and discarding the whole thread without reading any message bodies, if the title was pointing to support inquiries, commit messages or technical issues, and the thread did not contain a large number of messages. Threads not discarded were then opened and the message body of the first e-mail was read. If any association to process or tools could be made, the whole thread was read and coded where appropriate. If no association could be made in the first e-mail of a large thread, then the biggest sub-threads were sampled and checked whether the discussion topic had changed.
- Given innovations, key people or terms in general associated with innovation introductions in a given list, the mailing list was then searched for these and related terms to extract all pertaining messages. For instance, if a discussion was found in the project to switch the source code

<sup>34</sup>If e-mails contain data pertaining to multiple distinct innovation introductions, these are kept apart by separating the codes attached to a single e-mail using semicolons (see Section 3.4).

<sup>35</sup>If we compare this approach to Li et al., who studied decision making in Open Source projects and segmented finer inside of e-mails, we notice that this amount of detail was unnecessary as their analysis used units of decisions rather [313].

management system, all occurrences of the proposed replacement were searched for and then likely coded.

Both modes of operation should help to uncover most of the innovation-introduction-specific e-mails on a list, but unless all messages are diligently read, no guarantees can be made that all occurrences of all introduction-specific behavior have been found.

### 3.2.1.3 Coding Meta-Model and Epistemological Foundations

Before proceeding on the question how to work on the set of such interesting and introduction-specific e-mails, this paragraph is to establish an epistemological foundation and outline the coding meta-model used to capture insights on the conceptual level.

This research assumes an optimistically and for social scientists probably naïve view on the epistemological question whether and how insights about the world can be attained. While this research agrees with Popper's Critical Rationalism that truly objective knowledge is philosophically impossible [408], it remains a feasible endeavor to attain insights about reality that provide practical value and can be independently verified by others through their own observation and interpretation of the data at hand<sup>36</sup>.

As a general world view we assume that any system consists of active agents working concurrently and collaboratively on passive artifacts. For each agent and each system-external observer, a specific subset of the world is directly observable. In the context of an Open Source project, for instance, each participant is able to access the publicly available data of the project and has also access to his thoughts and resources. Yet, large parts of the world including tacit knowledge are inaccessible to the actor himself. Communication of an actor then can be regarded as a mapping of this visible subset of reality onto language.<sup>37</sup> As we know from Wittgenstein, others might not be able to map back from this language onto their world sufficiently [551].

To achieve a sound epistemological basis, we need to distinguish two types of mappings from realities to language. We call a mapping from realities to language a description if observers can agree that the resulting language does not include anything not present in reality. In the case of a written document such as this dissertation, the researcher will foreclose such an agreement on behalf of his potential readers. If an addition has taken place, we call the mapping an interpretation. Again, philosophically such a distinction cannot be cleanly made, since only by language are the described things created as such (compare with Nietzsche's "Chemie der Begriffe" [365]), and from this perspective there can be no neutral description, just additive interpretation. By talking about something, we actually only then give it shape and define it. In the pragmatic viewpoint that science needs to operate, we assume that it is possible to agree whether a description and a reality are compatible for the sake of advancing our knowledge in a field.

Description vs.  
Interpretation

For this work, we allow all products of description and interpretation by project actors themselves<sup>38</sup> as well as the publicly available artifacts to be primary sources of insights. This implies that while we are aware of the fact that actors might purposefully deceive us or merely err in their interpretations of reality, we still use all language products of theirs as source for generating insights. A distinction between "reality" and "the discourse" in its implication for structuring our reality [192, 225] is not made, but rather everything is discourse.

As a coding meta-model, the researcher thus starts with a set of *primary documents* such as in the case of this research e-mails, web pages or Internet Relay Chat transcripts. These documents describe (or literally contain) *phenomena* of interest which can be abstracted to *concepts*. While explicating and understanding concepts by looking at phenomena, the researcher will aim to structure a concept using a set of abstract *properties*, which should map to concrete *values* for each phenomenon.

Coding  
Meta-Model  
Primary  
Documents

<sup>36</sup>Grounded Theory Methodology provides us with the means to allow for such by the reader.

<sup>37</sup>Both the tacit knowledge and the inaccessible parts of the world can shape such a mapping, but only indirectly so.

<sup>38</sup>Language, unfortunately, does not offer good terms for distinguishing description and interpretation as mapping to language and the products of such mapping, which are called description and interpretation again.

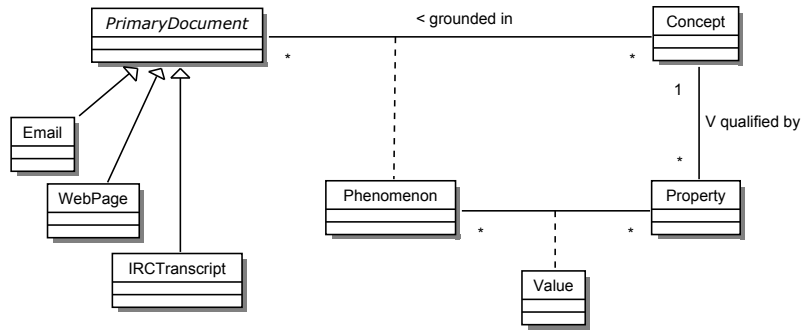


Figure 3.1: The coding meta-model used in this research for conducting research according to GTM. Most central is the many-to-many relationship between primary documents and concepts at the top, which is driven by the occurrence of phenomena in the data. Abstract properties of concepts and the associated values in concrete phenomena help to further qualify and structure the concepts. One important constraint exists: Phenomena can only be associated to those properties belonging to the concept of which the phenomena is abstracted from, which can be expressed in Object Constraint Language (OCL) [369] as: `context Phenomenon inv: self.properties->forAll(p | p.concept = self.Concept)`

*Episode* For example, one of the earliest concepts to be established was the concept of an *episode*, i.e. the aggregation of everything pertaining to a particular innovation introduction attempt, of which one central property is the *outcome*. Both of these could only arise to be of importance, because in concrete cases, such as in a set of messages *regarding the use of branches for patch management* at the project *ArgoUML*, we observed these messages as belonging to an innovation introduction attempt and noted a particular value for the property *outcome* (in this case the proposal was *rejected* and thus the attempt *failed*).

The resulting coding meta-model is represented as a class diagram in the Unified Modeling Language (UML) [371] in Figure 3.1. In this diagram, phenomena and the values of properties are modeled as association classes. This highlights that phenomena and values are essential for our bridging from data to concepts but also rather ephemeral constructs, which are just there, because a researcher is looking at data from a certain perspective.

### 3.2.1.4 Open Coding Structure and Syntax

Given the three previous paragraphs, we now know how to segment a large corpus of data into individual units to be regarded, how the researcher can find among these units those being of interest, and how in GTM we map from data to conceptual entities via phenomena and their property values. The next step in the Open Coding process is to start labeling the phenomena represented in these interesting units to start the process of developing theoretical concepts representing the concretely occurring phenomenon in the abstract. Such labeling can first proceed as plain text, most probably a list of names for phenomena occurring, but the drawbacks of such an unstructured approach is easy to spot:

Apart from giving each phenomenon a name, the researcher wants to discuss this name, delineate it from others, describe how it occurred in the data, or define which aspects in the unit of coding constitutes the phenomenon. To facilitate these desires and provide an accessible and searchable way to work with the conceptual labels assigned to a unit of coding, a coding scheme was devised. The coding scheme initially started as a comma separated list of key-value pairs: The key was the conceptual label of the phenomenon and the value any plain text pertaining to the occurrence of the phenomenon in the unit of coding in particular and the concept assigned to it in general.

Key-value Pair  
Scheme

For instance, consider an e-mail in which a project participant proposed to the project to start using a new *branching scheme* for the project's source code management system. If this proposition is made right after a previous version of the software had been released [geda:4072], we can deduce that one

concept of interest is the timing associated with this proposal. To code such an insight, we could first choose an appropriate label, for instance “timing of proposal”, and probably describe the particular importance timing had in this case as one of the primary arguments in favor of the introduction.

timing of proposal: "Proposing directly after a release provides the innovator..."

This scheme already gives the researcher the possibility to assign a code to the phenomenon, and provides scratch space to ponder the occurrence and its conceptual implications. It also allows to search for all occurrences of a particular phenomenon, but has still three main drawbacks:

1. The scheme has no way to represent specialization and generalization of concepts. In the above example, for instance, a particular branching scheme was proposed, which we would like to compare to (1) other such schemes and (2) all process innovations of source code management use in general.
2. Phenomena cannot be related to each other except by an opaque label, as for instance in the above example where a concept of timing is associated with the concept of a proposal, which both might be combined with other concepts as well. In the case of a proposal as conceptual entity, we might want to find the “author of a proposal” to be an important theoretical constituent as well.
3. The coding scheme does not distinguish spaces of discussion for the concept behind a phenomenon and the phenomenon itself. Rather, it only has one textual space in which all thoughts, descriptions, and ideas are mulled together. This problem could have been solved by structuring the text itself, for instance using headlines, but it was decided it was better to raise this distinction to the level of the coding scheme.

To achieve this goal, the simple comma separated list of key-value pairs was extended in three ways:

1. The possibility to describe conceptual labels as specializations of a more general concept was added by separating labels by dots. In the above example, an innovation relating to a branching scheme of a source code management system was coded as “innovation.source code management.branching scheme”. This offers the possibility to label a phenomenon on all levels from the abstract to the concrete via single inheritance categorization. Specialization

The use of a concrete categorization for a phenomenon down to the unique individual occurrence might seem counterintuitive at first sight. Did the coding scheme not try to provide a way to abstract from the concrete to the conceptual? Yet, it turns out that beside abstracting from the concrete we also need identification as a second important aspect. For instance, if we have found one message to include a proposal for an innovation introduction—let’s say, fictitious Bob is proposing the adoption of the branching scheme—we often find other messages relating to that particular proposal. If we code down to the individual entity (for instance using the code `activity.propose.Bob` if Bob only proposed once), we can use the concept as a way to highlight all places in which this entity occurred (for instance marking the e-mails containing counter-arguments to Bob’s proposition using this very same code). In particular, this technique was used to aggregate all messages belonging to the same innovation introduction attempt into an *episode*.

2. To relate concepts to each other, hierarchical nesting was allowed by offering composite values using the following syntax: Hierarchical Coding Scheme

```
code: {
  subcode1: "textual description",
  subcode2: {
    subsubcode2.1: "...",
    subsubcode2.2: "..."
  }
}
```

The semantics of this syntax is similar to that of the key-value notation: A unit of coding annotated with a certain code implies that this unit of coding contains the description or reference to an instance of the type of objects labeled by this code. To stick to the above example, we might attach the code of “*activity.propose*” to this e-mail in which the author Bob proposed to introduce the branching scheme, and thereby create a conceptual instance of a conceptual class of proposing activities. This instance can then be further qualified by describing it with additional sub-codes. For instance, the sub-code “timing” may be used to denote that a phenomenon regarding time is of importance for this instance of a proposition. By linking thus two instances of two concepts to each other, the concepts start building a relationship.

The use of colons and curly braces draws heavily from *JavaScript Object Notation* (JSON)<sup>39</sup> and *YAML Ain't Markup Language* (YAML)<sup>40</sup> [39] and can easily be converted into both notations. The syntax was designed to be easily writable and intuitive to users of other notations.

Description,  
Definition, and  
Memo

3. To address the third shortcoming of the coding scheme, the lack of possibilities to distinguish discussion about an instance of a phenomenon from a discussion on the class to which this occurrence belongs, three special sub-codes were created: *desc* (short for description) is used to informally describe the instance categorized by a concept (e.g. talk about Bob's particular proposition), *def* (short for definition) is used to give an informal definition of the concept attached to the instance (e.g. define the concept of making a proposal), and *memo* (short for memorandum) is used for all pondering at the conceptual level (see Paragraph 3.2.2.2 below for a detailed discussion on memoing).

With these three improvements to the coding structure, the coding scheme provides a formal representation to fulfill the requirements which posed itself for Open Coding in this research: To capture phenomena occurring in data using conceptual labels and stipulate several ways to discuss the phenomenon, the concepts, and theoretical pondering. Little has been said so far about the semantics of building a set or even hierarchy of codes for capturing what occurs in data. In particular, two questions soon arise when the number of codes of concepts grows with the wealth of phenomena occurring in data: Can and should I capture everything, and how important is it to cleanly delineate all concepts from each other? We want to postpone this discussion until the reader has a better understanding of the next practice of GTM—Axial Coding—, because it is here that concepts start to interact with each other for the purpose of building theory and the relevance of these questions can be appropriately assessed (see Section 3.2.2.3).

### 3.2.2 Axial Coding

Thickness  
Focused  
Coding

If Open Coding as described in the previous section is the practice of discovering and labeling phenomena occurring in data using codes to give rise to a conceptual entity, then Axial Coding is the practice of understanding and exploring such conceptual entities in depth. Thus, instead of focusing on individual mailing lists associated with an Open Source project, the focus is now on developing and understanding individual concepts scattered across mailing lists. The goal of Axial Coding is to arrive at conceptual *thickness*, which implies the concept to be well understood by the researcher, related to other concepts, and richly connected to occurrences in data. To achieve this, the following workflow of an Axial Coding session has been kept to for most of the time, which I call *focused coding*:

1. Select a topic, idea or concept as the one under scrutiny (the concept of interest). For example, one might decide that the concept of *hosting* is of interest, because during Open Coding different types of servers could be identified on which innovations are being run.
2. From the list of codes built in Open Coding select those that are related to the concept of interest by searching or scanning through the list of codes. Often, codes will already exist that are named in close relation to the concept of interest. For instance, we might look for all codes that contain

<sup>39</sup><http://www.json.org>

<sup>40</sup><http://www.yaml.org>

the word 'host' and find that we have coded `forces.hosting` for occurrences of hosting as a decisive force leading to the adoption or rejection of an innovation.

It is not necessary to execute this step with high accuracy or exhaustively by looking for all codes that are related to the concept of interest. Rather, the goal is to find codes with (1) the highest relevance to the concept of interest, and (2) a medium number of occurrences of the code. The second criterion for selecting codes is aimed at avoiding drowning in a flood of messages that need to be read. Continuing with the example of *hosting* as the concept of interest, we will not pick a broad concept like *innovation*, which—while relevant—has hundreds of messages attached to it. Instead, we rather pick `forces.hosting` or `activity.sustain`, which are closely related, but only contain five to ten messages.

3. For each code in the set derived from the previous step, all messages are revisited and reread with a focus on the concept of interest. Often, this will entail the rereading of the enclosing episode to understand the context the message was written in. Such an expansion in the focus of reading—from single message to a whole episode—is time-consuming and the researcher will be tempted to use an opportunistic strategy to read only until sufficient context has been established. Using this workflow for this research has shown that this strategy is bad practice. Instead, the researcher should rather take the time to write a descriptive summary of the episode, which can be reused when revisiting individual e-mails later. Indeed, the researcher should be aware that the development of a narrative summary of an episode is one of the primary strategies for making sense of an episode [299, p.695] and should always complement the application of Grounded Theory Methodology. As a second benefit, such a summary also helps when shifting the unit of analysis to the more coarsely-grained episodes (compare with the discussion above on Segmentation).

One important aspect of making sense of an episode and confirming that an existing understanding is plausible has been found to be the temporal distribution of the individual e-mails belonging to a single episode. In particular, a purely threaded view obscures how long a reply took unless the researcher consciously looks at the date an e-mail was sent. To facilitate better accessibility of the temporal aspect of e-mail communication, either e-mails should be viewed sorted by date or attached to a time line view (see Section 3.5).

4. The goal of rereading is to gather insights about the concept of interest or, as it is called in the lingo of Grounded Theory Methodology, *to develop the concept*. There are two types of such insights:

- (a) We might find that the concept we have been looking at, exists in different variations along some attribute or property. When discovering such a property, we can improve the *dimensional development* of the concept. Considering for instance the example of *hosting*, we might discover that there are different types of *locations* that provide hosting such as private servers and community sites.

Dimensional vs.  
Relational  
Development

- (b) The second kind of insight relates the concept of interest to other concepts, thus helps with the *relational development* of the concept. For instance, the influence of the concept of *hosting* extends beyond setting-up and using an innovation and often affects the discussion of an innovation proposal. *Hosting* in other words has a *conceptual link* to the concept of *forces*, as such influencing factors have been labeled.

Conceptual  
Links

5. One particular goal in concept development can be to understand how an occurrence of a phenomenon arises in data and what effects it has from a process perspective. GTM to this end offers *the paradigm* as a template for structured exploration. Using this template can provide the researcher with support when being overwhelmed by the complexity in the data to return to essential questions such as: How did this phenomenon occur? How does the context surrounding the occurrence interact and influence the phenomenon? What are the implications of this occurrence? While this constitutes the core idea of the paradigm, there is sufficient complexity associated with its use that a separate discussion is deserved in Section 3.2.4 below.

6. Once concepts have been developed in terms of attached properties and relationships to other concepts, a more in-depth study of individual relationships becomes possible. In the above example, we might be interested to learn whether the property of “*location of hosting*” is an important influence on the question whether hosting is a positive or negative *force* during an innovation discussion.

While this describes the workflow during Axial Coding that has been followed for most of the analysis in this research, four aspects are still unanswered: (1) How do we store such insights systematically for further development and retrieval? (2) How can the researcher be supported in this process? (3) How do we use these insights in our further work? (4) How do we present it to interested parties such as the research community, Open Source participants or corporate stakeholders?

The first question is discussed in the paragraph below on coding syntax for Axial Coding, while the second question is answered when the tool GmanDA and its support for concept development is being presented in more detail in Section 3.4. The discussion of the third and fourth question is given in Section 3.2.3 on Selective Coding.

### 3.2.2.1 Axial Coding Syntax

Axial Coding as described above adds two additional requirements to be addressed by the coding scheme. First, properties and their values in concrete cases as the result of dimensional development have to be stored, and second, relationships between concepts arising during relational development need to be expressed.

Coding of  
Properties

To fulfill the first need, the coding was extended by the convention to let all codes starting with a hash symbol (#) denote properties and interpret all sub-codes of properties as values of this property. For instance, if an e-mail contains a *proposition* with a large *enactment scope*, the following coding might be used:

```
activity.propose: {
  #enactment scope: {
    def: "The enactment scope of a
          proposal is...",
    large: {
      def: "A large enactment scope
            is...",
      desc: "This proposition is scoped...",
    }
  }
}
```

While this extension to the coding scheme captures all aspects of dimensional development that occurred for this study, finding a similar way to express relationships between concepts did prove more difficult, in particular, because until the end of this research no common typology of relationships emerged. Rather than formalizing the complex interplay between concepts inadequately in a coding scheme, textual memos were used to explore the relationship between a concept of interest and another one, which will be discussed next.

### 3.2.2.2 Memoing

Writing memos is “the pivotal intermediate step between data collection and writing drafts of papers” [78, p.72] and as such holds particular importance in the research process of using GTM. In essence, a memo is a written account of the insights and questions the researcher is currently pondering. As such a reflection of the research process, a memo is in most cases highly transient and contextual. What’s



more, because a memo is intended to be an intermediate step, a momentary scaffold for further thinking, little effort should and needs to be invested to perfect the thoughts discussed therein.

For relational development in Axial Coding memos have been found to be essential for capturing insights. The researcher in this mode visits the occurrences of a concept and whenever an insight occurs, writes a conceptual memo to capture this insight. Following this mode, a chain of memos will trail behind the researcher's progress through data. To raise insights to a higher level, such a chain is then usually retraced and reread, potentially prompting new insights to be written down as memos again.

Since following and understanding such a memo chain is crucial to this mode of insight, memos should (1) be assigned a date of writing to facilitate sorting memos later on, and (2) important aspects be highlighted prominently. Using the coding scheme discussed above, memos have been presented by the following convention (in this example, a memo about different types of innovations is shown partially):

```
memo.on innovation type: {
  date: "2008-09-15T20:45",
  desc:
    "This little excursion into innovation typology was triggered by two
    innovations that concern *coding style*. How can we categorize these?
    ..."
```

The keyword "date" has been used to capture the ISO-8601 [262] formatted date and time when the memo was written, and \*text\* as a known e-mail syntax convention for highlighting [414].

### 3.2.2.3 On Building a Code Hierarchy

With a detailed understanding of Axial Coding fresh in mind, we now revisit two remaining issues from Open Coding. They posed themselves in the context of increasing numbers of codes assigned to phenomena occurring in data. The researcher would increasingly be forced to spend time managing, defining, and delineating codes. This raised two questions: (1) Given the possibilities to categorize and discuss codes and their associated concepts, how much effort should be spent and strictness be applied when building a hierarchy of codes, and (2) what needs to be coded at all?

Some answers could be gained from looking back at the first coded e-mails for this research project. The initial instincts of the researcher appears to point in the direction of developing a set of codes which *exhaustively* and *orthogonally* conceptualize and describe what is happening in the data<sup>41</sup>, i.e. the researcher aims to capture everything which occurs in data and tries to do so by using concepts which have as little overlap between them as possible. Such an approach leads ultimately to what can be called a *foundational concept base* [444], i.e. a set of precisely delineated concepts which describe behavior down to a high level of detail<sup>42</sup>. Yet, after the fact and considering the resulting theoretical insights of this research, these foundational concepts have mostly vanished and been replaced by much higher-level concepts aligned closely to the topic of interest. These concepts alas are neither orthogonal to each other nor do they exhaustively cover all the phenomena which were observed in the data. Rather, they focus on the most interesting aspects and are highly integrated with each other, often revealing multiple perspectives on the same phenomena. With orthogonality and exhaustive coverage gone at the higher levels of conceptualization, is it then worth to invest in a cleanly structured foundational layer of concepts for the basic levels of interaction? Looking back, the following purpose can be associated with such an investment:

- Foundational concepts are useful as pointers into and markers in the data to find all occurrences of a certain phenomenon of interest. When exploring, for instance, how hosting played a role in the process of introducing an innovation (see Section 5.5), the codes for *sustaining* and *executing*

Foundational  
Concept Base

Foundational  
Concepts as  
Markers

<sup>41</sup>The words *conceptualize* and *describe* were intentionally used at the same time, because there is no precise boundary between these terms, rather they have different values along the dimensional property of abstractness.

<sup>42</sup>The description of the initial stages in developing foundational level codes for this research can be found in [372].

activities were two of the most important candidates for finding e-mails from which insights about hosting could be derived. Thus, the concepts of *sustaining* and *executing* themselves did not become expanded and explained to form high-level concepts resulting in theoretical insights on their own, but rather they became associated concepts of the more central concept of *hosting*.

#### Foundational Concepts as Starting Points

- Foundational concepts serve as starting points to develop properties. For instance, when looking at two *episodes* related to *proposals* of bug tracking processes, I found that they resulted in different *outcomes*. To explore this difference, I started investigating the properties of the proposals made, and arrived at the concept of *enactment scopes* of a proposal. During the course of this analysis, the importance of the original concept (proposal) became increasingly smaller and the embedded property (enactment scopes) developed a life as a concept in its own right. I believe that such a concept could have emerged already in Open Coding, but using foundational concepts offers a more systematic approach and less strenuous way to uncover it.

On the other hand, the danger of becoming stuck in foundational concepts should never be underestimated, because some concepts can be more easily derived as properties than others. For instance, if we consider the concept of *partially migrated innovations* (see Section 5.3) as those innovations that replace only part of an existing innovation, it is unclear where we would attach a property such as partiality when we are coding using foundational concepts.

#### Concept Explosion

- Foundational concepts help to prevent concept explosion. By providing better defined concept boundaries than the ad-hoc concepts that appear in GT while analyzing data, foundational concepts restrict the numbers of concepts that are in use. To that end, the foundational concepts' orthogonality and precise definitions are useful.

Having explored the useful aspects of a foundational layer of concepts, one final word of caution must be given: The development of a consistent, orthogonal, and well-defined set of foundational concepts can easily take months if not years of dedicated effort and in itself will not cause many new insights about the research subject. Thus, either one must restrict the amount of effort going into the foundational layer and accept imperfections in the delineating of concepts, or reuse existing foundational layers of concepts [444] with the associated dangers of importing ill-suited ideas and ways to think about the world as GTM often cautions [78, pp.165f.][105, pp.39–41].

### 3.2.3 Selective Coding

#### Core Concepts

Selective Coding is the last of the three core practices of GTM and focuses on the final step of arriving at a written document containing the results of an analysis. Such a step is necessary because the result of Axial Coding is only a densely interlinked web of thoughts and insights, but not a linear structure suitable to be written down. To achieve this last conversion, GTM suggests to select a *core concept*, which is used as the central anchor onto which the results of the previous stages are connected.

In this research we never chose a concept for such a final integrative step (“innovation introduction” certainly being the obvious choice), but rather performed Selective Coding on a series of concepts which constitute the main results of this research (see Sections 5). Each concept (for instance the concept of *hosting* in Section 5.5) has undergone the following steps:

1. The concept had to be consciously chosen for Selective Coding. The choice for this had already taken place in most cases during *Axial Coding*, where a particular concept would tickle the researcher's investigative instincts to pursue a comprehensive explanation for the reader. For instance, for the concept of *hosting* one particular episode had made the researcher curious to find out how hosting would affect innovation introductions discussions. Exploring *hosting* during Axial Coding then resulted in the concept becoming deeply embedded into a network of other concepts such as *rights* (the degree to which one project participant can use an innovation) and *control* (the degree to which the potential capabilities of an innovation can be used by the project as a whole). Such an intricate web of concepts then becomes the starting point for Selective Coding and also marks the big difference to picking a concept during Axial Coding:

In Axial Coding one is picking up an undeveloped concept to uncover its relationship to other concepts and give it substance in explanation, definition, and meaning. In Selective Coding one is selecting the most interesting concept with a rich analytical founding in data and a strong relationship to other concepts to arrange the presentation of research results around this concept. Thus, while Selective Coding presents the last step of a long journey of analyzing a concept, Axial Coding is merely the first step beyond having identified the concept to be of interest.

2. In this study I preferred to start Selective Coding by drawing a *diagram* of the relationships of other concepts to the core concept, because after Axial Coding the insights of the analytical process are scattered in memos and need to be integrated. Starting with the core concept in the center of the diagram, all related and associated concepts were added in association with the core concept and then linked to one another. Diagramming

The result of this process is similar to a Mind Map [71, 173] or a Concept Map [368, 367] in that it provides a graphical overview of a central concept.

Gathering related concepts is usually primarily achieved by revisiting the memos attached to the core concept, which should contain the result from relational development. When adding relationships into the drawing as arrows or lines between concepts, these should in general be named and linked to the primary documents or episodes in which each relationship became particularly apparent. This helps to organize the sources for grounding insights in data when producing the write-up.

Diagramming also provides the researcher with another chance to consolidate the set of concepts to present. The development of intermediate concepts is necessary to attain the insights finally gathered, but concepts at a foundational level, after the fact that the researcher built results upon them, rarely serve a purpose (see also Paragraph 3.2.2.3 above on foundational concepts). The advice from [78, p.161] can only be seconded here: "Fewer, but novel, categories<sup>43</sup> give your writing power and become concepts readers will remember."

3. A drawing resulting from the previous step then provides a one-page summary of the central results of the analysis of a core concept and can be used as a foundation for the write-up. The key challenge is now how to present the aggregated results of the diagram, given the format restrictions of a paper-printed dissertation. We are primarily restricted to text; figures and tables can merely be supportive means, and textual references can only be made to numbered entities such as sections, pages, figures and tables. The literature suggests that *discussion with intertwined narrative* is the main presentation form for Grounded Theory studies with page-sized tables giving examples of developed core concept serving as support (see for instance [264, 20, 69]).

The most obvious idea to turn the diagram into writing is to order it in the same way insights were first uncovered from data. Moving step by step along the *analytical development* of a concept is a canonical order and should be easy to follow for the reader. Problems arise only if the researcher in the analytical development encountered too many detours and culs-de-sac which need smoothing out, or if the development took too long and there is danger that the reader might miss a concluding condensation into results which are comprehensible and valuable on their own.

As an alternative, the researcher might use the diagram to think about an incremental way of uncovering associated concepts and relationships. The critical aspect is then to include concepts bit by bit in ever increasing detail without having to jump ahead with concepts yet insufficiently understood by the reader.

The result of this step should be an ordered list of bullet-points, which can be turned into draft sections and reordered easily.

4. Expanding the list of bullet-points into draft and then final sections is the last step in the process of Selective Coding. For this research, I tried to follow the respective chapters from the literature

<sup>43</sup>A category is a well-developed, well-integrated concept.

on GTM [78, pp.151–176][105, pp.275–295] and found that discussion with a group of peers over initial drafts helps considerably to clarify and focus a write-up.

5. To support the write-up, I found the following devices to be useful:

- **Verbatim quotations and links into data**—The most basic way to help the reader understand a conceptual development is to provide direct, verbatim quotations from data. Throughout the thesis you will find these in many places, in particular if the mailing list participants themselves have attained a high conceptual level with their remarks. More often than verbatim quotations though, the thesis will contain references to the actual data from which results were derived to provide the necessary grounding on which GTM relies. Since for this research all e-mails are available via Gmane.org, I was able to put hyperlinks into the electronic version of this document that take the reader directly to the referenced e-mail. For the paper-printed version, the links given as `<projectidentifier>:<id>`—for instance `[kvm:1839]`—can be manually resolved by looking up the exact name of the mailing list in Section 3.6, Table 3.1 and then browsing to `http://article.gmane.org/<listname>/<id>` (for the above example the resulting URL would be `http://article.gmane.org/gmane.comp.emulators.kvm.devel/1839`).
- **Links to concepts/Glossary**—For each concept referenced and discussed in the research, it makes sense to include its definition in a separate glossary. Otherwise the reader has to maintain the understanding of the concept throughout the rest of the text or remember the location where it was explained in detail. Employing a glossary will provide for an easy way to look up a condensed explanation of a concept that might have lapsed from the reader's mind.<sup>44</sup>

For this research, such a glossary was exported from the software GmanDA used for managing the GTM process (see Section 3.4 for a discussion of GmanDA and Appendix A for the glossary). In the electronic edition of this thesis, all glossary terms are hyperlinked from the text directly into the glossary.

Altogether, it can be recommended to read the main result section of this thesis on a computer screen or tablet computer, as it enables to seamlessly jump to both the definitions of concepts and the data in which the insights are grounded.

- **Graphical representation**—Where possible and applicable I have tried to create a graphical representation of the data and resulting theory developed, which often is a reduced and cleaned version of the diagram produced in the second step of Selective Coding as described above (see for instance Figure 5.4 or 5.5).<sup>45</sup>
- **Descriptive overviews**—The analytical development of a concept is often brief about the actual episodes from which the theory was derived. Therefore, it was found to be beneficial to provide the reader with more detailed accounts of episodes someplace else. This led to Appendix A.1, in which all relevant episodes are explained. For the use in other publication venues where such an appendix is not possible, I also experimented with “info boxes” woven into the text (one box is included in this thesis, see Info-Box 1 on page 106). While it must always be possible to follow the theoretical development without a summary either in the appendix or an info box, this should enable an interested reader to explore the episode in more detail.

<sup>44</sup>In the electronic version, the use of the keyboard shortcut for going to the previous view (commonly ALT+LEFT) is useful to continue reading after such a jump into the glossary.

<sup>45</sup>In particular, such diagrams were focused to show the strongest, best explained relationships without forcing too much information onto a single display.

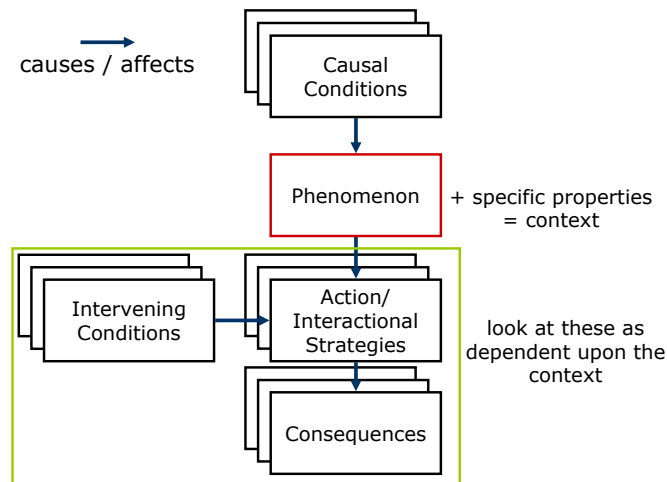


Figure 3.2: The paradigm model [493] graphically represented using a logic frame. Arrows pointing between elements represent that the pointed-to element is caused or affected by the pointing-from element.

### 3.2.4 The Paradigm

Strauss and Corbin describe the paradigm as “a perspective taken toward data [...] to systematically gather and order data in such a way that structure and process are integrated” [493, p.128]. To achieve this end, the paradigm suggests to take a concept of interest (called the phenomenon) and analyze for each occurrence of the phenomenon in data (1) the set of conditions that have caused the phenomenon and those that manifest its properties (those are called the context<sup>46</sup>), (2) the set of strategies employed by persons as a reaction to a phenomenon, and (3) the consequences of such interaction. Strauss and Corbin explicitly pointed out the danger of becoming stuck in the structure of the paradigm, following its prescribed model of analysis too rigidly, and reducing the analysis to simple cause-effect schemes that lack deeper analytical insights [493, 105].

The paradigm as given by Strauss and Corbin primarily improves our analytical abilities regarding the *reaction* to a phenomenon (as can be seen graphically in the representation of the paradigm shown in Figure 3.2) but is assuming a static and restricted view of the causes of the phenomenon. It can be hypothesized that this alignment towards reactive analysis is caused by the kind of phenomena that Strauss and Corbin investigated such as pain and dying, which are emergent from a context rather than the products of actions by people. In this research it is much more important to also understand how phenomena were caused by conscious action. That is, we want to understand the dynamic aspects that led to each phenomenon. For this thesis, I have therefore extended the paradigm and split each causal condition into three aspects: First, we need to know which actions and strategies have caused the phenomena to occur (*action/causing strategy*), second, we need to understand from which starting point these actions were taken (*the starting context*), and third, which conditions influenced the actions taken in a way relevant to affecting the phenomenon (*intervening conditions*). The resulting modified paradigm model can be seen in Figure 3.3.

A second insight about the paradigm is related to its ability to aggregate multiple occurrences of a phenomenon. From the description of the paradigm given above it might seem that the paradigm is a method for integrating several occurrences of a single phenomenon into a more aggregated abstraction. In this research I have used the paradigm only as a tool that can be applied to a single occurrence

<sup>46</sup>The term “context” is thus not used in the sense of “a context in which the phenomenon appeared”, but rather the context is the concrete instance of an abstract phenomenon in the data. The term highlights the importance of all phenomena only being of interest if they serve as a context to further action and events.

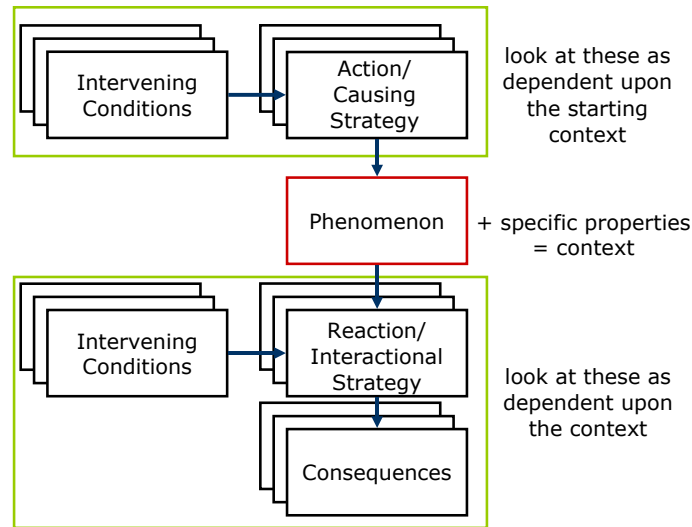


Figure 3.3: The paradigm model extended with three more elements to emphasize that the phenomenon is caused by *actions* that occur in a *starting context* and are influenced by a separate set of *intervening conditions*. Arrows pointing between elements represent that the pointed-to element is caused or affected by the pointing-from element.

of a phenomenon in the data, as can be seen in Figure 5.3. I have not found any indication that the paradigm provides any help in integrating such *frames of analysis* or occurrences of the phenomenon to construct higher conceptual abstractions beyond providing a comparable structure for each frame. Grounded Theory Methodology does not provide any explicit tool for these kinds of abstraction, but assumes that the generic process of performing GTM can be applied to the resulting frames again.

### 3.3 Use of GTM in Related and Open Source Research

A final look at the uses of GTM in Open Source research should give the reader a feeling for the types of questions being answered with GTM and how the respective authors use the methods to derive results.

West, O'Mahony and Dedrick conducted a series of studies on questions of commercial parties interacting with the Open Source world using GTM on interview data [386, 544, 139]. While the resulting theories from this research are rich, presented with quotations from the interviews and well researched, the process by which the results were derived is not explained to the reader. Neither are steps of Open, Axial, or Selective Coding presented nor are results phrased as concepts with properties.

Similarly, Dobusch discusses the migration of the city of Munich from Microsoft to Linux based on interviews with city officials, which he condensed into 692 paraphrases before coding [150]. Using a dichotomy from literature, he classified the resulting categories into two camps and then presented a quantitative overview of the most often used categories in each [150].

De Paoli et al. investigated issues surrounding licensing in two Open Source projects by looking at 298 e-mails from the developer and user lists [137]. Citing extensively from these e-mails, the authors discuss arguments and points the discussion participants made. Unfortunately, the authors start their analysis with a theory already in mind (namely Actor-Network Theory, see Section 6.3), and much of their analysis only consists of applying the theory's concepts, for instance "boundary object", to phenomena occurring in data [78, cf.165ff.].

Elliot and Scacchi investigated the GNU Enterprise project (GNUe) for beliefs, values, and norms using GTM on e-mails, IRC transcripts, summaries of both as offered by the Kernel Cousins website, the

project website, and interviews [160, 161, 162, 163]. Similar to the other studies they quote from their sources extensively, but also provide a conceptual diagram, which appears to show the application of the paradigm (see previous Section 3.2.4).

Shah studied the motivations for participation in Open Source projects. By using GTM on interviews, e-mails from mailing lists, and information from the project website, she succeeded in deriving some new results that were missing in the extensive literature until then [461]. The study also provides some details about how participants for theoretical sampling were selected and how coding was performed. She describes that after naming concepts she looked for additional occurrences and discarded concepts if they could further be substantiated. The steps from the resulting set of concepts and sub-concepts to theory are not explained except that she analyzed the concepts associated to individuals based on intra- and intergroup differences [461].

Sandusky and Gasser looked at bug-reports and using GTM focused on developing the concept of negotiation as pertaining to the discussion between bug-reporter, developers, and other interested parties [445]. By focusing on a single concept from the start, their work can go into more detail and is the only paper I encountered which describes properties and dimensions of the concept under study (for instance, the issue being negotiated, the level of intensity or the resulting effect). Instead of quoting selectively along the discussion of a certain point, Sandusky and Gasser take the time to present three complete examples, in which they annotate their quotes with the associated issue type identified, and after each example shortly discuss the derived insights [445, p.191–93].

To sum up:

- GTM is a method which has found some uses in Open Source research.<sup>47</sup>
- The use of interview data for GTM is more common than direct observational data, most likely because interviewees can raise issues to higher conceptual levels.
- Paraphrasing has been observed as a technique to reduce the amount of data prior to coding [150].
- Only rarely are intermediate steps from Open or Axial Coding presented in publication. Equally rare is the use of conceptual diagrams to summarize a resulting theory. Rather, the dominant way to present results is via extensive quotation in support of the theoretical discussion.

### 3.4 GmanDA – Tool Support for Qualitative Data Analysis of E-mail Data

When Grounded Theory Methodology emerged as the primary research method of this thesis, the desire for tool support to perform qualitative data analysis became quickly apparent [442, cf.p.238] because of two related factors: (1) Just as with any quantitative method it is necessary to organize and manage the data used in the analysis. (2) The tasks and workflows of following Grounded Theory Methodology and their resulting intermediate outcomes need to be captured and streamlined to become reproducible and efficient (see the previous Section for how GTM was used). It should be stressed that there is nothing in GTM which precludes it from being performed without any software support, but that the amount of data and the typical workflow can easily profit from specialized tooling [105, p.xi]. For instance, the first month of coding was performed without any tool except an Internet browser and a text editor, using Internet URLs for identification of data and textual search for returning to previous intermediate results.

Unlike quantitatively analyzed data sets, which often can be stored conveniently in spreadsheets and in most cases will have a uniform structure given by the research design and measurements being taken, forcing qualitative data into such a structure is likely to fail. This is due to (1) the exploratory nature

<sup>47</sup>A study on the use of empirical methods in OSS found 3 of 63 studies to have used GTM [490], while a literature survey by Crowston et al. found that 10% of studies each used interview or observational data, which is likely to be an upper bound for the application of GTM [119].

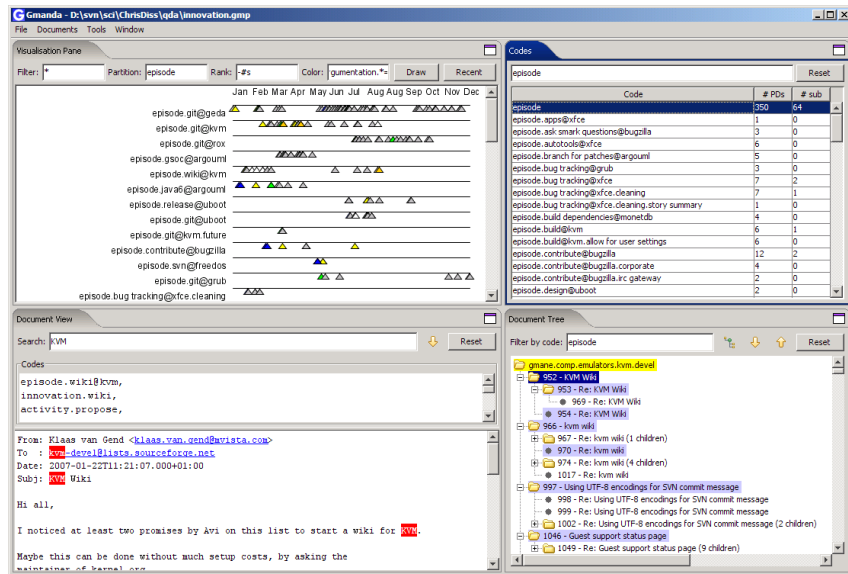


Figure 3.4: A screenshot of GmanDA showing the four views (clockwise from top left) Visualization, Codes, Document Tree, and Document Detail.

that many qualitative research endeavors entail, and (2) the wealth of qualitative phenomena that would result in sparsely filled matrices, if one tried to put a concept each into a column of its own. Thus, it is common practice to deploy coding schemes and explicitly note the occurrence of a phenomenon labeled by a code.

GmanDA

To enable such a coding-based data analysis, I began to develop the qualitative data analysis (QDA) tool *Gmane Data Analyser* (*GmanDA*), after trials with available QDA tools like ATLAS.ti [357] and the Open Source Weft QDA<sup>48</sup> proved unsuitable for dealing with the large number of hierarchically-ordered primary documents of which a single mailing list can consist. Other commercial alternatives such as MaxQDA<sup>49</sup> and Nvivo<sup>50</sup> (the successor of Nudist) have not been considered because their basic assumptions about the magnitude and nature of data under analysis appears to be similar to the one of Atlas.ti.

Over time, GmanDA has grown beyond being a tool to import mailing list data and attach codes to e-mails. In particular, it has since been improved to help the researcher manage both a growing set of codes and conduct Axial Coding via visualization, memoing, and cross-tabulation, explained in detail below.

GmanDA is licensed under the GNU General Public License version 3.0 (GPLv3) [196], consists of 17,600 lines of code excluding documentation in 291 classes, and is available for download from <http://gmanda.sf.net>.

Future Work

As future work, one might consider to extend GmanDA (1) to integrate support for data from source code management systems, such as for instance in [138], and combine these into the mailing list displays, (2) to enhance social network analysis possibilities which are currently available only via exporting to external applications (see below), and (3) to think about automating the workflows the researcher currently has to do manually<sup>51</sup> to enable more large-scale and fine-grained analysis of online communities [24, p.162].

<sup>48</sup><http://www.pressure.to/qda/>

<sup>49</sup><http://www.maxqda.com>

<sup>50</sup>[http://www.qsrinternational.com/products\\_nvivo.aspx](http://www.qsrinternational.com/products_nvivo.aspx)

<sup>51</sup>This last point is a sizable challenge likely to consume a number of Ph.D. students [429].



### 3.4.1 User Interface

GmanDA is intended to be used as a tool for streamlining the work following Grounded Theory Methodology on mailing list data. It is highly specialized for use with mailing list data in `mbox`<sup>52</sup> format which it can download from [Gmane.org](http://Gmane.org) (see Section 3.6) directly, but able to support data collections consisting of textual documents obtained from other sources as well. GmanDA is focused on supporting Open and Axial Coding in the Grounded Theory Methodology and aids Selective Coding by providing a well-structured interface to access data annotated with codes. GmanDA is not and does not attempt to be a general knowledge representation system, possibly including reasoning and inference, nor does it provide an ontology language or system. Rather, it assumes that the researcher must be supported to manage a collection of concepts and must perform all inference herself.

An overview screenshot can be seen in Figure 3.4 which shows the document tree, coding view, document view, visualization, and the code list, which will be discussed in turn.

- **Document View**—The central data structure used by GmanDA is called a primary document (PD) representing a single e-mail sent to a mailing list. Each document consists of the message body and a meta-data header which lists important information such as the author and title of the e-mail. GmanDA's document view performs primitive syntax highlighting on commonly used conventions in e-mail communication [414] such as enclosing text in a star symbol \* to denote an emphasis, or the use of the greater-than sign > to quote<sup>53</sup> from the replied-to e-mail.
- **Document Tree**—In contrast to related software such as Atlas.TI in which primary documents are presented in a list, the particular focus on mailing list data has caused GmanDA to organize primary documents into a tree structure in which child nodes represent replies to the parent PD. The document tree is fully searchable by a filter text field, which can reduce a large number of e-mails into a subset matching of message, threads, or e-mails and their replies. Filtering is possible by (1) the codes which have been attached while performing GTM, (2) the text of the e-mails, and (3) their metadata.

Aggregate statistics can be exported for selected messages, threads, or whole mailing lists in the following format: (1) As a list containing the number of e-mails written per author, (2) as a tabular summary of all e-mail activity usable for statistical analysis outside of GmanDA, for instance in the statistical package R [261, 428], and (3) as a social network graph in GraphViz .dot format [165, 205] and GraphML [63] for use in social network analysis, e.g. [245, 32] (see Section 6.3).

- **Coding View**—During the initial conception of GmanDA, a level of granularity in which single e-mails are the unit of coding was chosen because it best represents the discourse level at which the analysis in this thesis is located (compare with the meta-model in Section 3.2.1.3 and see the discussion on segmentation in 3.2.1). Thus, for any primary document there exists a (possibly empty) set of codes attached to this document. This is unlike most other qualitative data analysis programs, in which the primary elements of the user interface are aimed at segmenting and coding the fragments of primary documents (called “quotations”) down to the character level [357, p.355]. By making this design decision, GmanDA is much more user-friendly when annotating large numbers of primary documents at a high rate, because user interaction involves only selecting a message and entering codes. In contrast, GmanDA lacks usability when analyzing individual large documents consisting of segments which the researcher wants to discussed separately. Yet, I found that if the researcher finds the need to code at a more fine-grained level than single e-mails, then the level of analysis is more likely to focus on the speech-act level and not on the innovation discourse level [81, pp.286f.].

When designing the interface to enter the set of codes attached to a primary document it was chosen to develop a syntax-sensitive text editor over a graphical user interface. This decision

<sup>52</sup>`mbox` is a mail archival format preserving the e-mail in representation according to RFC-2822 [424].

<sup>53</sup>For an overview of quoting in e-mail-based discourse see [250, 158].

has reduced the development effort, but the primary reason to use a textual interface was to maximize coding efficiency by turning it into the process of entering a comma separated list of codes with possible nesting using curly braces. Defining new codes is simply done by writing them down on the fly and no extra interface is necessary.

While my subjective experience with using GmanDA for the course of this thesis supports the choice of a keyboard-based user interface over a mouse-based one, this has been argued to be a common misjudgment by users about their own performance [506]. For more general tasks than coding such as administrative record keeping [352] or file management [545] the use of mouse-based input has been shown to be faster than keyboard-based operation via commands both for experts and novices. For more specialized tasks such as the entry of boolean queries, results indicate that textual entry can be both faster and more accurate when designed well [86].

As has been described in more detail in Section 3.2.1.4, the coding syntax used in this thesis and GmanDA can be paraphrased as a set of hierarchical key-value pairs or a tree structure of keys with values as leaves in the tree. The semantic interpretation of such a coding tree attached to a primary document is that the concept represented by the key at the tree's root is abstracting a phenomenon occurring in the document. Nested keys to such a root concept then provide further conceptualization to the given instance of the phenomenon. Attached values describe the phenomenon in concrete terms which do not need or cannot be abstracted into the general. For example, a project participant writes an e-mail containing the sentence "We should switch to Bugzilla as our bug tracking system!". This can be conceptualized as a proposition of the participant to introduce a new bug tracking system and be coded as:

```
activity.propose: {
  innovation.bug tracking: "Bugzilla"
}
```

In this coding example, the proposition is put into the center of the analysis with the innovation being proposed becoming a nested concept of the proposition. Second, the concrete bug tracking system being proposed (Bugzilla) is explicitly outside of the conceptual world by being stored as a descriptive value rather than a key.

To support coding using such a textual interface, the following mechanisms were integrated into the Coding View: (1) To maintain the hierarchical structure of codes in a readable fashion, the coding view allows the user to reformat the display to correct indentation and formatting of the code tree. (2) To support recall of existing codes and their definition, a drop-down box of all existing codes can be opened inside the editor, and prefix matching text completion (equivalent to tab completion in command line shells) is available for all existing codes.<sup>54</sup> (3) Common templates needed during coding such as attaching a memo to a code, adding a current date/time-stamp, or opening a nested code are available via hot keys.

From an implementation view, GmanDA uses a robust hand-written parser<sup>55</sup> which maintains a live internal syntax tree to keep textual representation and internal code model in sync at all times.

- **Code List**—The list of all codes used in the primary documents of a research effort is displayed in the code list view. The two initial uses of the code list were to aid the researcher in finding existing codes during coding in the coding view, and to allow to select codes to be used as filters in the primary document tree view. Since auto-completion is available in both views, these

<sup>54</sup>Extensions to the text completion implementation such as predicting the code the user wants to enter [70] or allowing codes to be abbreviated during completion [233] are well possible within the existing framework, but unlikely to be beneficial for the task at hand, because the key use case for which the text completion is needed is remembering a precise code already used to label the occurrence of the same or similar phenomenon in a different context. In this use case, having a predictable order of entries to look through is of a greater advantage than a text mining algorithm making suggestions, or saving seconds by omitting letters.

<sup>55</sup>The parser easily could have been generated by a parser generator such as ANTLR [400], but the additional control about intermediate parsing results and the easy coding syntax made it unnecessary to reuse existing parser technology.

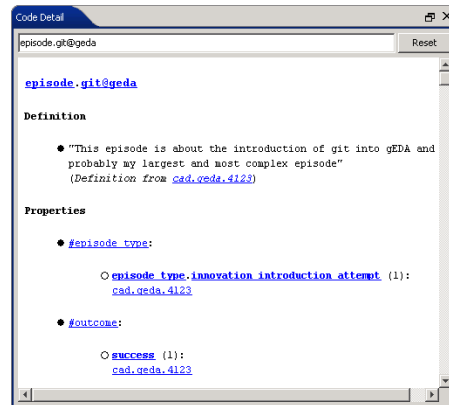


Figure 3.5: The Code Detail view in GmanDA. Its purpose is to provide information such as definitions, properties, and occurrences for a given code to aid with the dimensional and relational development of concepts.

intended roles of the code list have turned out to be of lesser importance, and the code list now primarily serves to browse existing codes and their sub-codes.

- **Code Detail**—This view (see Figure 3.5) is used in GmanDA to display all available information on a single code to support the dimensional and relational development of concepts. The view displays (1) the code's definition(s), (2) the properties of the concept behind the code, (3) the primary documents which were annotated using the given code, (4) the values assigned to the properties of the concept, and (5) any memos and sub-codes used to describe, discuss, or analyze the concept displayed. Because of this wealth of information on a single code, the code detail view has become the central view when performing Axial Coding in GmanDA. It is here that the researcher will reread existing insights gathered about each concept under scrutiny. To aid the researcher during Axial Coding, all references to other codes, primary documents, or definitions are linked from the view to facilitate quick navigation from data to data or code to code. By providing both such linked displays of code while retaining the expressive powers of plain text entry, it is hoped to achieve both a strongly verbal access to managing data and strong visual and hyperlink support [28].
- **Time Series Visualization**—To provide a view on message data that emphasizes the temporal aspect of e-mail-based communication, which is easily lost when looking at a tree structure of e-mail communication in which two e-mails follow sequentially no matter whether two hours or two weeks passed in between, the time series visualization view was devised. The processes and possibilities of visualizing primary documents and associated codes when those documents can be mapped to a date and time is described in Section 3.5. This short description focuses on the use of the interface available in GmanDA:

First, the researcher will filter the set of all primary messages to those matching a set of codes or a free form search term. For instance, one might be interested to see all e-mails coded with *concept.partial migration* containing the name of the maintainer of a project. Next, the researcher specifies a code for partitioning which takes the set of filtered primary documents and using the sub-codes of the partitioning code separates the data set into tracks displayed beneath each other. Alternatively to codes, the meta-data attached to the primary documents can be used as well. For instance, the names of the authors who wrote a set of e-mails can be extracted from the metadata of these e-mails and then be used to separate this set into tracks. Third, these tracks can be ranked by date or cardinality. Last, it is possible to colorize messages based on the codes present, for instance by coloring positive feedback of an innovation proposal green and negative feedback red.

Tabulation View

X: `opose, #enactment scope` Y: `#outcome,2` Group by: `episode` Filter by: `episode` [Update] [Recent]

☐ Show missing X properties ☐ Show missing Y properties  ☒ Remove empty rows & columns

#enactment scope → #outcome ↓	value.high	value.high and low	value.low
outcome.failed	<a href="#">design approval@bugzilla</a> <a href="#">bug milestone@xfce</a>	<a href="#">work groups@gnub</a>	<a href="#">gsoc@xfce</a> <a href="#">bug tracking@gnub</a> <a href="#">branch for patches@argouni</a> <a href="#">gpl3@geda</a>
outcome.success		<a href="#">git@geda.change log creation</a>	<a href="#">branching@geda</a> <a href="#">bug tracking cleaning@xfce</a>
outcome.unknown	<a href="#">ask smart questions@bugzilla</a> <a href="#">integrate document change log@xfce</a>		

Figure 3.6: The Tabulation view in GmanDA. It is used to analyze the relationship between two properties. In the figure, the properties *enactment scope* and *episode outcome* are tabulated after filtering and grouping for messages marked as belonging to an innovation introduction *episode*.

- **Tabulation View**—The code detail view discussed above can display all properties of a single code and the respective values of the occurrences of the associated phenomena, but it is often necessary during Axial Coding to see how the values of two properties are distributed. For instance, given the outcome of an episode as a property ranging from failure to success, one might be interested to see how abstractly the application of the proposed innovation is described to analyze and explore possible correlations and causations (this example resulted in a property called *enactment scope* which was developed as a concept and is discussed in Section 5.4).

To enable such operation, the Tabulation View as a place for cross-tabulating two codes has been created, which is shown in Figure 3.6 with data relating the aforementioned outcome and enactment scope. The figure shows that there is no clear-cut correlation visible in the data, even though there is no case of a successful innovation introduction after a highly abstract enactment scope. After it had helped to call doubts about the initial hypothesis that abstract enactment causes failure, the tabulation view proved a useful tool to structure the investigation into the individual cases.

Similar to the visualization view, a tabulation is created by first filtering for messages of interest. Then the concept of interest and two properties belonging to this concept can be given. The tabulation view will then plot each instance of this concept along the values of the given properties, possibly subsuming PDs under a common grouping. If no grouping is provided, then a list of all primary documents coded with the respective value is shown. Codes and primary documents are again hyperlinks causing GmanDA to filter for the codes or jump to the respective data.

### 3.4.2 Software Design

GmanDA uses a strict component-based architecture managed by a dependency injection container. This entails that every non-business object is created exclusively by dependency injection to facilitate loose coupling and a transparent design. The connection between the static application framework and the dynamic objects in the business domain is achieved via an observable framework. Both dependency injection and the use of observables are discussed below. This discussion is aimed at researchers looking to construct software for a similar purpose or extend GmanDA for their needs. Other readers are safe to skip to Section 3.5 on visualization.

GmanDA is built with a special focus of reusing existing implementation, so that all modules in GmanDA are application-specific or imported from libraries except for configuration handling and parts of the data

binding observables. GmanDA depends on the following libraries: GNU JavaMail<sup>56</sup> for processing of *mbox* files, XStream<sup>57</sup> for XML data binding, Batik SVG Toolkit<sup>58</sup> for SVG generation, Piccolo [36] as a 2D-drawing framework, GlazedLists<sup>59</sup> for observable list bindings, Apache Lucene<sup>60</sup> [239] as a search engine, Joda Time<sup>61</sup> as a replacement for J2SE date, Apache Velocity<sup>62</sup> as a templating engine to generate output such as LaTeX, DockingFrames<sup>63</sup> for managing dockable panels in the user interface, args4j<sup>64</sup> to manage command line parameters, and Apache Commons<sup>65</sup> for utility functionality missing in J2SE.

**Inversion of Control and Dependency Injection** One of the most basic problems in software system development is posed by the question of how an application can be assembled from components. This problem is essential since the separating of interface from implementation with the important advantages of abstraction and substitutability has thrown up the question of how we can choose an appropriate implementation for an interface [194].

Traditionally, the abstract factory pattern and the factory method pattern as described by [202] have been used to abstract the creation of an object of a certain implementation type. Apart from being difficult to use for the programmer [164], there are two disadvantages: (1) For abstract factories one needs to obtain a concrete instance of the factory, thus just another layer is added to the task of separating interface and implementation. Similarly, factory methods are typically implemented using static methods which cannot be used when a strict division between interface and implementation is desired, because interfaces cannot<sup>66</sup> include them [194]. (2) If factories are intended to be flexible in their possibilities to create objects of different types, and these types do require different dependencies, then factories need to be constructed with an interface which accommodates all these dependencies.

The solution proposed by the use of a dependency injection container is that control of providing an implementation for an interface should be reversed: Instead of putting the component in charge to acquire a suitable implementation, this responsibility is externalized to a central instance called the dependency injection container. From the viewpoint of an object interested in obtaining an implementation for a given interface using such a container entails that implementations are no longer chosen within the class, but rather injected from the outside by the container. This perspective from inside a single class has given the pattern its name.<sup>67</sup>

Modern implementations of this pattern such as Spring<sup>68</sup> or PicoContainer<sup>69</sup> commonly distinguish between constructor, setter, and field injection depending on how dependencies are provided. With constructor injection, dependencies are supplied as parameters to the constructors of an object, which has the advantage of dependencies being available during construction. Setter and field injection respectively inject dependencies into either annotated setters or fields after the object has been constructed.

GmanDA uses a combination of annotated field injection and constructor injection to create all singleton instances necessary to exist over the whole lifecycle of the application. This includes all UI elements, actions, observables, preference management, data abstract layer, and the GmanDA business logic

<sup>56</sup><http://www.gnu.org/software/classpathx/javamail/javamail.html>

<sup>57</sup><http://xstream.codehaus.org/>

<sup>58</sup><http://xmlgraphics.apache.org/batik/>

<sup>59</sup><http://publicobject.com/glazedlists/>

<sup>60</sup><http://lucene.apache.org/java/docs/index.html>

<sup>61</sup><http://joda-time.sourceforge.net/>

<sup>62</sup><http://velocity.apache.org/>

<sup>63</sup><http://dock.javaforge.com/>

<sup>64</sup><https://args4j.dev.java.net/>

<sup>65</sup><http://commons.apache.org/>

<sup>66</sup>This argument is Java-centric, but conceptually holds for other languages if a separation between interface and implementation is to be maintained.

<sup>67</sup>In the early days of dependency injection, the pattern used to be called “inversion of control”, which was deemed to general [193]. Rather, *inversion of control* is now seen as the higher-level technique and *dependency injection* as the concrete use of this technique.

<sup>68</sup><http://www.springsource.org/>

<sup>69</sup><http://www.picocontainer.org/>



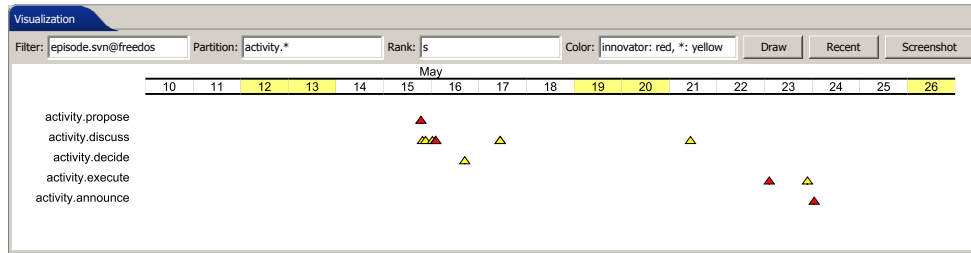


Figure 3.8: Example screenshot of the visualization built into GmanDA. A four-stage visualization pipeline with the operations filter, partition, rank, and color is used to generate the resulting visual display.

In the given example, the shopping cart would consist of a value observable for the total and a list observable for the items in the cart. From the perspective of the shopping cart implementor, the use of the observables replaces the use of plain variable fields.

In GmanDA, for instance, an observable value is used to hold the currently loaded analysis project. Interested parties such as the individual views in GmanDA or the title bar of GmanDA will thus be able to subscribe to the project observable being notified when the persistence layer set the value of the observable to a new value.

Since data binding observables add a lot of structured information to the observer pattern, additional possibilities arise. In particular, it is possible to use the contract defined by data binding observables to add additional behavior to derived observables such as buffering, delaying, vetoing, and filtering of events.

## 3.5 Visualization of Temporal Data

Before turning to the data sources used in this analysis in Section 3.6 we discuss the general question of how to visualize temporal data, which lies at the intersection of performing GTM discussed in Section 3.2 and using tool support discussed in Section 3.4. This section strives to abstract to some degree from both sections by providing a treatment of visualization of temporal event data that is independent from the question whether the underlying data are log entries from a log file or e-mails as used in GmanDA. A screenshot of the visualization support in GmanDA is given in Figure 3.8, whose underlying mechanisms are now to be defined. This section is kept in a formal style to be useful for the construction of tools for working with event data and can be safely skipped after taking in the following warning about using event visualization and regarding Figure 3.8.

Two words of caution should be given about using the visualization infrastructure: First, it is easy to come to wrong conclusions when using a temporal view on the events in the project, because much detail (including all the actual text) from the e-mail messages is lost in the graphical representation. The researcher should therefore take additional care to validate any results derived from temporal analysis by checking back with the full unfiltered mailing list. Second, graphical representations of information is preferred widely over textual ones when users of software are queried for their preference [302]. Yet, it is easy to miss crucial implications in complex graphical time series which would have been found, if textual summaries of the data on display had been available [302, 519]. Also, it often takes longer to comprehend graphical representations compared to textual ones [222].

It is thus recommended to avoid jumping directly from visualization to analysis result, but rather take the intermediate step of providing a descriptive summary. Such a summary helps to understand the underlying events, verify the information gathered from the graphical representation and avoids abstracting. Afterwards, this summary can then be used to make conceptual deductions. The process

in result should be akin to how GTM in general proceeds from raw unprocessed data via memos and diagrams to conceptual reasoning concluding in an integrated write-up of theory.

For related work on visualization infrastructure for temporal event data see for instance [534, 381, 7, 16, 87, 491]. It should be noted that the literature on visualizing e-mail conversations is separate, consider for instance [556, 523, 525, 279, 524].

**Events** We begin by defining  $E$  as the index-set of all events  $e$  of interest. For now, an event  $e$  should be treated as an opaque object without any internal structure. We then define  $\tau$ —the timing function—as **Timing**  $E \mapsto \mathbb{R}$  that maps events  $e$  to a real number that we take to represent a timestamp (for instance think of the number as the number of milliseconds since 1970-01-01). **Function**

**Tags** To attach meaning to the events, we define  $T$  as the set of all tags  $t$  that we might attach to an event  $e$ . For instance, we might use a certain  $t_1 \in T$  to represent a certain author, or another  $t_2 \in T$  to represent a certain type of output which appeared in a log entry.

**Coding** A coding  $C \subseteq T \times E$  is then used to represent which tags have been attached to which events. We found it useful to also define a partial order  $<$  on top of the set of tags  $T$  to denote a hierarchical sub-tag relationship (if a tag  $a$  is semantically a sub-tag of a tag  $b$ , then  $a < b$ ).

**Tag Filtering** The first operation we now introduce is the *filtering* function  $f : 2^E \mapsto 2^E$ , where  $\forall_{x \in 2^E} f(x) \subseteq x$ , which takes a set of events and returns a subset of them. In particular, a *tag filtering* function  $f_{t,C}$  is a *filtering function*, which returns the set of all those events that are tagged with  $t$  in the coding  $C$ . Formally,  $\forall_{x \in 2^E} \forall_{y \in x} y \in f_t(x) \Leftrightarrow \exists_{s \in T} s \leq t \wedge (s, y) \in C$ . Another useful filtering function uses the timing function  $\tau$  to reduce the set of all events to those which occurred in the interval  $[start, end]$ . Using such filtering, we can reduce the number of events we see on screen comfortably by naming the tags and time-periods we are interested in.

**Partitioning** The next operation we found useful is *partitioning*, which we define to be a function  $p_C : 2^E \times 2^T \mapsto 2^{T \times 2^E}$ , which receives a set of events and a set of tags and splits the set of events into subsets filtered by each tag (the *partitions*), i.e. for  $x \in 2^E$  and  $y \in 2^T$  we define  $p_C(x, y) = \{(t, f_{t,C}(x)) | t \in y\}$ . This operation is useful for splitting events along a second dimension (the first being time) such as the authors of e-mails or the programs from which a log entry originated. The most common application of this partitioning function is in combination with a tag  $t$  for which we want to partition along the sub-tags. This can be achieved by the set of all sub-tags for a given tag  $t$ , which we define as  $S_{\leq t} \subseteq T$ , with  $\forall_{x \in T} x \leq t \Leftrightarrow x \in S_{\leq t}$ . Also, the direct sub-tag function  $\bar{S}_{\leq t} \subseteq T$  is often useful: It only contains those tags which are direct sub-tags of the given tag  $t$ . For  $\bar{S}_{\leq t}$  the following needs to hold  $\forall_{x \in T} x \in \bar{S}_{\leq t} \Leftrightarrow x < t \wedge \neg \exists_{y \in T} x \neq y \wedge x < y < t$ .

**Nested Partitions** Further, we can also *nest* partitions by taking each partition and applying the partitioning mechanism again. For a similar mechanism by which partitions can be built, see the table algebra in [491].

**Ranking** As a last processing step before being able to draw the event data, we use a *ranking* function  $r : T \times 2^E \mapsto N$  to define an order by which to print the partitions from a partitioning induced by a set of tags. Ranking functions we have found useful have been:

- The starting time rank function  $r_s$ , which ranks partitions in the order of their earliest event, i.e.  $(t_1, e_1), (t_2, e_2) \in T \times 2^E$  holds that  $r_s(t_1, e_1) < r_s(t_2, e_2) \equiv \exists_{x \in e_1} \forall_{y \in e_2} \tau(x) < \tau(y)$  (and respectively end time, median time etc.).
- Ranking based on the number of events in a partition function, which was used to find those episodes containing a sufficient number of events. Formally we can define  $r_{|\cdot|}$ , where for  $(t_1, e_1), (t_2, e_2) \in T \times 2^E$  holds that  $r_s(t_1, e_1) < r_s(t_2, e_2) \equiv |e_1| < |e_2|$ .
- Alphabetical ranking based on the name of the tag of the partition.

**Alignment** Given such filtered, partitioned, and ranked data, one could in addition *align* the partitions according to the first/last/median occurrence of a certain tag [534]. If, for instance, each partition represents for each project the events in the introduction of a new version control system, then aligning these



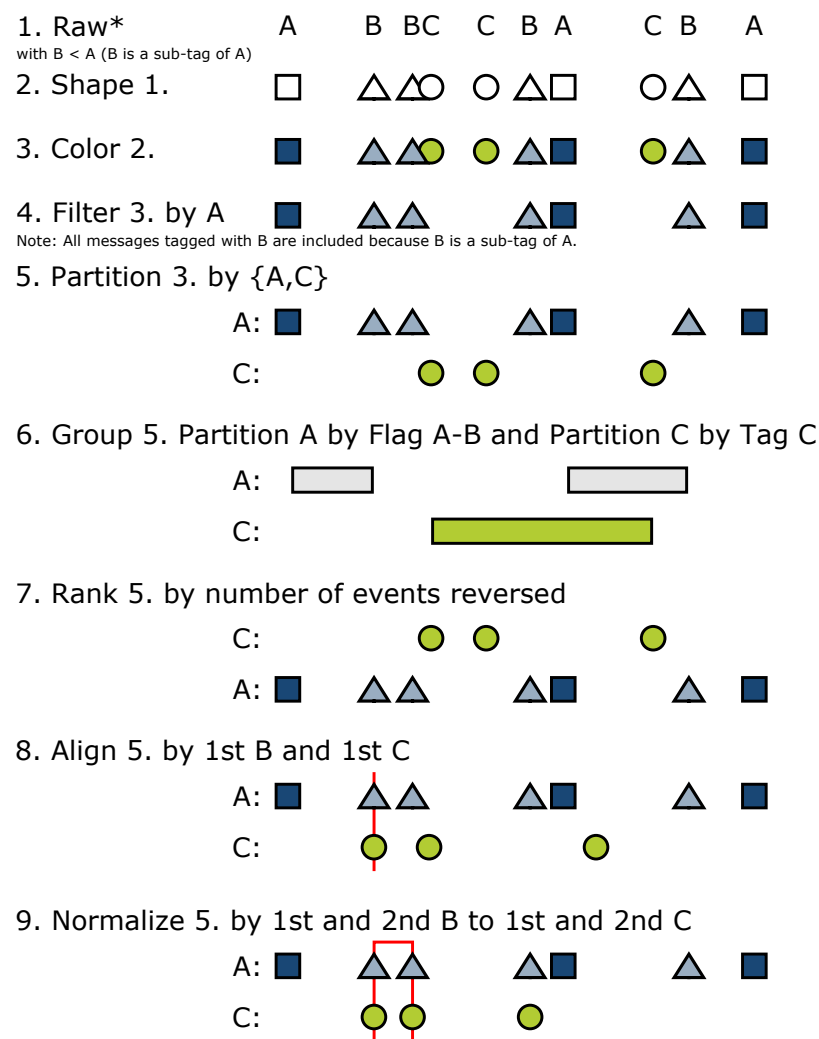


Figure 3.9: Operations for dealing with event data such as e-mail messages. The given example data uses three tags A, B, and C, where B is a sub-tag of A. Note\*: The letters in the first row (1. Raw) should be interpreted as events with one of the tags A, B, or C and positioned in time relative to their placement on the horizontal axis.

episodes to one another based on the e-mail, in which the introduction was first proposed, makes it much easier to compare both episodes, for instance regarding the duration of introduction phases or the moments when high-ranking project members were active in the introduction.

**Normalization** If interested in finding patterns, it might make additional sense to *normalize* the total time displayed in each partition (or the time between two events). In the above example this might help, for instance, to uncover the fact that even though one project took double the time to introduce the source code management system, the proportions for discussion, execution, and adoption were similar.

**Visual Mapping** As a last step in the processing pipeline we then take events and *map* them to visual marks. Currently, only coloring based on tags and a single type of marks is supported, but one could extend this to support changes to shape, size, orientation, color, and icons/textures [491] by tags as well.

If quantitative information is available about an event, this information can be used for styling the visual mark as well. Consider for instance the analysis of quotation depth of e-mail messages by Barcellini et al. [24]. The authors defined quotation depth of an e-mail as the maximum number of replies leading from this e-mail to a direct or indirect reply which still contains part of the e-mail as a quotation. Using this definition for a function  $q : E \mapsto \mathbb{R}$  which returns the quotation depth regarding an event representing an e-mail message, we can use  $q(e)$  for any given  $e \in E$  to scale the size of the visual mark appropriately. Thus, e-mails which are deeply quoted draw more attention.

**Event vs. Activity Visualization** Concluding, we want to relate this way of visualizing event data to the visualization of activity data that Salinger and Plonka use in their analysis of pair programming sessions [443]. Instead of having atomic events without any temporal extent, their visualization consists primarily of activities with beginning and end times. Both the preference in this thesis for events and theirs for activities can be easily explained through the type of underlying data used:

This thesis uses e-mails, which are atomically timestamped upon being sent, while Salinger's and Plonka's work uses video, which must be segmented into activities to capture the participants saying or doing something. Since the perspective on e-mails having an atomic time associated with them is not realistic, considering for instance at least the activity of writing the e-mail occurring in the background, two basic strategies will be offered to map events to activities:

1. Using two (thus semantically related) tags, we can interpret occurrences of events tagged with one of them as flagging the start of an activity and the other as flagging the end of an activity. Repeated occurrences of start events are ignored until an end event signals the end of an activity. For example, when working on e-mail data regarding the release process of an Open Source project, we can define the period in which features may be added to the code base (the *merge window*) as follows: It is the time between any number of consecutive e-mails stating that features may currently be added to the code base and the first following message stating that only bug-fixes and localization may be added to the code base (the *feature freeze*).
2. We can interpret the time between the first and last event marked by a given tag as the time during which an activity occurred. For instance, given three e-mail messages tagged as belonging to a certain innovation episode, we can define the period during which the episode occurred as the time between the first and last e-mail tagged with the episode.

An overview of the operations for dealing with event data is given in Figure 3.9.

**Zoomable UI** Once the events have been rendered to screen, GmanDA supports interaction using the zoomable user-interface paradigm based on the Piccolo Toolkit (successor of Jazz [37]). Most notably this includes the use of a semantic node for presenting the time line: First a time line shows the months of the observed year but when zooming in individual days become visible.

## 3.6 Data

After both Grounded Theory Methodology as the underlying research method and GmanDA as the used tool support have been explained, this section discusses how mailing lists as the primary data sources were chosen and which ones were included in this thesis. These aspects have to be seen in the context of the survey method having failed to gather responses as discussed in Section 3.1.2, and this thesis moving towards a retrospective qualitative analysis of past innovation introductions. At first, mailing lists were picked ad-hoc based on suggestions that I received upon asking for participation in the survey, but when it became apparent that performing GTM on such mailing lists was viable for this research, a more consistent and well-defined way to acquire mailing lists was sought to achieve the following goals:

1. Most importantly, I wanted to gather interesting data; data which GTM calls “rich” [78, pp.14f.] and which could serve as a foundation for interesting results.
2. The projects to be included should be “typical” Open Source projects to provide some basis for generalization (see also the discussion on validity in Section 3.7).
3. Finding projects and their mailing list needs to be possible efficiently, in a reproducible fashion and with a consistent interface for later data extraction.

I found that the use of a mailing list archive would help with the first goal (interesting results) and achieve the third goal (efficient operation). The first and second goal (typical projects) can then be achieved by defining a set of criteria by which projects and their lists are included in this research. Both the use of mailing-list archives and the selection criteria will be presented in turn, starting with the selection criteria.

### 3.6.1 Selection Criteria

The following criteria were used to select projects which are interesting (goal 1), typical (goal 2) and manageable (goal 3):

- **Open Source Development Model and Open Source License**—As given by the context of this thesis (see Section 1.2).
- **Aliveness**—While the Open Source development model with its open process and open collaboration is a requirement for interesting insights to be gathered, without a sufficient level of engagement in the projects<sup>72</sup> no interesting events to be gathered will occur at all.
- **Medium Size**—A small project is likely to be excluded on the base of lack of activity as by the previous criterion, but for this research we also explicitly want to exclude all big projects beyond 50 developers and in particular foundation and distribution projects as for instance Debian, KDE, the Linux Kernel, or Mozilla. The first reason for this exclusion is because these projects are big enough for dedicated leadership such as the board of the Mozilla Foundation and Mozilla Corporation, which steer and guide the project, making these projects sufficiently different from mid-size projects, driven by one or two maintainers and 5–20 developers. The second reason is that projects larger than 50 participants produce so much communication that it would have required dedicated work on a single such project to uncover innovation introduction attempts and their outcome (see Section 4.2 on the work Martin F. Krafft has done on process evolution in Debian).
- **Basic Infrastructure Installed**—Last, projects were chosen only if they had already progressed past the setup of the fundamental infrastructure services such as a website, a mailing list, and a source code management system. Projects lacking these, it was argued, had too much groundwork

---

<sup>72</sup>The actual metric used was e-mails per year on the mailing list with 500 as the initial lower bound. In the final sample though, three projects were included which had less communication but were deemed interesting nevertheless.

to do before reaching the stages of development in which discussion and decision making about innovation introductions would become important enough.

### 3.6.2 Data Collection Site

Given this set of criteria, an efficient way to find projects complying with it had to be found. To this end, Gmane.org<sup>73</sup>—a mailing list archive started in 2002 by the Norwegian Lars Magne Ingebrigtsen to allow for a bidirectional mail-to-news gateway without expiration of mails<sup>74</sup>—was chosen to find and extract mailing lists for this research from. Gmane.org hosts 11,000 mailing lists as of fall 2009 for a total of 88 million messages. Comparable services are Nabble<sup>75</sup>, Mail-Archive.com<sup>76</sup>, and MARC<sup>77</sup>, yet none hosts the same number of mailing lists and provides a convenient API for downloading mailing lists as files in *mbox* format<sup>78</sup> to be used in GmanDA (see Section 3.4). Gmane.org also provides a hierarchical categorization of mailing lists,<sup>79</sup> which was used to browse all mailing lists belonging in the *gmane.comp.\** sub-hierarchy containing computer software-related mailing lists.

Using a mailing list archive such as Gmane.org aids the goal of finding interesting projects, because it (1) provides a natural filter for the long-tail [5] of very small Open Source projects which easily outnumber the medium to large ones [326], and (2) because the statistical information a mailing list archive can provide over individual mailing lists can be used to efficiently exclude lists which show limited activity.

A mailing list archive provides a filter for these small projects, because only once sufficient need exists to read the mailing list in retrospect or keep a record for backup-purposes, will somebody register the list with an archive such as Gmane.org.

### 3.6.3 List of Projects

Given the criteria listed above, for each of the following 13 mailing lists all e-mails of the year 2007 were downloaded from Gmane.org:<sup>80</sup>

- *ArgoUML*<sup>81</sup> is a UML CASE tool written in Java. It originated from work done by Jason E. Robbins as a student at University of California, Irvine [431, 432, 433] from 1993 to 1999 [579] and has become one of the popular Open Source UML tools available. In February 1999, ArgoUML was released as Open Source and in January 2000 it was moved to the *project hoster* Tigris<sup>82</sup>, which specializes on hosting software development tool projects. Despite UML 2.x being first published in July 2005, there are no plans to extend ArgoUML's support beyond UML 1.4 [575]. ArgoUML's maintainer during 2007 was (and in 2009 still is) Linus Tolke.
- *Bochs*<sup>83</sup> is an emulator for x86 hardware supporting many Intel processors up to the Pentium 4 architecture written in C++. MandrakeSoft (today Mandriva) bought Bochs from its lead-developer Kevin Lawton and released it under the Lesser General Public License [197] in March 2000 [568]. In April 2001, the project moved to the project hoster SourceForge.net and entered a phase of high activity [576]. Unfortunately, while still being actively developed in 2007 by the then maintainers Stanislav Shwartsman and Volker Ruppert, the activity of the project on the mailing list slowed considerably, making it the least active project in this research set.

<sup>73</sup>Pronunciation is with a silent G: [mān] [263].

<sup>74</sup>See <http://gmane.org/about.php>.

<sup>75</sup><http://www.nabble.com/>

<sup>76</sup><http://www.mail-archive.com>

<sup>77</sup><http://marc.info/>

<sup>78</sup>The API is described on <http://gmane.org/export.php>.

<sup>79</sup><http://dir.gmane.org/>

<sup>80</sup>As a result of researching the origins of some of these projects, the Wikipedia pages of these projects were expanded where appropriate.

<sup>81</sup><http://argouml.tigris.org/>

<sup>82</sup><http://www.tigris.org/>

<sup>83</sup>Pronounced [bōks] [263], <http://bochs.sf.net/>

- *Bugzilla*<sup>84</sup> is a web application for bug tracking written in Perl. Bugzilla was created by Terry Weissman at the Internet browser company Netscape as a replacement for Netscape's internally used bug tracking software in 1998 [581]. When Netscape released its Netscape Communicator as Open Source following their announcement from January 1998 [231], Bugzilla was opened as well for an Open Source development model [581]. The Bugzilla project is part of the Mozilla Foundation and was led by Max Kanat-Alexander, Dave Miller, and Frédéric Buclin in 2007 [582].
- *Flyspray*<sup>85</sup> is a web application for bug tracking written in PHP. Flyspray was founded by Tony Collins as a bug tracker for instant messaging client PSI [583] and released under the LGPL [197] in July 2003. While in 2007 the project was active and maintained by Florian Schmitz and Cristian Rodriguez [583], the development pace slowed considerably after the observation period ended because the maintainers did no longer have time to concentrate on Flyspray [flyspray:6813].
- *FreeDOS*<sup>86</sup> is an Open Source operating system mostly compatible with Microsoft DOS released under the GPL [196]. The project was started in 1994 by Jim Hall as a student at the University of Wisconsin, River Falls [571]. Hall announced the project as PD-DOS in June 1994, from where it quickly grew when Pat Villani and Tim Norman contributed a working kernel and a command line interpreter respectively [571]. The project was renamed to FreeDOS in the beginning of 1998, moved to project hoster SourceForge.net in 2000, and reached 1.0 on September 3, 2006. Since then it can be considered mature [571]. FreeDOS was lead by Eric Auer, Bart Oldeman, and Jim Hall in 2007.
- *gEDA*<sup>87</sup> is a collection of tools for electronic design automation. The project produces one central set of software tools called gEDA/gaf for schematics capture and several smaller pieces of software. The three biggest ones of which are PCB (software for designing printed circuit board layouts), Icarus Verilog (Verilog simulation and synthesis tool), and GTKWave (electronic waveform viewer). The project was founded in spring 1998 by Ales Hvezda [584], who was still leading the project in 2007. gEDA is hosted at federated hoster Simple End-User Linux (SEUL)<sup>88</sup>.
- GNU *Grand Unified Bootloader* (GRUB)<sup>89</sup> is a boot loader for x86 PCs written in C and Assembler. GRUB was initially developed by Erich Boleyn as part of work on booting the operating system GNU Hurd developed by the Free Software Foundation [570]. In 1999, Gordon Matzigkeit and Yoshinori K. Okuji made GRUB an official software package of the GNU project and opened the development process to the public [570]. GRUB as an official sub-project of the GNU project is hosted at Free Software project forge Savannah<sup>90</sup>. In 2007, GRUB was led by Yoshinori K. Okuji and struggled with producing a version of GRUB2 that is perceived as a worthy replacement of GRUB legacy [grub:2705].
- *Kernel-based virtual machine (KVM)*<sup>91</sup> is an Open Source project funded by the Israeli start-up Qumranet which was bought by Linux distribution vendor Red Hat in September 2008. The software KVM is a virtualization solution written in C, integrated with the Linux kernel using the virtualization instruction sets of x86 hardware, so that virtual machine images can be run using Linux as their hypervisor [281]. The project is maintained by Avi Kivity and hosted on servers owned by Qumranet [564].
- *MonetDB*<sup>92</sup> is an Open Source database with relational and XML backend written in C [60, 58, 59]. MonetDB (initially only called Monet) was first created by Peter A. Boncz and Martin L. Kersten as part of the MAGNUM research project at the University of Amsterdam [58] and is led and hosted by the database group at the Centrum Wiskunde & Informatica (CWI) in the Netherlands [587].

---

<sup>84</sup><http://www.bugzilla.org>

<sup>85</sup><http://flyspray.org/>

<sup>86</sup><http://www.freedos.org>

<sup>87</sup><http://www.gpleda.org>

<sup>88</sup><http://www.seul.org>

<sup>89</sup><http://www.gnu.org/software/grub/>

<sup>90</sup><http://savannah.gnu.org/>

<sup>91</sup><http://www.linux-kvm.org/>

<sup>92</sup><http://monetdb.cwi.nl>

MonetDB is licensed under an adaption of the Mozilla Public License [355] and was released as Open Source in September 2004 [588]. MonetDB is maintained by Stefan Manegold, senior researcher at CWI [586].

- *ROX*<sup>93</sup> is a desktop environment like KDE, Gnome, or *Xfce* following the “everything is a file” metaphor of the Unix world [67]. Instead of linking to programs from a start-menu, in ROX whole applications are represented as a single file which can be moved by drag-and-drop. To achieve this, ROX is centrally built around the file manager ROX-Filer and uses a variety of independent tools and components to become a full-featured desktop environment.

ROX is written in C in combination with Python as a scripting language and is hosted at SourceForge.net and on a *private server*. ROX was started by Thomas Leonard while a second-year student at University of Southampton,<sup>94</sup> first released in 1999 [572], and has been led by him since [539].

- *Request Tracker (RT)*<sup>95</sup> is a web-based ticketing system written in Perl [589] for use in support and task coordination. It has a web-front end for use by the support staff and supports sending and receiving tickets and updates to them via e-mail. RT was created by Jesse Vincent while a student at Wesleyan University and first released under the GPL [196] in 1996. In 2001, Vincent founded Best Practical Solutions to provide support and development for users of RT [565]. The project is hosted by Best Practical and in 2007 was led by Jesse Vincent and Ruslan Zakirov.
- *The Universal Boot Loader (Das U-Boot)*<sup>96</sup> is a boot loader specialized for use in embedded systems licensed under the GPL [196] and written in C and Assembler. It supports a wide variety of architectures and motherboards. U-Boot originated in work done by Magnus Damm on an 8xx PowerPC bootloader called 8xxROM [566]. When Wolfgang Denk moved the project to the SourceForge.net, the project was renamed PPCBoot because SF.net did not allow project names starting with digits [566]. In November 2002, the project was renamed again, when support had been extended beyond booting on PowerPCs [567]. The project U-Boot is driven by DENX Software Engineering GmbH, a company specialized in embedded systems and led by its founder Wolfgang Denk [573].
- *Xfce*<sup>97</sup> is a desktop environment similar to KDE or Gnome which strives to be lightweight and fast. The project encompasses a window manager, desktop manager, panel for starting applications, a file manager (‘Thunar’), and many other applications. Most parts of Xfce are written in C, hosted at SourceForge.net, and licensed under GPL [196], LGPL [197], or a (3-clause or modified) BSD License [516].

Xfce was started in 1996 as a Linux version of Sun’s Common Desktop Environment (CDE) by Olivier Fourdan, but has since evolved to become independently designed and the third most popular choice for a desktop environment for \*nix operating systems [278].

These lists contain in total 33,027 e-mails in 9,419 threads for the year 2007. Of these I read 5,083 e-mails in 2,125 threads<sup>98</sup>, and coded 1,387 e-mails in 788 threads. All in all 134 innovation episodes were found in the data. The threads containing at least one read e-mail consist of a total of 12,714 e-mails, which represents 38% of all e-mails. The corresponding number of e-mails in threads in which at least one e-mail is coded is 4,836 or 15% of all e-mails. This gives a hint of the magnitude of innovation-introduction-specific behavior in OSS projects.

While the actual selection of these projects proceeded alphabetically based on the criteria discussed before, some attention was given to select projects pairwise based on product or process attributes. In

<sup>93</sup>An acronym for *RISC OS on X*, <http://roscidus.com/>

<sup>94</sup>Personal communication with Thomas Leonard, October 2009.

<sup>95</sup><http://bestpractical.com/rt/>

<sup>96</sup>*Das U-Boot* is German for *submarine* and probably a play on words on the German submarine movie *Das Boot* by Wolfgang Petersen, <http://u-boot.sourceforge.net/>

<sup>97</sup>Originally an acronym for *XForms Common Environment*, <http://www.xfce.org/>

<sup>98</sup>An e-mail was designated automatically by GmanDA as read when it had been selected at least once, and a thread if at least one e-mail in it had been read.

Project	Identifier	Mailing list name
<i>ArgoUML</i>	argouml	gmane.comp.db.axion.devel
<i>Bugzilla</i>	bugzilla	gmane.comp.bug-tracking.bugzilla.devel
<i>Flyspray</i>	flyspray	gmane.comp.bug-tracking.flyspray.devel
<i>FreeDOS</i>	freedos	gmane.comp.emulators.freedos.devel
<i>gEDA</i>	geda	gmane.comp.cad.geda.devel
<i>GRUB</i>	grub	gmane.comp.boot-loaders.grub.devel
<i>KVM</i>	kvm	gmane.comp.emulators.kvm.devel
<i>MonetDB</i>	monetdb	gmane.comp.db.monetdb.devel
<i>ROX</i>	rox	gmane.comp.desktop.rox.devel
<i>Request Tracker</i>	rt	gmane.comp.bug-tracking.request-tracker.devel
<i>U-Boot</i>	uboot	gmane.comp.boot-loaders.u-boot
<i>Xfce</i>	xfce	gmane.comp.desktop.xfce.devel.version4
Linux Kernel	kernel	gmane.linux.kernel

Table 3.1: Mailing list names of the studied projects. To manually resolve a reference in the format `<projectidentifier>:<id>` (for instance `[kvm:1839]`) first look-up the mailing list name using the project's identifier (in this case, the project identifier is `kvm` and the mailing list name `gmane.comp.emulators.kvm.devel`), prepend the URL of the Gmane.org article server `http://article.gmane.org/`, and append a slash (`/`) and the numerical identifier (1839) to derive the final address (`http://article.gmane.org/gmane.comp.emulators.kvm.devel/1839`).

the product dimension these are boot loaders (GRUB, U-Boot), desktop environments (Xfce, ROX), bug and task tracking web applications (Bugzilla, Flyspray, RT), desktop applications for design work (ArgoUML, gEDA), and virtualization and operating system software (FreeDOS, Bochs, KVM). From the development process perspective we can group Bugzilla and GRUB as belonging to larger software non-profit foundations [387] (Mozilla Foundation and GNU Project), KVM, U-Boot, RT as being sponsored [543, 544] and controlled<sup>99</sup> by a for-profit entity (Qumranet, DENX Software Entwicklung, Best Practical), ArgoUML and MonetDB as resulting from academic work, RT and ROX as resulting from student activities, and Bochs and FreeDOS as software which has reached maturity.<sup>100</sup>

## 3.7 Threats to Validity

To conclude this chapter on methodology let us critically assess the threats and limitations posed by using GTM and selecting data in this particular way.

### 3.7.1 Credibility and Internal Validity

With the use of GTM as the primary methodology, this thesis has an important advantage to demonstrate credibility and internal validity: Readers can follow at any time the hyperlinks to the actual e-mails (see Section 3.2.3) and assess for themselves whether to agree with the conclusions.

What remains is the question, whether to trust the data used in this thesis. Relying on publicly visible e-mail messages from the projects' developer mailing lists misses all other types of communication in the projects' discussion spaces such as private e-mail, Internet chat, conversation on other mailing

<sup>99</sup>West and O'Mahoney use the term *autonomous* to denote Open Source projects which are not controlled by third party entities such as for-profit companies or government entities [544].

<sup>100</sup>An Open Source software is called *mature* in contrast to being *stable*, when it is fit for production use, has reached the original intended purpose and is expected to no longer be expanded with regard to features. Less than 2% of all projects hosted at SourceForge.net designate their software as being in this stage [100].

lists, and communication occurring offline such as personal meetings at developer conferences. Such communication was regarded (see the bibliography on “Additional Non-e-mail Documents” on page 291), but only when explicitly referred to from the mailing list and archived publicly. Similar restrictions hold for communication in documentation space (e.g. on websites and wikis) and implementation space (e.g. in the project’s source code management system) [442].

A threat to internal validity could arise if the reliance on the e-mails from the developer mailing lists in connection with missing knowledge from other sources led to a misinterpretation of the episodes.

Two arguments speak against this. The first is that results were extracted from episodes which were understood with a high degree of certainty. Episodes in which the primary chain of events and discussion points was felt to be missing have not been used for deriving results. Thus, a threat to validity could only arise in those cases where an episode as traced from the mailing list is coherent and complete. I am confident that such did not occur.

Second, the amount of relevant communication which never leaves a trace on the mailing list can be argued to be rather small:

There is a strong argument in Open Source projects to keep discussion public and archived whenever possible [313, p.8] to reinforce the collaborative nature of this mode of software production. Communication media such as IRC [161], bug trackers [113], or forums [447] are certainly used, but no medium can match the importance of the developer mailing list as the “cross-roads” where participants meet and coordinate [300]. If some discussion is kept private, it is argued that this can easily hurt the open participatory governance if participants perceive this communication to be restricted to a “secret council” [190, pp.36f.].<sup>101</sup>

When circulating a draft version of one of the case studies in Chapter 7 among the maintainers and core developers of the project involved, one of the maintainers supported this with the following remark:

“At one point you emphasize that you used only publicly available information and thus wouldn’t know about activities behind the scenes. I can only speak for myself, but I rarely communicate with the other developers in private. Therefore, I am pretty confident that there were no discussions about the adoption of unit tests that were not held on the public mailing list.”<sup>102</sup>

### 3.7.2 Relevance and External Validity

Regarding external validity and relevance, I should reiterate the importance of considering GTM as a qualitative method with a focus on theory construction rather than generalization; as Yin puts it: “limited-case qualitative research can only be generalized to theory, not to a population” [557] cited in [31]. Proponents of GTM would thus say that drawing a representative sample from a target

<sup>101</sup>While this social side of the question is in preference of a public discourse, the technical answer to the question often leads to controversy focusing on the question whether to perform “header munging” or not. If such munging is performed, the “reply-to” header in the e-mail message is overwritten by the mailing list software to point back to the mailing list. Thus, if a user chooses to reply to the e-mail, he will automatically have his answer be sent to the mailing list. While this adds convenience and acts to increase the likelihood of discussions staying public, the act of “header munging” has the drawback to lose any reply-to setting the author of the e-mail defined. This has led to the technical opinion of avoiding munging in RFC 2822 [424]. Instead, the use of a List-Post header field defined in RFC 2369 [17] is recommended to indicate that the e-mail was sent from a mailing list. Then the user’s e-mail client can provide a button to “reply-to-list”. However, uptake of this feature in e-mail clients has been slow and it could be argued that even choosing between Reply and Reply-To-List is cumbersome from a usability standpoint.

Header munging, because of these conflicting implications, is a topic for a *holy war*, i.e. a passionately and bitterly led discussion with sharply separated factions which “cannot be resolved on the merits of the arguments” [190, p.136]. The particular holy war of header munging is raised typically when a discussion participant found the mailing list to be configured in an unexpected, cumbersome way for an intended task (see for instance [flyspray:5208]). To avoid such discussion, the best answer derived from *one episode* in the project *Flyspray* about header munging and Karl Fogel’s practitioner guide suggests to choose one alternative and then to stick to it [190, p.57].

<sup>102</sup>Private communication with FreeCol maintainer Michael Burschik, October 2009.



population is only a proxy for gathering results of interest at a theoretical level. The insights gathered from this and other GTM studies therefore only make the claim that they represent relationships between theoretical concepts which have been deduced from real world data and which have been found to be consistent with observed reality [104, pp.319f.].<sup>103</sup>

This implies that while I am confident that the research results of this thesis are consistent with the observations made in data and abstracted accordingly as theoretical relationships between concepts, these relationships must receive wider validation by additional research to show generalization beyond my sample.<sup>104</sup>

An innovator who wants to forego such validation and generalize the results from this thesis to other projects, should be warned: The population from which this study draws its sample was defined to include only medium-sized projects (see Section 3.6.1 on the selection criteria). Consequently, the results are not meant to apply to smaller and larger projects. The innovator should consider the following:

Smaller projects, while making up the vast majority of OSS projects by number [326, 228], do not benefit from an innovator: They have so little coordination and management overhead that an introduction of an innovation in contrast to just contributing code is likely to be ineffective. Even if an innovation introduction is attempted, the results of this thesis will not be necessary: With four participants or less, these projects are probably dominated by their maintainer, thus reducing the introduction into a conversation with a single person.

Big projects, on the other hand, have already established large numbers of processes, practices, and roles [269] and are likely to even have institutions which reflected upon how to design them [43, p.59]. Changing such embedded and established processes and practices, based on the complex interplay of institutions and norms, will be challenging. Even with the insights from this thesis and an in-depth knowledge of the particular project, the innovator will still need considerable time and experience to advance into a position to influence the leadership circles of a project. Krafft's work at Debian (see Section 4.2) provides a good overview of how the particularities of a large project such as Debian with its unique constructs like the Debian Constitution [293] will change the work of the innovator.

However, generalizing to other medium-sized projects should be possible in many cases because the results in this thesis are based on a diverse sampling from many application domains including both volunteer-driven and corporate-dominated projects, multiple programming languages, and maturity states. When comparing with data from self-assigned classification schemes such as the SourceForge.net Trove software map<sup>105</sup> [106], there is only one notable gap in the sampling: There is a lack of software targeting end-users such as games, and multimedia and offices applications, which make up 25% of all applications hosted on SourceForge.net [100]<sup>106</sup>.

In contrast to software for system administrators and developers, which make up two thirds of the software applications [100], the user-base of end-user software will be comprised significantly by users without a technical background—as illustrated by the personas “J. Random End-User and his Aunt Tillie” that Raymond conjures up [417, p.191] to distinguish them from power- and lead-users [530], hackers, and software developers.

Generalizing the results from this thesis should thus be done cautiously with regard to such projects. For instance, one can imagine that frustrated [77], inexperienced users produce support requests of such low quality that, if occurring in large number, could force a project into one of the following coping strategies:

<sup>103</sup>As a side note, this characteristic of the results gathered using GTM does *not* change if theoretical sampling, i.e. the gathering of additional data until the theory is found to be saturated, is employed. This is because even then GTM is focused on theory and not population and thus does not include mechanisms for achieving generalizability, but only consistency.

<sup>104</sup>This is no weakness of this study. Few experiments in software engineering can claim population-wide generalization, because they very rarely define the sample population adequately [464, p.741].

<sup>105</sup>Trove metadata include topic, intended audience, license, translations, user interface type, development status, target operating system <http://sourceforge.net/apps/trac/sourceforge/wiki/Software%20Map%20and%20Trove>.

<sup>106</sup>Even when looking beyond the number of projects, to usage, activity, and development status, does end-user-focused software maintain a similar degree of importance in the Open Source world [117].

Either it may spur innovation introductions to deal with increased demand for support [bugzilla:6540] or cause the developer community to isolate themselves with negative consequences for the suggestions and idea by an external innovator.

## Chapter 4

# Related Work

This chapter focuses on related work in the area of Open Source research and is presented before the results in Chapter 5. As the results—following GTM—originate from grounding in the data of the studied project and are thus not dependent on the related work, the reader has two options: (1) Read the related work now and judge the breath and depth of the main results of this thesis based on the accomplishments of existing research. This option is best to assess this thesis rigorously. (2) Skip the related work for now and read it after the main results in Section 5. With a firm grasp of the innovation introduction phenomena uncovered and conceptualized in the result section of this thesis, the reader can judge the related work better and pick the most valuable complementary ideas. This option is best to maximize the insights gained from this thesis.

First, the work of Stephan Dietze is presented, who built a conceptual model of Open Source development and then used it to suggest improvement areas (Section 4.1). Second, Martin Krafft's Ph.D. thesis on the diffusion of software engineering methods in the Debian distribution is discussed, the research question of which is most closely related to this work, but is asked in the context of a large-scale project and answered using a different methodology (Section 4.2). Third, a relationship is established to the work by Barcellini et al. on architectural design discussion in Open Source projects owing to their potential consequences for the collaboration in a project (Section 4.3). Fourth, Christian Stürmer's Masters thesis on building and sustaining a community is analyzed (Section 4.4). Finally, the chapter closes with four short summaries on related work of lesser importance to this thesis (Section 4.5).

### 4.1 Optimizing Open Source Projects

In his doctoral thesis at Universität Potsdam, Stephan Dietze derived a descriptive software process model of Open Source development and, based on this model, prescribed possible optimization to improve product and process quality [147, p.8]. Dietze used qualitative case studies to derive his results and studied the Linux kernel, the Apache httpd, the Apache Jakarta project, and the Mozilla project [147, pp.12f.]. In each case, Dietze looked at documents from the project such as FAQs or guides and existing secondary literature to enumerate in detail roles, artifacts, processes, and tools. This part of his work was published in a separate technical report [146]. His thesis uses Unified Modeling Language activity, class, and use case diagrams [371] to generalize processes, artifacts, and the association of roles and processes respectively [147, p.39]. The model is validated using the company-sponsored Open Source project *Pharma Open Source Community* (PhOSCo).

From his model and the studied cases, Dietze gained insights about Open Source development and several strengths and weaknesses.<sup>107</sup> He combines them with requirements put forward by commercial

---

<sup>107</sup>Insights: (1) Many activities occur redundantly, (2) projects often split into sub-projects, (3) processes for decision

stakeholders to theoretically derive possible optimization areas.

Optimization  
Areas

The optimization areas—the most relevant part of Dietze’s work for this thesis—are divided into processes, roles, artifacts, and infrastructure, of which the latter two mostly derive from the proposed processes and roles and are not further discussed here.<sup>108</sup> Eight process and role innovations are proposed:<sup>109</sup>

1. *Software Code Reviews* consist of (1) regular inspections on the existing code base and (2) mandatory pre-commit patch reviews [147, pp.92–95].
2. A *Request Reviewer* is a dedicated role for triaging, assessing, assigning, and handling bug, feature, and support requests. The goal is to keep the request trackers in good shape and improve user support. If necessary, sub-roles such as a *support reviewer* responsible only for support requests can be created.
3. *Extending the Build Process* is intended to (1) achieve publishing regular (for instance nightly) snapshot builds to increase user tests and peer review and to achieve the benefits of continuous integration [195] and (2) establish the role of release manager or build coordinator to improve the regularity of releases.
4. *Dedicated manual and automated testing* is suggested to be performed at regular intervals by participants in the roles of software tester and test case developer.
5. With the roles of *documentation manager* and 6. *communication reviewer*, Dietze proposes to put one person in charge of keeping project documentation such as FAQ, how-tos, manuals, and the project website current, and another one in charge of watching the project’s communication channels and ensuring that important information reaches all project participants and triggers appropriate processes.
7. As a complementary proposal to creating the role of a release manager, Dietze recommends taking the infrastructure responsibilities of the generic maintainer role and putting an *infrastructure maintainer* in charge. Delegating the administrative tasks is supposed to allow the maintainer to concentrate on organizing the community and steering development.
8. Last, Dietze suggests that *management processes* should be extended to include, for instance, explicit building of a community, defining a project strategy, establishing sub-projects, assigning core roles, establishing democratic procedures for conflict resolution, and incorporating the project in a foundation or similar.

These propositions leave four questions unanswered:

1. How to design such innovations so they can be successfully used? While Dietze uncovers the strengths and weaknesses using a well-defined process, the method by which the optimization areas are then designed is rather ad-hoc and makes you wonder how an engineering approach for designing innovations would look like.

---

making and conflict resolution are necessary, (4) developers focus on implementation and disregard (peripheral) processes such as design, quality assurance, support, and documentation, (5) releases are frequent, (6) source code must be modular and readable, and (7) Open Source tools are preferred for development.

Strengths: (1) The principle of natural selection guides software evolution, (2) processes rely on self-organization, (3) requirements specification is directly driven by users, (4) features are rolled out fast, (5) peer-review and user-testing are independent and widespread.

Weaknesses: (1) Peripheral processes are underrepresented, (2) redundancy implies inefficiency, (3) decision processes are slow and developer activity cannot be foreseen or planned, (4) releases contain many defects and no quality assurance guarantees can be made, (5) forking is possible, and (6) processes are not automated [147, pp.81–83].

<sup>108</sup>Most of the infrastructure innovations are nowadays included in the software offered by *project hosts* such as SourceForge.Net or project and build management solutions such as Trac (<http://trac.edgewall.org/>) or Maven (<http://maven.apache.org/>) [332].

<sup>109</sup>Dietze uses English terms for the roles, which were slightly adapted: (1) *environment manager* was changed to *infrastructure manager*, because the role only includes tasks affecting the project infrastructure and not the personal development environment, (2) *content manager* was changed to *documentation manager*, because the artifacts this manager is responsible for are documentation artifacts.

2. How to assess their applicability to a particular Open Source project?
3. What would motivate individual project participants to perform the associated tasks? For instance, the role of a request manager is likely to improve the speed and quality of user support request, yet unlikely to benefit the developers.
4. How to introduce such innovations in projects with unstructured and unenforced existing processes [147, p.91]?

These open questions can be seen as starting points and motivations of the work at hand in three areas: First, this thesis does include (an albeit small) treatment on the first question when discussing open access as a core principle behind *wikis* and *distributed version control* (see Section 8.2). Second, two of the case studies presented in Chapter 7 can directly be linked to Dietze's work and read as explications and evaluations of the proposals of (1) introducing a *documentation manager* and *communication reviewer* (see the case of information management at GNU Classpath in Section 7.1) and (2) using automated testing in Open Source development (see the case of automated testing at FreeCol in Section 7.4). Third, while this research was motivated by the fourth question of "how to introduce innovations?", the methodology has led it on a path that changed this question into a long journey toward the question of "how are they introduced?" The closest this thesis gets to answering the fourth question is given in the Section 8.1 on practical advice.

Dietze's work has the following limitations: First, validation of the model is done with a sponsored and not a community-controlled Open Source project [543, cf.] and covers only 10 of the 29 elements in the descriptive model directly [147, pp.76–79]. Second, unlike later work [318, 268, 171], Dietze did create a single generalized model of OSS development, which is now believed to be too inflexible to capture the "multiplicity and diversity" [316] of Open Source development [318, cf.]. Third, most generalized process elements are presented without referring back to actual data or existing scientific work, which often causes the elements to appear artificial in their formulation. For instance, the process modeling on change requests [147, pp.58–62] paints such an idealized picture: According to the model, the contributor only reproduces a reported bug to verify it. This ignores (1) the complex sensemaking [394] and negotiation [445] involving the bug-reporter, which has been found to ensue in many cases, and (2) the possibility for the contributor to skip most steps due to informal processes and lack of strict process enforcement [118]. Fourth, the identified insights, strengths, and weaknesses cannot comprehensively be supported by today's knowledge.

## 4.2 Innovation Adoption in Debian

In his dissertation Martin F. Krafft studies the influence factors regarding innovation adoption in the context of the Open Source distribution Debian, arguably the largest Open Source project with over 1,000 developers [293, p.10]. As an Open Source distribution, Debian performs two primary tasks: (1) to package software produced by other Open Source projects, so that they can be automatically installed with all dependencies via a package manager, and (2) to combine these packages into an out-of-the-box operating system. The Debian project uses the Linux kernel and the user space applications, libraries, and utilities of the GNU Project [475], therefore calling the resulting operating system Debian GNU/Linux.

The motivation for his thesis derives from the large number of "repetitive, minimally-integrated tasks" [293, p.xiii] involved in keeping each of the over 26,000 Debian packages up to date to the software produced "upstream" by the individual Open Source projects and compatible to each other. From this, Krafft formulated the goal of identifying the salient influences for innovation adoption in the Debian project.

As a method, Krafft used a Delphi study<sup>110</sup> with 21 Debian contributors in three rounds. Krafft notes Delphi Method

<sup>110</sup>The Delphi method [124] is an iterative approach for consolidating the "wisdom of the crowd" [497]—usually experts on the topic—and advances in rounds. In each round a moderator sends questions to a panel of experts and collects the results. For the next round, the moderator consolidates the answers and makes them available anonymously and/or adjusts

that this is the first use of the Delphi method in the Open Source research context to his knowledge despite what he identified as a “spectacular” fit to “the meritocratic spirit of Debian by focusing on contents of messages, rather than on who emitted them” [293, p.4]. This connection had been noted before by Raymond in *The Cathedral and the Bazaar*, where he describes the Delphi method as the underlying principle behind *Linus’s Law* (“given enough eyeballs, all bugs are shallow”) [415, pp.30f.].

To select the panel members, Krafft used a snowball sampling technique [465] and asked 162 Debian contributors to nominate peers for the panel. This resulted in 429 nominations, who were consequently asked whether they wanted to participate and then to rank themselves along five dimensions such as the size of their team or their interest in workflow improvement [293, pp.111f.]. Krafft then selected panel members to maximize variation along these dimensions. Panelists received gadgets worth EUR 250 as compensation [293, pp.97f.].

Over the three rounds, Krafft collected 24 influence factors for or against the adoption of innovations. He estimates that his panelists spent a cumulative total of 200 hours on writing responses to his questions and reading the summaries of previous rounds [293, p.71].

**Influence Factors** To present the resulting 24 influence factors, Krafft revisits the stage models of innovation adoption proposed by Rogers [436] and Kwon and Zmud [296] and combines them based on theoretical arguments into a cyclic stage model of individual and organizational adoption in the Debian project<sup>111</sup> [293, Sec. 7.1, pp.138ff.]. As this model is not empirically validated but merely used as a frame in which to present the 24 influences, it is not further discussed here.

At the beginning of the adoption process, Krafft distinguishes three influences which affect the way information about the innovation first reaches adopters: (1) marketing as information originating from the designer or creator of an innovation or from media in general, (2) “peercolation” as information which percolates through the network of peers, and (3) sedimentation as internalization of information about an innovation and understanding of the underlying problem by the individual.

From the remaining 21 identified factors, I would summarize the following 16 as innovation attributes [436, cf. Ch. 6]: (1) the ability to try out the innovation (*trialability*), (2) the innovation’s ability to be used for many tasks (*genericity*) and (3) at varying scales of use (*scalability*), (4) the *elegance* of the innovation and (5) how well it fits into existing innovation landscapes and modes of operation (*uniformity*), (6) *maturity*, (7) *transparency* as the attribute of how easy it is to understand and control how an innovation works and which implications it has, (8) *modularity*, but undecided on the question whether a modular or monolithic design is better for an innovation, (9) *compatibility* with other tools and processes, which is also called (10) *chunking*, when considering the innovation on a scale between evolutionary and revolutionary, (11) availability of *documentation* and (12) *examples*, (13) *sustainability* as the ability of the innovation to demonstrate a long-term perspective, and (14) the innovation’s *first impression*, (15) *cost-benefit*, and (16) *return on investment*.

The last 5 influences arise from an organizational perspective: Krafft identifies (1) resistance by project members, (2) network effects when usage expands beyond individuals, (3) the role of consensus building in the Debian project, and the embedding of innovation in (4) standards and norms and (5) tool-supported policy enforcement.<sup>112</sup>

To rank the identified influences with respect to importance, Krafft asked the panelists in the last round of the Delphi study to identify the three most salient influence factors, which resulted in tool-supported

---

the questions based on what was learned from the previous round. The method derives its strength from (1) anonymity, which takes many psychological and group dynamic effects out of the discussion and allows participants to reconsider their opinions freely, (2) controlled feedback by the moderator to keep the discussion focused on the topic, yet spread insights over the whole panel. More information on the Delphi method can be found in [383, 513, 91] and [293, chap.5]. An interesting application as a project risk assessment method in software engineering is given in [451].

<sup>111</sup>The stages in this model are: (1) knowledge, (2) individual persuasion, (3) individual decision, (4) individual implementation, (5) organizational adaptation, (6) organizational acceptance, (7) individual confirmation, (8) incorporation, and (9) organizational initiation, which feeds back into the knowledge stage to close the circle [293, Fig.7.1].

<sup>112</sup>Krafft uses the term quality assurance [293, Sec. 7.2.8.2, pp.228ff.].

policy enforcement being ranked first with 9.1 points<sup>113</sup>, followed by consensus (6.2), peercolation (6), network effects (5), genericity (3.6), and sedimentation (3.5). The remaining factors received fewer than three points each [293, Table 8.1, p.239].

Krafft's work has the following limitations:

- Focus is exclusively on the Debian project, one of the largest [293, App. A.4, pp.320ff.], oldest incorporated Open Source projects [387], with a distinct type of work (packaging) [339]. Furthermore, Debian is described as an ideological project with strong ethical principles [98], best exemplified by a quote from one of the panelists:

"Linus is purely pragmatic, and has no ethical ideas whatsoever. . . so he's fine with non-free stuff if it makes his life a little bit easier. We in Debian, however, have Principles with a capital, flowery 'P' (unreproducible in plain text)" [293, pp.21f].

Debian also is described as adhering to high quality standards [293, Sec. 2.2.4.1., pp.16f.] and as driven almost exclusively by volunteers in a meritocratic fashion [293, pp.14f.]<sup>114</sup> with a pragmatic if somewhat conservative attitude [293, pp.17f.].

Krafft makes no direct claim for generalization beyond Debian with its unique set of attributes, but readers still have to wonder to what degree results would transfer for instance to medium-sized projects or those with a more autocratic project leader or maintainer.

- The 24 influence factors are derived from the assessment of project members, which despite the use of the Delphi method might not reflect the reality of adoption decisions.
- The last round of the panel does provide an initial assessment of the importance of the factors, but their completeness, validity, relevance, and overlap still need to be investigated.

How is this thesis different from Krafft's work?

- The primary difference between both works is their different angle: This thesis focuses on concepts related to the *introduction* of innovations, while Krafft looks at influence factors on innovation *adoption*. For instance, hosting (see Section 5.5) was identified early on as important, but it was not yet clear what kind of relationship this concept would have to innovation introduction in general. It turned out that hosting can be an important argument in an innovation discussion, but also that hosting is often an essential requirement for innovation, a source of control over an innovation or even an innovation itself. By assuming this perspective, the thesis at hand remains more open for other conceptual relationships in the vicinity of innovation introduction.
- The numerous projects regarded in this thesis are much smaller, do have dedicated leadership, and represent different software domains such as embedded and CASE tools. Being small and more centrally controlled than Debian reduces the importance on diffusion processes, because communication is more immediate in a small project [1, cf.]. Both theses thus complement each other in the study of the Open Source universe.
- Last, there is a difference in methodology: This thesis uses Grounded Theory Methodology on developer e-mails; Krafft uses the Delphi method with 21 Debian contributors. Both methods have strengths and weaknesses: The Delphi method starts from a higher level of abstraction by letting participants conceptualize their experience. GTM—since performed on raw activity data—sees events as they occurred and it is the job of the researcher to conceptualize and increase abstraction. Consider for example the concept of *peercolation*, which appeared in the first round of the Delphi study as a verbatim quote from one of the panelists [293, p.140]. For such a concept to be derived in GTM would require a detailed and time-consuming study of

<sup>113</sup>Participants were allowed to give three groups of  $n$  factors, each of which then would receive  $1/n$  points.

<sup>114</sup>Krafft states that all participants of Debian are volunteers, some of which might be allowed to work on Debian during their regular jobs [293, p.14]. However, the absence of paid developers who in contrast to volunteers "profit directly from their effort" [435] would be surprising, as the large number of derivative distributions such as Ubuntu or Guadalinex—estimated at over 100 [293, p.1]—indicates commercial interest in the development of Debian.

communication processes by which peers exchange information in their social network. On the other hand, if a concept is derived by applying GTM, its foundation in reality is much more reliable. For instance, regarding the concept of *chunking*, as Krafft calls the preference for incremental vs. radical change, a similar tendency was observed also in the episodes in this thesis. Yet, GTM provides a richer picture, showing examples in which radical changes were performed, and gives us insights why these can be possible (see Section 5.9).

### 4.3 Innovation Introduction and Design Discussions

In this research, I have firmly excluded looking at the day-to-day business of Open Source development such as dealing with bugs, discussing about new features, or making a release, because these focus on the product. Instead, only those meta-level processes that have an impact on the software development process are considered such as the decision process of using a *decentralized source code management tool* over a centralized one or the process of adopting a *merge window process prior to a release*.

Target Project:  
Python

In this section, this division is put aside. Rather, it is considered to what degree matters of product and process can be combined by looking at the work of Flore Barcellini et al., who have explored design decisions as affecting the evolution of the software product [22, 24, 21, 23, 442, 26, 25]. Their project under study is Python, which produces an interpreted, object-oriented programming language [520]. They build on previous work by Mahendran on power structure [328] and Ducheneaut on socialization and joining [152, 153] in the Python project. The project is interesting for studying design decisions because it uses a structured way for changes to the programming language and standard libraries called Python Extension Proposals (PEP)<sup>115</sup> [26]. In essence, a PEP is a feature request for Python which specifies also a technical solution similar, similar to a Request for Comment [62] or a proposal under discussion in a Technical Review Meeting [127, 126].

The process for PEPs to become part of the Python language is described as follows [442, pp.232–34]: A *champion* writes a pre-PEP and submits it to the PEP editors, who can (1) accept a new PEP in draft status, assign it a PEP number, and enter it into the PEP database or (2) reject it, if for instance backwards compatibility is not addressed, the motivation for the proposal is missing, or the proposal is “not in keeping with the Python philosophy” [535]. The champion is then responsible for discussing the proposal with the community and ultimately convincing the project leadership to accept the proposal. If the champion succeeds, a reference implementation for the proposal can be created, which is reviewed by the project leaders. If the review was successful, the implementation is assigned the status final and included in Python. Proposals may also be rejected by the project leadership, or abandoned or withdrawn from their champion.

For their analysis Barcellini et al. select three PEPs, which they investigate in two lines of work: PEP 279 and 285 are discussed in [24, 442, 25, 23], while PEP 327, which was proposed by a user, is discussed in [22, 21, 26].

The methodology of their analysis is primarily quantitative on coded e-mails (discussion space) from the three PEPs, but also artifacts from documentation space (such as website pages or documents from the PEP archive) and implementation space (such as code from source code management) [24, p.147]. Second, they use hand-made temporal [22, Fig.2] and structural visualization [24, Fig.9f] of e-mail conversations, one of which is shown in Figure 4.1. Their main results are:

1. First, the authors investigate how participants can make sense of asynchronous and fragmented discussion and maintain coherence [250] in communication. While it had been known that fragmented, conversational turns are brought together again by threaded replies [523], Barcellini et al. find that quotations are better ways to maintain coherence [25, 23].

<sup>115</sup>A *process PEP* can be used to change a process surrounding Python [535] and a *meta-PEP*—as a special kind of *process PEP*—can be used for changing the PEP process itself [23, p.179].



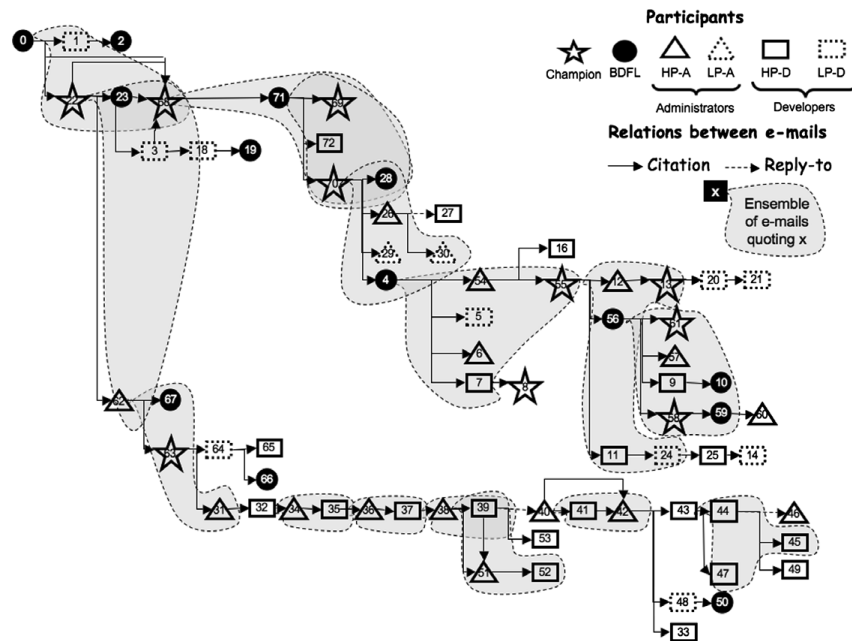


Figure 4.1: Conversation structure of PEP 279 showing both reply-to as well as quotation relationships between e-mails. Reprinted with permission [24, Fig.9].

Threading—they believe—is the coarser way of structuring conversation and may mislead on the question which topic is discussed in an e-mail. For instance, some replies continue to discuss an issue which is quoted by their parent message, but without referring to the point made in the parent itself. A second set of cases arises, if authors restart discussions by opening new threads and summarizing the most important points from a separate thread.

Barcellini et al. conclude that looking at threads is less useful in comparison to quotation-based analysis. My analysis yielded the same result that threading structure is inadequate for capturing all thematic links between messages, which are also often interrupted by technical problems such as misbehaving e-mail clients. Yet, using only quotation-based analysis ignores tactical actions by discussion participants and the ability of threads to act as focal points [450, 300]. Barcellini et al. particularly stress that a threading-based analysis of PEPs 279 and 285 splintered the discussion in fragmented threads and pushed “central” messages to “detached and peripheral” positions [23, p.181]. Yet, from my point of view these central messages would have been lost in the thicket of discussion, had their authors not deliberately restarted discussion by taking them out of their deeply and highly nested thematic context and put them into a new thread. Only there could they become pivotal to the discussion by summarizing opinions and listing design alternatives from a fresh starting point. This discussion is elaborated in Section 6.2 on the Garbage Can Model.

Method:  
Quotation-  
based  
Analysis

2. On the quantitative side, Barcellini et al. found that 91% of messages contain at least one quote in PEP discussions, 44% of messages were not quoted, 25% quoted once, and the remaining 32% quoted between two and six times. The champion of a PEP was the most commonly quoted participant.
3. E-mails written by the project leader or PEP champion more often open branches in the discussion;<sup>116</sup> core developers more often than others close discussion lines [23, p.184][24, pp.153,155].
4. The authors also found that despite multiple topics being discussed in concurrent discussion branches, the discussion was highly focused. They attribute this to key players such as the champion of a proposal who summarizes intermediate results and disallows off-topic discussion [24,

<sup>116</sup>All such frequency expectations were calculated using relative deviation (RD) analysis [45, 46].

p.161].

Concept:  
Synchronicity

5. The discussion of PEPs 279 and 285 showed a high degree of *synchronicity* with 50% of messages being first quoted within 60–136 min and 75% of messages within 5–7.5 h [23, p.181]. This points to a fast rhythm of e-mail communication [515]. Splitting each PEP into multiple topics of discussion,<sup>117</sup> these were discussed for 1 to 6 days (median 2) [24, p.158]. Six days was also the total duration of the longer of both PEP discussions.

Concept:  
Boundary  
Spanner

6. For PEP 327, Barcellini et al. found that several attempts were necessary for the proposal to succeed [21]. To investigate the reason for the failures, the authors separated the unsuccessful attempts from the one which was successful, analyzing in particular the discussion of each on both the user and the developer mailing list. They found that in the successful attempt (1) more participants are active (143 vs. 88), (2) who write more e-mails (5.2 vs. 3.6 on average). (3) The project leader is less active (19 vs. 31 e-mails written), but (4) follow-up between discussion is faster (30 days vs. 63 days) and (5) cross-participation between user- and developer mailing lists exists (5 vs. 0 participants are concurrently active in parallel discussion on the PEP both in the user- and developer-mailing list) [21, pp.61f.]. It is in particular the last point which they identify as the crucial difference: Cross-participants act as *boundary spanners* [467] to communicate the domain-problem experienced by the users to the developers who can act in the programming domain [21, p.62]. Figure 4.2 illustrates this boundary spanning for the successful proposals.

7. In [24], the authors then turned to the analysis of design activities contained in the e-mails of PEP 279 and 285 such as proposing, evaluating, and deciding on possible alternative solutions. They found that e-mails which contain more than one design alternative are more likely to cause respondents to open branches of discussion [24, p.158].<sup>118</sup> Since the project leader and the champion were also more likely to write e-mails containing such alternatives [24, p.158], this might provide a hint why they were responsible for more branches than other participants as mentioned above.

Second, both PEPs were markedly different with respect to design activities. In PEP 279, developers proposed 34 alternatives on five design topics, which caused developers to evaluate these in 57% of their comments, seek and give clarification in 13% of comments, and less frequently<sup>119</sup> decide on alternatives, summarize previous discussion and coordinate next steps. In PEP 285, on the other hand, the project leader as champion asked only to discuss three existing complex proposals. While evaluation was similarly common at 47%, this caused developers to seek and give clarifications in more cases (39%) and removed any need for summaries and decision making [24, p.158]. Barcellini et al. also discovered that each PEP discussion started on many topics, which quickly led developers to open specialized branches of discussion on each of them [24, p.160]. Developers also stayed focused on this topic except for a *meta-theme* which permeated the second PEP discussion and one *thematic drift* in the first PEP, which touched many orthogonal issues [24, p.160]. Last, the authors investigated the order of design activities and noticed common sequences; for instance, evaluations being followed by evaluations, which they hypothesized could indicate converging or diverging opinions [24, p.161].

8. In the second line of work on PEP 327, the authors then focused on coordination, social relationships, and knowledge sharing [22]. They observed that e-mails containing activities of coordination and social relationship are more prevalent on the developer mailing list (17% and 6.2%) than on the user mailing list (5.4% and 2.3%). They attributed this to the developer mailing list containing the discussion for coordinating and integrating the ideas and requirements of the users [22, p.566]. Looking at knowledge sharing, they discovered that the participants contributed knowledge about computer science and programming (28%), personal experiences (22%), examples (21%), links to knowledge provided by others (15%), and domain expertise

<sup>117</sup>Called thematic units in their work.

<sup>118</sup>Conversely, closing messages are more likely to contain one or no alternative.

<sup>119</sup>Exact figures are not given, but each type appears to occur in 5% to 7% of comments, with the remaining comments taken by proposals themselves and other comments such as humorous ones [24, Fig.17].

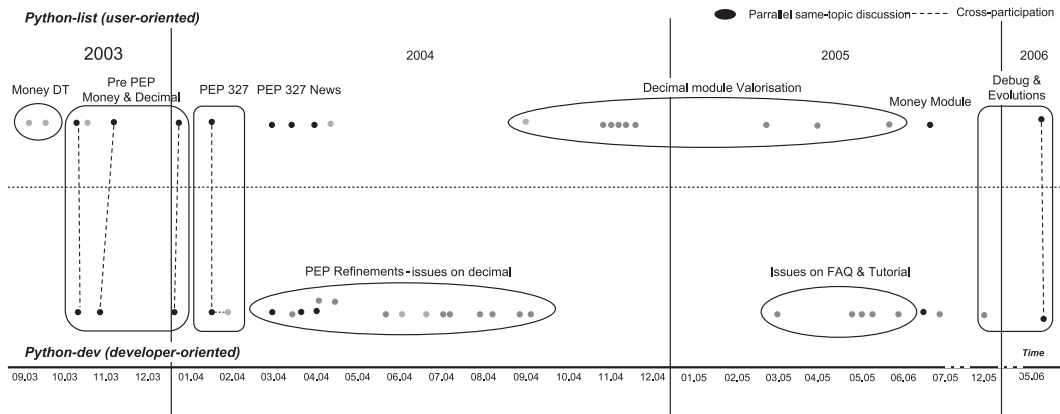


Figure 4.2: Boundary spanning between user and developer mailing list in PEP 327. The successful proposal shows significant interaction between both lists. Reprinted with permission [22, Fig.2].

(14%) [22, Table 5]. Yet, most users stayed confined to their list. It took the work of cross-participants as active and central participants in this PEP to span the boundaries of the two lists and carry important information on the application domain and programming between both lists [22, p.567].

Two limitations of the research exist:

1. The entire analysis is focused on the Python project which uses an established process for modification of their software. Not only is the discussion structured by this process, but the project leader also maintains tight control over decision and thematic coherence [24, p.161], which leads to the question if results would generalize to other Open Source projects lacking such a process or such a BDFL (“benevolent dictator for life”, as participants call the project leader Guido van Rossum [420]).
2. All the studies focus on three extension proposals out of the 133 which have been discussed by the Python project to the date of the studies [24, pp.148f.]. The analysis on design activities shows how different these proposals can be: For instance, one PEP discussion generated 34 alternatives and the other just three, leading to a different distributions in design activities. Also, all three PEPs are larger than the average PEP, focus on software features in contrast to software process (meta-PEPs), are successful despite one third of PEPs failing and a much larger number of proposals likely never to reach PEP status [24], and are thus not representative of all PEPs.

From a methodological perspective Barcellini et al. also discuss the large amount of effort involved in the detailed analysis of as little as three decision episodes.<sup>120</sup> They propose to automate part of the process [24, p.162]—a vision so far unrealized.

In comparison to this thesis, their analysis is more quantitative and with more detail on each episode, but also loses the breadth of analysis over 13 projects and 134 innovation episodes (see Section 5.1) and the depth of qualitative investigation.

Which of the results of Barcellini et al. are applicable to the discussion on innovation introductions?

1. Quoting and threading create thematic coherence, 2. key players maintain discussion continuity and open and close discussion branches, 3. e-mails with several proposals open branches—In the episodes I studied I found no indication that participants had problems with maintaining a

<sup>120</sup>Their analysis began in 2001 with a Master's project by Mahendran [328], and continued over the Ph.D. thesis by Ducheneaut [152] to the work of Barcellini [22, 24, 21, 23, 26, 442].

coherent understanding of the discussion flow.<sup>121</sup> However, thematic drifts into other topics (and into chaos) over the course of the discussion and *intentional* use of selective quoting and even ignoring of whole e-mails were observed (see discussion on the Garbage Can Model in Section 6.2). The latter can be considered an indication that participants are well aware that the communication medium e-mail has the ability to disrupt thematic flow, but use it consciously to steer discussion in their favor.

4. Synchronicity is high—Since most of the studied projects are smaller than the Python project, the number of participants and the intensity of the discussion was often greatly reduced. Yet, discussions quickly reaching a *thread-depth of over 20 in 36 hours* have also been observed, for instance in *gEDA* and *Bugzilla*. A discussion on timing in innovation introduction is presented in Section 5.8.
5. Cross-participants are important to span boundaries—This work has focused on developer mailing lists and in the cases when innovation introductions originated from users, the boundary spanning was thus already accomplished by the proposal being brought to the developer list. Since these cases are rare and the innovations proposed affect processes and only indirectly the product and therefore developers more than users, the role of the boundary spanner between developers and users is not as important as with feature discussions. The role of developers as boundary spanners between different Open Source projects on the one side<sup>122</sup> [467, 189] and between Open source projects and sponsoring organizations [247, 139] on the other hand is relevant because innovations and knowledge about innovations is often imported from other projects and external literature rather than being created inside the projects (see Section 7.4).
6. Different activities have varying importance for design discussions—It is difficult to assess whether the quantitative figures for different activities would be similar in innovation discussions, because the thesis at hand does not include a quantitative analysis of the number of occurrences of each type of activity.<sup>123</sup> Following GTM, a quantitative approach in an early phase of understanding the phenomena surrounding a research area is of little use, since concepts are still too ill-defined and too little understood to be used for counting occurrences. Nevertheless, all types of activities identified by Barcellini et al. have been coded at least once in the data of this thesis.<sup>124</sup> But since there were more interesting concepts to investigate first, said activities play subordinate roles in the work at hand.

There are two reasons to believe that innovation discussions are different from design and feature discussions and, more general, that innovation introduction is different from refactoring and implementation: (1) The impact on each developer when the project adopts an innovation such as a new source code management system is likely to be more disruptive than for many features given a certain degree of modularity. Certainly, a major change to the architecture could affect or even invalidate much of the knowledge developers have about a system, but given the effort involved with such a major refactoring, this is unlikely to be achieved quickly and without backing of substantial parts of the project (see Section 5.9). (2) Innovations—by their nature—are something new, at least for part of the project. Features to be designed and implemented can certainly hold equal novelty, but dealing with the open design space and the associated challenge of producing software is one of the central motivations for participating in Open Source [323]. Adopting an innovation—I would argue—is not.

Last, one paper by Sack et al. presenting the work by Barcellini et al. makes a fleeting point about “inverse Conway’s law” [442, p.246], as the suggestion is called that existing software architecture might

Inverse  
Conway’s Law

<sup>121</sup>Since e-mail is the primary communication medium in the Open Source world [554], anything else would be indeed surprising.

<sup>122</sup>Madey et al. call them *linchpin* developers [326].

<sup>123</sup>Such a quantitative analysis can not easily be derived from coding, since the qualitative investigation does not aim to code all events, but rather only the relevant ones.

<sup>124</sup>Coding of activity types is different: proposing alternatives (`activity.propose` and `activity.offer`), evaluate them (`capability` and `argumentation`), seek and give clarifications (`activity.ask for...` and `activity.discuss.explain`), explicitly decide (`innovation decision`), humor (`argumentation.humor`), explicitly refer (`activity.refer`), coordinate (several codes, for instance `activity.ask for help` or `activity.inform status`), build social relationships (several codes, for instance `offtopic.social` or `activity.thanks`).

affect organizational structure [562] (compare with the discussion in Section 2.3.8 which discussed several such sources from which the Open Source development process arises).

Most commonly, this idea is used to explain how a highly modular architecture could cause the loosely coupled process which is characteristic for Open Source development. Having well separated modules, which developers can work on highly independently and specialized by only considering the interface between their and surrounding modules, would—the argument goes—in combination with code ownership reduce the amount of coordination necessary [347, 395, 533]. An interesting twist is given to this in the project *Flyspray* in a *discussion about a new feature request* proposed by one of the maintainers. One of the core developers proposes that this feature request would be an ideal case to be realized independently by a third party via a plug-in architecture [flyspray:5666]. The other maintainer informs the core developers that he is already working on such a plug-in architecture, but the first maintainer notes three problems: (1) No third party for implementing the feature request exists currently, (2) the actual problem to be solved is still hard, and (3) that—once implemented—the burden of maintaining such a plug-in against an evolving architecture is likely to become the project's responsibility [flyspray:5670]. Thus, a modular architecture per se—according to the maintainer—would not reduce the amount of work, only potentially attract independently working developers, and might lead to legacy constraints [501, p.97] weighing the project down.

Precise boundaries should reduce coordination between modules, but inside each module we might expect those developers who work on this module together to communicate more intensely to share knowledge, visions, and responsibilities. As the development of the Linux kernel shows, precisely defined modules owned by individual maintainers can cause local sub-hierarchies inside of what Iannacci and Mitleton-Kelly describe as the Linux heterarchy [260].

If architecture has this ability to affect process, this helps to relate design and innovation: A conscious change to the architecture to affect process is an innovation itself. In the project *Flyspray*, for example, the proposal to introduce a plug-in architecture was an innovation because it could affect process by attracting independent third parties.

One study by Thiel has investigated the case of affecting change the other way: from (process) innovation to design [501]. Faced with existing architectures prone to security vulnerabilities, Thiel proposed an annotation scheme for highlighting architectural weaknesses and opportunities for refactoring. The goal was to provide an incremental path for architecture change and make each individual change a task, which could be executed independently and without being chief architect. The outcome of applying this idea to two Open source web applications is described in Section 7.5.

To summarize, the research on design discussion provides some insights which can transfer to the discussion on innovation introduction. For example, the importance of quotation-based analysis to complement threading-based analysis, the concept of boundary spanning or the property of episodes to occur with high synchronicity are all likely to equally apply to the innovation introduction context. Apart from that, the relationship between architecture and process via *inverse Conway's law* and process innovations for incremental refactoring has been discussed.

## 4.4 Community Building

Stürmer interviewed seven core developers and one community member from distinct Open Source web application projects on the question of how to successfully start and build an Open Source community [494]. This led to a list of seven attributes desirable in new community members or the community as a whole: (1) productivity, (2) self-motivation, (3) norm adherence, (4) altruism, (5) perseverance, (6) diversity, and (7) common vision. To attract and keep such participants, i.e. build a community, Stürmer identifies 14 innovations, innovation areas, and product attributes from his interviews, which he discusses for impact on (1) recruiting new developers, (2) enhancing collaboration

in the project, and (3) improving the software product. The third perspective is added because of the impact product success can have on community success, as is discussed in Section 2.3.10.

Some of these innovations and innovation areas are basic such as the use of collaboration infrastructure, release management [168], communication channels [554], or keeping a credits file, task list, and documenting the project [190, Chap.2]. Others are more strategic such as explicit marketing, *incorporation as a foundation* [387], or keeping the architecture modular (see Section 4.3). Then *physical meetings* [115, 43] and internationalization are discussed as chances for intensive and global collaboration. Last, the interviewees stressed code quality, user interface, and the installation procedure as important aspects of the product for project success.

The difference between Stürmer's study to the work at hand is that he lists the innovations and product attributes and explains some of the mechanisms by which these can have a positive effect, but without discussing how these innovations might be introduced into a project or the product attributes be achieved and then sustained over time. For instance, a *task list* is given as an innovation for enhancing collaboration and for motivating new participants, yet the issue of keeping the task list up to date is not discussed (compare the case study on the innovation manager in Section 7.1). As a product attribute, for example, the interviewees report the importance of an attractive user interface, yet give no insights how the project can achieve such (consider [363, 14, 390, 42] for discussion on the associated problems such as lack of expertise on UI topics). Only in the last section of his thesis did Stürmer approach the issue, but conceded that it was “neither expected nor explicitly investigated” [494, p.109]. He hypothesizes that organizational change in Open Source projects must be created bottom-up by community demand driving leadership action. He gives several examples of top-down management by project leaders who tried to anticipate problems, but failed to get their solutions accepted by the community [494, pp.109f.].

## 4.5 Other Related Work

The studies by Hahsler [228], Bach et al. [14], and Shah and Cornford [462] are presented in short to conclude the discussion of relevant related work.<sup>125</sup>

**Adoption of Design Patterns** Hahsler studied the adoption of design patterns [202, cf.] in 519 projects sampled from SourceForge.net [228]. He discovered that 12% of the 761 developers who added or modified more than 1,000 lines of code in these projects used the name of a design pattern at least once in their commits, which corresponds to 16% of projects using them. This points to only a very low penetration of state-of-the-practice software engineering inventions in Open Source projects. When looking to explain which projects use design patterns and which do not, Hahsler could detect no correlation to the development stage or project characteristics except for very large projects which are more likely to use design patterns. For developers he found the probability for design pattern usage to increase with activity.

Hahsler also observed that if design patterns were used, they were most often used only by a single developer in a project. This implies that knowledge about and affinity to using design patterns is bound to individuals rather than projects. A similar conclusion is reached in the case study on introducing automated regression testing in the project FreeCol (see Section 7.4).

**Improving Usability Processes** Bach et al. investigated ways to improve usability processes in Open Source projects [14]. To do so, the authors first identified barriers to improved usability and solutions based on the literature, notably from [363]. They combine these with a model of Open Source collaboration based on “control, trust and merit” [14, p.987] to derive four suggestions to improve the infrastructure of the Microsoft CodePlex—a hosting provider for collaborative development: First, to strengthen trust between usability experts and developers, Bach et al. suggest to raise usability activities to the same level in the CodePlex portal as bugs and code commits, which involves adding a usability role and a workspace for designers (akin

<sup>125</sup>The work by de Alwis and Sillito [135] on migrating to a decentralized source code management system is also relevant, but discussed in detail in Section 8.1.1.

to a bug tracker for designs). Second, to strengthen opportunities for merit, the authors recommend adding visibility to the activities for instance by a gallery of designs, measuring design activity to rank projects, letting users vote for designs, and offering a dedicated design forum. Third, Bach et al. show how the design workspace and its workflows could be integrated as an intermediate step between user suggestions and the bug tracker [390, cf.]. The goal here is to make design part of the development process. Fourth and last, the authors hypothesize that by offering state-of-the-art practices and tools in the design workspace, this alone could raise awareness and make usability a more important activity. Unfortunately, their work is so far restricted to suggesting these improvements only theoretically and building mock-ups [14, p.993].

Shah and Cornford have documented the early discussion in the Linux kernel project about which source code management system to adopt. This debate was between CVS (strongly detested by Linus Torvalds for its technical inadequacies) and BitKeeper (dubbed “evil” because it is not Open Source [462]). The authors describe how Linus first refused to use CVS against the generally perceived need for version control. This led to a CVS repository to be set up in parallel to the “official” tree as maintained by Torvalds. Since there was technically no difference between this CVS repository controlled by Dave Miller and the “official” tree except by social convention, further conflict in the kernel project could have led to a leadership change. Linus finally agreed to the use of a version control tool, but adopted BitKeeper, which was not Open Source but preferred by Torvalds on technical arguments. Many community members strongly criticized this use of a non-free tool [462]. More abstractly, Linus combined something desirable—improved collaboration and finally a tool for version management—with something detested by the community—non-GPLed software. The community could have balked at the latter, but eventually went for the former. Shah and Cornford argue that the use of BitKeeper—once established—caused a similar dependence on using BitKeeper, as a dependence existed on Linus Torvalds: To participate, one now had as strong an incentive to use BitKeeper as to accepting Torvalds as the project leader (this general idea is discussed further in Section 5.7 on forcing effect). From this case we can learn two things: (1) Who is wielding control in a project is not a clear-cut issue, as control derives ultimately from the socially assigned power. (2) Technical arguments can trump ideology, but only under considerable resistance.

Adopting  
Source Code  
Management  
Systems

Li et al. studied decision episodes in Open Source projects, but unfortunately excluded all decision episodes which do not directly affect software modifications [313, 242]. How far their results, which were already presented in Section 2.3.9 on decision making, are thus applicable to innovation introduction is an open question. Still, two key points are worth reiterating: (1) Open Source projects appear to have a “bias for action” [554] rather than for coordination and discussion, with 58% of decision episodes being resolved without evaluating alternative options. (2) The majority of the remaining episodes<sup>126</sup> were found to be highly complex to follow. Discussion is described, for instance, as looping back from decision to new proposals, multiple problems being discussed at once, no clear separation being maintained between individual issues, or suggestions bringing unrelated new decision opportunities into the front [313]. Li et al. liken these to the Garbage Can Model of organizational decision making, which is discussed in Section 6.2. Whether these results transfer to innovation introduction episodes cannot directly be answered in this thesis, because I collected no quantitative data on this question.<sup>127</sup> One could hypothesize though that (1) complex loopback episodes are more likely to be relevant for the future of a project and thus for this research, as they involve more participants and more agitated discussion, and (2) innovation introduction episodes are more likely to be complex, as the implications for each developer should exceed those of source code changes. These hypotheses can be supported by results from Bird et al. who studied the social networks of Open Source projects and found that on product-related topics discussion is more focused on sub-groups, while process topics are discussed more broadly [52, p.30]. In this thesis, understanding such broad and complex episodes will be an important challenge.

Decision  
Making

<sup>126</sup>29% of all episodes.

<sup>127</sup>One area where results are available is the failure rate of episodes, which is much higher than the 9% reported by Li et al. (see Section 5.1).





## Chapter 5

# Results

At this point we should take stock of what we know already before plunging into the main results of this thesis. First, an innovation is a tool, method, or practice which can become part of a software process (see Section 2.1). Second, Open Source Software development is a development model arising from a particular licensing scheme, in which participants (mostly volunteers) who are distributed widely and loosely coupled to each other collaborate on producing software in communities called Open Source projects (see Section 2.3). Third, the methodology which was used to derive the following results was Grounded Theory Methodology—a qualitative method for theory construction. Fourth, the data corpus consists of e-mails from 13 medium-sized, diverse Open Source projects which were analyzed for innovation-introduction-specific content (see Chapter 3). Fifth, we know about the related work which, with the work of Krafft, has begun to shed some light onto the question of how to understand innovation introduction behavior, albeit in a large software distribution project such as Debian (see Chapter 4).

This chapter is structured as follows: First, a small quantitative overview is given of the episodes on which the results are based (Section 5.1), followed by a lifecycle model of innovation introduction (Section 5.2), which is used to structure individual innovation episodes. After this, each section presents one concept or a set of related concepts uncovered and developed using GTM (an overview is given in Figure 5.1). Each such set of concepts is explored based on the episodes discovered in data and often follows the analytical development and the narrative in the episodes before being abstracted and related to innovation introduction. A diligent reader might want to take the opportunity to first read the descriptive summaries of the episodes offered in Appendix A.1, which should make the following discussion easier to follow. As discussed in Section 3 on methodology, the concepts have been kept independent of each other so that the reader can investigate the following results in any order.

### 5.1 Quantitative Overview

Quantitative results are not an objective of GTM at all and, due to the selection bias introduced by looking preferably at data which widens the understanding of the topic of interest, the internal validity of the following quantitative statements on the number of episode hidden in the sample of this study must not be overestimated. They are given here nevertheless to give the reader a sense of the amount of innovation-related activity which occurred in 2007 in the observed projects.

In total, there were 134 innovation episodes uncovered in the thirteen projects regarded. Of these, 76 were found to involve discussion and innovation introduction attempts, seven were episodes in which an innovation was announced without previous discussion, 27 were about *using and adopting* and twelve about *sustaining* previously established innovations. Three episodes revolved around phasing out an innovation and four episodes never made it to a proposal, but rather only involved problems being

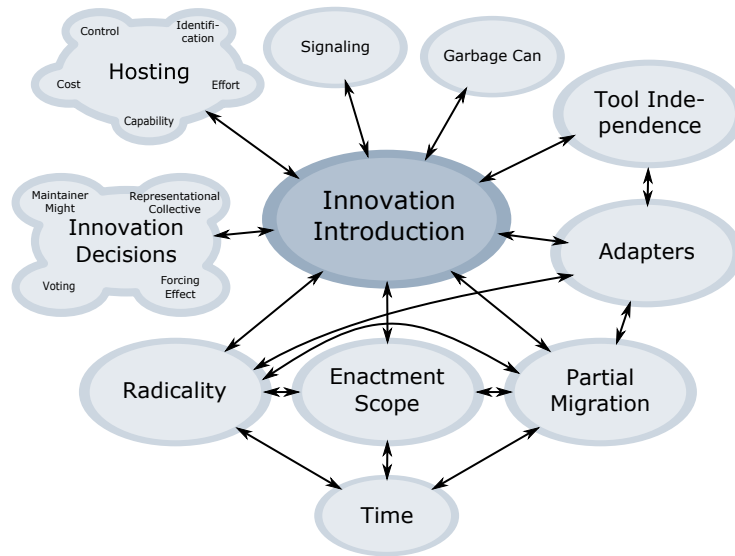


Figure 5.1: The primary results of this thesis concerning the central topic of innovation introduction. Arrows indicate conceptual relationships. Minor results are shown in a smaller font.

discussed. Five episodes could not be categorized in this scheme, for instance because they were about experimenting with an innovation without an attempt on introduction begin made.

Of the 83 episodes in which innovations were introduced or announced, there were 37 *failures* and 30 *successes*. The remaining 16 episodes had to be assigned to an *unknown* outcome, because for instance adoption could not be determined from the mailing list data. A breakdown by project is given in Table 5.1.

## 5.2 The Introduction Lifecycle

The introduction lifecycle is a theoretical model of the stages an innovation assumes during the process of being introduced. Discussion of this lifecycle could have occurred already in the introduction, but then it would have appeared as static and externally defined rather than as an early, valuable result of using GTM. On the other hand though, the lifecycle model is primarily useful as terminology for the structure of innovation episodes and thus did not receive the same in-depth analysis as the other result sections.

To characterize the lifecycle of an innovation introduction, we first turn to the different ways that *episodes* have been observed to conclude. Recall that episodes encompass all events related to one innovation introduction and that a successful episode outcome is likely the main goal for an innovator. Unfortunately, *innovation introduction success* is hard to define, as it is based on the general concept of success in Open Source projects which in turn is based on a specific interpretation by the innovator. The discussion in Section 2.3.10 has already revealed the general concept as polymorphic, and the innovator's interpretation is certainly grounded in varied individual and possibly hidden motives, which are even harder to assess.

In this study we define *success* in the following way:

**Definition 2 (Success)** *An innovation is successfully introduced, when (1) it is used on a routine basis and it has solved the problem it was designed to solve, or (2) it attained the goal it was designed to attain.*

Project	Success	Failure	Unknown	Total
<i>ArgoUML</i>	3	7	3	13
<i>Bochs</i>	1	1	0	2
<i>Bugzilla</i>	4	3	7	14
<i>Flyspray</i>	2	2	0	4
<i>FreeDOS</i>	1	0	0	1
<i>gEDA</i>	9	4	2	15
<i>GRUB</i>	1	7	1	9
<i>KVM</i>	3	3	0	6
<i>MonetDB</i>	0	2	0	2
<i>ROX</i>	2	0	0	2
<i>Request Tracker</i>	0	0	0	0
<i>U-Boot</i>	3	2	2	7
<i>Xfce</i>	1	6	1	8
Total	30	37	16	83

Table 5.1: Number of episodes per project and outcome.

The first part of this definition regarding routine usage is a direct adaptation of Denning and Dunham's key measure to innovation success—"adoption of a new practice by a group" [140]. The second part was added, because some innovations were found which did not aim for adoption, but rather tried to increase the likelihood of a certain event. One example for these non-adoption innovations can be given from the project *Bugzilla*, where one of the maintainers aimed for increasing the chances for users to become project members by optimizing the processes by which users would join the mailing-list.

The definition keeps the origin of the goal and problem to attain or solve intentionally open, so that success remains contextual. In the general case the innovator provides them and the project adapts them slightly during discussion.

Given this definition, three primary categories for the outcome of an episode arise out of data, which already have been given in the previous section: *success*, *failure*, and *unknown*. The *unknown* category is necessary because we cannot always determine whether *adoption occurred* or whether the innovation *achieved its goal*.

As an example for the difficulties to determine the success of an episode consider the following episode in the project *ArgoUML*, which is developing a UML CASE tool in Java and will serve as a source of examples for most concepts in this section. The maintainer had proposed to join the *Software Freedom Conservancy (SFC)* to avoid some of the hassle of handling money as an Open Source project. Shortly thereafter, the legal paperwork was filed and *ArgoUML* became part of the SFC. When *ArgoUML* participated in the *Google Summer of Code* and a money transfer from Google to the project had to be handled, the project used the SFC to take care of the transaction.

Foundation at  
*ArgoUML*

Using the conservancy to handle money is definitely a use of this innovation and a first step towards adoption. Unfortunately, it is hard to decide at which point we can say this has been incorporated sufficiently to be a practice or routine, because handling money in *ArgoUML* remains such a rare event that our sample period only includes this single case.

Measuring the success of the introduction by its goal—less hassle—is similarly difficult, since (1) the usage interactions with the innovation are separated from the mailing list, and (2) measures for such a subjective goal are hard to come by. Since the maintainer never reports on reduced work, one is left to wonder whether the introduction may be called a success.

To determine innovation success, I have tried similarly to be conservative in my judgment. Evidence was always gathered on goal attainment and the use of the innovation becoming routine and established behavior. Despite all efforts, it cannot be guaranteed that an innovation is not abandoned shortly after

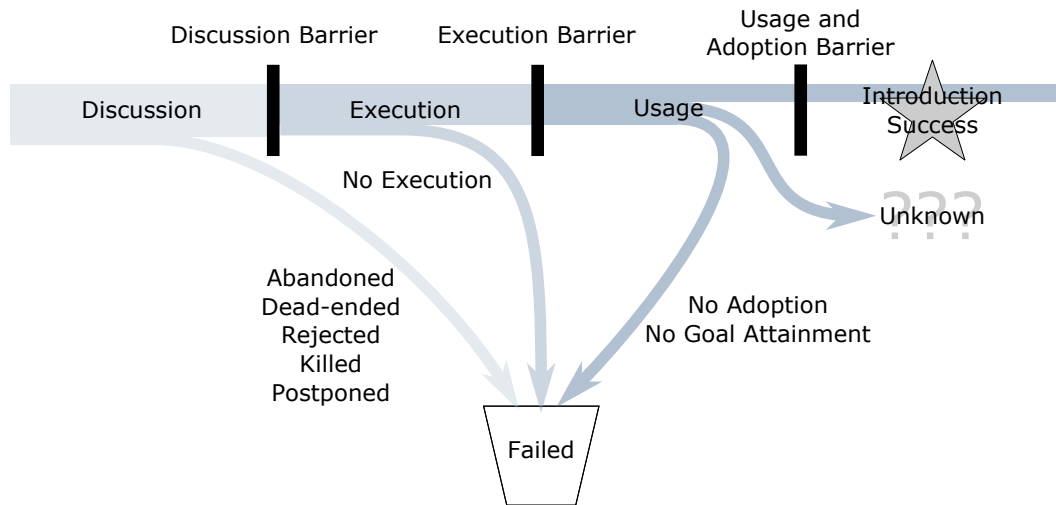


Figure 5.2: The reasons identified by which introduction episodes fail based on a simple phase model of innovation introduction. Failure reasons can be interpreted as barriers which the innovator needs to overcome to proceed into the next phase of an introduction.

the sample period ends or side effects appeared that prevented it from achieving its intended goal.

Considering these difficulties regarding the notion of success, we can come back to the question on how to structure the possible lifecycle of an innovation.

In most episodes, the following simple phase model of innovation introduction could be observed, which distinguishes between

1. *discussion* leading to an organizational innovation decision in favor of or against an innovation (see Section 5.7.1),
2. *execution* leading to a usable innovation, and
3. *usage* and institutionalization of the innovation.

While a successful introduction is achieved once usage becomes widespread and goal-attaining, each phase provides opportunity for failure, which will be illustrated with an example for each phase:

**Failure in discussion phase**—An example of a proposal that fails in the initial discussion phase can be given by an episode in which the maintainer of ArgoUML proposes the use of *branches for cooperative bug fixing* in contrast to the existing use of patches attached to bug tracker items. This proposal is tightly scoped (see Section 5.4) and directed specifically at the two highest ranking core developers (beside the maintainer himself). In the ensuing discussion these two core developers *reject* the given proposal by (1) a series of arguments such as inefficient operations in their development environment [argouml:4773] or the lack of an established tradition of using branches for experimental work [argouml:4784], and (2) by enlarging the *scope* of the proposal to include the whole project. The innovator tries to counter these arguments, yet the core developers do not return to the discussion, a phenomenon called a *dead end* where the introduction episode ends.

Branch for  
patches at  
ArgoUML

**Failure in execution phase**—Examples of failures to execute are rarest by phase, and the example given is not perfect: In one episode a *translator* proposes to join forces with the translation teams of distributions such as Debian or Ubuntu to get ArgoUML translated more rapidly [argouml:4691]. The idea is well received by two project members [argouml:4694,4696], up to the point that one of them declares that he will contact some other distributions about the idea. Yet, neither he nor the innovator ever report back on their attempts to get in contact with the translation projects of any distribution (or presumably never contacted them). This is a pointer to a first class of reasons why innovation

Translations at  
ArgoUML

introductions fail: Lack of commitment by the innovator. If we dig a little deeper, we learn another reason why this introduction failed: The second proponent—a core developer—explores the state of ArgoUML packages with the major distributions and finds that they are in such an outdated state that translating them would not make any sense. He thus *abandons* the execution of the discussed *episode regarding translating ArgoUML by using distributions* in favor of starting a new *one regarding packaging*.

**Failure in usage phase**—Third and as an example for an episode which failed in the adoption and usage phase, a short episode can be recounted in which the maintainer defines the new role of *an observer on the mailing list*. The project member to take on this role was proposed to be responsible for welcoming new project members and managing duplicated or incorrect information in the bug tracker. The innovator decides to create the role by his power as a maintainer, thus bypassing discussion, and defines (thus executes) it within the proposing e-mail. Yet, he *fails to get the innovation adopted* by the project, because nobody volunteers to take up the role.

Observer at  
ArgoUML

These three examples should provide a sense of the wealth of ways an innovation introduction may fail and that it is useful for the innovator to be aware of mechanisms that might prevent him from being successful.

### 5.2.1 Failure Reasons

By analyzing all episodes for the reason why they failed, three main categories could be identified:

**Failure by rejection**—*Rejection* by project members is the basic reason for episodes to fail: During discussion, the proposed ideas meet with resistance and the innovator cannot convince the other participants to decide in favor of his innovation. I found that such rejections contain two special cases: (1) If a proposal is rejected by a high-ranking project member, then the end of the discussion is often so strong and abrupt that I use the special term of a *killed* proposal. These drastically stopped introduction attempts might provide central insights for how to overcome the *strong influence of the maintainer*. (2) Some proposals are not outrightly rejected, but rather *postponed* for revisiting the proposal later. The innovator should carefully consider in such a situation whether time is working in or against his favor. In the project KVM, for instance, a maintainer waited four weeks when facing considerable resistance with an introduction. Then participants had become more comfortable with the accompanying technology and let him execute it.

**Failure by abandonment**—While this first group of failures highlights the ways that other project members can make an introduction fail, a second big group of failures highlights the importance of the innovator for each introduction episode. We call an episode *abandoned*, if the innovator fails to continue with the episode, even though no obstacle to innovation success is present except the innovator's own ability to invest time, execute, or explain the innovation. In fact, such episodes highlight that ideas are easier proposed than followed through. An archetypical example can be seen in the project *Bugzilla*, where the maintainer proposes an *optional design review process* to be added to the development process. Such a review was meant to be an optional process step for any developer who is unsure whether a contribution they want to work at would be accepted based on its design. The maintainer's goal with this proposal was to reduce extraneous effort spent on contributions which are rejected after implementation because of a "fundamental design problem" [bugzilla:6943]. The maintainer receives two clarifying (if somewhat critical) questions just six minutes after he sent his proposal to the mailing list, to which he fails to reply. Given that he as the maintainer has demonstrated time and again that he is able to convince others of his proposals, we are left to conclude that he has *abandoned* this proposal for reasons unknown to us. For an innovator we can conclude from this that his planning of available time and resolve to stick to the episode is a central prerequisite to introducing an innovation.

**Failure by dead end**—A third type of reasons for failure during the discussion phase originates again in the behavior of the innovator. In several cases the innovator failed to attract the interest of the project with his proposals or arguments. We call such a situation *dead ends*, in which the last e-mail

in an episode is written by the innovator. A *dead end* episode is always also an *abandoned* episode, because the innovator could have picked up on his unreplied-to message. However, a dead end provides different insights about the difficulties of capturing the interest of the other project members and the tactical dimensions of an innovation introduction, such as when to write a proposal (not shortly before a release<sup>128</sup>), how many proposals a project can digest at once, how to scope a proposal (see Section 5.4), whom to address [437], how much effort to invest before proposing [413], etc.

Out of a total of 83 episodes all 37 failing ones fit into one or more of these categories. The categories are not exclusive, because each associated failure reason implies a significant contribution to the outcome of that episode, of which there can be several. For example, consider the following episode at ArgoUML that was *triggered* during the *discussion of joining forces with the translation teams at distributions such as Debian*. One of the core developers explored the versions of ArgoUML included in distributions and proposed to discourage shipping unstable versions which might harm the reputation of the project [argouml:4697]. He received two replies, the first of which proposed to change the versioning scheme from *odd/even for unstable/stable releases* to a *milestone-based naming scheme* to make unstable packages more explicit [argouml:4698]. The second e-mail rejected this proposition to change the naming scheme (as in line with Open Source community norms) and also discounted the problem in itself [argouml:4701]. Given that the innovator does not return to the discussion, we can identify two significant contributions to the failure of this episode: The first being *rejection* by the project peers and the second being *abandonment* by the innovator.

A concluding overview of the reasons for failure of innovation introductions is shown in Figure 5.2.

### 5.3 Partial Migrations

Having gathered sufficient background knowledge about the goals and definitions of this research, a solid understanding of GTM and Open Source development, and the basic terms and phases of introducing innovation, we can now turn to the main results. I want to begin with the concept of “partial migrations” because the underlying phenomenon was the first to stop me from Open Coding and turn to Axial Coding instead to discover why it had occurred.

The discovery of the phenomenon of partial migrations during Open Coding is closely bound to a single episode occurring in the project *KVM* at the beginning of 2007. This project conducted the introduction of a novel source code management system while at the same time retiring its existing one.

**Definition 3 (Migration)** *A subtype of the introduction of an innovation, where an existing innovation is replaced by a newly introduced one.*

Yet, instead of migrating its existing set-up entirely from the old to the new innovation, the project chose to conduct the migration only partially, moving certain parts of the code base to the novel system while retaining others in the existing one. This approach of concurrently using two systems in parallel led to a series of problems such as duplicated effort in maintaining branches and increased complexity of determining which revision contains a certain change [kvm:1839]. Similar situations of partially introduced innovations occurred in other projects, which prompted the following puzzling questions:

- What causes partial migrations to occur, i.e. in which situations, because of which actions by project participants, and why does a partially migrated state arise?
- What are the consequences of partially migrated innovations?
- How and to which effect do projects and their members deal with these consequences? In particular, if the consequences are negative, why is the migration not completed?

<sup>128</sup>This is well exemplified by two nearly identical propositions shortly before and shortly after the release of version 4.4 in the project *Xfce*: The proposition shortly before the release fails to even draw a response [xfce:12700], while the proposition after the release by a peripheral developer is a success [xfce:12949].

These questions map directly to the development of a paradigm centered around the phenomenon of partial migrations as discussed in Section 3.2 on Grounded Theory Methodology. We thus set out to understand the phenomenon of partial migrations by understanding (1) the actions causing the phenomenon, (2) the starting context that they occurred in, and (3) any augmenting conditions influencing the actions taken so that the phenomenon occurred.

### 5.3.1 Partial Migration at KVM

The first episode in which we encountered a partial migration occurred in the project “*Kernel-based virtual machine*” (KVM) as mentioned above. KVM is a commercially-backed Open Source project of the start-up Qumranet, on which this company bases its main offering of a desktop virtualization product. KVM is a virtualization solution on top of the Linux kernel for x86 hardware, so that virtual machine images can be run on a single computer running Linux. The commercially dominated background is important to understand the history of KVM’s use of source code management tools. The demand for making KVM’s internally kept code repository publicly available first appeared at the end of 2006, when the project became more and more popular. After discussing internally, access using the centralized source code management tool *Subversion* was provided. This set the starting context in which the partial migration occurred.

In February 2007, the project leader announced that the parts of the project code interfacing with the Linux kernel had been migrated to the decentralized source code management tool *Git*. The user space parts, however, would remain in the existing system *Subversion*. The primary reason given was the inadequacy of *Subversion* to deal with the following changes in the project’s requirements: First, *Subversion* was said to be unable to scale to handle a full Linux kernel tree, which had become necessary when KVM code extended into places in the kernel beyond a single sub-directory. Second, *Subversion* required an account on the Qumranet server for each developer who needed write access to manage long-term branches. These reasons were perfectly fine reasons to migrate, yet they do not explain why the project migrated only partially and retained a large portion in *Subversion*. The only reason we can find is from a discussion related to the initial offering of *Git* to the project:

*Git at KVM*

“Git’s learning curve is too steep for me. I may dip into it later on, but I’ve got too much on my plate right now.” [kvm:187]

So, abstracting from this quote, we see that learnability and associated effort to learn a new technology in combination with an already high workload might be a primary impeding factor to a full migration. We can even abstract these augmenting conditions to two antagonistic ones: While certain reasons call for a migration, others make it difficult to do so, and the project maintainer escaped their full impact by migrating only partially.

Effort  
Management  
Strategy  
  
Deal with  
Antagonistic  
Forces

Because KVM is driven by a company, we did not observe any e-mails as part of the migration itself that could have provided further insight. What we saw on the other hand is that, one month later, the maintainer complained about the negative implications of the partial migration:

“Managing userspace in subversion and the kernel in Git is proving to be quite a pain. Branches have to be maintained in parallel, tagging is awkward, and bisection is fairly impossible.” [kvm:1839]

So, given the phenomenon of an existing partial migration, we become aware that within a month the context has changed: (1) The situation that was created just a month ago is now undesirable to the maintainer. (2) The maintainer has learned how to use *Git* sufficiently well to be comfortable with it [kvm:1853]. These changes in the context create room for new interaction, which the maintainer thus uses to propose to migrate the remaining part of the code by putting it into the *usr*-directory of the Linux kernel *Git* repository. At the same time, though, the maintainer directly adds a first intervening condition: The proposal feels “slightly weird” [kvm:1839], which sounds like a petty reason at first sight, but should probably rather be interpreted as the maintainer’s hunch that the proposal would cause pain and awkward situations in other regards. The second intervening condition that acts upon this proposal

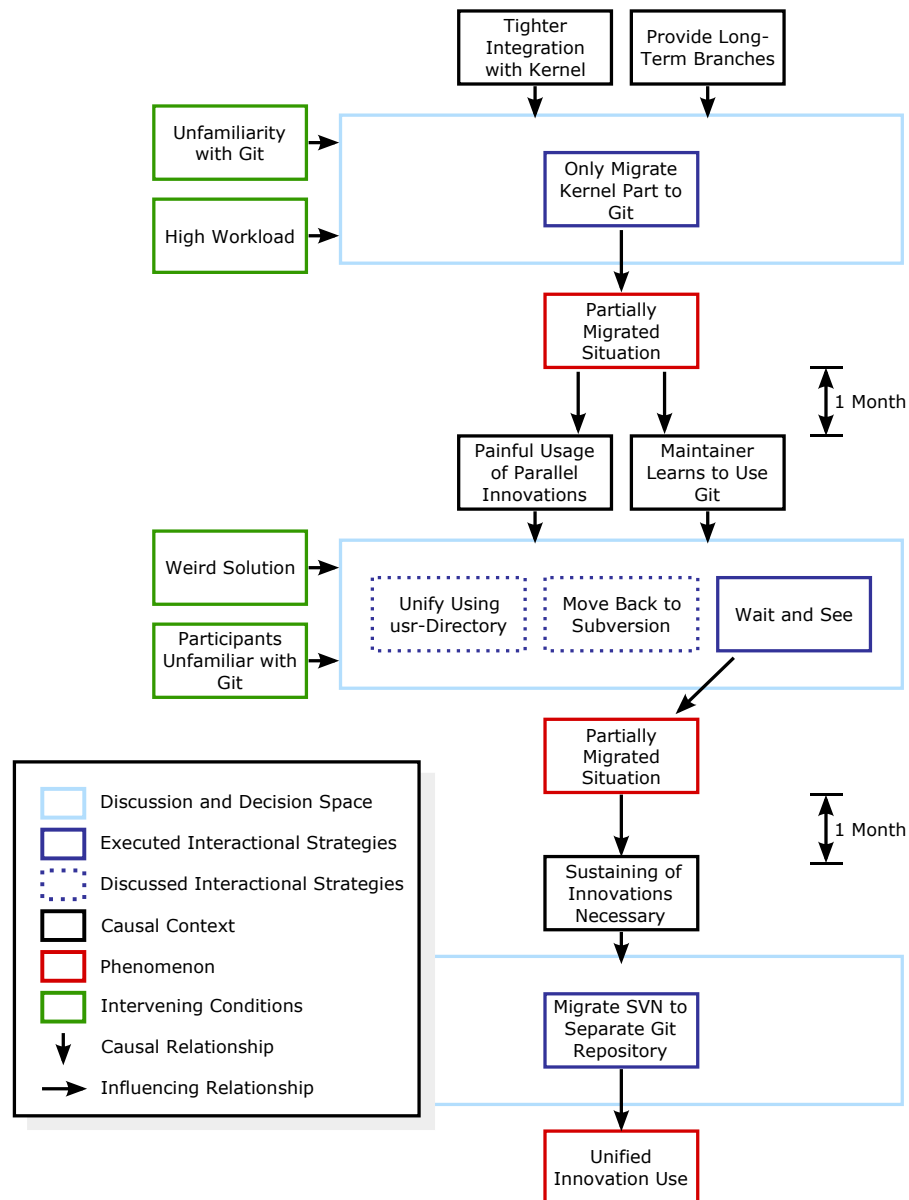


Figure 5.3: The *partial migration* in the project KVM presented using the Paradigm. The process by which the phenomenon of a *partially migrated* source code management situation arose and was resolved is depicted by its three discussions and decision frames.



is the knowledge discrepancy the other project members have regarding using *Git* in comparison to Subversion [kvm:1853]. Thus, while everybody appears to agree that the proposed solution is weird (for instance [kvm:1840]), all alternative solutions proposed involve undoing the migration to *Git*. In fact, developers suggest to return to Subversion by offering tools and processes for being able to use Subversion exclusively, such as scripts for managing patch sets against the Linux kernel [kvm:1845]. Given this situation, the maintainer withdraws his proposal and *postpones* a decision to consider the issue some more [kvm:1869]. If we interpret this “wait and see” strategy of his and deduce its consequences, we see that the maintainer is most probably balancing the effort necessary to adopt one of the proposed solutions against its effects on reducing his “pain” and the chance that the effort could be wasted, if a better solution arises.

Wait and See  
Strategy

It takes another four weeks before this better solution is found. Instead of unifying the user-space Subversion and kernel *Git*, the Subversion repository is migrated to a separate *Git* repository [kvm:2336]. The context in which this completion of the migration was done is given by the request of a user to enable http-access to both the *Git* and Subversion repositories [kvm:2184]. Thus, what the user is asking is to invest effort into maintaining both systems. Interpreting the partial migration as an effort management technique, it loses more and more appeal, if such requests continue to arise. Thus, the maintainer uses the opportunity and migrates to a separate *Git* repository. After two months of using *Git* and having overcome initial resistance, this even draws a cheer from a project participant [kvm:2434].

A graphical summary of this analysis of the partial migration occurring at the project KVM is shown in Figure 5.3.

To conclude the discussion of the partial migration at KVM, we come back to our initial questions:

- *What causes partial migrations to occur?*  
The existence of antagonistic forces such as shortcomings of existing tools and lacking knowledge about novel ones may be the cause of partially migrated innovation introductions.
- *What are the consequences of partially migrated innovations?*  
Partially migrated innovations do cause painful duplication of effort to manage and sustain both innovations, yet at the same time enable the project to become comfortable with both innovations.
- *How and to which effect do projects and their members deal with these consequences?*  
The project actively engaged in discussion when the maintainer mentioned his pains with managing both innovations in parallel. It was quickly agreed that it was important to resolve the situation. Yet, the project members did only provide solutions based on the old innovation, and it remained the task of the innovator to resolve the issue.

### 5.3.2 Partial Migration at ROX

To discuss the concept of partial migrations, we choose as a second episode a migration occurring in the project *ROX*, a project which develops a desktop environment that is closely tied to the file-system. Again, this is a migration away from centralized *Subversion* to decentralized *Git* and the decision to migrate is also made unilaterally by the maintainer of *ROX*, so that we do not see any decision processes regarding the migration, but rather only get informed by the maintainer about his rationale to migrate the core component of *ROX*—the *ROX-Filer*—to a decentralized version control system (see Info-Box 1 for a detailed discussion of this rationale). Reasons why all other of the sub-projects such as the window manager *OroboROX*, the session manager *ROX-Session*, or the library for shared *ROX* functionality *ROX-lib* remain in *SVN* are not given directly as part of this proposition, but hide in the ensuing discussion during the execution of the migration.

*Git at ROX*

A first indication why the strategy to migrate only partially was chosen can be found in the e-mail by the maintainer in which he retrospectively *narrates the conversion* to *Git*: The maintainer describes that the technical capabilities of both SCM tools make the partial migration more attractive: In Subversion it is easy to extract parts of a repository and in *Git* it is easy to combine two repositories. While this is

no causing condition, it is augmenting, making the strategy more appealing. So, in addition to the module boundaries and boundaries of responsibility (Kernel code vs. User Space in KVM) we can add technologically-indicated boundaries as possible candidates along which a partial migration might occur.

When the maintainer of *ROX* proposes to migrate the repository of the central component ROX-Filer from Subversion to Git, his argumentation on the highest level of abstraction revolves around enumerating disadvantages and problems with the existing and advantages of the proposed innovation [rox:9368]. He describes these from the viewpoint of (1) the contribution processes for non-core developers as a whole (process-centric view) and (2) certain important individual tasks (capability-centric view) such as keeping up to date with an evolving repository, collaborating without the presence of a project core member, and merging a set of changes with full history.

His argumentation also elegantly separates the discussion about a solution in the abstract (using decentralized version control) from the discussion about a solution in the concrete (using Git): First, he makes sure that he has convinced the project to use a decentralized version system and then he provides an additional set of arguments favoring Git over other DVCSs (all these arguments are based on external validation such as being in use at the Linux Kernel).

Info-Box 1: Rationale given for the *migration from Subversion to Git in the project ROX*.

If we look at an answer of the maintainer to the question whether there were plans to complete the migration [rox:9380] and take the answer—no there weren't any—as indication why the migration occurred only partially, we find two more conditions that are involved: (1) Finishing the migration would require a lot of free weekends [rox:9384]. Abstracting from this puts *lack of time* and *associated effort* on the table of causing conditions for partial migration. (2) He first wants to see how using Git for the ROX-Filer will work out, from which we can deduce an unfamiliarity with the new technology (similar to the one occurring during the introduction of Git in KVM) that is causing the maintainer to run a trial.

If we next look for the consequences of the partial migration, we find that they are not as clear as they were with KVM, because nobody is really complaining about having pains with the migration being partial. Rather, we notice two problems that were caused by the migration being only partial: (1) As not all content managed in Subversion is deleted after the migration to Git, there are some commits that end up in the wrong repository and need to be manually transferred to Git [rox:9404]. This can be abstracted as increased usage problems. (2) A more complex problem might be caused by the duplicate learning effort. In [rox:9424], a developer asks for some patience with providing a patch using Git, because he is still in the process of migrating in his personal development from CVS to Subversion and has not had time to learn how to use Git. If a migration is only partial, this developer consequently cannot just stop learning about Subversion and start learning to use Git, because he still needs the related skill to use the parts not migrated.

More consequences probably exist, but they remain unspoken of, so that we are left to wonder whether they would cause anybody to want to complete the migration or whether other strategies were adopted. As noted, the maintainer has opted to use a “wait and see” strategy of first running a trial with ROX-Filer and Git [rox:9384] before considering to migrate the rest of the project. We do not get any answers to the question why this strategy was chosen and what it accomplishes, yet, the most likely reason seems to be effort and risk management on the side of the maintainer: Not investing effort in a new technology or innovation unless there is a clear indication to do so. Another conservative strategy can be observed when a developer adds new features to a sub-project remaining in Subversion and creates at the same time a new sub-project to complement the existing one in Git without any obvious problems [rox:9499].

We find that the only developer who appears to be interested in extending the migration to more sub-projects is the one who initially asked the maintainer whether he had plans to complete the migration. When he gets a negative response from the maintainer [rox:9384], he pursues two interesting strategies (this developer is already experienced with the use of Git [rox:9370] and has access to hosting [rox:9380]

as two important intervening conditions):

First, the developer offers to the project to provide an *adapter* that allows to access an existing Subversion repository using a Git client (a git-svn mirror) [rox:9385]. While such a system cannot copy the workflows enabled by distributed version control system, it enables its users to have a homogeneous tool environment. For the innovator it is an interesting strategy to support the introduction of Git because the adapter can be set up independently and without legitimization by the project. In fact, the concept of an adapter has not only proved useful in making partial migration appear like a completed one, but also it has been used as a powerful way to introduce an innovation (see Section 5.6).

Adapter  
Homogeneous  
Tool  
Environment

The second strategy this developer employs is to migrate individual parts of the Subversion repository to Git using his own hosting infrastructure [rox:9399]. While he manages to migrate several such repositories [rox:9403], looking back one year later, we uncover that his efforts must have been in vain, because the repositories have disappeared, are still being managed in Subversion, or have been migrated again by the maintainer [rox:9647]. The only effect of his activities seems to have been that he and the maintainer learned how to migrate repositories to Git more effectively [rox:9409]. We will return to discussing private hosting strategies in Section 5.5.

To sum up this investigation into the partial migration of Subversion to Git in the project ROX:

- *What causes partial migrations to occur?*  
Lack of time and lack of expertise are the most likely reasons why the maintainer decided to conduct a trial and not complete the migration despite compelling advantages of the new innovation.
- *What are the consequences of partially migrated innovations?*  
ROX did not suffer any visible negative consequences like KVM from the partially migrated project state except for a couple of minor usage problems and possibly some duplicated learning effort.
- *How and to which effect do projects and their members deal with these consequences?*  
Because there are no negative consequences, we see conservative strategies with the goal to manage the expedited effort regarding the migration of existing sub-projects such as (1) keeping the status quo of existing sub-projects and only migrating novel ones and (2) using adapter technologies to enable homogeneous tool environments without migrating all sub-projects.

I believe that this analysis served to convey some of the reasons and implications of partial migrations in Open Source projects. Even though not saturated with a large number of cases, we have still seen that (1) partial migrations do occur in Open Source innovation introduction for diverse reasons such as a effort management strategy, (2) they are not without merit, as they give projects opportunities to learn about the new innovation and to experiment with its uses, and (3) they provide the innovator with a strategic opportunity:

**Strategy 1 (Migrate Partially)** *Migrating only partially can be a suitable effort management strategy and give the project members time to experiment and learn about a novel innovation without losing all productivity based on the existing solution.*

## 5.4 Enactment Scopes of Process Innovation

A second exploration into a concept was triggered by two proposals which occurred within two weeks on the mailing list of the desktop manager *Xfce*. Both proposals suggested similar process innovations, yet their outcomes were entirely different. The first proposal asked to define a target release for each outstanding bug, so that planning and tracking of progress towards the release would become easier [xfce:12700]. The second proposal asked to close bugs already fixed and outdated with the current release [xfce:12949]. In essence, both proposals asked to visit all open bugs and assess their status in one or the other way. The first proposal—thirteen days before the second—failed to attract any attention of the rest of the development team, while the second proposal was accepted by the

project and enacted successfully by its innovator. What might have caused this difference in reception by the project members?

Looking at the reputation of both innovators in the project reveals something puzzling: The successful innovator is not even a developer in the project. Without any commits or patches and only a third as many e-mails written, he is still more successful than the innovator who failed. Standing with the project, it seems, cannot explain the difference in outcome.

What was striking during the analysis was that the successful innovator proposed only a single enactment of the process innovation for the current release, while the failed innovator proposed vaguely, implying all future releases. This looks like a plausible difference which might explain the difference in response to the proposals: Being asked whether one wants to perform the process activities once is certainly a less difficult decision than deciding whether one wants to perform them now and for all future releases.<sup>129</sup> More formally we can define:

**Definition 4 (Enactment)** *The execution of activities mandated by a process by one or several agents [175, cf.].*

And based on this:

**Definition 5 (Enactment Scope)** *The set of situations in which a process should be enacted.*

Given these definitions, we can hypothesize:

**Hypothesis 1** *Restricting the enactment scope of an innovation proposal is positively correlated with introduction success.*

Which suggests the following strategy to be used by an innovator:

**Strategy 2 (Keep the Enactment Scope Small)** *To increase introduction success for process innovations, an innovator should limit the enactment scope of the innovation.*

A directly visible advantage of such a strategy would be that after the initial enactment of the process the innovator can assess much better whether the introduction is worthwhile to pursue. If not, the effort spent to convince the project to adopt the process for an abstract set of situations can be saved.

In order to evaluate whether the hypothesized relationship holds between the probability of failure to be accepted and size of enactment scope and can be used to formulate the proposed strategy above, I looked at ten other episodes in which *process innovations* were proposed. I then arranged these episodes in GmanDA using its tabulation view (see Section 3.4) across the dimension of *enactment scope* and *outcome* of the introduction.

To begin with, I found two more episodes which exhibit the proposed correlation: Consider first one episode in the project *gEDA*, in which an innovator proposes to adopt a stable/unstable branching scheme and focuses his proposal on the current situation [*geda:4072*]. He describes the implications of branching in the immediate future, nominates a certain person to be the maintainer of the new stable branch, chooses version numbers for each novel branch, and gives detailed instructions for how development would continue. This proposition with small enactment scope is accepted, and a stable branch is created within a month [*geda:4350*]. Following this first enactment, yet without any renewed discussion or involvement of the original innovator, the branching procedure is enacted two more times more over the next six months [*geda:4498,5629*], thus strengthening the idea that an innovator might want to focus on getting a first enactment of a process innovation introduced and then use its success to extend to a larger enactment scope.

Branching at  
*gEDA*

Design  
Approval at  
*Bugzilla*

A second episode contains the rare event of a lead developer failing to gather support for a proposition of his, when he keeps the proposal too vague and thereby largely scoped [*bugzilla:6943*]. This is interesting

<sup>129</sup>It should be noted that proposing such a correlation only makes sense, if we also assume that the *expected effort* of adopting a process innovation is a central factor correlating with the acceptance of this innovation. While I think this is a reasonable proposition, it is not yet grounded in coded observations.

because even though the developer proposes only an optional step to be added to the quality assurance process (in this case an *optional design review* before the mandatory patch review), his proposal is challenged on the notion that the proposal would cause effort and complexity:

“And why a comment from us is not enough?” [bugzilla:6944]

So again, we have found that a large or vague enactment scope correlates with fear of expected effort.

When studying the cases that do not fit the proposed correlation, we find three episodes in which an enactment scope of a single enactment still leads to rejection.

In the first case, a maintainer proposes to two core members to use a separate branch for collaborating on a feature, when they start having problems with managing their patches via the issue tracker [argouml:4772]. This proposal is rejected by the two core developers arguing in the following way:

*Branch for  
Patches at  
ArgoUML*

1. The first developer directly states that he deems the use of branches too much effort [argouml:4773]. The second one says the same, but indirectly so by noting that branches have not been used for years to his knowledge [argouml:4784]. I interpreted this as primarily stating that the use of branches would have to be learned again. If we interpret both developers as pointing towards effort as reasons for rejecting the process innovation, we can clarify our current hypothesis that a reduced *enactment scope* can help to alleviate fears of overburdening effort: If even a single enactment of a process innovation is expected to be too cumbersome, the proposal will still fail:

**Hypothesis 2** *When reducing the enactment scope of a process innovation proposal, the innovator still needs to achieve acceptance for a single enactment.*

2. The second argument used against the process innovation is not associated with effort expectancy, but rather picks up on the small enactment scope of the proposal: Even though the innovator asks the two developers whether they want to use a branch in the *current* situation of collaborating on a given feature, the second developer enlarges the enactment scope by asking whether the innovator was proposing to start using the branches again in *general*. He then goes on discussing the appropriate situations for branching and possible dangers. He does this under the implicit assumption that a single enactment of the process innovation can become the default case for the future. Doing so, he counteracts the strategy of using a small enactment scope by discussing the implications of a larger one. As a defensive strategy for the innovator:

**Strategy 3 (Protect against Scope Expansion)** *The innovator should guard his proposal against opponents who expand the enactment scope of an innovation by addressing concerns in concrete terms and situations.*

Yet, while the proposal to use branches for collaboration on patches is rejected by these two arguments, the maintainer continues to advocate the reinstatement of branches as a means for collaborating. He does so by making them the default development approach for students under the Google Summer of Code program two months later (see Section 8.1.2). Taking this indirect approach, he achieves within a month that the use of branches has been adopted by the two previously opposing developers, and that over time even one of them explicitly suggests to a new developer to use branches for the development of a new feature [argouml:5681]. In a way, the small scope approach thus succeeded eventually.

The second and the third case of rejected proposals, in separate projects each, each with small enactment scopes, extend these two insights: In the second case, we find an innovator proposing the participation of the project in the *Google Summer of Code* and notice this proposal fail because of lack of time of the maintainer and bad previous experiences with participating [xfce:13244]. This is in line with the hypothesis derived from the previous case that reduction in enactment scope still needs to achieve acceptance of a single case. The third case shows another example of attacking a small enactment scope. When the innovator suggests using the project's *wiki* as infrastructure for keeping track of bugs found just before a release, the opponent challenges this proposal by expanding the enactment scope to bug tracking in general:

“btw, I think the wiki is not very convenient, something like a forum would be much better.”  
[grub:3268]

From there the discussion quickly loses any association with the initial process-oriented question and becomes tool-centric.

When, in contrast, looking at the three episodes in which the enactment scope was large (i.e. should be used continuously or enacted in all future cases of a certain kind), we find that all of these proposals were (1) proposed by the maintainers of the project (2) in an attempt to influence general behavior such as being more disciplined with keeping a changelog [xfce:13533], being more friendly towards new developers [bugzilla:6190] or more careful when asking questions [bugzilla:6540], and (3) were all decided in favor of. Yet, their general nature and, therefore, broad enactment scope made it impossible to determine whether they were adopted as intended, and thus successful. Since we cannot say whether these proposals successfully changed the behavior of the project members, we can only speculate whether smaller scoping—for instance in combination with praising a particular friendly developer or reprimanding another for a missing changelog—could be more successful.

The two last episodes I want to present show another interesting aspect related to enactment scoping: Both proposals contained small and large scoped aspects at the same time. In the first episode in the project *GRUB*, an innovator proposed that developers interested in a common feature should work more closely together as *work groups* (in the literature on team effectiveness such a work group would be called a self-managing work team [95]) to speed up development on these features. He first did so with a large scope, targeting a possible wide variety of such work groups:

“Therefore I would propose that we would set up a group of people that would concentrate on specific issues related to implementation (like a work group or something).” [grub:3236]

Only later does he reduce the scoping by naming two such “specific issues” that would need work groups right now. Similarly, in the second episode, a developer proposes a new scheme for managing changelog messages, first by describing the general consequences of not adopting his innovation, and then becoming concrete in describing how to execute his new scheme [geda:4330].

This second episode of combined small and large enactment scoping was accepted and successfully used by the project while the idea of work groups fails in an interesting fashion: When the innovator proposes to establish them, the project members in the discussion directly start talking about their desire and abilities to participate in the two concrete work groups that had been proposed. Such a *skipped decision* of jumping into the adoption of an innovation without any discussion is an interesting strategy for an innovator to bypass opposition. Since the work groups in this particular case—formed without a decision—failed to go beyond an initial declaration of interest and their momentum quickly fizzled out, the verdict is still out whether the possibility of a skipped decision is an advantage of a certain type of innovation or whether it rather leads to premature enactment when the consequences of the innovation are not yet well known. To remain on the cautionary side, we might leave this as two open hypotheses to investigate:

**Hypothesis 3** *When proposing a process innovation with too small an enactment scope, this can cause the constituents of the project to skip decision and directly enact the innovation.*

**Hypothesis 4** *To skip decision and directly adopt/use/enact an innovation is negatively correlated with introduction success.*

As a last remark on enactment scopes note some similarities between the concept of *enactment scopes* and *partial migrations*: With partial migrations we have seen that tools are not introduced for all parts of the project, but rather that some parts remain under the old technology and some are migrated to the new one. In a way, we could also interpret this as a reduction of enactment scope along the dimension of modules in contrast to the reduction of scope in the dimension of future enactment situations. To discuss this relationship in further detail remains for future work.

To summarize this section: (1) From two similar episodes with contrasting outcomes the property of the enactment scope of a proposal was constructed. (2) Using a small enactment scope, i.e. proposing an

Work Groups  
at GRUB

Changelog for  
Git at gEDA

Skipped  
Decision

Premature  
Enactment

innovation to be applied to a limited set of concrete situations, was hypothesized to be correlated with reduced expectations of effort and increased introduction success. (3) After showing two more episodes that exhibit the proposed correlation, first, the limits of reducing the scope of a proposal were discussed with an episode, which failed with a single proposed enactment, and then, the dangers were shown, when a concrete proposal led to premature enactment without decision. (4) The counter strategy of scope expansion was demonstrated to have made a tightly scoped proposal fail, and (5) a conceptual association to partial migration was drawn.

## 5.5 Hosting

While studying the concepts of enactment scopes (Section 5.4) and partial migration (5.3), another concept struck me as intriguing because of one episode in the commercially-backed project *KVM*: Some participants from this project had repeatedly asked for a wiki to be introduced by the sponsoring company and eventually created one by themselves when their calls remained unanswered. Despite this wiki being perfectly usable and available, the sponsoring company then hosted its own *official* wiki shortly thereafter on a server run by the company, and never acknowledged the system set-up by the developers [kvm:966]. This official wiki turned out to be so popular that within a week the network bandwidth of the server running the wiki was insufficient, which caused problems with the source code management system being run on the same machine [kvm:1173].

*Wiki at KVM*

Thus, where and how an innovation is *hosted* and why so may be important. We define:

**Definition 6 (Hosting)** *Provision of computing resources such as bandwidth, storage space, and processing capacity to the use by an innovation.*

Following the terminology of Barcellini et al., who distinguish between (1) discussion space on the mailing list and in IRC, (2) the implementation space in code, patches, and repositories, and (3) documentation space on the website and in wikis [25], one might associate the provision of resources with a fourth space, namely the server or infrastructure space of interaction.

We ask the following questions:

- Which kind of strategic and tactical choices regarding hosting do innovators have when introducing an innovation?
- What are the consequences of a certain kind of hosting for the introduction and usage of an innovation?
- Which other relationships exist between hosting and the innovation introduction process?

I started a series of coding sessions focused on the concept of hosting. In contrast to the two previous sections in which I had used the paradigm for partial migrations and tabulation diagrams<sup>130</sup> for enactment scopes, this section employs a literary style, because the analysis is more exploratory on the conceptual development than suitable for the other two approaches. Tabular diagrams as used for the discussion of enactment scopes particularly require a small set of alternatives, while the paradigm focuses too much on cause-effect analysis rather than embedding a concept of interest into a larger set of related concepts. Analysis was conducted using *focused coding* as described in Chapter 3:

I first searched the list of existing codes for the ones that were related to hosting such as “innovation.hosting”. I then looked at each e-mail labeled with this code and wrote memos in which I developed the concept of hosting and attached properties to it. After having written 36 memos regarding hosting in this way, I reread the memos and created a mind-map linking the developed concepts. The following discussion describes the properties of the concept hosting and its relationship to other concepts.

As a first step to understand the concept of hosting, we want to regard which types of innovation (see Section 2.2) have been found to use hosting:

<sup>130</sup>See Section 3.4

- Predominately, it was discovered that innovations with a *service* component such as a centralized source code management system, a bug tracker, mailing list, or the project website require hosting. As an interesting sub-case I discovered that projects which develop web applications also frequently host their own software for two reasons: (1) Their product is an innovation such as a bug tracker and is used by the project itself as part of their development process (“eat your own dog food”). (2) The product is hosted as part of another innovation such as providing a demo system to facilitate easier beta-testing.
- Second were innovations that require *documentation* to be accessible conveniently and up-to-date by developers and users. For instance, in the boot-loader project *U-Boot*, the maintainer wrote a design document which outlines the primary principles that have to be followed when developing for U-Boot, and then hosted it on the project server [uboot:29737].
- Third, it turned out that *tool* innovations are also frequently hosted by the project, so that developers can easily download a designated or customized version of the tool [argouml:4909].

Types of  
Hosting Next, the most basic question to ask about hosting is, which *types of hosting* can be distinguished:

- *Forges* such as SourceForge.net or Berlios are platforms for collaborative software development that offer a set of popular development tools, such as a SCM, a mailing list, bug tracker, and web-space. The projects hosted on such forges are usually not thematically related to each other.
- *Service hosts* such as repo.or.cz provide specialized hosting for a particular type of *service innovation* (in the case of repo.or.cz, Git hosting) [geda:2893].
- *University servers* often provide web-space and computing capacity to students and faculty free of charge. For instance, in the project gEDA, a participant used the *Student-Run Computing Facility* (SRCF) to host an adapter innovation making the Subversion repository of the project available via Git (see Section 5.6) [geda:2799].
- *Private servers* owned, rented, or operated in the name of and by individual project members. For instance, in the project *Flyspray* one of the maintainers sets up a service running the latest developer snapshot of the project’s software on his private server gosdaturacatala-zucht.de [flyspray:5422].
- *Affiliated hosting* occurs when a project asks a thematically related project to use its already available server or service infrastructure. For example, some developers in the project *KVM* at one point considered hosting a *wiki* on kernel.org, which is the umbrella website for all Linux-kernel-related projects.
- *Foundation hosting* occurs when a project is part of a larger group of projects that are joined under a shared legal entity. These groups or their umbrella organization then often operate servers for their member projects. An example would be the hosting used by the Bugzilla project, which is part of the infrastructure provided by the Mozilla Foundation, from which the Bugzilla project originated.
- *Federated hosting* is a lesser kind of *foundation hosting*, where the relationship between the individual projects is not based on a legal entity, but rather on a common goal or friendship and cost-sharing between the maintainers. For instance, the project *gEDA* is hosted by the *Simple End-User Linux* (SEUL) project, which provides a home to projects which want to do “development for user-friendly software for Linux, and more generally for high-quality free Linux software of all kinds”<sup>131</sup> [geda:2899].
- *Private PCs* are privately owned computers which do not have a permanent Internet connection.

These types were easily found but puzzling in their diversity. What are the attributes that make them unique, important for the introduction of innovations and distinguish them from others? Using GTM, I identified five concepts relevant for most of the decisions people make towards picking a host. These are

<sup>131</sup><http://www.seul.org/pub/hosting.php>



(1) effort, (2) control, (3) identification, (4) cost, and (5) capability. As the concepts all relate to and depend on each other, I will weave their explanation and definition together in the following paragraphs.

It is best to start with the concept of effort, which offers the most clear cut distinction why a decision towards a certain type of hosting would be made or not. Effort

**Definition 7 (Effort)** *The amount of time, mental concentration, or physical energy necessary to do something.*

In the case of hosting, such effort can include (depending on the innovation) the time necessary to install the software on the machine, migrate data to be used by the innovation, configure network settings, watch for security updates and apply them in time or deal with hackers and spam attacks, create back-ups, take care of overfull disks and hardware or software failures, move the server to other virtual or physical locations, inform the project about the status of the innovation being hosted, and answer questions about any of these. That maintainers should strive to minimize these activities should come as no surprise, and maintainers find strong words how they feel about the effort hosting can cause:

“I dont [sic] need to devote my time administering yet another mail server, doing so is a no option unless someone pay for my time” [flyspray:5411]

“People often tend to think about software quality, but the really important thing in services is the maintenance. It is surely easy to set up a server, but it is very tough to maintain such a server for years (especially without being paid) [...] Please consider my words carefully, before proposing not using savannah.” [grub:3273]

Yet obviously, given the above example of the introduction of a wiki in the project KVM, less effort or less anticipated effort cannot explain why the maintainers reject hosting the wiki using existing infrastructure on the *affiliate hosting* at Kernel.org, but rather invested much effort in installing a wiki on a KVM server. If effort is to be avoided, why is it that many projects host their services on their own servers instead of on big forges like Savannah or SourceForge.net, which have dedicated staff for administrative tasks? In the example of KVM, the decision not to use the wiki at Kernel.org caused a lot of effort to troubleshoot the software failures reported by users, and eventually required a possibly costly increase of bandwidth for the server hosted by the project [kvm:1168].

To answer this question, the following episode at the project ROX provided two more insights: Here, the project was experiencing such a slowdown of the website that beside proposing to update the version of the content management system used, the maintainer also suggested to migrate the web-page away from SourceForge.net to a *private project server* [rox:9507]. A caveat: as hosting on a private server costs money, the maintainer asked the project whether they would accept paying for such a server, for instance, with money from showing advertisements on the project website. This reveals two more variables which influence hosting choices: *cost* and performance (or rather more abstractly *capability* of the hoster). While hosting, which is “free as beer” [flyspray:5411]<sup>132</sup>, appears to be widely preferred, in this episode, the members of ROX were not opposed to trade dependence on some form of income such as from advertisement for a faster website. Cost

Capability on the other hand is a broad concept extending beyond performance. Most basically, it is a binary criterion on the existence of a certain service offered by a host, or the set of features offered by this service. If a host has insufficient capability for the needs of the project, then project participants need to look elsewhere. Consider the following exemplifying quote from the migration of Subversion to Git in the project ROX (see Section 5.3.2): Capability

“> Does SF provide git hosting now?

No, and given how long it took them to support svn I'm not holding my breath ;-)” [rox:9373]

<sup>132</sup>Adapted from the explanation given for the term Free Software: “you should think of free as in free speech, not as in free beer.” [478]

Capability of a hoster also includes more scalar aspects such as the *bandwidth* (as seen negatively in the episode at KVM), *storage space* [rox:31112], or *quality of service* attributes such as availability of a host [rox:9428]. The latter, for instance, is a drawback of private PCs which are not connected to the Internet permanently.

Yet, neither capability nor cost explains the KVM episode, as Kernel.org is both free of charge and a perfectly fine wiki on a solid infrastructure capable of handling a large number of downloads of Linux kernel snapshots every day. Upon further searching for episodes related to hosting, I came across the following in the database project *MonetDB*: One developer noticed that the results from the nightly regression test suite were provided on the web-page without an expiration time, so that a browser would cache the pages and not provide the latest results [monetdb:121]. When notified about this, the maintainer replied that a fix “must be done in the server” [monetdb:122]. Thus, the server is certainly capable of sending out appropriate HTTP headers, but somebody needs to be able to perform the corresponding changes. The developer proposing the improvement, alas, did lack the authorization to log in on the server and add the HTTP headers. So the episode failed. Abstracting, we arrive at the concept of *control* which the project members exert over a certain type of hosting:

**Definition 8 (Control)** *The degree to which the capabilities of an innovation and host are actually available to the project.*

Having sufficient control to perform certain tasks with the innovation such as configuring the innovation, backing-up its data, troubleshooting it, installing updates, and more, represents the first concept that explains why the KVM maintainer rejected the Kernel wiki: This wiki—while offering all necessary capabilities at a low price—is not ultimately controlled by KVM, but rather by Kernel.org.

Control only refers to the potential to perform an activity, but not that it will actually be performed. In the above episode, the maintainer could have configured the server to correctly set the expiration time, but didn’t [monetdb:122]. Be also aware that control is often bound to individuals who are authorized to perform certain activities. For instance, when one developer set up a Git-Web system to show changes to the Git repository of *gEDA* (see Section 8.1.1), he alone was in control of this system. He consequently had to ask project members to be patient with him and wait for the updates which he needed to perform manually [geda:9428].

If control is lacking, then the project members become dependent on hosting staff, are unable to resolve issues with hosting, or have to exert higher levels of effort:

- When the maintainers of *Flyspray* lost the password to administer the mailing list, their hosting provided them so little control that they did not have any other choice but to abandon their existing platform and migrate to another hoster:

“> So far, the old list was a better option for me.

for us, is no option, we had zero control over it” [flyspray:5411]

- In the database project *MonetDB*, one of the core members made a mistake while committing to the project repository, which rendered the repository unusable for the project. The situation could only be resolved by filing a support request with the staff of the hosting provider and waiting for them to handle the issue [monetdb:468].
- During the introduction of *Git* into *gEDA*, the innovators ran a Git-CVS adapter innovation to show Git’s usefulness to the project. In retrospect, the innovators reported that this had not been easy “due to lack of git servers we control” [geda:3068], pointing towards a relationship between lack of control and increased effort.

Such instances of lack of control highlight its implications for the use of innovations and offer another variable by which we can understand why not every project is using forges such as SourceForge.net with their low level of associated effort. Forges, we learn, provide less control than privately owned servers, on which the owners can work at will. A notable exception to this is the forge operated by the GNU project (Savannah): Since the software used to operate it (Savane) is Open Source itself,

the users of the Forge have the possibility to extend the capabilities of Savannah by contributing to Savane [grub:3273]. The effort can be substantial though.

Until now I have discussed control mainly as pertinent to administrative tasks, but the definition given above also includes another aspect: Are the project members sufficiently in control of the innovation to actually use it? For instance, in the context of centralized source code management it is common to employ a user-based access control scheme to only allow changes to the repository by project members with commit rights [monetdb:6]. If we consider a project member who has sufficient control to assign or revoke commit rights (sometimes called *meta-commit rights*), we can deduce that hosting (via its associated control) can be one of the sources of power in the project. It is likely that to establish such power is the essential driver in the episode at *KVM*, whose funding company Qumranet must maintain some control over the project (for instance via an intellectual property regime) to use it as a platform for commercial offerings [542].

When saturating the set of concepts that relate to the concept of hosting, I found one more interesting phenomenon to join the previously found concepts of cost, effort, capability, and control:

In the project *Flyspray*, the maintainer had become dissatisfied with the low level of user-feedback he received on a new experimental feature he had proposed to the project. When repeating his request for feedback, he offered to the project that he would be willing to set up a server on which the latest development reversion could be run. This would allow each project member to beta-test the software without having to install an unstable developer version themselves [flyspray:5395]. When the users responded positively [flyspray:5399], because many of them did only run stable versions [flyspray:5409], the maintainer set up such a development demo system on his *private server* [flyspray:5422], yet marked it as temporary:

“I have now set up a temporary development BTS (until we get one on our server) at

<http://gosdaturacatala-zucht.de/devel/>” [flyspray:5422]

What this hosting option was lacking in comparison to a permanent solution was none of the four previously mentioned concepts. Its only flaw was that it failed to fit the *identification* of the community. Identification Conversely, a participant in another episode, who was also hosting on a private server as well, explained that he did so to avoid being seen as “official” and rather give the hosting location the notion of being “experimental” [rox:9428]. Identification with the social norms of the Open Source community and adhering to its standard can also be subsumed in this concept: When the maintainer of *ROX* asked the project whether he could run advertisement on the project homepage to cover the expenses of moving to a paid hoster, we might interpret this as him being sensitive to the underlying norms in the project.

The fact that projects seek *identification* for their hosting options might also explain why *federated* and *foundation* hosting can occupy a good compromise between *private server hosting* (high control, high capability) and *Forges* (low cost, low effort) which both have low identification.

The analysis of these five concepts related to hosting constitutes the main insight generated by this investigation. We have identified cases in which each concept was an important factor during a discussion related to choosing an appropriate hosting: (1) *KVM*'s maintainers favored a hosting with high control, even though they were presented with an innovation hosted at a site with little effort, no cost, sufficient capability, and high identification. (2) In *GRUB* the maintainer used harsh words to remind the innovator that abandoning the existing forge for a certain service would cause a lot of maintenance *effort*. (3) In *ROX* insufficient performance of the existing host made the project members trade their existing low-cost solution for paid hosting with better capabilities. (4) When forced to look for a new mailing list host because losing the password caused lack of control, the project *Flyspray* decided that no cost would be a central requirement for the new service. (5) One maintainer of *Flyspray* realized that hosting on a private server, despite being cheap and fully functional, would provide insufficient identification with the project and thus declared the service as only temporarily hosted there.

The categorization in this section was found to accommodate all episodes in which hosting played a

prominent role and should therefore be adequate for innovators and researchers alike to use it as a framework for understanding hosting. Nevertheless, these concepts represents just one possible way to categorize the variables that influence a complex topic like hosting. It was not the focus of this dissertation to validate the completeness or usability of the developed categorization. Most likely though, if and when additional concepts arise, one will be able to add them or explain their appearance as a combination of the given ones. For instance, during the introduction of the source code management system *Subversion* in *FreeDOS*, one of the project veterans noted that one should host the project repository on SourceForge.net to “keep all FreeDOS-related resources in one place” [freedos:4826]. While defining a concept such as *co-locality* or *unification* of hosting resources makes sense, it is best to explain its motivation as a combination of the other five concepts. In this case, unified hosting is driven by reduced maintenance effort, more centralized control, and improved identification, since all services are kept in one place. Capability is affected by co-location as well, but it is unclear whether it would cause higher risk as a single point of failure or enable synergies. Cost is likewise irrelevant, as SourceForge.net is free of charge.

### 5.5.1 Strategies for Hosting

Because each identified concept has been identified to be important during at least one innovation episode, it is difficult to derive any strategic advice for an innovator regarding hosting. Different situations and different innovations will require different hosting solutions. Two basic strategies for non-maintainer innovators should nevertheless be stated:

**Strategy 4 (Prioritize Control)** *The more reliant an innovation is on hosting, the more effort should be spent to convince those project members in control of the project infrastructure.*

**Strategy 5 (External Hosting)** *To overcome both discussion and execution barriers, the innovator can start an introduction on a private server, offer the solution to the project and then over time achieve integration into the project.*

Yet, I have seen both the first and second strategy fail, and an innovator must be cautioned that they are not final solutions in themselves. The first strategy, it appears, is particularly dependent on the attitude of the person in control over hosting resources (often the maintainer), which implies that the innovator should explore this attitude in advance by looking at innovation cases in the past. Should it turn out that the maintainer is too conservative, other strategies must be used. A common problem for the second strategy is that the innovator considers his job done after setting up the innovation. Unfortunately, the innovation often is not able to overcome adoption barriers by the advantages it provides alone. The innovation introduction then dies a slow death.

### 5.5.2 Relating Hosting to Innovation Introduction

Last in this discussion of hosting I want discuss the relationships between the concepts of hosting and innovation introduction. So far, we have interpreted hosting as a concept on which the innovator can have a strategic influence by making a choice on the location where to host an innovation. Implicit to this was the conceptual relationship that hosting is (1) a *requirement* for the usage of an innovation and (2) a *task* during *execution* (execution is the set of activities necessary to make the innovation usable by the project).

In fact, the provision of hosting and the subsequent announcement of availability of a service often constitute the only activities that the innovator will be involved with. For instance, one developer of *Xfce* set up an instant messaging server to provide low-cost and secure communication to the project by installing a software on his private server, and announced this via the mailing list [xfce:13133]. There was no public discussion, decision making, or advertising the innovation, just the provision of the service resource and the announcement to the list. Even though the introduction in this case failed to find any

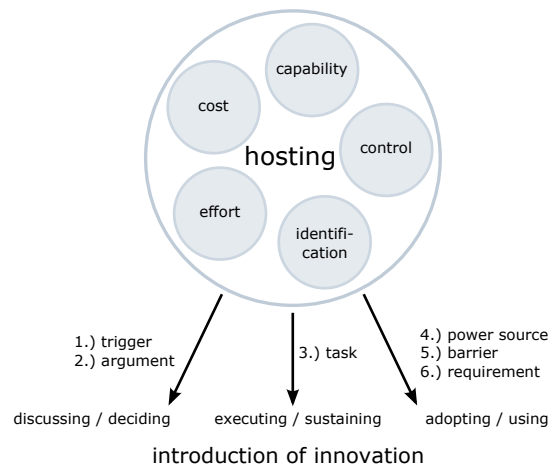


Figure 5.4: The conceptual relationships between the concept of *hosting* and the central activities performed during the introduction of an innovation. Hosting can serve as (1) a *trigger* for or (2) *argument* in the discussion and decision phase of an innovation introduction and as (3) a *task* during the *execution* and *sustaining* of an innovation. During the *usage* phase, hosting can act as (4) a *source of power* for its controller, (5) a *barrier* to adoption, and (6) a central *requirement* for the usage of the innovation.

adopters, we can take note that the achievement of hosting is the central activity of executing and possibly the minimal introduction scenario for a service innovation. Conversely, if the innovator fails to acquire hosting that is acceptable, the introduction as a whole will probably fail.

Despite the primary importance of hosting as a requirement and task during execution, there are also four minor conceptual relationships worth being mentioned:

The first conceptual connection between hosting and innovation introduction can be derived through the concept of control: We have discussed in the last section that control is often an important factor when making a decision against or in favor of a certain type of hosting, because it determines the ability to utilize the capabilities of the server. When considering these capabilities in relation to the use of the innovation, it turned out that control can lead to power, for instance when only certain people in the project can assign commit rights. Conversely, control can be a *barrier* to equality of access in the project (compare with Section 8.2).

Hosting as a  
Source of  
Power

Hosting as a  
Barrier

Second, I found that because achieving hosting is vital to achieving a successful introduction, it also often features prominently as an *argument* during the discussion and decision making. For instance, when a developer proposed to switch from CVS to Subversion in the project FreeDOS, the only restriction brought up was the demand to keep resources of the project unified on SourceForge.net [freedos:4826]. Thus, it might happen that instead of discussing the potential effects of an innovation, the only topic for discussion might be related to hosting.

Hosting as an  
Argument

Third, hosting can be a *trigger* to a new discussion and decision process. In particular, when hosting breaks or changes its status, this is a plausible point in time to consider alternatives to the current system. For instance, during the introduction of Git in the project gEDA, the innovators hosted an adapter innovation to allow project members to get to know Git. When this adapter got broken by the administrators, it was a natural moment to ask the project members whether there was any interest in moving the whole development to Git [geda:2889].

Hosting as a  
Trigger

Fourth and last, hosting can also be a source of countless *tasks* during the *sustaining* of an innovation (much of the discussion about effort of a hosted innovation derives from these sustaining tasks). In fact, it might be that such tasks are a much more considerable burden in the long run than the one-time

Hosting as a  
Source of  
Tasks

costs of setting the innovation up.

An overview of the proposed relationships is given in Figure 5.4.

### 5.5.3 Summary

The concept of hosting has been introduced as the provision of resources for an innovation and then connected to the innovation introduction process. First, we have discussed the observed hosting choices such as a *forge* over a *private server* based on the five determining concepts of (1) cost, (2) control, (3) effort, (4) capability, and (5) identification. Each concept has been shown to have played an important role in at least one episode, and any strategic suggestion to the innovator necessarily needs to take all of them into account. Last in this section, we have abstracted the discussion some more and considered hosting beyond being a requirement for enabling an innovation as (1) an argument or (2) barrier during discussion, (3 and 4) a source of tasks during execution and sustaining of an innovation introduction, and (5) a trigger for new innovation introduction episodes.

## 5.6 Adapter Innovations

During the analysis of partial migration (Section 5.3) and hosting (Section 5.5) another concept appeared which relates to both of them and is interesting enough to be discussed independently:

**Definition 9 (Adapter Innovation)** *An innovation used to make an existing innovation accessible by another one innovation.*

For instance, in the project *KVM* one developer used the adapter tool *Tailor* to import changes from the official *Subversion* source code repository into his private *Mercurial* repository [kvm:997]. In the following, I use the same terminology as the Gang of Four design pattern with the same name: The existing innovation (*Subversion* in the example) is called the “adapted innovation”, the innovation adapted to (*Mercurial*) is called the “target innovation”, and the innovation performing the adoption (*Tailor*) is called the “adapter” [202].

If we ask why such an adapter was used, I can provide a first set of cases in which the adapter can be seen as increasing the independence of tool choice, i.e. it gives each developer more choice regarding the set of tools he wants to use during development (compare Section 5.10 on tool-independence in general). I found three motivations for such independence:

1. As with the example just given, some developers have a personal preference for certain tools over others and thus need to adapt the officially offered system to be usable by theirs [kvm:997]. In some cases such a preference might be unqualified, in others it might have practical considerations such as additional features only found in the target innovation [bugzilla:6157].
2. Partial migrations might fragment the landscape of innovations being used by a project. Thus, some developers feel the desire to use adapters to make their tool environment homogeneous. For instance, in the project *ROX* an adapter allowed all parts of *ROX* to be accessed using *Git*, even though only parts had been migrated away from *Subversion* [rox:9385].
3. Certain tools might have high entry barriers for some project participants if they require complex installation or certain platforms. For instance, in the project *Bugzilla* the maintainer installed an *IRC Gateway* “for people who can’t easily get on *IRC*” [bugzilla:6263] which enables to join the developer discussions from within a browser.

Beside this set of reasons why an adapter was used, I found one case in which an adapter was used as a strategic device throughout an innovation introduction in the project *gEDA*: Here, the two innovators used adapters as part of their many activities to get *CVS* replaced by *Git*. Six advantages could be identified for using an adapter innovation as an intermediate step to achieving an introduction:

Tool  
Independence

*IRC Gateway*  
at *Bugzilla*

*Git* at *gEDA*

1. The innovators were able to set up the adapter without needing access to the project hosting or getting their actions legitimized by the project. Thus, adapters can help to avoid decision and execution barriers.
2. The adapter enabled them to start using Git in their daily work, thereby becoming familiar with the technology and learning about its problems and advantages.
3. Using Git enabled them to demonstrate to others first hand which impact the innovation could have if adopted project-wide [geda:2918].
4. Instead of having to convince all project members at once to introduce Git for the project, the innovators focus on individual developers and incrementally convince, demonstrate, and teach one after the other [geda:3635].
5. Because Git was available to the innovators, they could use it to introduce or propose additional innovations on top of it. For instance, they installed a *Git-Web* repository browser [geda:2799] for looking at the source code changes in the project and proposed Git as a staging system for the work of students in their summer projects [geda:3068]. This made the target innovation even more interesting, or plain necessary.
6. The existence of the tool provided repeated opportunity to talk about the innovation. In the observed case, even negative events created such opportunities, for example when the adapter innovation broke [geda:2889] or the data in the target innovation became out of date [geda:3930].

There were other reasons beside the ones given above why the innovators achieved an introduction. Yet, the examples highlight that the use of an adapter innovation<sup>133</sup> can be a supportive strategy:

**Strategy 6 (Adapter)** *Use an adapter to (1) bypass discussion, execution, and adoption barriers, (2) run a trial to demonstrating the usefulness of an innovation, and (3) create a homogeneous innovation landscapes for users.*

## 5.7 Forcing, Compliance, and Decisions

In this section the decision making processes involved while introducing an innovation are explored. I found in agreement with Rogers's distinctions in this area [436, Ch.10] that decision making about an innovation introduction can be divided into two major components: the *organizational innovation decision* and the *individual innovation decision* (Fichman calls this distinction the *locus of adoption* [182]). The organizational innovation decision is given by the decision of the project as a whole to acquire and commit to an innovation which then enables each individual to decide whether to adopt this innovation or not. This distinction is important because the individual's adoption is not automatically implied by the organization's decision. Rather, *assimilation gaps* between the organization's commitment to an innovation and the individual uptake might occur [185]. The organizational decision often has to precede the individual one because in many cases the implementing the innovation can only be achieved via the organization, but it is also possible to see individual adoption precede and promote an organizational decision. Both decisions will be discussed in turn.

Locus of Adoption:  
Organizational and Individual Innovation Decisions  
Assimilation Gap

### 5.7.1 Organizational Innovation Decisions

Well known from hierarchically structured organizations is the *authority innovation decision*, which is "made by a relatively few individuals in the system who possess power, high social status, or technical expertise" [436, p.403, Ch.10]. Such authority to unilaterally decide on the organizational adoption of an innovation is usually located with the project leaders, maintainers, or administrators of the project infrastructure. For instance, in the project gEDA the maintainer decided to migrate to *Git* as the source

Authority Innovation Decision

*Git at gEDA*

<sup>133</sup>As a minor point: The technical complexity of an adapter can be rather small, if the adapted and target innovations have well-specified interfaces [bugzilla:6629].

code management system making the following announcement:

“Here’s the plan going forward:

- I do want a stable/unstable branch/release arrangement going forward.
- I do not want to use CVS to maintain this though.
- So, I am going to go ahead and setup a git repository as the official repository of gEDA/gaf.” [geda:4123]

**Maintainer Might** Often, the *maintainer’s might* is so strong that the maintainer never has second thoughts about making a decision in such a unilateral way. If maintainers involve their project participants, it is often only to ask them for opinions about or objections to the plan, but not to constructively involve them.

**Representational Collective** The second most frequent way organizational innovation decisions have been made is by a *collective* or more appropriately by a *representational collective* innovation decision. Rogers defines the former as “choices that are made by consensus among the members of a system” [436], but I feel that the term consensus is inadequate to describe the fact that in most cases with Open Source projects a large part of the project is absent from the discussion and decision making process. Rather, the subset of the project participants who are involved with the discussion assume the role of representing the project as a whole. This is a necessary mechanism because of the decentralization of and fluctuation in participation. It is also directly connected to meritocratic principles: If a project member does not participate, this should not stall the decision making.

**Licensing Schemas at gEDA** For instance, in *gEDA* one of the maintainers of a sub-project proposes to clarify the licensing scheme used for graphical symbols. The proposal is made by posting an updated licensing disclaimer to the list [geda:3122].

Looking at the number of e-mails written in *gEDA* in 2007, only nine of the top twenty participants in the project participated in the discussion. The innovator then took the results from this discussion to revise the document and presented it again for a final round of feedback, in which he received four additional positive comments, two of which came from four of the remaining top twenty members.

**Legitimacy** A possible danger involved in invoking a representational collective arises, if this collective is too small or staffed with project participants too low in the hierarchy. The collective then lacks the legitimacy to actually make the decision. This problem is particularly pronounced in projects with strong maintainers.

**Git at GRUB** For instance, in the project *GRUB* a set of project participants had already decided to adopt a new versioning tool and, after a waiting period for objections, began to talk about executing the innovation. It was only then that the maintainer voiced his concern [grub:4116], thereby withdrawing legitimacy from the collective to cause the innovation introduction to be pushed back into a discussion about whether to adopt the innovation at all.

**Voting** The third mechanism found to drive an organizational innovation decision is *voting*. Famously known in the Open Source community is the Apache style minimum quorum consensus where each project member can vote “+1” (in favor of) or “-1” (against) for a proposed change. At least three votes in favor with no vote against are necessary to achieve a code change [186].

**Merge Conflicts at U-Boot** The most interesting example of the use of *voting* occurred in an episode in the project *U-Boot*: The vote happened when the project leader rejected a change to the code format in the build-files of the project [uboot:30660]. One developer, who had not participated in the discussion so far, informally triggered a vote by simply posting the following reply:

“I vote for the new, single-line version suggested by Kim & Jon.” [uboot:30667]

This way of starting a vote, however, does not define who can vote, how the result of the vote is going to be determined, and how long it will be run. Rather, all aspects are implicit, as are the no-vote of the project leader and the yes-votes of the two innovators. Despite being so badly defined, the effect of the call is immediately noticeable and in line with Fogel’s observation about voting in Open Source projects:



"It ends discussion, and [...] settles a question so everyone can move on" [190, p.94]. Thus, it then only takes two more votes in favor, which are equally terse, to make the maintainer accept the decision:

"D\*mn. Seems I'm on the side of snake eyes tossed against the side of seven again  
:-(" [uboot:30789]

Li et al. in their study of decision processes in Open Source projects observe that because of the informal nature of the voting, the maintainer is not really forced to accept the result of the vote in any binding way [313, p.7]. Rather, an obligation to be bound by the decision must arise from the social context of the project. In the case of this voting episode, the number of important project members participating (1) generates enough social pressure on the maintainer, and (2) makes it difficult for the maintainer to ignore the decision without overstepping his legitimacy as a project leader. If projects do define more formal rules on voting than in the preceding example, as is the case in the Apache foundation, it might make sense to distinguish between formal votes and informal polls, even if the above example suggests that an informal "show of hands" might be treated as binding [190, p.96].

There are not enough comparable episodes to derive strategies. Nevertheless, three hypotheses appear to be warranted:

**Hypothesis 5** *Voting can be an effective tool against individual high-ranking opponents such as a maintainer if the general opinion in the project is in the innovator's favor.*

**Hypothesis 6** *Voting speeds up the decision process by reducing opinions to yes and no.*

**Hypothesis 7** *Voting derives its binding effects from informal social mechanisms.*<sup>134</sup>

Lastly, a series of innovation introduction episodes was found in which organizational innovation decisions did not occur at all, which was categorized as "*just do it*"-innovation decisions.

Skipped  
Innovation  
Decisions

Such *skipped* organizational decisions were particularly common when the innovator was able to execute the innovation to a large degree independently, for example when setting up mirrors of existing systems [rox:9385], *writing a new piece of documentation*, or *suggesting a tool for cleaning up white space issues*. It is a good indication of the pragmatic nature of Open Source projects [139] and the flat hierarchies that project members feel empowered to invest time and effort to execute an innovation even though the project has not given them a mandate to do so.

Given these decision types, an interesting question arises: Which type is used in which situation, introducing which innovations? Unfortunately, only the obvious correlation between the number of people affected by an innovation and an increased tendency to see voting or representational decision making could be observed. Yet, deviations are common with far-reaching innovations being decided by authority innovation decision or skipped decisions and large collectives discussing peripheral issues.

To sum up, several different types by which innovation decisions can be made at the organizational level have been portrayed and their implications for the innovator been discussed.

## 5.7.2 Individual Innovation Decisions

After the project as a whole has adopted an innovation at the organizational level, I typically saw a set of activities performed to enable the use of the innovation by the project. For example, data needs to be migrated [freedos:4835,xfce:9378], software must be installed [geda:2799,rt:4295], and documentation has to be written [geda:5069,rox:9373]. Such activities have been labeled *executing activities*. Once these activities are completed, each project member can now start using the innovation. The question of whether a project member does actually adopt and use an innovation, is called the individual innovation decision. I want to discuss the results of looking at the extrinsic factors affecting individual innovation decisions, namely the concepts of *forcing effects* and *compliance enforcement*.

<sup>134</sup>In contrast to formal and technical ones.

Following Denning and Dunham's model of innovation introduction and its central concept of adoption [140], I was surprised to see that some innovations did not incur any noticeable adoption periods. Instead, adoption happens fluently and without any active resistance. Considering the traditional view that "the [OSS] community is primarily a loosely coupled network of individuals with no organizational forces in terms of economy or management that can force other individuals to behave in a certain way" [44, p.310], this poses a conundrum. How can it be explained that for other innovations the adoption is the most central aspect of the whole innovation? One answer that goes beyond relative advantage of one innovation over the other is the concept of *forcing effects*.

Forcing Effects

**Definition 10 (Forcing effect)** *A property or mechanism of an innovation that promotes the use of the innovation itself.*

Consider for example the introduction of the legal innovation GNU General Public License (GPL) v3 in the project *GRUB*. The maintainer proposed in July 2007 to switch the sub-project GRUB 2 from GPL Version 2 to Version 3 because of a recommendation issued by the Free Software Foundation to switch all software of the GNU project to the latest version of the GPL. After letting the project participants discuss about the proposed license migration, the maintainer unilaterally decided [grub:3380] and executed the switch [grub:3385]. This was possible because the Free Software Foundation holds the copyright to all parts of GRUB and can thus execute such a change. If we now consider the adoption of this innovation by the project members, we see that the nature of the GPLv3 forced them to adopt the GPLv3 implicitly. This is because the GPL is a viral license and ensures that only code may be contributed to the project which is under a license which is compatible with the latest version used. In this case, project members could no longer contribute GPL v2-only code to the project without being in violation of the terms of the GPL. In other words, the innovation itself has forced its own adoption based on a *legal* mechanism and the ultima ratio of excluding project members from participating.

GPLv3 at  
GRUB

Legal Forcing  
Effect

Such forcing effects which make participation contingent on adoption were observed to derive their power from two other mechanisms beside the *legal* mechanism: First is the existence of a *data dependence*, which is best explained by the example of migrating one source code management system to another. Because the repository from which the "official" project releases are made is changed to the new system, project participants cannot continue contributing towards such a release unless they contribute to the new system. If somebody continues to commit to the old repository, his changes are lost. In the project *Git at ROX*, for instance, parts of the existing Subversion repository were *migrated one after the other to separated Git repositories*. During this migration, one developer accidentally committed to the old Subversion repository after the Git repository was announced. Because he had not had time to adopt Git as a client tool yet, he needed to ask the innovator to move his change from Subversion to Git for him unless he wanted his work to be wasted [rox:9404].

Git at ROX

Data  
Dependence

Code Is Law

The second mechanism by which innovations can achieve strong forcing effects is by their use of systems restricting participation. An example for such a "*code is law*" [307] mechanism is the change of the mailing list in the project *Bochs* to a subscriber-only mode. By changing the configuration of the mailing list to accept only those members who have registered with it, the maintainer forced all project participants to sign up, because the mailing list would otherwise reject their e-mails [bochs:7272]. Again, we see that the underlying mechanism uses the desire of participants to participate to achieve its forcing effect.

It should be noted that forcing effects are not necessarily perceived as negative by the project participants. Rather, and in line with results regarding the motivation for participating in Open Source projects [210, 298, 238], one participant remarks that force has positive effects such as promoting learning of new technologies [rox:9369].

Several examples of forcing effects can be found in the literature on Open Source as well. Krafft, for instance, discusses the case of Lintian—an automated policy checker for the rules of creating packages in the Debian distribution [293, Sec. 7.2.8.2, pp.228ff.]. Krafft reports strong positive sentiment towards the tool's ability to help the Debian project to make incremental changes when issues such as outdated or non-portable scripts are detected in the way a package is built. On the other hand, there

are critical voices who feel that the forcing effects that Lintian exerts are “going too far nowadays” and that the project is focusing on quantitatively measurable issues instead of soft issues such as good documentation [293, *ibid.*, pp.230f.].

A second set of examples of forcing effects is given in the Open Source literature which used Actor-Network theory as the underlying framework for analysis (see Section 6.4). Shaihk and Cornford, for instance, mention the role of the source code management system in the Linux kernel project as an obligatory passage point “through which all other interests must pass” [462]. De Paoli et al., similarly, discuss the role of the project license as an obligatory passage point, focusing in particular on one discussion in which the license is sought to be changed [137].

To sum up: There are three mechanisms—*legal*, *data dependence*, and *code is law*—by which an innovation can generate a strong forcing effect to be adopted. In all cases, these mechanisms cause participation to be conditional on adoption.

If we regard other innovations which do not contain such strong forcing effects, we become aware of two more classes of innovations: Those with *expected* use and those with *optional* use:

First, innovations with an *optional innovation decision* are defined as those in which each individual can make the adoption choice independently and the project does not hold any expectations towards adoption. An example might be a *script to correctly indent source code*, the use of which is optional as long as the submitted code adheres to coding standards.

Optional  
Innovation  
Decision

Second is the class of innovations for which the adoption is *expected* by the project members, but not automatically required by forcing effects as above. For instance, in the project *U-Boot* it was proposed to change the coding standard used for build-files. In particular, it was suggested to list each build item on an individual line instead of four items on a single line. The intent of the change was to reduce merge conflicts when individual items in the lists changed [uboot:30646]. This was agreed on and executed as a new coding standard after the project members voted their maintainer down, who had opposed the change [uboot:30789]. To represent how such an innovation can become adopted when it does not exert forcing effects, the concept of expected behavior was derived from this episode in two different ways: (1) Adopting an innovation at the organizational level can cause behavioral norms to come into place which make it a social obligation to act according to expectations. (2) Once a decision to adopt an innovation has become embedded in the code as a new status quo, the code is able to have sufficient normative power of the factual to uphold an expectation for others to maintain it.

Expected  
Innovation  
Decision

In an introduction episode of *peer reviewing* in the project Subversion—as noted by Fogel in [190, pp.39ff.]—the importance of habituation as a third possible source for establishing expectation can be exemplified: Here, the innovator chose not to aim for a discussion on whether to perform peer reviews, but rather set an example by “reviewing every single commit” [190, p.40]. Over time, developers started to expect the review and became worried when it was not performed [190, p.40].

Two other mechanisms which can generate usage expectations are (1) guidelines, which formalize norms into written documentation (see for instance [argouml:4663]), and (2) *pleas*, which represent the most basic form of creating an expectation in another person towards a behavior (well exemplified in [geda:3988]).<sup>135</sup>

### 5.7.3 Compliance and its Enforcement

The next concept to understand arises, if developers do not comply with a given expectation. If no forcing effects exist, then additional mechanisms outside of the innovation are required to detect and correct such violations. This leads to the idea of compliance and *compliance enforcement*:

**Definition 11 (Compliance)** *The act of following a given norm or standard of using an innovation.*

<sup>135</sup>The language of this sentence does explicitly not focus on the act of communicating or expressing an expectation, but rather highlights that expectations must become internalized or created in the other person to affect behavior.

**Definition 12 (Compliance Enforcement)** *The act of ensuring that a given norm or standard is complied with.*

Gate Keeper In the above example, the project U-Boot uses a *gate keeper* strategy to ensure compliance with the norm and status quo of single items per line: To contribute to the product, the contribution needs to pass through the watchful eye of a module owner (called ‘custodian’) or project maintainer. If a violation is detected, the custodian or maintainer in their role as gate keepers will reject the patch (or correct it themselves) [uboot:32452]. If we reason about the underlying mechanism from which the gate keeper as a compliance enforcement strategy derives its power, we can see that the strategy also uses a data dependence on the project repository and the underlying contingent participation. The origin of the power of both strategies is therefore the same.

A second compliance enforcement strategy that uses the contingent participation as the ultimate source of power was found in the project MonetDB, where the maintainer sends an e-mail to the list after the opening of each release branch, reiterating the rules of which branch to commit to. To enforce these rules, the maintainer then states:

“Any violation of these rules might be ‘punished’ by a forced undo of the respective changes.” [monetdb:54]

Forced Undo The *forced undo* of a commit to the repository is thus another way to invalidate effort and sanction those activities that are in violation of the rules set by the project.

A hint on the relative importance of both strategies can be found in the study on decision making for source code modifications by Li et al., in which they studied the order of phases in the decision process and found that only a third of decision situations on the mailing list did include evaluation of the solution akin to the gate keeper strategy [313].<sup>136</sup>

As a last point, the relationship between expected and (en)forced behavior can be clarified:

1. Both the *forced undo* and the *gate keeper* enforcement strategies are reliant on detection by a watchful guardian of the project repository. The most well-known mechanism providing such detection is certainly peer review. Yet, it should come as no surprise that Open Source projects should also attempt such detection by technical means. During the migration to *Java 5* in *ArgoUML*, one of the core developers proposed to restrict usage of automatic conversion of primitive types into object types (a language feature called auto-boxing [219]). This language feature leads to more concise code [argouml:4981], yet, the core developer has reservations because auto-boxing incurs a performance penalty which is hard to notice by a developer [argouml:4967]. However, the innovator himself notes the dilemma of proposing a coding guideline and thereby defining expected behavior immediately: If the use of auto-boxing is hard to detect for the author of code, it will be even harder to notice for a reviewer, and thus difficult to enforce its exclusion from code. The technical solution which the innovator proposes is to use a static code analyzer such as Checkstyle<sup>137</sup> [argouml:4967]. Where a *code is law* compliance enforcement strategy would directly enforce by means of a software system, a static code analyzer merely aids the detection of being in violation and requires a mechanism such as a gate keeper to enforce compliance.
2. As a second point concerning the conceptual relationship of enforcement and expectancy of “correct” innovation use, consider the following: Just as the presence of a police officer will prevent crimes which would happen in his or her absence, the awareness about enforcement strategies such as a *forced undo* will most likely prevent most deviations from the norm. Thus, compliance enforcement can be another effective mechanism to maintain expectations towards the behavior of project members.

An overview of the forces affecting the adoption of an innovation by individuals is shown in Figure 5.5.

<sup>136</sup>Since their study operates on a structural level without looking into each decision episode, they do not attain conceptualization on a strategy level as in this work (compare Section 4.5).

<sup>137</sup><http://checkstyle.sourceforge.net>

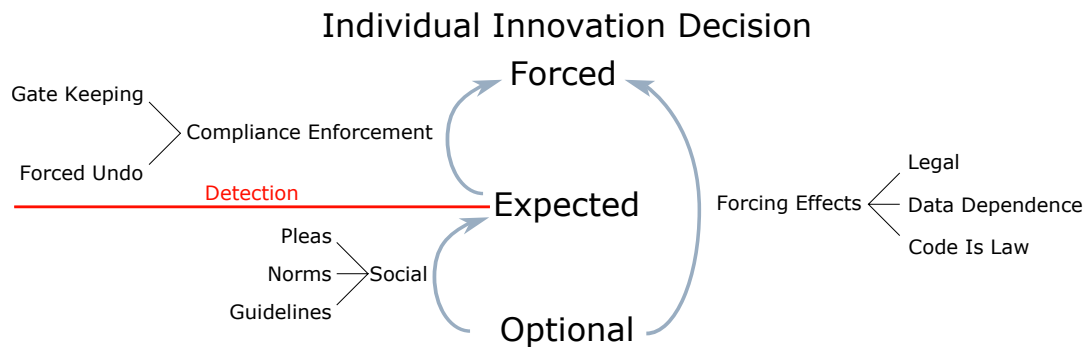


Figure 5.5: The proposed model explaining how an individual innovation (adoption) decision is shaped by social actions (left side) and by attributes contributed by the innovation (right side).

Abstracting from the strategies, we see that both strategies are similar in that they work primarily to ensure some quality of the code. It is unclear, however, how a compliance enforcement strategy can help in assisting in the adoption of innovations that do not manifest in code.

Process innovations, on the other hand, which lack any such relation to code, require strategies from the innovator that go beyond forcing effect and compliance enforcement. This is illustrated by the following quote from a discussion about how to encourage the users of the project to report bugs directly to the bug tracker:

“I don’t think we could be more explicit or efficient in discouraging the use of the mailing list for bug reporting without the risk of being perceived as discouraging in this.” [argouml:4752]

Put more abstractly, while the forces to achieve adoption draw their power from restricting participation, the innovator has to carefully balance the use of this force, because if forced too much, the participants may withdraw, and if consciously so, they can turn their participation into a power source as well. If we consider the forking of the projects GCC, XFree86, and TWiki, we see that in each case the project leaders had restricted—or threatened to restrict—participation so much that project members quit and forked the project<sup>138</sup>. For instance, in the project TWiki the project’s backing company locked out all developers from participation and required that they agree to new terms and conditions before resuming their work. This demand for a transfer of rights to the company and its leader, upset the community so much that a large number of project participants including many of the long-term core developers forked the project and created FosWiki.<sup>139</sup>

As a last point of discussion, I want to relate the insights about the concepts of forcing effect and enforcement to the model of decision making discussed at the beginning of this section.

If we make the distinction between organizational and individual innovation decisions, then the forces—as discussed above—primarily affect the individual’s decision whether to use the innovation or not. But if the forces are anticipated by the individual, this anticipation can also shape the discussion leading up to the organizational decision. In summary, forces can have ambivalent effects as a strategic device for an innovator:

**Hypothesis 8 (Forcing Effects)** *While forcing effects and mechanisms for compliance enforcement can speed up individual adoption, they can increase resistance during organizational adoption.*

<sup>138</sup>Unfortunately, there is only anecdotal material on forking in the Open Source world such as Rick Moen’s “Fear of Forking” [http://linuxmafia.com/faq/Licensing\\_and\\_Law/forking.html](http://linuxmafia.com/faq/Licensing_and_Law/forking.html) or Wikipedia’s article on forking [http://en.wikipedia.org/wiki/Fork\\_\(software\\_development\)](http://en.wikipedia.org/wiki/Fork_(software_development))

<sup>139</sup>An account given by the forking developers can be read at <http://blog.wikiring.com/Blog/BlogEntry28>, while no official statement by the TWiki project leaders can be found about the fork.

## 5.8 Participation Sprints and Time-dependent Behavior

When considering how Open Source projects adopt innovations, it is important to not only look at the conscious activities of participants and the role of established structures and processes in shaping the outcomes of introduction. The concept of time also needs to be analyzed and entered into the equation. From this conceptual perspective, time is not just something that goes by while an innovator acts, but something that can be consciously used. In Garud's and Karnøe's words: the successful innovator can "marshal time as a resource" [206, p.21]. Exploring the concept of time in the studied episodes revealed many different associations and related concepts which are difficult to sort out. As before, we develop the concepts surrounding time by looking at individual episodes.

On November 9 and 10, one core developer in the project *GRUB* wrote 53 e-mails to the mailing list. Compared to his overall yearly activity of 270 e-mails or the overall average of 5.9 e-mails per day on the mailing-list, this intense two-day period is well characterized in the developer's own words as "spamming this list like crazy" [grub:3902]. More formally, I call a period of high activity by a project member a *participation sprint*. In most of the 53 e-mails the core developer reviews and comments on patches, replies to left-over e-mails, and encourages mailing list users to participate in the project [grub:3885,3883]. However, also two innovation proposals are made [grub:3902,3905].

Participation  
Sprint

Task Tracking  
and Bug  
Tracking At  
GRUB

Both of these proposals address important and reasonable changes to the *task-* and *bug-tracking process*, which the core developer just enacted in multiple cases and probably will have felt were in need for an improvement (see Section 8.1.5 for a detailed discussion of the latter episode). Unfortunately, the participation sprint ends shortly afterwards. One week later, the core developer returned to write 20 e-mails in less than three hours and then disappeared from the list for an entire month. It is not surprising that one of the episodes failed right away, the other only being successful because another core developer revived it almost two months later [grub:4047].

Variability and  
Fluidity of  
Participation

For the analysis of time as an important concept, this episode emphasizes the high variability and fluidity<sup>140</sup> in activity that average statistics, such as that 70% of participants spend less than ten hours per week on Open Source [211], cannot pay justice to. Second, neither intense participation sprints nor prolonged absence appear to be very compatible with the asynchronous rhythm of participation and e-mail communication [515] in Open Source:

Overloading

1. During a participation sprint, it is likely that a developer is overloading the communication, decision, and assessment capacities of the project. Faced with so much e-mail in a single day, information might easily be lost and decision opportunities squandered: "So many messages all the time. I can't really read it anymore, there's just so many discussions about all sorts of things related to it,..." [531], cited after [468, p.30].

Brief Flame

2. The developer runs the risk to become a "brief flame" [84], i.e. to overtax his capabilities to deal with the backlash of communication when people respond to a large set of initial e-mails.

Absence and  
Underloading

3. Prolonged absence conversely leads to underloading of the project's capacity by leaving questions unanswered for too long and stalled discussion without stimulus. The direst effect of underloading induced by absence is that the project forgets already discussed options, preferences, and decisions within months [grub:3927].

It is important to realize that both participation sprints and prolonged absence are largely specific to Open Source development. This is because (1) the low average engagement of less than ten hours per week allows for scaling up the number of worked hours manyfold, for instance during vacation, on a free weekend, or between jobs and (2) the volunteer engagement allows to stay away for extended periods of time, which would be impossible in most jobs.

Both overloading and absence deserve a little more elaboration. In the preceding episode, overloading was caused by a single active developer who acted on a wide variety of discussion topics. In addition, episodes were found in which the project as a whole exceeded its own capacity regarding time and

<sup>140</sup>See Section 6.2 on the Garbage Can Model for the concept of fluid participation.

attention. This phenomenon is discussed in detail in Section 6.2 on the Garbage Can Model and leads to the suggestion that strong leadership is needed to maintain focus and redirect the energy of a project in heated discourse [24, cf. also]. Looking at individual members, overloading of each individual can easily occur when participation is reduced while responsibility abounds [grub:3514]. This might cause episodes to fail because key people are unavailable [xfce:13244] or might lead to episodes in which innovations are introduced without objections being attended to [grub:3380].

It might be worth noting that slight overloading of the project as a whole is likely to be a common phenomenon, as the project is assaulted with new ideas, bug-reports, and user support [bugzilla:6766]. Thus, a successful project and a successful innovator must be able to structure innovation processes to fit the temporal capacity of a project. Section 5.9 on radical vs. incremental innovation expands on this point.

Absence as the primary cause for underloading has less important implications. Certainly, the absence of the innovator as described above can easily cause an innovation introduction to be *abandoned*. But in many cases, the innovator can simply reactivate the episode. In the episode above in the project GRUB, the issue of overhauling the bug tracker had to be raised four times before it was resolved [grub:3266,3391,3902,4047]. In a second example from the project *Bugzilla*, the maintainer restarted an episode to improve contribution to the project more than 12 months after the problems and potential solutions had been discussed [bugzilla:6190]. The question is, why would innovators start episodes when a stretch of low activity or absence is foreseeable? In the project *ArgoUML* we find an episode, which presents an answer. Here, one developer asked whether to use a certain feature of a previous innovation [argouml:4965]. Being forced to react before the request expired and the developer would start using the feature, one core developer responds with the proposition of a new innovation making the use of the feature potentially more safe [argouml:4967]. Yet, this proposition occurred at a time when the core developer is acting with low activity in the project, and after this e-mail he is absent for more than two weeks. In other words, the episode at GRUB occurred at the culmination point of an intensive stretch of participation and smells of a strategic mistake, whereas the innovation proposal at *ArgoUML* occurred in a decision time frame which the innovator had little influence on.

Checkstyle At  
ArgoUML

The concept of absence reveals another important property of how time is perceived in Open Source projects. When the core developer in this episode at *ArgoUML* is absent for two weeks, this absence is not recognized by the project. The core developer did not announce it and the project members did not notice. In the episode, the proposal then *fails* because the core developer is expected to act next. Abstracting, the phenomenon can be described as a trailing shadow of presence. Analyzing the cause for the failure, we realize that even though almost none of the project participants engage continuously over the year<sup>141</sup>, the most responsible maintainers are the only ones who announce their absence to the project. As with other processes in Open Source development, it appears that presence is informally and implicitly managed.

Trailing  
Shadow

The main disadvantage of this lack of explicit coordination is that participants wait for responses, not knowing that somebody might have left. One additional instance of this problem is described in detail in Section 7.4 on the introduction of unit testing at FreeCol. Here, the innovator had to explicitly signal to other project members that he would be withdrawing his engagement in order to get the project to take over his responsibilities in unit testing. The advantage of a lack of absence coordination, on the other hand, is that it reduces communication overhead and is a more natural fit for the loosely coupled participants who contribute only sporadically.

Signaling

Looking at this phenomenon of informal time management more widely, it can be argued that Open Source projects operate mostly without using precise reference points in time. Rather, temporal arrangements retain a large degree of vagueness, for instance, innovators proposing to perform a change “some day” [monetdb:18,rox:9384,bugzilla:6190] or “as soon as we have forked off the new stable branch” [monetdb:12], without indication whether this might happen next month or next year. Again, this is understandable when considering the volunteer nature of participating, which constitutes a “spare

Vagueness of  
Time  
Management

<sup>141</sup>A manual check on the twenty most active developers in the overall sample showed that 75% of them had at least a single two weeks period of absence from the project.

time exercise” [argouml:5424] in which contributions are highly variable and deadlines or schedules are rarely considered. Consequently, innovations seldom include elements of time behavior. Two rare exceptions should be described: (1) The *Google Summer of Code* program, in which students are sponsored over the summer to participate in Open Source projects (see Section 8.1.2), imposes a deadline for applications both on projects and students [xfce:13218,13242]. (2) Internet Relay Chat (IRC) can be used to hold scheduled project meetings. Yet, despite IRC being technically a synchronous communication device, it allows participants to drop in and out of the chat at any time or remain idle on the channel while working on other things. In a project such as Bugzilla, which held IRC meetings once a month, keeping meeting notes was therefore used to desynchronize the discussion to some degree, e.g. [bugzilla:6261,6320,6571,6754].

Timed Releases

Lack of time management has one important drawback: How can tasks and software releases be managed if their completion cannot be based on reliable commitments of individual participants? This question has led to the paradoxical answer that software releases in Open Source projects should rather be based on a timed schedule instead of a set of features. This is because individuals cannot be relied upon to complete an individual task timely, and a release based on features thus will thus be likely to take much longer than initially anticipated. Using a rough schedule for releases (commonly at six month intervals) can still be imprecise, but loosens this dependence on individuals. A detailed analysis of release schedules from the perspective of release naming schemes is given in Section 8.1.4.

What can innovators do when faced with variable time commitment in Open Source projects? A reasonable approach is revealed in every studied introduction of the version control system Git (discussed in detail in Section 8.1.1): Innovators themselves executed the central part of the introductions within very short periods of time (in *ROX* within 24 hours), but then gave their projects long periods to become accustomed to the change (even as long as four months after switching to Git, individual aspects of the migration were still not completed). Not relying on others to participate in the execution helped to avoid coordination effort; not relying on others to migrate in lockstep eased adoption pressures.

**Strategy 7 (Act on Multiple Time Scales)** *The innovator should act quickly and independently to enable an innovation, but spread out the time and opportunities for adoption.*

Project Events

A second strategy could be observed when the project was faced with an established tradition of not using branches [argouml:4784]. Here, the innovator found that waiting for a better opportunity to introduce the innovation to new project members and leaving the core developers undisturbed [argouml:4976], helped to gain sufficient time to demonstrate the innovation and learn about its intricacies. Such opportunities could be found surrounding the events in the project’s development process, for instance, before [argouml:5685] or after [xfce:12949,geda:4072] releases or after a branch was opened [monetdb:12]. Such events can *trigger* introductions because they free resources for change, make need for change more visible, or generate change to be piggy-backed on. Building on the “wait and see”-strategy from the section on partial migrations:

**Strategy 8 (Wait and teach)** *Waiting for the right opportunity to introduce an innovation—for instance in the vicinity of releases—buys valuable time to promote and understand an innovation.*

Urgency

Last, one point can be made on the tactical level: Since Open Source projects are not managed on schedules, they also usually do not experience much urgency (the aforementioned *Google Summer of Code* is a notable exception). Looking at situations in which they do, it seems that (1) on-going development can steam-roll many processes, an important example of which is discussed in more detail in Section 5.9 where radical changes to the software architecture were found repeatedly to be overridden by ongoing development, and (2) the capacity of individuals to keep track of local changes such as patches and tasks is limited, which creates urgency if processes are stalled [geda:4117]. Innovators who act tactically can use insights into both concepts to synchronize the timing of their introductions to project events, thereby avoiding urgency.

To summarize the theoretical associations made to the concept of time in this section: (1) Open Source participants are mostly volunteers and their invested time is often variable. (2) Variable time investment leads to participation sprints and periods of absence, which may overload or respectively underload the



project's capacities for innovation processes. (3) Time is informally managed and frequently lacking precise boundaries. (4) Innovations which rely on schedules are rare and mechanisms such as meeting notes must exist to desynchronize them. (5) Innovators can use time by pursuing different time scales for different introduction phases, align introductions to project events, and attend to progress to avoid urgency.

## 5.9 Radical vs. Evolutionary Innovation

This section explores the question whether an innovator should aim for radical or rather for incremental evolutionary innovations. Typically, these alternatives arise before proposing an innovation and during considering the applicability of an innovation to an Open Source project. The results of this section were published in detail in [380].

### 5.9.1 Reimplementation in Open Source Development

To start the discussion about radicality in Open Source development, we will first leave the area of innovation introduction and turn to software reimplementation and redesign (see Section 4.3 for the relationship between software design and innovation introduction), because several interesting episodes and insights can be gathered there.

The first episode to regard occurred in the project *Bugzilla* in which the maintainer of the project questioned the viability of continuing development using Perl as the programming language [bugzilla:6321]. The maintainer effectively proposed to search for a new programming language and reimplement Bugzilla from scratch. In the large, ensuing discussion, which was heated and in stretches personal [bugzilla:6394], a core developer brought up one striking reason against rewriting: He recounted the story of how Bugzilla had already faced the choice between rewriting and incrementally repairing before, and history had proven the incremental repair to have succeeded, while the rewrite failed [bugzilla:6767].

Similar points about incremental development were raised in two other projects on three other occasions by leaders or senior members of the respective projects: (1) During a discussion about task proposals for participating in the *Google Summer of Code* program (see Section 8.1.2 for a discussion of this program) in the project *ArgoUML*, the maintainer of the project praised agile methods and incremental development and cautioned against giving students any tasks, which would imply replacing existing code rather than improving on it [argouml:4912]. The primary argument against a reimplementation was seen in the high probability of additional work being necessary to polish a potential replacement vs. building upon the known value of the existing solution. (2) In *gEDA* the preference for incremental development was raised twice in the context of setting course for the future development of the project, because it was seen as the only viable way to move forward [geda:3004,3979]. Consequently, the maintainer of the project mandated that all radical changes must be broken down into “a set of controlled small step refactoring stages” [geda:3016].

In *U-Boot* one developer argued that it was in fact natural for the development to become incremental: “u-boot has been around and refined for quite some time now and the changes have become a lot more incremental” rather than “earth-shattering” [uboot:31353]. Based on this argument, he proposed to switch to a date-based version naming scheme, which better expresses continuity in development. This proposition was initially rejected, but eventually accepted in 2008 (cf. Section 8.1.4) when the realization had spread in the project that incremental development indeed was now the standard mode of operation.

Looking into the literature on Open Source development we can find accounts and analyses of several failures to achieve reimplementation:

Østerlie and Jaccheri give the most thorough account of such a failure by describing the trouble the Gentoo distribution underwent when reimplementing their package manager Portage from 2003 to

2006 [392]. After three major attempts within three years to replace the existing system by a more robust, better structured, and faster code base, yet no success on this goal, they find strong evidence that Open Source projects should prefer evolutionary over revolutionary development strategies.

In their paper, the authors provide us with four reasons why the reimplementation failed: (1) Over-indulgence in discussion which drains scarce development resources (compare this to the discussion on the Garbage Can Model in Section 6.2). (2) The failure to provide a prototype version that can serve as a starting point for mobilizing the community or—in the words of Eric S. Raymond—which provides a “plausible promise” that “convince[s] potential co-developers that it can be evolved into something really neat in the foreseeable future” [417]. (3) Competition for resources from other reimplementation efforts and the day-to-day business of dealing with bug-reports by users and of attracting new developers. (4) The inability to balance the need for a stable starting point to achieve a rewrite against the need for progress to manage the day-to-day business.

In their concluding section, Østerlie and Jaccheri condense these reasons down into the following substrate: Reimplementation is “limited by the installed base”, i.e. limited by the existing users and their systems running particular versions and configurations of the distribution, and only possible “through a continuous negotiation with the installed base” about what is possible to say, expect, and do [392]. The failure to reimplement is thus seen primarily as a failure to provide a transition strategy which takes the realities of the existing situation into account.

A similar account is given by Conor MacNeill, one of the maintainers of the project Ant, in his discussion of the history of Ant [325]. In his cases, there were multiple competing proposals for rewriting Ant as a version 2.0, none of which eventually succeeded. He argues that these proposals were typical results of the second-system effect [65], in which the developers became aware of the deficiencies of their initial implementation and asked for too much in terms of new features and architectural capabilities. The project was able eventually to resolve the splintering in different proposals, but even then it did prove impossible for the reimplementation to keep pace with the development of the trunk. Finally, it was accepted that incremental changes were more likely to result in the desired architecture and features [325]. Today, Ant is at version 1.8.0, paying tribute to this realization.

Jørgensen, as a third example, surveyed the development process of the Open Source operating system FreeBSD and found it to be highly incremental and bug-driven, making the development of radical new features difficult [273]. Two reasons were uncovered: (1) Radical changes and reimplementations put the code base of the project in an unusable state, making intermediate releases based on the principle “release early, release often” [417] impossible. This lack of releases has two negative consequences: First, the user’s perception of progress in the project will decrease, making the project seemingly less attractive to users which in turn will demotivate developers. Second, maintenance effort in the project will increase, as bugs need to be fixed in two code bases which are more and more diverting, stretching the limited developer resources. (2) In radical feature development the number of subtle bugs emerging from architectural complexities rises. This makes parallel debugging break down, because the number of developers who consider these bugs to be “shallow” (in Raymond’s terms) is becoming too small [273, p.14].

Similar to this last point, Benkler makes a theoretical argument about the importance of modularity, granularity, and cost of integration as limiting factors in collaborative production [41]. He argues that for Open Source to be successful in attracting participants, the independence of individual components must be maximized and the granularity of tasks for each such module—as well as the cost of integrating them into the product—must be minimized [41]. Radical features and reimplementations as described above hurt granularity and exclude contributors with small scopes for activity [455].

One danger of reliance on evolutionary development is discussed by Weber based on a path dependence argument [538, p.36]: Open Source development has been able to overcome the path-dependent lock-in caused by proprietary software such as Microsoft Windows by providing a cheaper alternative and collecting contributions from users. But, would Open Source projects be able to break their own paths as caused by evolutionary processes based, for instance, on an inferior architecture without hierarchical

control? The projects in the studied sample obviously answer this question with yes by stressing the viability of evolutionary refactoring.

#### 5.9.1.1 A Possible Remedy?

An important question arising from this preference for and better ability to achieve incremental modification is how to accomplish large-scale and architectural changes. In one of the case studies reported in Chapter 7 this question is addressed by Thiel in the context of software security. In an initial assessment of the status quo of protection against SQL injection and cross-site scripting he found that architectures already exist to help developers to avoid security-related defects. To achieve a reimplementation and overcome the resistance to a radical change to the existing security concept and database access architecture, Thiel devised an incremental update path in which architecturally unsound code location would first be annotated using a coding scheme. It was hoped that splitting the architectural changes into manageable work packages would appeal to a group of developers interested in getting the security record improved. Developers could then pick individual annotated locations and fix them. Introducing this innovation for improved architectural change into two Open Source projects failed and gave further insight into the reasons why some types of changes might be difficult in the Open Source development paradigm. Most importantly for this discussion, Thiel found that the open nature of software had made other projects become dependent on certain methods in the source code of the project. Changing the architecture would have removed these methods. However, without a well-defined interface, the breadth of this feature-use was intransparent to the project and led to an effective legacy constraint and structural conservatism to avoid any radical change even if executed incrementally.

A similar legacy constraint could be observed in the above-mentioned discussion in the project Bugzilla on reimplementing in a new programming language. Here, one mailing list participant interjected that companies were selling customized versions of Bugzilla to customers with particular needs. These customized and non-public versions were reliant on incremental updates, which can be merged painlessly. If radical changes would occur, the developers of this software would likely be stuck with an old version of the software (eventually leading to a fork) [bugzilla:6354].

After all these negative examples of failed rewrites, one example should be given where a rewrite was successful. In the example given above from the project *ArgoUML*, the maintainer had cautioned against assigning tasks in the Google Summer of Code to students whose aim it was to rewrite existing code. Yet, this advice was not followed by the core developer who still offered to mentor the task. When a student was found to work on this task [argouml:4987], the outcome predicted by the maintainer occurred: The student was able to rewrite the existing code but failed to get it integrated into the trunk: Some users were still using the old implementation and converters were necessary to overcome this legacy constraint [argouml:5636,7492]. After the Summer of Code ended in August 2007, almost 18 months passed before the new implementation was integrated into a release [argouml:7905,8209].

Why did this reimplementation eventually succeed? First, the core developer who mentored the change put sufficient energy into getting the change integrated [argouml:5686,7911]. He thereby defeated the primary argument of the maintainer who had assumed a lack of resources for it. Second, an incremental update-path was chosen in which the new implementation of the feature was integrated alongside the existing one [argouml:5686] for bleeding edge users [argouml:6254]. Third, there was absolutely no concurrent development on the legacy feature [argouml:5507,4903], leaving the feature unusable in current versions of ArgoUML [argouml:6955]. If parallel development on the existing feature would have occurred, it might have out-paced the reimplementation in a similar way which ultimately caused the reimplementation in Ant and Portage to fail. Fourth and last, the Google Summer of Code paid the student to work concentratedly on the task for three months, thus raising the student's possible task granularity to a level where the reimplementation could be achieved.

## 5.9.2 Radical Innovation Introductions

Considering the link between software design and innovation introduction discussed in Section 4.3, the question arises if similar preferences for incremental change can be found in the context of innovation introductions.

First, though, it is imperative to understand what radicality means in the context of innovation introductions. In the context of software development discussed in the previous paragraphs, radicality was associated with reimplementing and large and complex new features, and therefore linked to the tangible code base. Similarly, radicality in innovation introduction is linked to the development process, which is less accessible and thus makes this discussion more difficult.

*Self  
Introductions  
at Bugzilla*

The first and most obvious defining aspect of a radical innovation introduction is the degree to which it causes changes in the project. Such changes might alter processes in general, or specifically modify social structures, development processes, tool usage, or data managed by the project. For example, consider the innovation of *self introductions* which was established in the project *Bugzilla*. In this innovation, new developers joining the mailing list are asked to introduce themselves by sharing social and occupational facts [bugzilla:6549]. This innovation is radical in the social dimension, because people are asked to reveal hitherto private information about themselves [bugzilla:6554].

Second, the suddenness by which a change is introduced into a project affects the perceived radicality of the introduction. If large changes are spread out over a long time, they are likely to be perceived as incremental and evolutionary.

Third and fourth, the concept of radicality appears to have minor associations to the perceived scope of the innovation (see Section 5.4), in particular to whom it might apply and to the amount of effort it might cause [xfce:13027]. For instance, in the case of the self introductions at *Bugzilla*, the social radicality, criticized for touching the participants' privacy [bugzilla:6554], was limited by making participation voluntary [bugzilla:6549] and giving participants control about the extent to which they want to share "some information about themselves" [bugzilla:6555].

**Definition 13 (Radical Innovation)** *An innovation which has a sudden, widespread, and effort-inducing impact on the way the project collaborates.*

*Bug Tracking  
at GRUB*

A good episode to explore the concept of radicality occurred in the project *GRUB*. The project had long suffered from the lack of a usable bug tracker and project members were discussing remedies (see Section 8.1.5). Five proposals were made which can be ranked by increasing radicality, starting with an entirely non-radical one: (1) Stick to the status quo of not using a bug tracker and keep the existing flow of bug-reports being sent to the maintainer or the mailing list [grub:3934]. (2) The team members could return to using the existing bug tracker [grub:4047], which, at the time of the debate, was filled with bugs for a legacy version of the software product. In this option, the existing bugs would either have to be confirmed for the current version or closed. (3) The project considered altering the existing bug tracking software to distinguish current and legacy bugs, thus avoiding the work to confirm or reject the legacy bugs [grub:3391]. (4) Some members suggested moving to an entirely novel bug tracking system such as Bugzilla [grub:3902], which already included the capability of managing multiple software versions. This option would provide additional features over the existing bug tracker. (5) It was suggested to use an *integrated bug tracker* on top of a new version control system [grub:4082,3934]. Systems in this last category, such as Trac<sup>142</sup>, simplify bug tracking procedures by giving developers the ability to control the bug tracker via commit-messages and enhance the bug tracker using links to version control.

Considering the degrees of radicality of the propositions, the first one is of course without any radicality as it demands to maintain the status quo. From here, radicality increases continuously according to the scope of the proposed solution. The second can be assessed as modifying primarily data of the project, the third changes infrastructure instead, the fourth alters infrastructure and data (as a migration of data

<sup>142</sup><http://trac.edgewall.org/>

is needed into the new system), and the last shakes up both data and infrastructure of the bug tracking data and also data and infrastructure of the source code management system. While all proposals are favored at one point or another during the discussion, it is the least radical proposal (which leaves the status quo) that is ultimately executed by a core developer [grub:4093].

Unfortunately, the discussion in the project *GRUB* does not give much indication why deviations in certain dimensions are perceived as more radical than others. For instance, why is modifying data perceived as less radical than the change to the software? Only the reasons for favoring the existing bug tracker over a new one (with or without SCM integration) [grub:3273] and a rationale for not using the bug tracker at all [grub:3934] are extensively explained by the maintainer. In the first case, the maintainer argues that the burden of maintaining a server over the years is unreasonable in comparison to using an existing hosting platform [grub:3273], an argument which connects radicality to effort. In the second case of using no bug tracker at all, the maintainer argues that bug trackers are unsuitable for discussion in comparison to e-mail and provide no advantage over a wiki in terms of taking note of important tasks to be done [grub:3934]. This rejection is thus not connected to radicality but rather to capability of any solution.

Nevertheless, the following hypothesis suggests itself:

**Hypothesis 9** *Open Source projects prefer innovations that incrementally improve on the existing situation in preference to radical innovations that cause major disruptions.*

Such a preference does not imply that an incremental innovation is automatically likely to succeed. In the project *Bugzilla*, for instance, the maintainer proposed a highly incremental change to the development process in which developers could additionally and voluntarily ask for a *design review* before spending time implementing a solution which might be rejected later due to design issues [bugzilla:6943]. This introduction fails despite being modest in the consequences it would have.

*Design  
Approval at  
Bugzilla*

Next, we inspect the innovation introductions which seemed most radical at first and in which projects switched from a centralized to a decentralized source code management system (cf. Section 8.1.1 for details). The switch of a version control system is radical at first sight because it invalidates the tool set-up for all participants, potentially makes new processes necessary, affects *hosted software*, renders long-honed skills and knowledge partly obsolete, requires changes to existing data, and in the case of distributed version control can even affect the power relationships between project members. The last point needs some expansion: *Distributed version control systems* are potentially disruptive to the power balance in the Open Source workflow because every interested person can obtain an identical copy of the project repository. With this it is possible to supplant the existing flow of contributions coalescing into the project's software by social mechanisms alone. Structural power relationship as known from centralized version control, in which commit and meta-commit rights have been handed out by the project core selectively, can be invalidated and replaced by reputation and trust between all interested parties if decentralized processes are adopted.

How could the four projects which introduced the distributed version control system Git achieve such a radical change? The following reasons could be identified:

- Execution of the data migration, clean-up, and server set-up were done within short periods of time, confronting the projects with a new status quo (in *ROX* execution is completed within one day after the decision, in *gEDA* within sixteen days, and in KVM the project is presented with the finished execution without any prior announcement of intent). This quick execution is possible because it is performed by the project leaders [rox:9371,geda:4322,kvm:1399], who control the hosting resources and can act based on their legitimacy.
- The adoption of the technology, on the one hand, and adaptation of auxiliary processes and tools on the other hand is spread-out considerably over several months. For instance, contributions were still received using the old system [rox:9404], tools were converted only as needed in the context of the next release [rox:9543,9434] and processes were slowly adjusted to support the new realities [uboot:25853].

- Adapters (see Section 5.6) enabled the use of a new technology prior to the migration [geda:2893] as well as the use of a corresponding old technology after the migration [geda:4322]. Spreading out opportunities for all participants to learn and, at the same time, preserve their ability to contribute can reduce much of the radicality associated with a sudden change.
- All projects performed the switch partially (see Section 5.3), starting with central components.
- Not all projects switched to a distributed workflow of pulling changes, but some retained centralized operation in which commit rights are used to access one official repository [geda:4335].

Taken together, all these points reduce the radicality of introducing a distributed version control system sufficiently. What it takes to be successful appears to be mostly decisiveness and strong capabilities in execution. In particular, adapters and partial migrations can sufficiently stretch out the introduction, limit the scope of the migration and make it less radical.

In summary, this section has (1) shown evidence for the preference of incremental development in Open Source projects and (2) hypothesized a similar preference in the context of innovation introduction. It was next argued that (3) the volunteer nature of Open Source participation limits the capabilities of a project to enact radical changes both in design, code, process, and tools. (4) Several possibilities for reducing the radicality of introductions such as adapters were discussed, which could be used by an innovator to overcome the natural preference for incremental solutions.

## 5.10 Tool Independence

As a last and minor result, when looking at all episodes combined, a surprising lack of innovations for the individual developer was discovered. In order to explore the reason for this lack of tools and support for the individual developer, all episodes were categorized using the *type of the innovation* which was proposed therein. Only in five episodes were new innovations proposed that could be marked primarily as tools in contrast to server-side innovations, new processes, or changes to social relationships. Two of these, in the projects *Xfce* and *U-Boot*, only consisted of small scripts being provided to the fellow developers for *removing white space* and *assembling a commit message* for which no subsequent usage incidents could be found. In the first of the three remaining episodes, it is suggested to start using the static code analysis tool *CheckStyle*<sup>143</sup>, a tool which is primarily run by each individual developer prior to check-in to assure consistency with coding guidelines. This suggestion is not adopted by the project. In the last two episodes, one developer of *gEDA* is suggesting the use of the high-level support tools *Cogito* and *StGit* instead of the low-level commands of *distributed version control system Git*. In both cases the reception is lukewarm, particularly for *Cogito*, which later on is even discontinued by its developer [geda:4392].

Certainly, there are several episodes of new source code management systems being adopted by projects (see Section 8.1.1), which as part of the organizational decision to use the new *service* involve the use of new tools by each user [462]. Nevertheless, the number of episodes involving tools is small and it is reasonable to wonder how projects that “want to constantly try to find the best way to do things” [argouml:4788] should ignore such an important aspect of developer productivity. Investigating this question revealed that the freedom to choose tools individually and independently is one of the highly valued norms in the Open Source world. In the words of the maintainer of *ArgoUML* “every developer is free to make his own choices [of tools] as long as it doesn't cause problems for others” [argouml:4788].

How is the conflict between the desire to optimize a project's processes and the desire to maintain tool independence resolved in the studied projects?

1. It appears that tool independence finds its boundaries in the use of open standards. Developers can use whatever tools they want as long as their tools comply to the mandated standards. For instance, the maintainer of *Bugzilla* wanted to improve the ability of the project to communicate

<sup>143</sup><http://checkstyle.sourceforge.net>

via IRC. In the line with the concept of tool independence, he did not improve support for any particular IRC client but rather installed a browser-based *IRC gateway*, which is accessible for all developers who use a regular browser implementing the web-standards HTTP and HTML. In a similar vein, e-mail client choice and configuration is not a topic that is raised to any prominence in any of the projects. Rather, it is assumed that everybody uses a standard-compliant e-mail client and is able to operate it correctly.

2. Tool independence is achieved using commonly and platform-independently available systems. All source code management systems used by the Open Source projects fall under this category of systems for which wide availability is an important argument during decision making [freedos:4824]. By mandating such common systems, the project allows each developer to choose particular implementations or helper tools to be used on-top. Both episodes in the project *gEDA*, in which a core developer advertised the use of StGit and Cogito, belong in this category [geda:4227].
3. If a set of tools is mandated, for instance by the choice of a particular SCM, then the use of adapters (see Section 5.6) provides another viable way for developers to use the tools they prefer with some independence from the project's choice. One developer of *U-Boot*, for instance, preferred using the SCM Mercurial instead of the officially supported Subversion and thus used an adapter to achieve such tool independence [kvm:997].
4. Projects use soft means such as documentation and easier set-up routines to optimize usage for a particular tool and maintain independence of tool choice. For instance, one developer of *ArgoUML* noticed that renaming a file twice in the source code management system using a particular SCM adapter inside the integrated development environment Eclipse would lose the meta-data history [argouml:4824]. Even though the project was stressing tool independence as an important principle [argouml:4852], it nevertheless chose to support Eclipse in this particular configuration. This was done by documenting the issue for future reference and thereby aid the particular set of developers [argouml:4851].

This last strategy for how to balance tool-independence against efficiency deserves some more discussion. On the one hand, it is reasonable when users of a particular tool give advice to others on how to use it [geda:4321], but it also leads to a predicament: If users give advice on a particular tool, or document its usage in the context of the project in detail, this makes the tool more attractive to future developers. Path-dependent adoption is a plausible outcome of such action, leading possibly to problematic lock-in (cf. Section 6.1): Developers might no longer want to change their tool-set-up [grub:4155] because it took them possibly years to have become accustomed to them [geda:5426]. If such a tool becomes obsolete [geda:4392] or superseded by a more powerful alternative, this lock-in of accumulated investment in documentation, set-up, and know-how might cause reduced productivity.

This problem of tool lock-in is particularly pronounced if a client-side tool requires or benefits heavily from project-wide configuration, like in the project *ArgoUML* where one developer created a "Project Set File" to make it easier for users of a particular SCM client to check-out the project locally [argouml:4858]. Even if a project realizes that by bundling such a project-wide configuration with the source code management system it will cause lock-in and maintenance effort to keep such a configuration current, there is no easy escape: If the configuration is not bundled, one still needs to explain to other developers how to configure it to achieve the optimization benefit [argouml:4911]. Build-tools, which are technically client-side, can be seen as an extreme case of such project-wide configuration [xfce:13023]: Due to the amount of effort to maintain them, they are rarely configured by each developer individually. If, on the other hand, projects intend to retain tool independence and thus optimize the usage of popular tool choices, then effort becomes multiplied [argouml:4897].

The consequence of all this is that despite the efforts of the project to maintain a balance between tool-independence and efficiency, it is easy to get caught in these possible negative effects. Most projects thus appear to not promote particular tools officially. Rather, only occasional user-to-user support is occurring.

To summarize this section: (1) Only a small number of innovation introductions are aimed at establishing

tools for the individual developer. (2) Such tool-independence appears to originate from the norms and values of Open Source development, but also from inefficiencies associated with mandating tool choices. (3) To avoid lock-in on certain tools, projects will often assume policies of not officially supporting any tools. (4) An innovator should consider at length whether a plausible case for a tool innovation exists, even if such a tool is platform-independent and widely available.



## Chapter 6

# Comparison with Classic Models

Organizational and social sciences<sup>144</sup> provide a number of theoretical models which help to understand processes such as the adoption of an innovation. In this section, I present four such models in detail with the following goals:

1. To illustrate the application of cognitive models, perspectives, and theories of decision making during innovation introduction and to judge their applicability to real world situation as given by the collected innovation episodes.
2. To stimulate theoretical sensitivity and deduce novel concepts for a theory of innovation introduction.<sup>145</sup>
3. To compare and match the existing concepts in the theory of innovation introduction to those provided by the theories from the organizational sciences, to give the former more validity and embedding into literature.

From a methodological standpoint the first goal necessarily deviates from Grounded Theory Methodology: Using existing external concepts on a corpus of data is highlighted by GTM theorists as one of the pitfalls of GTM as it may easily lead to “importing and imposing packaged images and automatic

<sup>144</sup>I will continue using primarily the term organizational sciences.

<sup>145</sup>If this increased sensitivity added insights to the concepts presented in the chapter on results, these were incorporated there. Novel concepts on the other hand are presented in the current chapter.

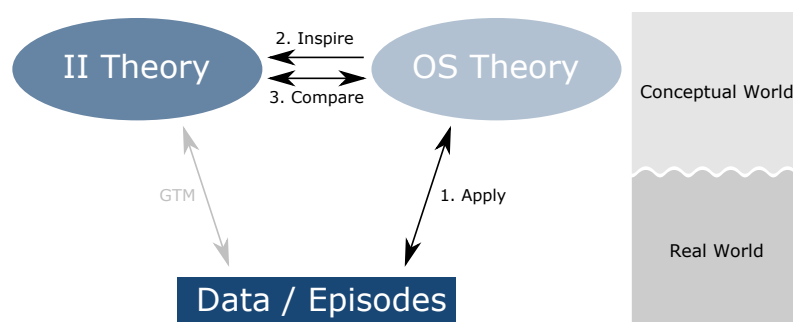


Figure 6.1: A schematic overview of the three uses of theories from the organizational sciences (OS) within this chapter. First, the theories can be applied to episodes as tools for the innovator, in the process of which the data must be lifted into the conceptual world of the respective theory. Second, they can be used as inspiration for building a theory of innovation introduction (II). Third, the concepts of the theory may serve to verify the relevance of existing concepts in the theory of innovation.

answers" [78, p.135][493]. Even worse, the researcher often interprets and stretches data to conform to the imported concepts and ideas, and assigns meaning where the data provided none. Yet, of course there is no excuse for ignoring the literature even in GTM [496]. Rather the prescribed way to proceed following GTM is via the second goal by strengthening theoretical sensitivity and then have the concepts "earn their way" into the theory, a process alike to rediscovering the external theory [105, p.324].

This dissertation proceeded with the first goal (illustrate application) without rediscovery for each of the existing theories because it was deemed more important to take an unobscured view on the application of each theory. Each theory was followed closely enough so that the body of work behind it remains well accessible. If, for instance, the concept of *choice opportunities* from the Garbage Can Model became integrated into the theory of innovation introduction as a special type of *trigger* and the discussion of the concept advanced under this conceptual association, then this might make it harder for the innovator to align his conceptual thinking of the Garbage Can Model to such discussion. Instead, the sections below explicitly separate the methods used for illustrating the use of theory from those gaining new introduction-specific insights. To illustrate the application of an existing theory, (1) the concepts of this theory will be applied as closely as possible to their definition, but (2) still much more interpretative reasoning will be used than appropriate for theorizing using GTM. The second point is necessary to illustrate at all how the theories can be applied in practice. Figure 6.1 shows a conceptual overview of the goals of this chapter.

Each of the following four sections on (1) Path Dependence (Section 6.1), (2) the Garbage Can Model (Section 6.2), (3) Social Network Analysis (Section 6.3), and (4) Actor-Network Theory (Section 6.4) discusses one theory from the organizational sciences and is structured as follows:

- The model or theory is presented based on the original literature.
- The Open Source literature is sampled for references to the model and this work is presented briefly.
- The applicability of the model is verified (in case it is based on presumptions) and operationalized into concepts and questions that can be worked with.
- Multiple episodes are discussed with a focus on the model. Two types of case selection strategy were used: (1) The episodes were chosen at random in the case of Path Dependence and Social Network Analysis, because the applicability of either theory on an episode could not be determined in advanced. Rather than duplicate effort, this was seen as a chance to adopt the viewpoint of an innovator who could equally not chose and foresee whether a particular theory or model would be appropriate for a particular episode. (2) For the Garbage Can Model and Actor-Network Theory episodes could be found which fit these models particularly well. Cherry picking the episodes analyzed has the advantage of maximizing the insights gained from the analysis while possibly suppressing the difficulties and downsides of applying a model or theory on average or badly fitting episodes.
- Insights for the innovator are condensed and summarized to illustrate where the model provides help and guidance.

## 6.1 Path Dependence

Path dependence originated in research on historical economics as a concept for stochastic processes whose resulting distribution is dependent on the processes' history [130]. The term was popularized by the example of the technology diffusion of the QWERTY keyboard layout which has persisted in the market despite allegedly<sup>146</sup> being inferior to other layouts such as Dvorak [129]. How did this happen? Do not the economic theories predict that efficient markets and rational actors should favor a unique

<sup>146</sup>While the case of QWERTY brought the idea of path dependence into the economics literature, the account of the superiority of the Dvorak layout has been disputed [314] and fought over repeatedly [130].

optimal solution? The basic proposition put forward to answer this question is that many processes can only be understood as being path-dependent, i.e. strongly influenced by the sequence of historical events that they undergo. Any attempt to reduce the outcome of such processes to a small set of initial conditions or attributes of the technology is then necessary to fail. As Arthur—one of the original proponents of path dependence—puts it:

“Where we observe the predominance of one technology or one economic outcome over its competitors we should thus be cautious of any exercise that seeks the means by which the winner’s *innate* ‘superiority’ came to be translated into adoption.” (Emphasis added) [8]

To go beyond this “broad” tenet the research on path dependence has first tried to uncover general mechanisms by which processes become path-dependent. How is it possible that such processes—like in the example of QWERTY—result in outcomes which ex-post appear inferior<sup>147</sup> to possible alternative outcomes without a striking reason why those alternatives should not have succeeded?

In the following, a list of possible origins of path dependence is given, albeit with one caveat: The list is strongly dominated by the phenomenon of *increasing returns* as it has been singled out by many researchers [9, 407] as the primary *generalizable* factor causing path dependence and to which all other factors are reduced. But as Mahoney noted in his discussion of path dependence as a perspective for sociology, this utilitarian perspective might “[...] fail to theorize many potential intriguing features of path-dependent sequences, [...]” [329, p.525–526] and thus researchers should strive to look beyond *increasing returns* as the single mechanism for path dependence.<sup>148</sup>

Increasing  
Returns

*Increasing returns* or *self-reinforcement* denote the property of a dynamic processes which causes steps of one direction to induce further steps in the same direction. When considering for instance an adoption process between two competing technologies, then each adopter opting in favor of one technology will typically strengthen this technology by spending money acquiring it. This enables the manufacturer of the technology to invest further into the development or marketing, which should increase its appeal to future adopters. Thus, self-reinforcement can enable one technology to prevail over the other based on the timing and ordering of adoption decisions, which makes this process a path-dependent one.

Self-  
Reinforcement

Arthur notes the following mechanisms from which increasing returns can result [9]: (1) Economies of scale usually improve as output increases and fixed costs can be spread further [9]. (2) Many technological choices show strong positive *network externalities* or *coordination effects* by which the choice of one unit of adoption in favor of a technology increases the utility of the technology for all others [277]. This is best exemplified by adoption of telecommunications technology where each new subscriber makes the network more attractive. Network externalities can also arise indirectly, for instance if the growth in the network makes it more profitable for third parties to offer complementary service such as training. It should be noted that externalities can also have negative effects (in the example of telecommunication networks, additional subscribers can also cause degradation in availability or service quality). (3) The number of adopters of a technology should be positively correlated with the manufacturer’s ability to improve the technology from experience gathered in real-life use [9]. As an example Cowan notes the development of light water reactors in the U.S. after the second world war. The author identifies the use of light water technology in the U.S. Navy submarine propulsion programme and the desire to rush a peaceful use of nuclear energy to market as delivering sufficient head-start to the technology that subsequent learning effects from developing and operating light water reactors were caused the technology to dominate the market [107]. (4) Since agents act under incomplete information, they might extrapolate current prevalence into the future and use current market share as signs of quality [277].

Network  
Externalities

A second condition noted by David in causing path dependence is the “quasi-irreversibility of investments” where, once a decision is made, the associated investment in money, time, or effort cannot be easily recovered [129]. Such “sunk costs” (for instance due to transaction costs or because there is no market

Quasi-  
Irreversibility

<sup>147</sup>Inferior is typically seen from a technical point of view or regarding economic welfare [9, p.112].

<sup>148</sup>Mahoney distinguishes between self-reinforcing sequences of historical events dominated by increasing returns logic and reactive sequences of “temporally ordered and causally connected events” [329].

for second-hand goods) make it often more practicable for adopters to stick to their existing decision until their first choice fails. Accordingly, the durability of the technology becomes an important aspect leading to potential path dependencies [457].

A third condition which can cause path dependence can be found if assets are interrelated with each other and have overlapping service lives such as a railway track and the associated trains [130]. Investors looking to change a highly interrelated technical attribute such as the track gauge (width) or the size of wagons might find that each time a train fails or outdated track needs to be replaced the decision to stick to or change the status quo goes in favor of the existing technology.

Two basic implications arise: (1) Because all effects strengthen the technology which moves first, path-dependent processes are often found to be highly sensitive to initial conditions. (2) Path-dependent processes by the same nature might “lock-in” to a resulting state that cannot be easily escaped by the system itself without “shock” or “external force” [130].

Both properties deserve some cautionary words to prevent misunderstandings of the nature of path-dependent processes. (1) If a path-dependent process is said to be highly sensitive to initial conditions, this does not imply that the initial conditions are “responsible” for the outcome or, in other words, are their cause. Rather, a path-dependent process always attributes the final outcome to the chain-linked steps that enabled a particular path to come into existence and persist over time against all odds and culminate in this particular outcome. This is well expressed by Mahoney’s dictum for researchers on path dependence to “ruthlessly move back in history to uncover a point in time when initial conditions *cannot* predict the outcome” (Emphasis added) [329] cited after [111]. (2) It is easy to overestimate the associated difficulty of breaking free from a path that the term lock-in casually implies. Rather than being a fate that is overly costly to escape, several authors have argued that what looks like an entrenched path, bearing no escape from inside the system, is easily overcome when a path dissolves or is deviated from by time or changes outside the system [499], for instance the entrance of a new technology [550].

With these remarks in mind one should be able to avoid interpretation of path dependence as a historic perspective on decision processes and a passive view on actors which become locked-in by the minor differences in initial conditions of alternatives [111]. Yet, the question how path dependence accommodates the individual actor and gives him room to affect the outcome of a process is still open at this point [206]. Garud and Karnoe have thus proposed to investigate how actors can intentionally affect the creation of paths or break free from given ones instead of just being passive while paths emerge. The authors emphasize in particular the role of the entrepreneur in Schumpeter’s ideal as somebody who can mindfully deviate from existing conditions and create new realities by “creative destruction” [453].

In a case study on the invention of the self-adhesive Post-It Notes, Garud and Karnoe emphasize six activities of such mindful deviation to achieve “path creation”: (1) The entrepreneurs or innovators must detach from their own knowledge and “unbelieve” held assumptions and points of reference to see possible paths and pursue them. (2) Once they have overcome such internal barriers, they need to change the relevance frames of others and overcome their resistance to believing in alternative outcomes. (3) To bring people with diverging interests together into a shared space of understanding and acting, the ability to span boundaries is next important. (4) From here the entrepreneurs must build momentum by gathering champions and drivers of change. (5) Such change then needs to be paced for co-evolution of people and structures for which Garud and Karnoe suggest a flexible approach based on chunking progress and technology into pieces meaningful for participants. (6) Last, the authors highlight the importance of seeing time as a resource in a change process, which can be used by a skilled entrepreneur or innovator.

This list is certainly given on a high level of abstraction<sup>149</sup> and I want to complement it with a concrete episode in my data which exemplifies a couple of the aspects. In this episode, the maintainer of *ROX* migrates part of the project’s existing central source management system to *decentralized Git*. From

<sup>149</sup>For further reading I recommend [452].

his first e-mail [rox:9368] to announcing the migration complete [rox:9371], less than 24 hours elapsed. If we assume the perspective of Garud and Karnoe and reread the first e-mail, we first notice that the maintainer must have spent considerable time prior to writing it, research the issue, leaving existing mental models of centralized version control and understanding the new workflows and possibilities of Git [rox:9368]. In the same e-mail he also spans the boundaries of his position as a maintainer and the developers' as his audience by describing the workflows as they would pertain to developers, stressing their advantages more than his own [rox:9368]. In this episode, the maintainer builds momentum basically on his own by directly starting the migration of one core repository and pulling one enthusiastic developer right with him [rox:9399]. On the other hand, only migrating a part of the repository can be seen as chunking the migration into affordable pieces (compare Section 5.3 on partial migrations). Last, when one developer asks when the rest of the repositories will be migrated [rox:9380], the maintainer uses time to his advantage by proposing to let the initial migration become accepted first [rox:9384]. Thereby, he can let the project evolve to match the structural change he performed by learning how to use Git and incorporating it in their workflows.

Originating from this line of thought comes the applicability to the research in this work.

### 6.1.1 Literature on Path Dependence and Open Source

Most uses of path dependence ideas in the Open Source literature have taken a perspective at a distance and regard the Open Source movement as a whole. Several authors have noted that this movement has been able to challenge the lock-in into which software companies such as Microsoft have led many software markets [538, 56]. For instance, Dalle and Jullien have explored how already small local networks of Linux usage can achieve much of the same benefit from network externalities as participating in the dominating network of Microsoft Windows usage [125].

Using path dependence at the more local level of individual projects, Weber contemplates whether the preference for incremental change might not cause Open Source projects to fall prey to path-dependent dynamics such as becoming locked into inferior architectures [538] (a question discussed in more detail in Section 5.9).

#### 6.1.1.1 Applicability of the Path Dependence Perspective

To apply the path dependence perspective to innovation introductions, the context in which we invoke this perspective needs to be specified. In contrast to most of the existing research, which used path dependence ideas for reasoning about Open Source, we always consider the context of a single project and the adoption decision of an innovation inside this single project. The effects of the adoption decision inside the project for the diffusion of the innovation in a larger context, such as the Open Source movement at large, are not of interest. If, for instance, we would discuss the introduction of a novel system for bug tracking into a project, no consideration will be given to the implications of this adoption to the adoption in other projects.

This is because there is only one episode in which references to wider implications of an introduction are made: In the project *GRUB* the maintainer rejects a *proposal* for the aforementioned novel bug tracking system on the grounds that the existing system in use was Open Source itself and should therefore rather be extended than abandoned [grub:3273]. As the discussion—until then lively—ends with this rejection, a possible interpretation is that participants perceived the cost of extending the innovation as higher than the perceived benefit this single project would derive from it. This is in line with (1) Cowan's analysis of technology choice under uncertainty which concludes that users do not have incentives to select a technology on the notion that their adoption improves it [108, p.811] and (2) David's suggestion that adopters disregard their own impact of adopting on the technology (possibly wrongly so) [130, p.40].

We thus conclude that contributions to the technology due to adoption are rare. Since most technology, ideas, and practices deployed in Open Source projects are also freely available and widely used already (see Section 8.1), we can deduce that increasing returns for the adopted innovation are small. This is not to say that the processes leading to dominance of one innovation used in the Open Source movement over another are not path-dependent, yet that they are of limited use for the case of innovators attempting adoption in a single Open Source project.

Given this context, path dependence as a theoretical model for processes can be applied to two different scopes of interest for a possible innovator: (1) How did the status quo in an Open Source project as a starting point for an innovator arise and more importantly are path-dependent effects at work which need to be broken? (2) How do the actions of the innovator affect the outcome of an introduction episode? To operationalize the application of the path dependence perspective on these two scopes for the innovator, the following five questions are derived from the model of path dependence (the first four can be summarized under the question “did lock-in occur?”):

- Are mechanisms of increasing returns noticeable?
- Are decisions associated with transaction costs or other features which make them hard to reverse?
- Are mechanisms of technical or institutional interrelatedness visible which strengthen certain options?
- Are expectations extrapolated into the future without considering the effects of coordinated action by the project participants?
- Are alternative realities conceivable within the bounds of the project that could have resulted in superior results with little difference in effort?

The following section will discuss these five questions with regard to three innovation episodes and prepare the ground for the discussion of implications for the innovator (see Section 6.1.3).

## 6.1.2 Path Dependence and Innovation Episodes

The following three episodes were chosen at random from the list of introduction episodes found in data for which their *outcome* could be determined. The selection is random because the applicability of a path dependence perspective could not be determined beforehand. Rather it was deemed more important to judge the applicability of the path dependence perspective for any innovator, who will likely not have a choice whether the particular improvement he wants to achieve is affected by a path effect or not.

### 6.1.2.1 Episode Smart Questions at Bugzilla

*Smart  
Questions at  
Bugzilla*

This episode occurred in the project *Bugzilla* when one of the core developers, prompted by a series of low quality questions on the user mailing list, writes a condensed version of Eric S. Raymond’s guide on the question of how to ask smart questions<sup>150</sup>. With the document written he looks for feedback on his guide and asks for inclusion into the process by which users can write support requests [bugzilla:6540]. He receives positive feedback by the maintainer [bugzilla:6544] and the document is integrated into the workflow of posting to the support mailing list without further ado [bugzilla:6543].

**Status Quo** The status quo from which the innovator begins his introduction is not characterized by any effects of technological lock-in or irreversible decisions, because support requests by users are usually independent events which do not impact each other. Inferior quality of such requests can thus

<sup>150</sup><http://www.catb.org/~esr/faqs/smart-questions.html>

be reduced to a problem of individual instances of communication. Neither could feedback cycles, nor instances of interrelatedness be observed.<sup>151</sup>

Indeed, we find that the innovator is not concerned with the reasons why the status quo came into existence at all or what is causing the status quo. Rather, he describes the problem as a subjective impression: “[W]e have recently been getting a large quantity of (what I see as) low quality requests for help” [bugzilla:6540]. Instead, the innovator’s behavior shows how unencumbered he is by the past when he devises his solution: He can write a new document, post it to his personal website, and then announce it to the project, without having discussed an associated proposal and without getting it accepted prior to execution. Li et al. in their study of decision making in OSS projects have called such a sequence of (1) problem identification, (2) execution, and (3) announcement an implicit-evaluation path, because evaluation by the project is not done before announcement [313].

Only if we consider what alternative realities are easily observable, we see that the innovator’s decision to tackle the problem of low-quality support requests by means of condensing Eric S. Raymond’s document and integrating it into the workflow is just one of many possibilities, thus hinting at path dependence effects at work. Rather, the innovator could have rewritten the document from scratch, develop a web-based form into which users have to enter the most often missing information about their requests, start an initiative about improving the FAQ so that the number of requests decreases, etc. Why he chose to improve an existing document over these possibilities remains a question we cannot answer by some attributes of each possibility, but which is probably rather linked to the past steps leading up to the decision of introducing this particular innovation. This line of reasoning shows a potential problem with a path dependence perspective, because going beyond certain events to discover their original reasons is not always possible.

**Adoption** Regarding the introduction effort of the innovator, we also do not see much applicability of path dependence thinking. First, this is because his changes do not cause any feedback effects which perpetuate a certain development. Second, because his solution is so lightweight, requiring just a website update and altering a configuration setting in the mailing list program, any effects of technical interrelatedness and irreversibility must consequently remain minimal as well. If in the future another innovator comes along to improve the support request scenario which would require the removal of the document from the support workflow, we cannot conclude that any lock-in effects would prevent this.

#### 6.1.2.2 Episode Google Summer of Code at ArgoUML

*Google’s Summer of Code program* has been one of the most frequently mentioned innovations during the year 2007 (see Section 8.1.2). The project *ArgoUML* had previously participated in 2006 *Google Summer of Code at ArgoUML* and when the application deadline approached in 2007, the maintainer made a proposal to repeat the success of last year’s participation [argouml:4846]. The project maintainer then unilaterally applied to the program and only after this application had succeeded did lively debates ensue around the possible ideas that students could work on (for instance [argouml:4917,4931,4936]). In the end, the project attracted forty-two applications [argouml:4956]. Six students eventually won scholarships [579] for the three months of the summer term of the American college year. The results of the program were positive and five of the students finished their work [578], but only three of them received developer member status beyond the GSoC [argouml:5513].

The following discussion of the GSoC at ArgoUML will also include some aspects of the Summer of Code episodes in the other studied projects to widen our view (compare Section 8.1.2 on the GSoC in general).

<sup>151</sup>One scenario of increasing returns can arise in user support scenarios if the support team is overwhelmed by requests [394]. This decreases the time the support team has to provide help documents and general advice, which increases the number of requests, creating a vicious circle of decreasing support quality. Alas, this episode does not contain any hint on such a scenario.

**Status Quo** The Google Summer of Code program is unique among the innovations regarded in this thesis, because it involves a large external factor influencing the adoption success: Google decides both whether the project can participate and which student will work in which project. Since this decision is outside of the scope of the Open Source project which applies, this analysis does not focus on the question whether path-dependent effects occur with-in Google when choosing projects to participate (see Section 8.1.2 for a global discussion of the Summer of Code program).

If we consider the costs for introducing the innovation, we see that ArgoUML benefits a lot from increasing returns from having participated before. Convincing the project members to participate, getting accustomed with the application process, and formulating project ideas are almost effortless to the maintainer who thereby ensures a \$3000 stipend for the project in a matter of days (in addition to the \$1500 each student receives per month, the project as a whole gets \$500 per student for mentoring). This is in stark contrast to what happened in the project *Xfce*, which got rejected in 2006. When the application deadline in 2007 approached and a student voiced his interest in working as a GSoC scholar this summer, the maintainer chose not to get involved with applying a second time stating “I’m not sure Xfce would fit in Google’s interested projects.” [xfce:13244]. A path-dependent interpretation of this behavior could be that the maintainer perceives the transaction costs of conducting the application, gather ideas and mentors, as too high when considering the potential rewards of a successful application. Instead of seeing increasing returns by applying year after year, Xfce sees transaction costs, which prevent an attempt being made.

As with the episode in Bugzilla, we do not observe any past decisions or already spent transaction costs causing an innovation trajectory that led the project ArgoUML and its maintainer to keep participating in the GSoC. Rather than having features of a quasi-irreversible decision, the question whether to participate can be made independently of the past. This point cannot be illustrated in this episode because ArgoUML participates, but I found a matching case in the project *GRUB*. Here, the project members almost pass up the chance to participate because of a lack of involvement [grub:2688]. In the end, they only participate because the student from the previous year volunteers to be a mentor [grub:2699].

Similarly, we do not see any technical or institutional factors being in favor of certain options in ArgoUML, but can make out other episodes in which institutional association did matter: *GRUB* could participate with little effort, because their parent foundation—the GNU Project<sup>152</sup>—applied as an umbrella organization. This took care of the administrative hassles of applying, which prevented *Xfce* from participating. The reverse case of an institutional factor preventing an innovation introduction to be attempted can also be illustrated with an episode from the GNU Project. In the first iteration of the Summer of Code project, the GNU Project did not apply because their philosophy forbade them to participate in a program that was perceived as discriminating against projects that use the term Free Software in contrast to Open Source [585]. Thus in 2005, GRUB would have felt strongly deterred to participate because of this philosophical difference.

If we look at the introduction with a focus on little changes which could have had a large impact, we find such a case in a statement of the project maintainer after the student proposals have been received: “[O]ne thing that is not so good has happened. We have had very few suggested proposals and many students have chosen one of them” [argouml:4975]. ArgoUML had 42 applicants for just six project ideas, which could have been changed without much effort by putting a little bit more effort into the creation of suitable project ideas.

**Adoption** The GSoC is unique again when identifying an adoption phase. Since the scholarship program has a defined set of stakeholders (students, mentors, and backup administrators), there is little activity necessary to convince them to use the innovation. In ArgoUML, all mentors stuck to their commitment, the backup administrators never had to be involved, and all students who opted to participate by applying and then won a scholarship did participate. Thus, there was no assimilation

---

<sup>152</sup>[www.gnu.org](http://www.gnu.org)



gap between the organizational adoption decision and individual adoption decision at the beginning of the coding period. Only for one of the six students who won a scholarship [579] did this situation change before the end of the program, because he could not finish due to workload in another job and failed the GSoC program [argouml:5363]. For all the others, the end of the program also meant the end to their special status as paid participants and prompted the question whether they will stay within ArgoUML as contributors.

First, I analyzed the case of the student who failed the GSoC from a path dependence perspective, but could not find convincing evidence that his failure can be attributed to a negative increasing returns scheme. On the contrary, there are initial conditions that already pointed to a possible failure: (1) The task the student received was the least inspiring from the list of ideas [577], targeting only to close as many bugs as possible from the tracker. (2) The student needed to familiarize himself with much of the ArgoUML background literature such as the UML standards and the cookbook before he could begin to work on the project [argouml:5107]. (3) After the summer the student was unable to spend sufficient time on ArgoUML because of a separate job as university faculty [argouml:5363].

In the fourth week the student started to post weekly progress reports, which gives us some insights about his status in the GSoC but also is another indicator of possible problems since all other students have been posting these reports from the first week. Over the next eight weeks, the student posted the reports diligently, and similarly to the other students received little to no feedback or encouragement on the mailing list. Beyond his weekly reports on the other hand, the student does not write a single e-mail. Deducing from the reports, the student made progress but had particular problems with an issue in the second half of the participation period [argouml:5173]. At the end of the summer, the student publicly acknowledged his failure and even then received no public feedback. Since the private communication between the student and his mentor are not accessible for this study, there is no way to verify whether other reasons might be involved in the failure. For instance, it might have been that the student got stuck in a vicious circle of negative returns from his work: Having been assigned overly difficult tasks could have demotivated him, stalling his progress, making his mentor more prone to ask for results, thereby demotivating the student even further, etc. Given that we do not see any intervention from the mentor beyond the suggestion to post progress reports, it appears most appropriate to attribute the failure of the student primarily to initial conditions, such as his skill. Thus, his process would not be path-dependent.

The second case in which I employed a path dependence perspective is based on the question why three of the six students who participated in the GSoC received commit rights, but the other three did not. Unfortunately, it turns out that the best way to explain the division into the group of students who succeeded to attain committer status and those who did not is by the number of e-mails the students had written. The students who succeeded wrote 29, 43, and 43 e-mails respectively, while those who did not, wrote 8, 10 and 19 e-mails over the course of 2007. It is particularly striking that those developers who did not achieve commit rights did not write a single e-mail during the Summer of Code program to the mailing list that was not a progress report (see Figure 6.2). The students who ended up as committers in the project, however, voiced their problems with understanding existing code [argouml:5116,5265], discussed feedback regarding their implementation [argouml:5233], gave advice to others [argouml:5294], ask for help for how to implement certain requirements [argouml:5262], and reported issues found in the code [argouml:5310].

Thus the student's level of activity is the best explanation why commit rights were assigned. Looking beyond the end of the Summer of Code and into 2008 shows that the engagement of the three successful students is much reduced for two of them (even though one is selected for the GSoC again in 2008). This points to the difficulty of attracting long-term members to an Open Source project. We did not find any other negative factors that could explain why the three students who failed to be accepted ended their engagement with ArgoUML right after the Summer of Code was over.

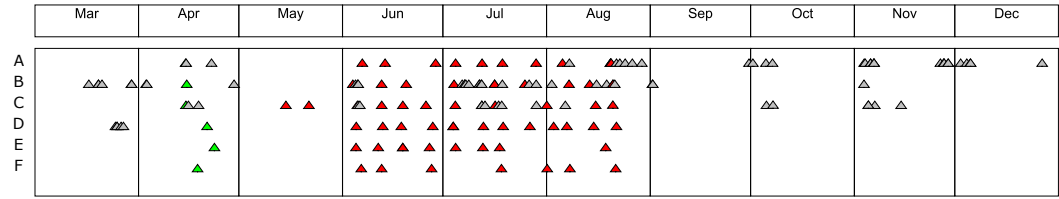


Figure 6.2: E-mails written by students participating in the *Google Summer of Code* in the project *ArgoUML* sorted by number of e-mails written. E-mails marked red denote progress reports written by the students, green ones the student's introduction to the mailing list. It is well visible that the students A, B, and C—who received commit rights—were involved beyond the participation, while the students D, E, and F were not. Participation outside of the Summer of Code was reduced for all student.

6.1.2.3 Episode Personalized Builds at U-Boot

Episode  
Personalized  
Builds at  
U-Boot

During an episode for improving the robustness of the build-file of *U-Boot*, a co-developer and user of the U-Boot system proposed to improve the build-file to accustom private changes [uboot:30760]. In particular, the developer had modified one configuration variable and now experienced merge problems. This proposal was strongly rejected by the project maintainer and a core developer [argouml:30769, 30870]. Both argued that private changes should not exist, but rather that code should always be published for inclusion in U-Boot so that the community could benefit from it.

**Status Quo** If a path dependence perspective is used in this introduction episode, several indicators in favor of a path-dependent situation and several against it can be found. First and foremost in favor is that the status quo can be interpreted to show signs of lock-in: Both the maintainer and the core developer did not criticize the innovator for his innovation, but criticized that he withheld code and was attempting to make withholding code easier using an innovation:

“Just taking a huge amount of code that’s publicly available for free, adding your own little stuff and never returning it to the community is something what happens here and there, but it is definitely NOT the way the free software community works, and in my opinion also pretty unethical.” [uboot:30872]

The maintainer then stresses how important it is that all project users and participants contribute external code into the U-Boot repository no matter, how trivial it is [uboot:30872]. He names a long list of features in the project which started in private code, then became incorporated and evolved into useful features available to others [uboot:30872]. The innovator retorts that adding the private change would bloat U-Boot unnecessarily and that he preferred to keep his code proprietary [uboot:30799].

Richard Stallman has spelled out the difference between these two positions particularly well in his comparison of the LGPL (Lesser GNU Public License) to the GPL (GNU Public License). To choose the LGPL which allows the usage of an unmodified version of the library in proprietary products over the GPL for a software library boils down—in his words—to a “matter of strategy” [477]. Using an *exclusionist* strategy given by the GPL, i.e. prohibiting proprietary use and discouraging external development, can on the one side strengthen the software, and the Open Source movement as a whole by requiring others to reveal their changes. But on the other side, the GPL can also hamper the use of the software by developers of proprietary software because they cannot not use it as a building block as they could with the LGPL. Conversely, the LGPL denotes an *inclusionist* strategy which might miss out on the chance to force the conversion of a software to Open Source but attracts more developers to use it in proprietary software [477].

Despite the harsh rejection by the maintainer in the face of these two alternatives, one could imagine that the whole project could well be inclusionist, in particular, because most project participants work

as paid developers for firms using the U-Boot firmware. Just as the Apache foundation<sup>153</sup> or the Codehaus project<sup>154</sup>, which are strongly connected to supporting proprietary use of their technology by using business-friendly licenses such as the Apache Software License or the BSD license, U-Boot could use an inclusionist stance and argue that everything should be done to make the inclusion of U-Boot in proprietary products even easier and more convenient to attract the largest possible user base from which the U-Boot project could then draw contributions. Yet, U-Boot uses the viral GPL, which contains a legal mechanism to enable proliferation of itself (see Section 5.7 on forcing effects). A strict inclusionist strategy enabling use in proprietary products is thus not possible. Still, the GPL leaves some space for a compromise and, legally speaking, can only force the private derivative work to be made public once software based on it is distributed. From a path dependence perspective both strategic alternatives show features leading to lock-in, because (1) both strategies should attract developers with likeminded attitudes and repel those with opposing ones (increasing returns logic), (2) changes as proposed in the above episode should be executed to make it easier or more difficult to keep changes outside of the project, further strengthening the prevailing strategy (again increasing returns), and (3) a switching of licensing terms is a difficult, yet possible endeavor (quasi-irreversibility).

In the above episode, a sensible middle ground is then pursued by both the project leadership and the innovator: The maintainer argues strongly for the code to be contributed but does not risk alienating the developer by forcing the issue, and the innovator does not insist on the innovation to be deployed. This prevents some of the increasing polarizing we would associate with a path-dependent lock-in.

When looking at the other characteristics of path-dependent processes, we find less support. First, the decision to not provide any support for personalized builds is not associated with quasi-irreversibility on the technical level as it only requires small changes to the build-scripts. For instance, the innovation could be achieved just by including user-definable variables in certain places. Second, technical arguments do not appear anywhere in the discussion, discounting the possibility of technical interrelatedness shaping the discussion.

**Adoption** Since the innovation is rejected in discussion and never executed and adopted, an analysis of the adoption is impossible.

#### 6.1.2.4 Summary

These three randomly chosen episodes demonstrate that path dependence as a perspective on innovation introduction can further our understanding of the forces that exist both at the beginning of an innovation episode and those that come into existence during the adoption of an introduction. Yet, path dependence does not offer a panacea: (1) The episodes start from situations that are not strongly locked-in per se such as the episode of personalized build-scripts which was both technically feasible and associated with only little transaction costs. Yet, the situation turned out to be locked-in once the introduction discussion started, because of strategic alignment by the project leadership. (2) While I feel that it is worthwhile to call attention to the process steps for explaining the episode's outcome, the analysis in *ArgoUML* about attaining commit rights showed that initial conditions such as strong coding skills rather than path-dependent steps can exert strong influences that ex-ante could have predicted some of the outcomes.

### 6.1.3 Implications for the Innovator

To conclude this analysis of path dependence as a strategy for the innovator, three points can be deduced: (1) The narrow conception of path-dependent processes as those that can only be explained by the series of steps they took, requires much investigative legwork with unclear benefits for most

<sup>153</sup>[www.apache.org](http://www.apache.org)

<sup>154</sup>[www.codehaus.org](http://www.codehaus.org)

innovators. (2) Rather, the broad line of thinking that the status quo is a product of the past will be more beneficial for the innovator, because it leads to a concise analysis of the forces which lock-in the project at the present. (3) Using path dependence does not guarantee finding an explanation for all outcomes that were achieved, since many attributes and properties of processes remain hidden in the mist of history.

I thus agree with Garud and Karnoe in their criticism of path dependence as too passive a theory for giving advice to the practitioner and can not recommend it as a perspective on innovation introduction in most cases.

## 6.2 Garbage Can Model

The Garbage Can Model (GCM) provides some explanation for organizational decision making in institutions characterized by organized anarchy, i.e. organizations which have “inconsistent and ill-defined preferences” for making choices, which operate with “unclear technology” that is not thoroughly understood, and which suffer from “fluid participation” where participants have varying amounts of time to commit to solving problems and frequently join and leave the decision processes [94]. Given this setting, the GCM proposes that problems, solutions, participants, and choices—as the basic ingredients of decision processes—should be regarded as highly independent “streams” that often have a life of their own; problems arise out of nowhere, discovered solutions are waiting for suitable problems to come along rather than being designed to solve a specific problem at hand, choice opportunities arise and disappear again, and so on. When ingredients come together (in the figurative Garbage Can), a decision can then be made.

The allure of the model lies in postulating (1) the independence of each stream and thereby escaping classic thinking about rational and orderly participants who in strict order react to problems by finding solutions and then decide on their implementation, and (2) that we can abstract an organization for the purpose of modeling decision processes into a set of Garbage Cans which act as arenas of discourse. The second point implies that a strategic actor might utilize the existence of different Garbage Cans to capture and direct the attention and work of certain actors. An example from Cohen and Olsen's treatment of the American college presidencies suggests that the college leadership might provide Garbage Cans with the explicit goal to attract problems and solutions produced by the organization:

“On a grand scale, discussions of overall organizational objectives or overall organizational long-term plans are classic first-quality cans. They are general enough to accommodate anything. They are socially defined as being important. An activist will push for discussions of grand plans (in part) in order to draw the garbage away from the concrete day-today arenas of his concrete objectives.” [93, p.211]

Importantly, the GCM intends not to be a general theory of decision making in organizations, but rather it “present[s] one way of looking at organizations—ideas assumed to be useful for some purpose and to capture some organizations, activities, and situations more than others” [384, p.191]. Thus, to find the GCM following Cohen et al. to be applicable to an organization is to discover that some of the decisions made by the organization can be best explained as “highly contextual, driven primarily by timing and coincidence” [384, p.193].

### 6.2.1 The Garbage Can in the Open Source Literature

The Garbage Can Model of organizational decision making has found its way into research on Open Source only timidly. In 2003 Sebastian Spaeth published a Ph.D. thesis proposal which uses GCM as a theoretical starting point for a qualitative study on the decision making strategies in Open Source projects [468], but his dissertation in 2005 then used Social Network Analysis (see Section 6.3) and quantitative methods to assess the collaboration in Open Source projects, not mentioning the GCM [469].

Li et al. have studied decision processes in Open Source projects using qualitative methods and have identified a phase model of decision making including the possibility to loop back to earlier phases. They found three major and three minor types of decision making paths to be of relevance [313] (see Section 4.5 for details). The authors identified 29% of the observed discussion episodes as highly non-linear with looping back to prior phases [313] and associated these with the decision processes as described by the Garbage Can Model [242]. Li et al. hypothesize that these complex GCM decision episodes can be attributed to a lack of formal leadership in Open Source projects, yet their analysis does not include a treatment of this specific idea.

### 6.2.2 The Garbage Can and Innovation Episodes

If we want to transfer the GCM to the domain of Open Source projects as closely as possible, we first need to assess whether a project qualifies as an organized anarchy. Fluid participation is most easily accepted as a property of Open Source participation because of the high variability of time spent on Open Source by individuals ranking from ten hours per week to full working loads [211, p.21] and observed phenomena like participation sprints, in which selected participants spend several days working on the project with high intensity and then disappear again for long stretches (see Section 5.8 for an in-depth discussion).

Fluid  
Participation

The usage of unclear technology can similarly be found in many cases. While often one or two participants are versed in using a new technology such as new *source code management systems* like *Git*, there is rarely thorough knowledge about how an innovation works and what implications its usage will have.

Unclear  
Technology

Problematic preferences are the most difficult property of organized anarchies to assess in the context of Open Source projects. The Open Source world has preferences, such as the preference for Open Source tools [171, pp.20f.] or adherence to community norms regarding e-mail conduct on mailing lists [97, 142]. Yet, it is hard to assess them as problematic, and other areas such as project goals are often much less well-defined and rather develop as the project continues as a result of the on-going dialogue between developers and users [447]. The same can be said about decision making processes in which I have found some order such as with voting schemes, but also a lot of chaos.

Problematic  
Preferences

In their criticism of the GCM, Bendor et al. attack the model because of its lack to consider “authority, delegation and control” as basic organizational mechanisms for achieving the goals of the organizations [40, p.173]. I would like to turn this criticism around and use it as a defining moment for an organized anarchy: If an organization is somehow unable to establish structures of authority, delegation, and control but is still able to collaborate on a common goal, it can be characterized as an organized anarchy.

Open Source appears to fit such a model, because authority, delegation, and control are weakened by the volunteer nature of participation where every project member is working on self-selected goals only roughly shaped by a possible project agenda. Delegation is almost absent from Open Source project coordination unless mandated by access structures such as password protection or commercial backgrounds of participants. Control mechanisms also work rather to separate the project inside from the project outside, than keep checks on individual participants [229].

In the next section of using the Garbage Can Model of organizational decision making as a perspective, the following questions will be asked:

- Is the project well described as an organized anarchy, i.e. does it show signs of fluid participation, unclear technology, and ill-defined preferences?
- Are signs of an independence of participants, problems, solutions, and choice opportunities visible in introduction episodes?
- Can Garbage Cans as discussion spaces for low energy discussions be identified?
- Which strategic options are available to the innovator?

One remark on internal validity needs to be made: Since an innovation introduction is defined as the set of messages which can be associated with the introduction of one innovation, a discussion of one or several problems which never culminates into a solution being proposed is not captured as an innovation introduction episode. Thus, the results in the section below are skewed towards episodes which include solutions without problems and does not include any problem-only episodes. This implies that GCM, which includes problems as primary entities, might be more applicable than the following examples illustrate.

This section will start with three episodes which fit the GCM particularly well to discuss the advantages of adopting a perspective that sees actors, solutions, problems, and choice opportunities as highly independent streams before concluding with the implications for the innovator.

### 6.2.2.1 Episode Contribute at Bugzilla

*Contribute at  
Bugzilla*

This episode in the project *Bugzilla* was driven by the maintainer who wanted to *make it easier to contribute* to the project. The maintainer to this end proposed changes to aid recruiting such as *writing a contributor guide* and *putting a request for help into the software Bugzilla visible to administrators* and simplifying the *access to the project IRC channel by a web gateway*.

This episode provides three insights, if we adopt a Garbage Can perspective.

1. The whole episode is driven not by a problem or a need, but rather by a goal and vision for the future: The maintainer wants to increase the number of reviewers who are as active as he was when he was at his most active to 10 over the course of the next 10 months [bugzilla:6190]. Each of his sub-proposals is then framed strictly as a solution, which can be well exemplified with his proposal to adopt a friendly attitude towards new developers: “be nice to them! Don’t be mean to them, please. :-)” [bugzilla:6190]. The intent is to prevent jeopardizing the effect of the proposed measures, but from the viewpoint of the Garbage Can we are left to wonder whether this can qualify as an independence of problem and solution. If we assume that a solution such as *being nice* indicates a corresponding problem—in this case alienation of new developers by harsh words—then a separation becomes difficult. One possible solution to accommodate March and Olsen’s view on solutions being independent from problems is to require a problem instance to have occurred (a case in which a new developer did not start participating because existing members were unfriendly). This would imply distinguishing preemptive, as in this case, and reactive solutions, of which the former are hard to analyze using the GCM.
2. The maintainer introduces his proposal by referring to a discussion about simplifying contribution to Bugzilla at the end of 2005. Thus, the episode has been lying dormant for more than one year and is reactivated without any particular, visible *trigger* such as an acute problem to solve, indicating the independence of choice opportunities. Even more, the episode continues to proceed in bursts without any predictable pattern: After the proposal, which does not involve any notable discussion (see below), the episode continues one month later with an announcement of the availability of *an IRC gateway* for enabling easier access to the IRC channel of the project [bugzilla:6263]. One month later again, a retrospective about the success of a conference booth triggers a discussion about the viability of doing outreach activities for recruiting new contributors [bugzilla:6306]. Another three months later the maintainer overhauls the documentation for contributors and announces his plans for people to introduce themselves on the mailing list when joining it [bugzilla:6549]. None of these sub-episodes depended on any of the others, thus strengthening the case for independence of choice opportunities. Another perspective on this long delay can be that the available time of developers is highly correlated with their ability to engage in innovation activities (discussed in Section 5.8).
3. The maintainer employs a Garbage Can when he separates the e-mail describing the context of his proposal (“A long time ago, some of you may remember...”) and his goal (“My goal is to have 10 active reviewers from a growing pool of contributors”) from the actual proposed ideas for change

by putting them into a bug tracker ticket [bugzilla:6190]. A direct effect of this separation is that the e-mail is vague on solutions and subsequent respondents need to drag ideas from this Garbage Can onto the mailing list [bugzilla:6191,6192,6193]. The maintainer then uses separate e-mails to announce the individual solutions as being *executed*, further separating goals, proposals, and actual implementation of an idea.

### 6.2.2.2 Episode Licensing for Schemas at gEDA

This episode in the project *gEDA* was triggered by an outspoken developer who was confused about the license used for *gEDA* symbols [geda:3108]. Shortly afterwards, a core developer jumped into the discussion and helped clarifying the text used on the web-page by pointing to the license used by the Free Software Foundation (FSF) for handling fonts. Similar to fonts, *gEDA* symbols are included into files created with *gEDA* and the question thus arises whether such inclusion causes any licensing implications. This core developer as the innovator then opens a Garbage Can for “comment/flare” to let the project participants exhaust themselves in discussion [geda:3122] of the proposed wording. The resulting thread of 31 replies by nine project participants reaches a thread depth of 20. The innovator refrains from this discussion, asking only one clarifying question [geda:3123].

*Licensing for  
Schemas at  
gEDA*

This discussion can be regarded as a Garbage Can to attract solutions, problems, and participants outside of the actual decision space using three arguments: (1) The innovator offers a revision of his innovation improved by comments from the Garbage Can after only seven of the 31 replies have been written, thus mostly ignoring the ongoing discussion. (2) The maintainer (arguably the most powerful person in the project) stays away from the whole thread but rather makes his comments on the issue in a new top-level thread to escape the Garbage Can. (3) The discussion shows both signs of unclear technology as represented by the complexity of the legal aspects involved [geda:3170] and unclear preferences as given by the question of how to balance between the freedom of users to redistribute and the desire of the project to harness contributions made in the process [geda:3143].

In the Garbage Can, only one alternative brought up by another core developer receives some discussion. In the end, though, the chosen solution is to stick to the revised version of the Font license exemption by the FSF [geda:3145] as proposed by the innovator.

A similar episode is reported by Barcellini et al. in [23] in a discussion on a Python Extension Proposal (see Section 4.3). There, the champion of a proposal and the project leader retract from the discussion, which then continues for more than 10 turns before the project leader forcibly ends it [23, p.184]. Similarly, in a separate PEP the project leader as the champion engages during the first few days, then retracts and only reappears to close the discussion [24, p.158].

These two examples illustrate the strategic possibilities of using<sup>155</sup> a Garbage Can to channel discussion away from actual decision, particularly in those cases which are heated and controversial. Famously, “bike shed” discussions<sup>156</sup> with low complexity can be particularly heated, prompting the innovator to seek a Garbage Can to contain them. Possible alternatives for opening such Garbage Cans are numerous: First, an innovator can restart threads as in the episode above, but secondly can also move them to bug trackers as in the first episode at Bugzilla. Third, Internet Relay Chat (IRC) allows for fast exchanges to exhaust topics more quickly with less persistence than archived mailing lists. Fourth, additional forums and other mailing lists can be used to contain discussion. The downside of these alternatives is that if discussion is split into several venues, boundary spanning might become necessary to transfer relevant insights and outcomes [22].

<sup>155</sup>To be able to create and use Garbage Cans in such an intentional way in contrast to them being emergent phenomena of organized anarchies certainly requires a lot of skill. I think though that the preceding examples illustrate the possibilities.

<sup>156</sup>A “bike shed discussion” is one in which a detail of a decision to which people can relate such as whether to build a bike shed consumes the attention and energy of the discussion participants in contrast to more important, yet complex matters such as whether to build a nuclear power plant, which get decided with little controversy. Also called “Parkinson’s Law of Triviality” [398, p.24–32].

Unlike “trolling”<sup>157</sup>, where a controversial message is written with no purpose but the personal enjoyment of the author, a Garbage Can is not necessarily something bad or sinister. Rather, keeping separate Garbage Cans or arenas might be beneficial or even necessary for projects to maintain competing interests or agendas. In the gift giving case study (see Section 7.2) and in Thiel’s case on security annotations (see Section 7.5) such Garbage Cans were caused for instance by the maintainer’s desire to keep an inclusionist agenda to motivate participants while at the same time keeping the mailing list as an exclusionist arena where the resolve and commitment of an individual would be tested. An innovator who anticipates and uses the possibilities of Garbage Cans can achieve a clean separation of the decision process and the discussion of concerns, for instance by condensing a sprawling discourse into a newly opened thread (the anti-Garbage Can strategy):

**Strategy 9 (Manage the Garbage Can)** *When an innovation discussion gets out of hand in depth and topics covered, the innovator should refocus it by condensing opinions and offering steps forward in a new thread.*

This strategy can be deduced from both the episodes presented here and by Barcellini et al. [23, 24], but it can also be found as a central activity of the information manager role discussed in Section 7.1.

### 6.2.2.3 Episode Static Code Analyzer at ArgoUML

Java5 at  
ArgoUML

During the migration to *Java 5* in the project *ArgoUML*, one of the peripheral developers was unsure about the status of the migration, because he could not detect any usage of Java 5 language features and thus whether he could start using them. He received permission from the maintainer, but one core developer proposed to restrict usage on one language feature for automatic conversion of primitive types into object types (a language feature called auto-boxing [219]). While this language feature leads to more concise code [argouml:4981], the core developer had reservations because auto-boxing incurs a performance penalty which is hard to notice by a developer [argouml:4967]. Yet, the innovator himself immediately noted the problem of proposing a coding guideline and thereby defining expected behavior: If the use of auto-boxing is hard to detect for the author of code, it will be even harder to notice for a reviewer and thus difficult to enforce. The technical solution which the innovator thus proposes is to use a *static code analyzer* such as Checkstyle<sup>158</sup> to aid in detection [argouml:4967].<sup>159</sup>

Checkstyle at  
ArgoUML

There is only a single reply to this proposal, in which one peripheral developer supports the use of the feature by noting that even though auto-boxing can cause code which is more error prone, he prefers the concise code it enables: “But, I like the feature and tend to prefer less code” [argouml:4981]. The innovator had already indicated in his message that he was not *dedicated* to his proposal and the episode ends abandoned on this comment. The GCM offers two concepts by which such behavior can be conceptualized and which will be discussed in turn: (1) *energy requirements* and (2) *fluid participation*.

1. In the computer simulation of the original treatment on GCM decision making, Cohen et al. do not treat solutions as one of the four basic ingredients of decision processes, but adopt a “simpler set of assumptions” [94, p.3]: Instead of modeling a stream of solutions, they introduce an *energy requirement* necessary to resolve a problem-laden choice situation. Under this model, participants provide their energy for decision making to a particular choice in a discrete simulation step. Once the total sum of energy spent by participants on a choice opportunity exceeds the energy requirement caused by the problems attached to this choice, then a solution is said to have been found.<sup>160</sup> In this sense, the lack of interest of the core developer to pursue the episode

<sup>157</sup>A fishing term for letting a bait trail behind a slowly moving boat to attract a catch; adapted for the Internet to denote the activity of individuals who write deliberately offensive or discussion evoking messages for their own enjoyment [151].

<sup>158</sup><http://checkstyle.sourceforge.net>

<sup>159</sup>From a conceptual standpoint it is interesting that in this episode a software solution does not enforce compliance using a *code is law* mechanism (see Section 5.7), but that the style checker supports a social mechanism which relies on detection.

<sup>160</sup>In the computer simulation the energy supplied to a choice is both additive over all participants who supply it and conserved over time, which probably is only a crude approximation of reality but in line with the goal to keep the simulation simple [384, p.192].



could be conceptualized as a period of low energy. To explore this possibility, Figure 6.3 shows the messages written by both the core developer and developer over time. It is well visible that the density of messages written by the core developer given by the black line is already low when the proposal is being made (marked red bold) and further diminishes in the following week. If the core developer had started the innovation episode independently, this would probably be called a tactical mistake, as insufficient energy is available to further the episode. But since the choice opportunity for the introduction of Checkstyle arose in relation to the episode regarding *the use of Java 5 language features* and the core developer as an innovator could not choose the time for his proposal, this highlights the problems of timing in Open Source projects with fluctuating energy.<sup>161</sup> A nascent implication for the innovator is to ensure that sufficient energy is available for on-going and opportunistic engagement.

**Strategy 10 (No Introduction without Commitment)** *Innovation introductions should only be started when sufficient time and energy are available in the months to follow.*

2. The second concept appropriate for reasoning about the behavior of the innovator is *fluid participation* and captures the “sometimes unpredictable exits and entrances of participants” [310, p.194]. Looking again at Figure 6.3, we can notice that the innovation proposal is the last e-mail of the innovator before a hiatus from the mailing list lasting nineteen days. This marks the second longest absence from the mailing list of the innovator and directly points to a general dilemma: How can it be assessed whether a project participant has left the project or is merely active with low energy? This problem of assessing the presence of a participant [300] has two reasons: (1) The correlation between the occurrence of artifacts of activity such as e-mails or commit messages and the expenditure of activity by their author is variable. If, for instance, a project participant is working over weeks to produce a patch or is following the mailing list passively without writing e-mails, such activity cannot be captured by looking at the artifacts produced by the participants. Conversely, regular e-mails written by a participant can give the impression of activity even though the time spent on coding and reading the e-mails of others might be minimal. (2) Activity and involvement are often managed implicitly, i.e. longer stretches of absence, for instance due to changed job situations or vacations, are seldom explicitly voiced (see also Section 5.10 for a discussion of tool usage which is a similar no-topic in Open Source projects).

Both factors implicate high levels of ambiguity for participants when judging the presence, absence, and level of involvement of others. Following this line of thought, the concept of fluid participation and activation energy needs to be modified in the context of Open Source projects. In companies or universities attendance in meetings or vacation calendars can cause awareness of the fluidity of participation and thereby stipulate the independence of participants from problems, solution, and choices in the GCM (for instance, by assigning substitutes for missing participants). In contrast, the communication mechanisms in Open Source projects reduce the perception of fluid participation, making it harder to assess whether a project participant is still involved in a discussion. This strengthens the ownership of issues by participants and solidifies the association between participants, problems, and solution.

**Hypothesis 10** *The independence of participants from solutions, problems, and choices in Open Source projects due to fluid participation is counteracted by the difficulty to assess the changes in participation.*

This is not to say that a high degree of participant turnover does not occur, but just that it does not have the same implications for decision making. The implication of this hypothesis that lack of transparency strengthens ownership is that Open Source development gains stability of who is working to solve which problem and thus indirectly some structure.

<sup>161</sup>In the computer simulation of the original Garbage Can paper, the authors explore the effects of varying the amount of energy a participant can supply in relationship to the number of choice situations a participant can get involved with. Unfortunately, the results of this variation are not discussed [94, p.7–9].

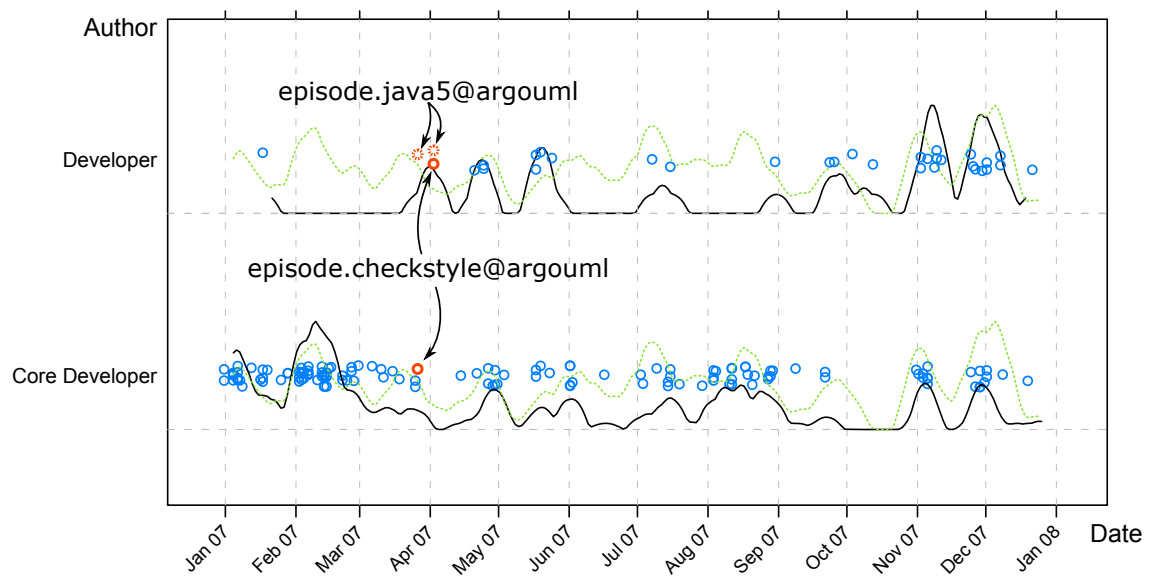


Figure 6.3: Messages written by the core developer and developer plotted over time with a highlight on those messages written in reply to the proposal to use a *static code analyzer* to detect the use of a problematic *Java 5 language* feature. The black density line for each developer was created using an Epanechnikov kernel with a smoothing bandwidth of seven days. For comparison the dotted green line represents the project wide message density using the same smoothing kernel. It is well visible that the episode is located after a strong peak of activity of the core developer in February and at the end of a much less pronounced participation in March. After the core developer wrote the proposal he entered one of his longest periods of hiatus in 2007, lasting two weeks. It is not surprising that the proposal failed in the face of the innovator's absence.

Taking a step back, this episode highlights three ways in which solutions and problems can become related to and independent of each other:

1. **Solutions trigger problems**—The initial confirmatory character of the episode (“is it OK to start using Java 5 features?” [argouml:4965]) triggers the problem-centric discussion about auto-boxing despite the general decision to adopt Java 5 already being made in the past. On the one side this triggering can be seen as a discussion opportunity which gives the core developer a forum to voice his concerns, but it also highlights that a solution in the abstract (“to use Java 5”) can remain too separate from a particular problem (unintended performance penalties due to auto-boxing) until actual execution and adoption of a solution is attempted.
2. **Multiple Streams**—The perception of the problems and benefits associated with auto-boxing diverges between the developer and the core developer. While the core developer puts emphasis on the performance penalty aggravated by the difficulty to detect the use of auto-boxing, the developer uses this difficulty as a reason why auto-boxing causes defects and ignores the performance point. On a micro-scale of analysis this might well be interpreted as a tactical point, but on the macro level of discussing solutions and problems it can point to independent streams about a subject.
3. **Complexity**—The core developer’s proposal to use a static code checker to enforce avoiding of auto-boxing shows how the relationship between problems and solutions can easily reach a high level of complexity within a short discussion. To paraphrase: Because somebody could not detect any features of Java 5, this triggered the question whether Java 5 may be used, which triggered the question of whether to use a certain feature which is hard to detect when being used, which triggers the question of whether to use a tool for enforcing the non-use of such a hard to detect feature.

Taken together, these different ways in which a problem might interface other problems or solutions may easily cause the impression for an external observer (in the GCM) that problems and solutions are chaotically intertwined. Only when analyzing the interaction in detail can the structure between solutions and problems be revealed. More bluntly, when the GCM is being used as a model to explain phenomena, one should be aware that it offers simplistic macro-level explanations without much explanatory value for complex, yet rich micro-level events.

### 6.2.3 Implications for the Innovator

The independence of problems, solutions, and opportunities is pronounced in many innovation introduction episodes, which prompted the comparison to the GCM. Solutions are developed and thought-up often outside the boundaries of the project and imported by participants who are only temporarily active and then disappear again from the project, while problems drift into the discussions similarly often by chance.

What the innovator can learn from the GCM is primarily to keep a flexible mind-set towards the innovation process. Rather than pursuing a strict stage model from problems to solutions and implementation, the innovator can—and should—assume a relaxed attitude towards exploring each aspect with relative independence. Even if a problem is not yet entirely understood, it makes sense to discuss possible solutions or even already start implementing a solution by running a trial. Seizing an opportunity related to getting a solution adopted is additionally stressed by the GCM even to the point that a fierce debate including a wealth of problems does not indicate that the solution could not be adopted by simply skipping all these problems during decision.

One of the central criticisms aimed at the GCM is targeted at the idea of independent streams. The criticism notes that people are the central entities that bind all the other three together, because people have problems, call for meetings to resolve them, design solutions, implement them, push for decisions, thereby bringing all ingredients together and making them dependent on each other [40, p.172]. If we consider participants in the context of Open Source innovation discussion, we might expect that similar to proprietary software development in firms, there should be problems associated with inheriting

tasks, innovation, and goals from former developers such as understanding their intent, scope, and difficulties [310, p.195][284]. Looking at the episodes studied, we find the participants of discussions to change frequently and each member's involvement to be volatile, but the set of innovators and executors in most episodes to be stable. One possible explanation is the lack of delegation, mentioned above, which should bind solutions strongly to individuals who are knowledgeable and motivated to introduce them. Thus, we might expect the independence of solutions and problems from individuals to be diminished in comparison to organizational anarchies with hierarchical power structures such as described by Cohen et al. Since this thesis only looked at one year of mailing list activity in each Open Source project, a conclusive assessment of the effects of changes to the long-term participants such as core members and maintainers can not be made. The best discussion on the issue is in Section 7.4 on the case study of introducing automated regression testing in the project FreeCol. Here it was discovered that signaling is an important aspect when withdrawing from a project as an innovator.

From a practical perspective, the skilled innovators who supported the creation of Garbage Cans for channeling discussion were successful with achieving their goals and it can be recommended to explore these tactical aspects further in future research.

The best, yet abstract conclusion at the moment for the innovator might be that the independence between problems, solutions, participants, and choice opportunities should be carefully considered. They might be both more independent than expected, as seen in the episodes discussed, and less so than expected, as the case at FreeCol illustrated.

## 6.3 Social Network Analysis and Social Network Theory

Social Network Analysis (SNA) is a method for looking at data representing social relationships [282]. The methodology originated in the social sciences at the beginning of the 20th century from "sociometrics" [199] and has studied many types of relationship data [456] such as those of local communities of fishermen in Norway [27], the spread of AIDS through sexual contacts [283], the interlock of enterprises via debt or share holders [34], or collaboration among authors of scientific publications [359], to name a few. By taking a structural perspective on such data, several general effects and principles of social networks were uncovered such as the small-world phenomenon [341, 537, 99], 0-1-2 effect [15], preferential attachment [360], or triadic closure [291].

SNA is founded on graph theory and uses measures and attributes of graphs such as betweenness, diameter, distance, density, betweenness centrality, degree centrality, eigenvector centrality [64] etc. (see [360, 361] for an introduction) to derive statements about the social world represented in the graphs. While mathematics can be a powerful tool to uncover attributes of networks, we are cautioned by Scott to always search for the underlying social phenomenon which gives a number derived from SNA explanatory value [456, pp.112f.].

For a historical overview of SNA refer to [199].

### 6.3.1 SNA in the Open Source Literature

The most relevant studies using Social Network Analysis on Open Source projects are presented below:

- Madey et al. were among the first researchers to perform a study of Open Source projects using SNA [326]. Their study was focused at the large scale and looked at all projects hosted at SourceForge.Net. Developers were linked in their networks if they were participating in the same project [327]. Certainly, this was a crude assessment of network structure as it disregarded all information about communication, activities, or roles of the developers in the project. Since the network of developers for one particular project by definition is also always fully connected, the

analysis can thus only be global. Madey et al. found that the size of the giant component<sup>162</sup> in the graph was 34.6%, while conversely also a third of all projects are isolated [327]. Looking at the network from a qualitative perspective, Madey et al. called developers who participate in more than one project “linch-pin” developers and suggest that they are valuable for the projects as boundary spanners [327]. Running simulations, Madey et al. found that they could match the evolution of the network at SourceForge.Net, if they modeled new developers to show preferential attachment to larger projects and if they used a variable fitness function for new up-start projects with a fast decay to simulate some new projects to become successful [553].

Linch-pin  
Developers

- Crowston and Howison used SNA on the bug tracking communication of all projects from SourceForge.net up until April 2002 with more than 7 listed developers and 100 bugs [113]. In these 120 projects they constructed social networks based on whom people replied to in the comment section of each bug report. In particular, they looked at the out-degree of participants, i.e. the number of people somebody has replied to and computed centrality of a whole project as the average difference between out-degree for each person and the maximum out-degree for the whole graph normalized by the maximum out-degree. Thus, for each project a score between 0.0 and 1.0 (excluding) can be calculated, where a centrality of 0.0 represents a graph in which all people have the same out-degree and a value close to 1.0 a graph with one person, whose out-degree is larger than those of all others. Looking at the resulting distribution of centrality measures for the 120 projects, they find centrality scores to be normal distributed (mean=0.56, sd=0.2, min=0.13, max=0.99). Unfortunately, centrality as calculated in relation to only one person with maximal out-degree is of course highly dependent on this single person. The analysis is furthermore limited by collapsing all activity into a single network and not considering the evolving structure of the network over time.<sup>163</sup> Also, the authors found a correlation to project size [113], in that it is less likely in a large project to have one person who is dominating replies. While Grewal et al. similarly find heterogeneity of network structures [223], the question arises whether such uncovered variance in centrality can really be taken as an indicator for different communication structures in the Open Source world.
- Lopez-Fernandez et al. looked at data from source code management systems in three large Open Source projects (KDE, Gnome, Apache) [321]. In particular, they studied the repositories at the module level (using top-level directories as a proxy for this) and linked two developers if they contributed to the same module at least once. Conversely, they also studied a module network in which two modules were linked if one person contributed to both of them. Unfortunately, their study did not reveal more than the networks being loosely connected and showing small-world characteristics [320, p.45].
- Spaeth performed SNA on data from source code management systems of 29 Open Source projects for his doctoral thesis in 2005 [469]. His work mirrors previous results but looks at medium-sized projects. He found similar results of modifications being caused by a small group of core developers [469, pp.58f.] and that most files were only modified by a small number of developers, pointing to code ownership [469, p.68].
- Similarly, de Souza et al. also looked at data from source code management of five medium to large Open Source projects but did so over long periods of time and using static call-graph analysis to better understand the evolution of authors and their detailed contributions [138]. The authors found shifts in participation from the periphery to the core of a project and vice versa, as well as changes to the ownership of files over time.
- Bird et al. looked at social networks from five large Open Source projects created both by developers who worked together on the same files and via reply-to relationships on the mailing list [52]. They then showed that (1) the communication network was modular, i.e. sub-groups could be identified, (2) developers discussed product-centric topics in a smaller group while other topics were discussed more broadly in the community, and (3) people who interacted on the

<sup>162</sup>The largest connected cluster in a network.

<sup>163</sup>This problem was stated by one of the authors in a private communication.

mailing list were also much more likely to work together on the same files in the repository. On a methodological side Bird et al. found that identifying software modules to repeat the product-centric analysis on a more coarsely-grained level is difficult because software concerns often cross-cut through modules and skew the results [52, p.32].

- Ducheneaut conducted the only ego-centric SNA study on Open Source that I am aware of. In this study, the socialization of a new developer in the Python project called Fred is studied using SNA among other things [153]. Ducheneaut uses a hybrid network of mailing list communication and work on files in the repository to illustrate how Fred is able to move into the center of the network over time. Ducheneaut then traces back this move into the center to Fred's first question on the mailing list, fixing bugs, gaining commit access, contribution to discussions about Python extension proposals, and becoming respected in the project in the end. By combining both such an ethnographic perspective and SNA, Ducheneaut can provide a rich description of how Fred became naturalized in the project Python. He then generalized this results to joining as both a learning process and a political process for instance by explaining identity construction, join scripts, and the process of probing the black-box that is the project [153, pp.349ff.].

**Summary SNA** To summarize, social network analysis has been performed on data from Open Source projects such as project association [326, 327, 223], association to implementation space artifacts such as files or modules [321, 138, 469, 52], communication relationships on mailing lists [52] or bug trackers [113]. I have not found any studies which used surveys or interviews to construct social networks as suggested in [110]. Most studies presented a socio-centric view on one or more projects rather the ego-centric view on a single developer within such a network. Results indicate that the networks exhibit small-world characteristics and Pareto distributions of contributions [51, p.138].

**Criticism of SNA** On the downside, SNA can easily be criticized for abstracting reality too highly [43, Sec. 1.3.2]. For instance, reply-to relationships were often used to model an association between participants without giving much thought to the fact that the content of an e-mail or quotation patterns [25] might be more significant for judging the association, or that association via communication also decays over time [291]. Guy et al., for instance, found that using e-mail communication intensity for modeling weighted social networks often misrepresented the “the hidden business links, old connections being kept on ‘low fire’, or friends from work with whom no everyday tasks are being performed” [224, p.401].

### 6.3.2 Social Network Analysis and Innovation Episodes

**How were networks constructed?** To explore whether SNA could be used as a new perspective on innovation introduction, two episodes from the projects *Bugzilla* and *ArgoUML* were selected at random from episodes which involved both core and peripheral developers, and their respective social network was constructed from the reply-to relationships on the mailing list in 2007 of each project. The hope was that choosing these episodes would maximize the potential for seeing interesting effects in the networks. Each network was exported from GmanDA (see Section 3.4) as an undirected, weighted graph and rendered using the GraphViz package [165] with the following visual aspects:

1. Each vertex represents one mailing list participant and was labeled accordingly.<sup>164</sup>
2. The importance of an individual mailing list participant is approximated by drawing the corresponding vertex as a circle with an area proportional to the total number of e-mails written by this participant. This proxy is rough and easily disturbed by “bike shed discussions” [190, pp.135f.] and “noisy minorities” [190, pp.138f.], but it is justified by the reliance on e-mail for assessing individuals—paraphrased by Fogel as “you are what you write” [190, p.122].
3. The weight of an edge is proportional to the number of e-mails replied-to between two mailing list participants, giving a sense of the strength of interaction between the two participants.

<sup>164</sup>I took care to unify vertices for participants who wrote to the mailing list using multiple names and e-mail addresses, but only inspected the resulting figures as needed for this analysis. Additional heuristics as proposed by [51, pp.138f.] should be used if exact results are needed.

4. The core developers of the project were drawn inside a central cluster, whose members were calculated to have been active in more than two thirds of all months with at least one e-mail. This cut-off is rather arbitrary and could be improved with a precise assessment of the core members of a project [120, cf.], but the resulting sizes of the respective cores are comparable with related work [120, 287, 347]. I also experimented with an approach suggested by Bird et al. in [52] to use a community finding algorithm [409, 397] as described in [362] and [90] to find sub-groups based on the network (see Figure 6.6). The results are less convincing, because the identified clusters appear to include too many miscategorizations.
5. Each project participant was colored to represent the number of months the participant had been active in the project.

The results can be seen for two projects in Figure 6.4 in large and in Figure 6.5 for nine of the remaining projects. These are too small to read in print, but can be enlarged easily in the digital version.<sup>165</sup> The Figures for the projects *KVM* and *U-Boot* were too complex for GraphViz to handle.

All figures show a similar structure of (1) a tightly integrated core, (2) a loosely collected set of co-developers which are strongly oriented to the core but share some links between each other, and (3) a periphery of project participants which are only connected towards the project core. In contrast to the onion model of influence and role advancement (see Section 2.3.8) the communication social network is more appropriately named “sun, moon, and stars”, with the project core being the sun around which most communication revolves, the co-developers forming a crescent-like set of “moon” developers around the core, and the peripheral participants are only dotted as stars around the core.

Such a model supports the notion of a qualitative difference between the long-term, stable participation in the project core and the fluid participation of co- and peripheral developers [300]. A first quantitative analysis confirming this subjective impression as significant was published in [378] and found that co- and peripheral developers discuss less among each other than their share of communication would lead us to expect. Further research would be necessary to explore the implications of this result and to strengthen validity.

For related work on visualizing social networks consider in general [245, 32, 198] and in particular the projects Tesseract [446] and GraphMania [382], which formulate a vision to improve coordination and cooperation in a project based on visualization of data from the development process. It should be noted that the visualization of e-mails in contrast to constructing social networks based on them is a separate field of research [556, 523, 525, 279, 524].

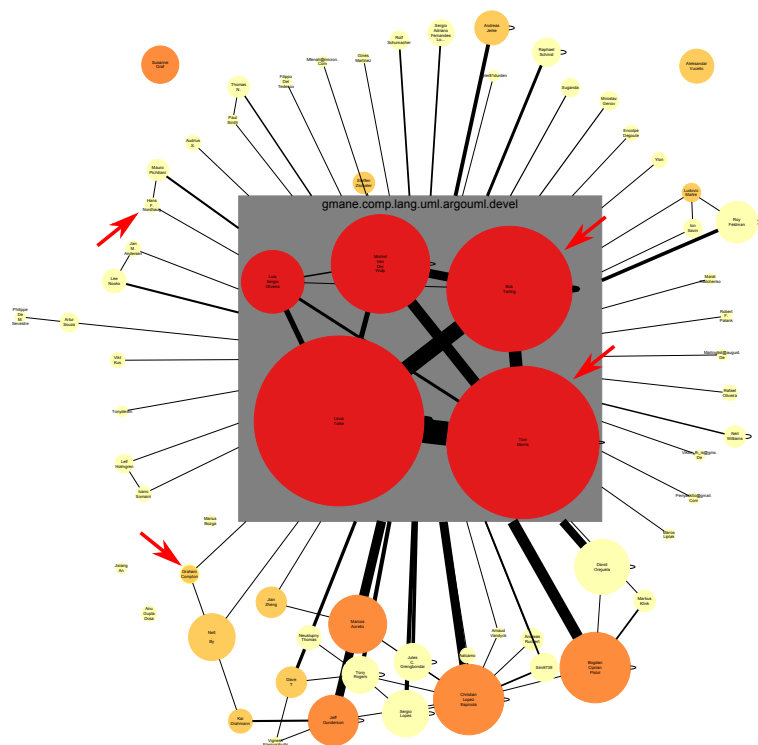
### 6.3.2.1 Episode Packaging at ArgoUML

This episode in the project *ArgoUML* involves two core members and two peripheral members who have been highlighted with arrows in Figure 6.4a. The episode is triggered by an episode in which the least frequent participant of the four on the list proposed to *improve the state of translations* of *ArgoUML*, but the project found that the versions of ArgoUML included in popular distributions were outdated [argouml:4696] and thus any improvement of the translations would not directly benefit users. Therefore, one of the core members of ArgoUML opens a new discussion on how to improve the state of packages within distributions and proposes to discourage inclusion of unstable packages [argouml:4697]. The other core developer suggests that the *stable/unstable release numbering scheme* of ArgoUML might be part of the reason why unstable releases get distributed at all and proposes to switch to a *milestone-based numbering scheme* [argouml:4698]. The second peripheral participant brings up arguments against both proposals, and the episode ends in a rejected state [argouml:4701].

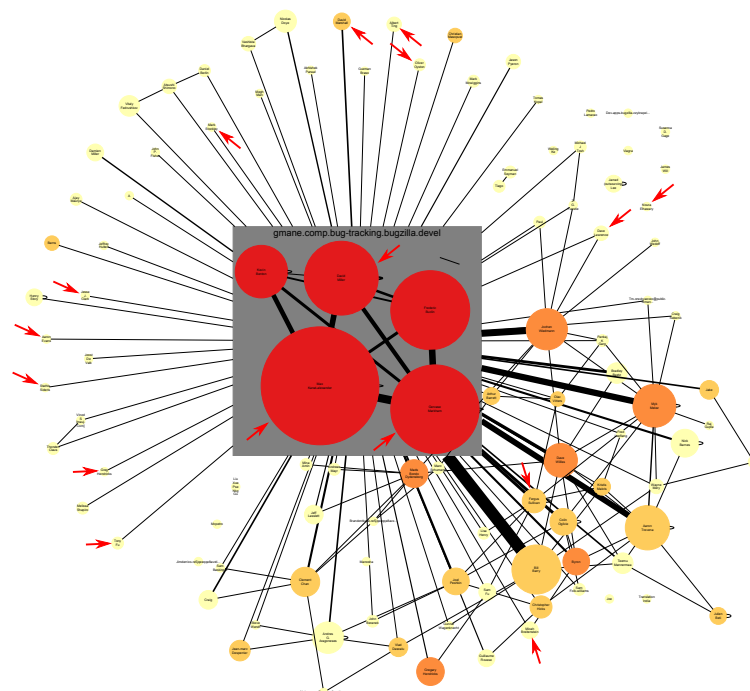
Packaging at  
ArgoUML

Looking at the social network in this situation, we can see that while the second peripheral developer was an insignificant participant according to the social network, he was able to argue against two of the

<sup>165</sup>This dissertation is available digitally from <https://www.inf.fu-berlin.de/w/SE.OSSInnovation>.



(a) Project ArgoUML



(b) Project Bugzilla

Figure 6.4: Social network graphs of e-mail communication in the year 2007. Size of vertices and width of edges are proportional to the number of e-mails. Color was assigned based on the percentage of months a person was active in the project: Darker colors indicate longer activity. Participants highlighted with a red arrow are involved in the episodes discussed in the text.



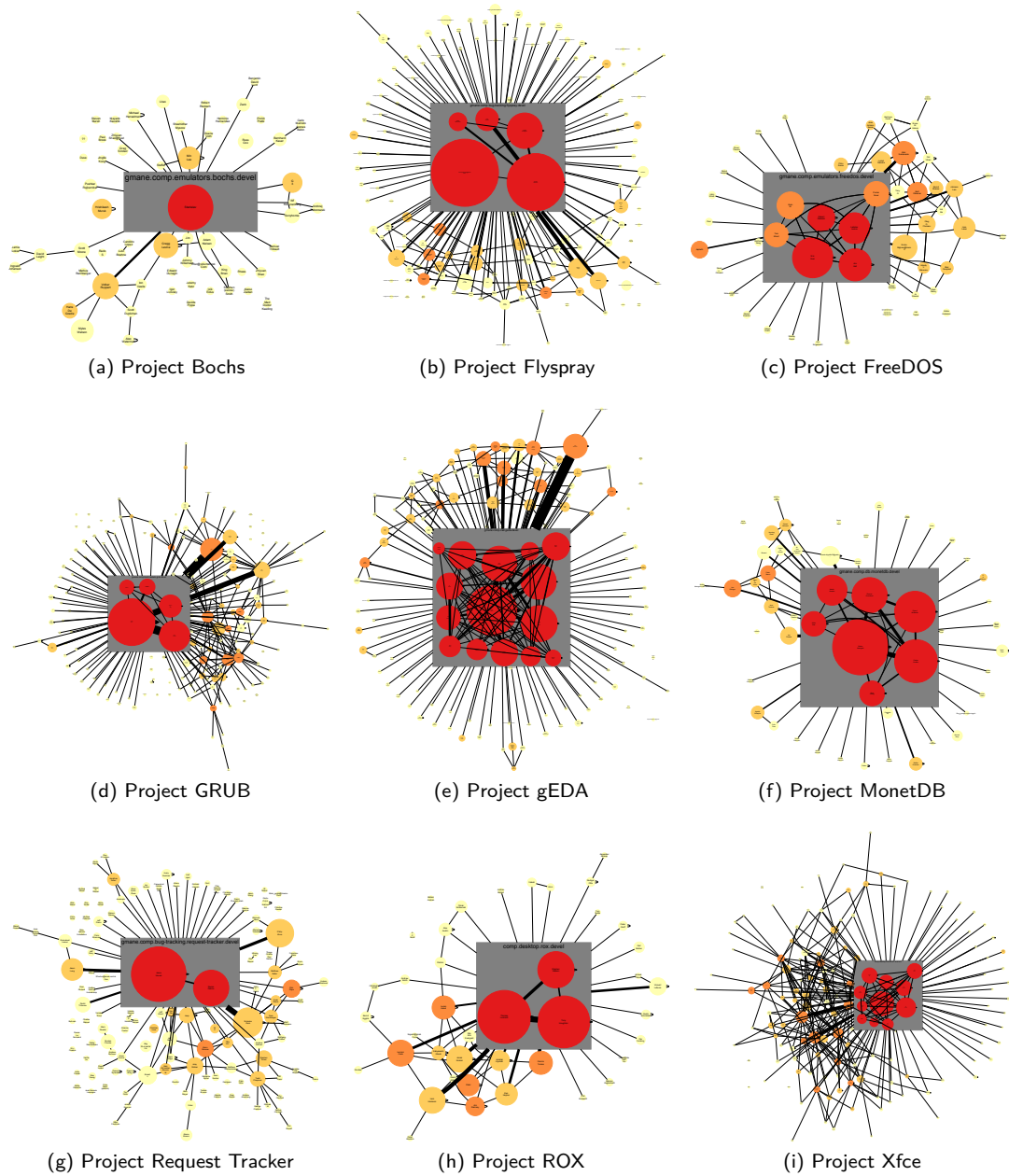


Figure 6.5: Social network graphs of e-mail communication in the year 2007. Size of vertices and width of edges are proportional to the number of e-mails. Color was assigned based on the percentage of months a person was active in the project (darker colors imply longer activity)

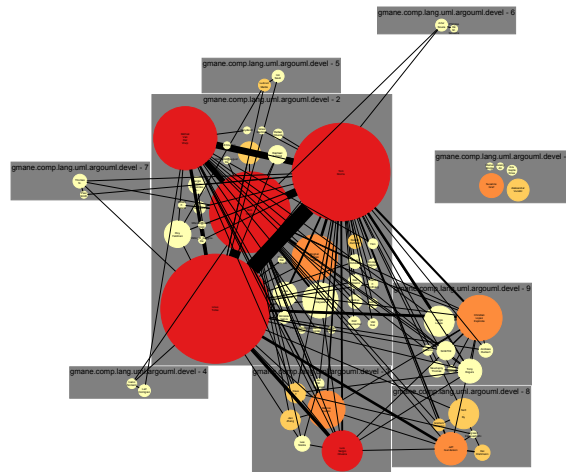


Figure 6.6: Social network of the communication in the project ArgoUML in the year 2007 with communities identified using the algorithm by Clauset et al. based on graph modularity [90].

most central core developers in the project. This cautions us to deduce the importance of participants in individual episodes from a generalized social model.

### 6.3.2.2 Episode Self Introductions at Bugzilla

*Self  
Introductions  
at Bugzilla*

As a second example of using a social network an episode from the project *Bugzilla* was chosen, which was triggered by the maintainer of the project when he proposed that *everybody should introduce themselves* when joining the project to strengthen the social fabric of the project [bugzilla:6549]. Over the next couple of months, there are 16 such introductions by the participants highlighted in Figure 6.4b. The social network graph shows that only three of those are from the five core developers in the project and that only one participant of the remaining 13 was able to build a set of social ties beyond the project core. All other mailing list participants who introduced themselves did not show much activity beside the introduction. Thus, we need to ask how well suited self introductions are for community building. Certainly, there are instances in which surprising and interesting information is revealed during introductions such as common interests in making music [bugzilla:6559] or specifics about the largest Bugzilla installation known to the project [bugzilla:6607,6612]. While there is not enough data to discard the innovation and only little reason to believe there is any harm in suggesting self introduction to mailing list participants interested in becoming project members [bugzilla:6554], the social network view has revealed a mixed record so far.

### 6.3.3 Summary and Recommendations for the Innovator

With the current scientific knowledge about Open Source development it is challenging to use social network analysis on data from a particular project to derive insights about a research topic such as innovation introductions. While SNA can provide a good overview of the relative importance of individual participants on a mailing list and reveal their relationships to other developers, this information is abstract and might miss crucial aspects of the social relationships existing between project participants. For the researcher, it would thus be important to explore in which ways the social network changes if two project members are associated by participating in the same discussion or when quoting rather than reply-to relationships are being used [25, cf.]. Also, formal project roles such as maintainer or committer should be modeled and additional data sources added such as the source code management systems [52], or bug trackers [113].

For an innovator a graphical overview of the social network of a project might be the best way to get a quick initial impression of the social power structure. The central players can easily be identified as strongly connected and continuously active. Peripheral figures can be equally assessed via their relationships to co-developers. But as with all tools which an innovator can use, the cost-benefit ratio must closely be regarded. For instance, a global assessment of the relative importance of individual members can also be deduced from the number of messages a person has sent to the mailing list with little tooling except a search engine. Following the work by Ducheneaut [153], the innovator can also track his own embedding into the social network of the project.

## 6.4 Actor-Network Theory

Actor-Network Theory (ANT) [303] can be seen as an extension of the Social Network approach based on the following three ideas: (1) non-human artifacts should be treated just as any other human actor in the network, (2) what looks like a single actor is often only a simplification and punctualization<sup>166</sup> of a large aggregate of actors, and (3) to uphold the network and its effects, actors need to actively maintain it (an act which ANT calls “translation”).

It is in particular the principle of the heterogeneous network which assigns to artifacts the possibility to be actors that can provide a stimulating fresh perspective on data. ANT does so because it realizes the importance of objects in mediating and translating the action of intentional human actors. So while a password-protected server is not acting based on free will or a set of internal goals, it can play an important role when innovations are being managed on this server. As another example consider a lengthy and complex manual page given to project members when adopting a new tool. ANT—by assigning actorship to the manual—can recognize that this manual is the source of work and hassle for participants attempting to learn to use a novel tool, but that it can also serve as a mediating and multiplying factor when used well by the innovator to speed up adoption.

Heterogeneous  
Network

Simplification and punctualization are principles that come as no surprise to the computer scientist who is used to think about generalization and, in particular, abstracting from the inner details to a simplified interface. The most typical example where in the course of the research we use punctualization to simplify our thinking about the world is when we simplify “the Open Source project” as a single actor, despite knowing that the project is the ill-defined and ever-changing sum of participants, the project infrastructure, the versions of source code produced and packaged by it, etc.

Simplification  
and Punctual-  
ization

As a last important term translation is used to explain how actors can engage in the network. ANT keeps this notion intentionally broad to include all actions which generate “ordering effects such as devices, agents, institutions, or organizations” [303]. An example for a powerful translation can be given by the act of using an Open Source license for source code [137]. By choosing such a license, the network potentially changes in multiple ways, suddenly it allows for co-developers, user bug reports, forks, maintainers, the whole wealth of the Open Source world. Also, translation often makes heavy use of punctualization: Rather than forcing every participant to fully understand the legal implications of Open Source licensing, the translation still works if users only understand Open Source as software which is free to download [511].

Translation

### 6.4.1 ANT in the Open Source Literature

Three relevant uses of Actor-Network Theory have been found in the Open Source literature:

1. Thomas Østerlie uses ANT in a study on distributed control in the Open Source distribution Gentoo [391]. In line with the theoretical discussion in [229] he observes that hierarchy cannot be used to impose control over others. In particular, a project leader cannot command by his

<sup>166</sup>For instance, to talk about “an organization” as an actor in a story is a punctualization of those hundreds of employees, their rules, bosses, and computer systems who make up this organization.

position on top of the hierarchy that project participants should behave in certain ways. Østerlie discovers that the framing [167] of problems as shared rather acts as a central mechanism of control. If a problem is framed as shared, then other participants can be led to feel responsibility for working on it. Control arises from the ability to frame problems appropriately. At this point in the argument Østerlie invokes ANT to explain how the framing of problems is the product of many heterogeneous actors including non-human artifacts. This can be exemplified by the arrival of a bug report in an Open Source project. Just by sitting there in the bug tracker it can cause developers to feel responsible to deal with it. It exerts control even though no classical means of control are involved.

As a second implication of invoking ANT to understand the distributed nature of control in Open Source projects, Østerlie notes that control by his definition cannot be unidirectional but always is reciprocal by the means of the shared problem space.

2. De Paoli et al. use a case study on two Open Source projects to discuss the actor nature of Open Source licenses [137]. The authors first analyze Sun's CDDL and the GNU GPL from an ANT perspective as licenses used in the projects they studied. Both licenses, they noticed, define boundaries when used in a project by including or excluding certain source code under other licenses and restricting the possibilities of joining any particular software eco-system. Also, the patent provisions in CDDL are shown to have actor abilities in creating a software eco-system protected from patent trolls.

Looking then at e-mail discussions in the projects related to license use, they find several attempts to change the licensing regime. From an ANT perspective such an attempt can be interpreted as an exercise to mobilize the network to change the existing translation regime. In the project GRASS, which produces an Open Source geographical information system, the proposed change was from the existing use of the GPL to the less strict LGPL to enable proprietary applications to be built on top of GRASS. In the case of Open Solaris, an operating system produced by Sun, the proposed change was from CDDL to GPL to increase compatibility with the large GNU GPL eco-system of software. From an ANT perspective both changes are portrayed by De Paoli et al. as having to undergo Callon's *moments of translation*, namely (1) problematization, (2) interessement, (3) enrollment, and (4) mobilization [73, cf.]. In the studied cases, though, the spokesmen of the change fail to achieve the translation, sometimes even not raising enough interest for a discussion to last more than two days.

Problematiza-  
tion,  
Interessement  
and  
Mobilization

3. Lanzara and Morner have studied the Apache project and Linux kernel project to understand how Open Source projects organize themselves [300]. They start their discussion by presenting both evolutionary theory [2, 74] and Actor-Network theory as theoretical starting points for their inquiry but in the rest of their article focus on the concepts of variation, selection, and stabilization from the evolutionary theory. They regard source code, version repositories, mailing lists, and source licenses as artifacts and systems from which the Open Source development paradigm originates when using evolutionary principles as guiding mechanisms. While closing the gap to ANT should not have been difficult, they use ANT primarily to conclude on the web of technology into which the individual Open Source participant and his contributions are enrolled as source for the emergent coordination in the projects [300].

### 6.4.2 Actor-Network Theory and Innovation Episodes

In the previous section on Social Network Analysis it has already been discussed how the network between participants could be used to understand innovation introduction or aid the innovator. This section on ANT will focus more on (1) the role of artifacts in the network, (2) phenomena of punctualization, and (3) noticeable occurrences of translation.

On the first aspect of artifacts as actors in the network, the reader is also asked to refer back to the discussion on forcing effects in Section 5.7, which has explored the role of artifacts on individual

innovation adoption and prior to this discussion of ANT ideas.

Again, we look at two episodes which were specifically chosen to test the applicability of ANT thinking.

#### 6.4.2.1 Episode Online Demo System at Flyspray

This episode in the project *Flyspray* is triggered when one of the maintainers receives little feedback on a feature proposal of his [flyspray:5395]. Exasperated about the silence from the mailing list the maintainer as a last sentence in his e-mail suggests that he might provide an *online demo system of the developer version* to reduce the hurdle of using a “bleeding-edge” [528] system. In the ensuing discussion the reasons for this hurdle become visible: (1) Some users are having a hard time to distinguish the different versions of Flyspray in use [flyspray:5399,5406]. A request to test a feature in the developer version fails for simply this reason. (2) Keeping an up-to-date developer version, which should not be deployed for production use [flyspray:4923], is especially hard for users accustomed to using package managers [flyspray:4531,5409], because they have to start using version management, understand which branch to work on [flyspray:5330], move to a newer version of PHP [flyspray:5845], etc. Within a couple of days after these reasons have been voiced one of the maintainers sets up a demo system running the developer version straight from the source code management system [flyspray:5414]. Over the remaining seven months of the year this online demo system is used in at least 17 different threads to discuss features, configurations, and bugs, making it a successful innovation to elicit more feedback from users.

*Online Demo  
System at  
Flyspray*

From an ANT perspective, we can draw three conclusions:

- The existence of a developer set-up is—in the terms of ANT—an obligatory passage point for an actor who wants to participate in the process of providing feedback. By identifying this as a weak point in the network which asks too much of many users, the project leadership can create the online demo system as an alternative connection in the network by which actors can generate feedback.
- Understanding software versioning in Open Source projects is particularly hard, because the open access to source code management and availability of patches, snapshots, and frequent releases cause different versions to float into the interaction space of project participants [171]. From an ANT perspective versioning is best understood as a punctualization. Various files in particular revisions, often changing as time progresses, come to constitute a software version. These punctualizations often involve tacit and implicit information which not all participants are aware of, as the difficulties of the users to understand and identify the different available versions of Flyspray illustrate. Providing the users with stable identifiers such as a URL to the developer version being run on a demo system is a strategy for hiding the complexity possibly associated with an extensional term such as “bleeding-edge developer version”.
- Analyzing the introduction process from ANT and the way that the innovator established a translation of the existing network results in four observations: First, several e-mails can be found preceding the proposal, in which the problem of having access to a developer version is lingering [flyspray:4531,4611,4617], thus setting the stage for the problematization of the lack of feedback in [flyspray:5395]. Second, by putting up the proposal to set up a demo online system, the maintainer in the role of the innovator can be seen as testing for interest by the project’s community. Third, this interest is provided by only a single co-developer in three e-mails over a single day [flyspray:5399,5401,5405]. Two other peripheral mailing list participants are present in the discussion but only as a reaction to the problematization and not in reply to the innovation proposal. Thus, a single person is able to represent the project community with several well-made e-mails (another instance of punctualization) and let the maintainer agree to set up the demo system. Fourth, we do not see any formal enrollment in this episode. Rather once the system is *announced*, it is instantly used by mailing list participants who test the developer snapshot and report issues and give feedback [flyspray:5423,5426,5431,5434,5438,5441]. Mobilization without enrollment highlights the independence of each participant on an Open Source mailing list.

### 6.4.2.2 Episode Branch for Patches at ArgoUML

*Branch for  
Patches at  
ArgoUML*

This episode in the project *ArgoUML* involves only a short discussion by the maintainer and two core developers on the question whether branches in a *source code management system* are suitable for collaborating on patches [argouml:4772]. As discussed before, both core developers *reject* the given proposal by (1) a series of arguments such as inefficient operations in their development environment [argouml:4773,4784] or the lack of an established tradition of using branches for experimental work [argouml:4784], and by (2) enlarging the *enactment scope* of the proposal: The maintainer had proposed the use of branches primarily in the given situation, but one of the developers brings up the implication of using branches in general as an effective argument against the proposition.

Using ANT as a perspective, we can first note that the innovator failed right during problematizing the use of patches when he could not stir interest in using branches in this specific situation. Rather, both core developers were able to enumerate the problems with the suggested alternative and thereby stopped the innovator's attempt in the first stage. Interpreting the enactment scope used by the innovator, we could argue that this constitutes a use of punctualization to hide complexity and scope from the proposal made. In this episode though, we could say that the core developers reveal the network behind the punctualization by talking about "developers" in general [argouml:4784] and thus counteract the innovator.

Looking beyond this spat between maintainer and core developers in which the core developers prevented the innovator's proposal to use branches because they perceived it as inefficient for their particular situation, the episode gets another twist. When the project participates in the *Google Summer of Code* (as discussed from a Path Dependence perspective in Section 6.1.2.2), the maintainer revives the idea by asking students to work in separate branches [argouml:4976]. This time enrollment and mobilization of branch use happens without any noticeable discussion or resistance. Certainly, these can be attributed to the maintainer of a project stipulating rules for incoming "newbie" students, but from an ANT perspective the notion of an obligatory passage point—the source code management as a gate to participation—is equally plausible an interpretation.

Once the students have been mobilized, the adoption of the innovation spreads quickly through the network. The aforementioned core developers as mentors of several of the students are quickly involved in the use of branches for evaluating and transferring the students' work results. The experience thus gained with using branches as a result of being embedded in a network of innovation use then leads one of them even to explicitly suggest to a new developer to use branches for the development of a new feature [argouml:5681].

### 6.4.3 Summary

The two episodes discussed have offered a glimpse at the possibilities of using ANT to interpret existing episodes. In particular, the vocabulary provided via terms such as punctualization, mobilization, or intersement has been helpful to highlight points for the interpretation of these episodes. But the hope that ANT would provide entirely novel insights must be declared a false one. Similar to the other theories, I found the concepts offered to be elegant, yet nondescript. For instance, how can we explain that the same innovation becomes an obligatory passage point in one situation, when it did not even catch intersement two months before? ANT does not lend itself to answers to such questions but rather only enumerates concepts of interest which can occur in the interaction of actors. For the innovator I suggest the following take-away insights:

- Do not underestimate the power of non-human actors such as servers with access control or scripts running in daily cron-tabs when introducing an innovation.
- Considering the implications of an innovation especially on obligatory passage points in general or forcing effects more specifically can point to adoption pathways which can be pursued with better chances of success.

## 6.5 Self-Organization and Structuration

The social sciences provide an array of additional theories which could be studied in the same manner for their ability to help a researcher studying innovation episodes and an innovator conducting them. This thesis discussed four such theories and models above and will only present four other interesting ones in passing. More could be named such as complex adaptive systems [12, 356], dissipative structure [364], or stigmergy [254] to pursue in future work.

Self-organization and structuration are presented together because both are concerned with the high-level question how social systems come into existence and maintain themselves over time. The central idea in both is the concept of structuration as the set of those properties of a system that are involved in its continued existence and reproduction [212]. The ability of a social system to be involved in the creation of such structurational properties, thus being involved in its own creation and maintenance in a self-reflexive way, is then called autopoiesis (self-creation) [322]. From such a reflexive perspective it is also possible to resolve the relationships of social structures as determining interaction and conversely interaction as establishing structure as a recursive duality [460] of which Giddens gives three essential ones: (1) communication interaction is linked to structures of signification via interpretation of language, (2) power interaction is linked to structures of dominance via facilitation of resources, and (3) sanction interaction is linked to structures of legitimation via norms of morality [212].

The work by Heckman et al. is one of the rare articles in the area of Open Source to have adopted a structuration perspective. It explores the role of leadership in Open Source projects [243]. Several propositions are offered about how signification, domination, and legitimation are affected by daily activities, are fed back to establish the day-to-day routines, and influence the efficiency of Open Source projects.

For the researcher and innovator the ideas of self-organization and structuration primarily should provide the insight to have balanced perspectives on agency and structure as both important aspects to understanding innovation introduction. For instance, understanding leadership in an Open Source project from a perspective of structure that puts a designated maintainer on top of a hierarchy using control of servers to maintain power will miss the large amount of interaction the maintainer conducts with other project participants to establish his position as a leader by providing technical, managerial, and collaborative directions. The leadership change in the project *GRUB* conversely illustrates the resilience of structure under dwindling interaction: Here, the project maintainer was reducing his own communication with other project members without stepping down from his position as project leader, which led to introductions performed unilaterally by him without discussion [grub:3380] or him blocking innovation introductions after the other project members had already decided to adopt them [grub:4116]. As a result of his reduced communication, one of the core developers became the most active contributor and communicator on the project. As the existing structure of leadership prevailed though, only in August 2009 did this core developer gain maintainer status by receiving privileges to assign commit rights.

## 6.6 Heterarchical Organizations

A heterarchy is an organizational structure in which multiple hierarchies are interdependently connected without a clear predominance of any of them [482, 549]. Some authors speak of nested hierarchies [260], but I think it is better to consider as defining (1) the existence of multiple clusters or centers, (2) strategic responsibilities of each part towards the whole and (3) relative independence for each part, (4) several dimensions along which to understand leadership<sup>167</sup>, and (5) integration based on normative rather than coercive control [244]. The heterarchy as an organizational structure originated in research

<sup>167</sup>For instance, a heterarchy might use a top-down hierarchical organizational chart but have a much different assignment of additional leadership dimensions based on technological skill or social network embeddedness of its members.

on the topology of nervous networks [337] and then has been primarily developed in the context of high-level management structure [244, 482].

In the context of Open Source only the work of Iannacci and Mitleton-Kelly makes explicit use of the idea when investigating leadership in the Linux kernel project [260]. The authors argue that while patches flow in a hierarchical fashion to the top represented by Linus Torvalds, the developers interact in locally existing decision contexts which might have different dimensions of hierarchy [260]. Two such dimensions of relevance for Open Source might be given by the amount of contribution to relationship-oriented and task-oriented communication [243].

## 6.7 Communities of Practice

A community of practice (CoP) is a group of people who interact with each other to learn about a shared craft [541]. The concept originated in the learning theory research of Lave and Wenger when studying the learning of apprentices, which did not involve just the master but also the peers of the apprentice or older journeymen [301]. In such a setting where people focus on a shared domain in which they work practically, a community can be an effective way to share insights and help each other based on the shared identity deriving from the work.

In the context of Open Source, this craft is assumed to be computer programming around which a community could arise [555]. For instance, Ye and Kishida argue that the open collaborative model and role progression from periphery to core allows people to participate legitimately and learn in the process [555]. Elliot and Scacchi on the other hand describe how they preferred the term of an occupational community in contrast to a community of practice, because the community members work together on a software project rather than being loosely associated by their passion for programming [161]. A researcher looking to explore Open Source development in general or innovation introduction in particular should thus be cautious whether a CoP is really an appropriate conceptualization and conclusions can be transferred.

## 6.8 Summary

In this section I connected the results of this thesis to four models, theories, and approaches from the organizational and social sciences with varying outcomes. On the positive side, each model and theory has been found to be applicable in understanding the events which occurred. For instance, for the Garbage Can Model, Open Source projects were analyzed to match well with the notion of organized anarchies. When applying each theory to a small set of episodes, several of the concepts from the theories could be used to explain the phenomena occurring in these episodes. For instance, the concept of an arena from the GCM was found in several episodes to well describe the way the discussion was split by knowledgeable innovators. On the negative side though, the initial hope of using the theories and models as a new perspective which would generate further insights into the data was not fulfilled. Certainly, the theories provide a way to think about the episodes which can often be beneficial, yet no new core category or main result was uncovered. In [410] I proposed that this is due to a mismatch between the specific, concrete findings in this thesis based on the grounding on data and the general, sweeping perspective provided by the theories.

For example, consider the results from Section 5.7 on forcing effects, which explain how power structures in Open Source projects surrounding innovation usage are created and mediated using technological and social means, such as commit-rights and gate keepers. Looking at all theories studied in this section we only find the concept of an obligatory passage point in ANT to discuss similar issues. Yet, this concept is so vague that we can fit all forcing effects under its umbrella without gaining any new conceptual insight to understand their differences, origins, and implications.



Also, the danger to fall for preconceived ideas from the theories could be felt strongly, because the theories themselves construct a space of terms and meanings which can easily occupy the vocabulary of the researcher and prevent a new conceptual world to arise from GTM. I would thus argue that the researcher should be aware of the fundamental positions behind each theory, such as the relevance of inanimate objects to relay a “translation” in Actor-Network theory, but at the same time avoid using the theories as building blocks for their own theories or risk ending up with “hollow” results that merely mirror existing ideas.

This chapter has shown that the concepts uncovered in this thesis go beyond what existing theories can offer, in wealth of phenomena covered, applicability to observed episodes, and usefulness for an innovator. It therefore strengthens this thesis and confirms the use of Grounded Theory Methodology as a more effective way of studying innovation introductions in comparison to the theory-driven approach used in the preceding sections.



## Chapter 7

# Case Studies

Over the course of my research, five innovation introductions were conducted in cooperation with students at the Freie Universität Berlin. While these studies cannot compete with the depth of insights generated by the Grounded Theory Methodology discussed in Chapter 5, they nevertheless can enrich our knowledge about innovation introduction. The cases achieve this in particular by being conducted from the perspective of an innovator, following his steps through the introduction process. There they touch aspects which are difficult to observe by mailing list analysis, such as how to design a novel innovation, whom to contact with an innovation idea (Section 7.3), or how much effort to invest prior to contacting (Section 7.2). Historically, these studies precede the use of GTM for this thesis and the problems with them eventually led to the use of a passive qualitative method. Details of this development in methodology are given in Chapter 3 and were published in [410].

If not otherwise noted, these studies were designed by me and conducted by students under my supervision.

### 7.1 Information Management in Open Source Projects

This first exploratory case study conducted by Robert Schuster and myself in 2004 marks the beginning of my innovation introduction research. As such it has markedly contributed to opening this area of research. Schuster approached me with the idea of helping the project GNU Classpath<sup>168</sup> he was already involved in. What occurred to him as part of his participation was that the project had a striking need for dealing with the wealth of information being created as part of the daily software development. Not only did more and more information accumulate as the project grew older, but also because the project had grown in size and cultural diversity, had its management overhead notably increased. Yet, OSS projects in general and GNU Classpath in particular seemed ill-prepared to handle the burden of recurring questions about how to join the project, of keeping roadmap information up to date, and of codifying policy about coding standards and patch formats. There are two main reasons: (1) All studies regarding the motivation of OSS participants have shown that it is in particular the joy of learning, programming, and sharing skills [211, 238, 252] which motivate. “Arduous” tasks such as documenting, summarizing, and collecting information are far removed from such “Just for Fun” [508] activities. (2) The primary tools for communication in the Open Source world—mailing lists and *IRC* [554]—are well-suited for efficient information exchange but rather ill-prepared for information

---

<sup>168</sup>GNU Classpath was founded in 1998 with the aim to write a Free Software/Open Source version of the Java class libraries which are needed to run software written in Java. This need was prompted by Sun Microsystems’ (now past) policy to keep their implementation free of charge but under a restrictive license. This implied that software written in Java could not be run on an entirely free system; a situation Richard Stallman has called “the Java trap” [479]. As GNU Classpath matured progressively, the attitude of Sun slowly changed and a release of the Java class libraries under an Open Source license took place in May 2007 [379].

management. Particularly, for IRC with its resemblance to oral discourse in which messages often scroll off-screen within seconds [427, p.509] any single piece of relevant information quickly drowns in a sea of chatter. But also three drawbacks exist with regard to managing information for mailing lists: (1) threaded discussion delocalizes information, thus making it hard for readers to locate and extract relevant information, (2) few mechanisms for summarization of discourse are available [423, 253], and (3) since all content is archived without possibility of modification, conflicting messages may arise if information changes over time [379]. For a more detailed introduction to knowledge management practices in Open Source see [441].

#### Information Manager Role

As a possible remedy for this identified weakness of the Open Source development model, we designed a light-weight role-based process improvement named the “information manager”. We chose a role-based improvement because this would allow us to define a set of tasks which help to resolve the information problems and then connect these tasks to relevant goals and benefits. By doing so and assigning a name to the role, defining clear boundaries and noting its benefits for the information manager himself, we hoped that the role would become interesting, well-defined, and rewarding enough to be taken up by one or several project members.

The task associated with this role at a high level is to collect and aggregate project-relevant information, which can be broken down into (1) identifying recurring and unanswered questions and supporting the project in resolving them, (2) summarizing and publishing the outcome of discussions or supporting them being concluded, and (3) maintaining a knowledge repository. As goals of the information manager we highlighted (1) lowering of entry barriers for new developers, (2) improving overview of current project status, (3) enhancing communication, and (4) supporting the project in performing information management itself. Later we found that such tasks had been proposed by Dietze based on a theoretical argument surrounding his descriptive process model of Open Source development (see Section 4.1) [147]. Thus, our work can be seen as putting these theoretical ideas into practice, defining the resulting role and substantiating the motivations that could drive a person assuming the role.

During the design of the information manager we tried to keep the process improvement as light-weight as possible and to adhere to Open Source community culture and norms [487, p.294][486, p.3] as much as possible. In practice this entails that (1) the information manager role was described in a short guideline document<sup>169</sup> without any formal notation<sup>170</sup>, (2) the information manager was designed to rely solely on voluntary participation, and (3) the improvement was described independently of any particular software or tool (see Section 5.10).

We then introduced this information management innovation with the project GNU Classpath. To this end, Schuster drafted a proposal e-mail describing the innovation, its implication for the project, and his motive as a student and researcher [454, pp.33f.]. In particular, he noted the restricted time frame of his involvement due to the length of his thesis and that he was respecting the project's privacy, but that he would keep notes of the publicly archived communication unless project participants opted out. He then first discussed the proposal with the maintainer of GNU Classpath. Upon receiving positive feedback from him, Schuster sent his e-mail to the mailing list. The ensuing discussion went in favor of introducing the innovation, with two core project members and the maintainer favoring the innovation, but it also raised interesting points: One core member wondered for instance whether the information manager should be somebody who is not working on code and could therefore concentrate “full-time”<sup>171</sup> on information management. Another one remarked that he was in favor of the innovation because it would help with introducing beginners to the development, but also stressed that information management should not restrict experienced developers in their work.

Thus sensitized, the *execution of preparatory steps* for conducting innovation management in the project GNU Classpath began. In particular the question arose which technical infrastructure should be used as

<sup>169</sup>See [454, p.35–40] or online at <https://www.inf.fu-berlin.de/w/SE/ThesisFOSSIMMediationManual>.

<sup>170</sup>See for instance [147] for process improvements for Open Source development drawing heavily on UML diagram notations.

<sup>171</sup>“Full-time” in the context of participation in an Open Source project implies that all of the disposable time for participation is used for a particular task, not that a full workweek will be invested.

a knowledge repository to host the documents to be created and maintained as part of the innovation. A wiki system was chosen rather than maintaining documents in the revision control system of the project, because the wiki's ease of use for editing, built-in hyperlinking to create structure, and open access paradigm are a natural fit to support all tasks of the information manager and mesh with OSS community norms.

After the wiki was *hosted*, Schuster created a basic structure for conducting information management, collected a couple of information items from the mailing list, and *announced* the availability of the wiki to the list. Over the next three months Schuster assumed the role of the information manager, filled the wiki with relevant information, encouraged to participate in information management, and frequently referred project members to existing information in the wiki.

### 7.1.1 Assessing the Information Manager

Up to this point, we were primarily motivated (1) by assisting GNU Classpath to be a more “successful” Open Source project (see Section 2.3.10 for definitions of success in the Open Source world) and (2) to create a well designed process improvement.

On both counts we think we were successful. First, we conducted an exit survey among the project members of GNU Classpath which resulted in eleven replies and showed a positive impression of the information management work. For instance, eight developers strongly agreed that new developers would benefit from the work done by the information manager, five developers agreed strongly and five weakly that a wiki had been a good choice for managing information. We also learned that there was already a good understanding of what the information manager does (nine agree), that three project participants had added new content to the wiki and five had updated already existing content in the wiki. Second, from performing and introducing innovation management into the project we learned much about fine-tuning the role. (1) We found that a wiki—as any new place to keep information—has a tendency to cause information to be duplicated and that preventing this should be an explicit goal of the information manager. We could often resolve such issues in favor of the wiki, as it was agreed in the project to move the duplicated information exclusively to the wiki. In all other cases the wiki proved flexible enough via hyperlinking to reference the information instead of duplicating it. (2) Wikis turned out to be not only viable knowledge repositories, but by their easy editing features to be able to support discussions by subsequent edits. Yet, for the information manager the disadvantages from a knowledge management perspective are clear. If people open more arenas for discourse, then the communication and the transparency of discussion via the mailing list in the project will be reduced. To counteract this tendency it was agreed in the project to discourage discussions in the wiki (just as discussion on the bug tracker is frequently frowned upon [190, p.76]). This decision can also be supported by the relative weakness of wikis for discussion [330] compared to more structured forums or mailing lists, which would lead to inefficient conduct. Purely wiki-based projects such as Wikipedia had to create explicit discussion pages attached to each wiki page to facilitate such discourse [403, 495], others had to establish guidelines for how to discuss an issue in *thread mode* in contrast to presenting information in *document mode* [292]. Technical solutions have also been proposed to solve this problem such as qwikWeb which turns mailing list discussions into wiki pages to gain the benefits of both mediums [169]. Yet, for GNU Classpath the easier solution appeared to lie in establishing a social norm of conducting discussion on the mailing list. (3) For managing content in the wiki, Schuster had created several metadata schemes. These would define for a certain type of information which kinds of data should be collected by the information manager. For instance, when capturing past decisions from the mailing list, the date of decision, links to relevant e-mails, and affected parts in the source code were included in such a metadata scheme. Yet, it turned out that in most cases this structure was too rigid to accommodate the diversity of actual content to be managed. The metadata schemes were thus reduced considerably. (4) Not all information needs to be managed. We observed that in some cases information would intentionally be omitted from the wiki, for instance when the learning effects for new developers were deemed too important to hand them all the details on a silver platter. (5) An active stance for the information manager is beneficial. We noticed that it is useful for the information

manager to explicitly ask the project members for relevant information to be noted in the wiki from time to time instead of just collecting information from the mailing list.

Yet, two open research questions remained. (1) Can the information manager be transferred to other projects or is the combination of Schuster as an innovator and GNU Classpath as a project a unique success? (2) How would the information management role evolve beyond Schuster's engagement as a dedicated innovator? What kind of innovation was actually managed and was it worth to do so?

To answer these questions, we first sent a mass marketing e-mail to 76 Open Source projects proposing to them to perform information management. Then we performed a 22 months long-term observational study of the role of information management at GNU Classpath. These will be presented in turn.

### 7.1.2 Mass Marketing the Information Manager

To improve on external validity of our results, we took the idea of information management as packaged in the information manager guide<sup>172</sup> and asked 76 Open Source projects whether they would find this interesting for their work and could imagine adopting the role. We chose medium-sized projects using the *project hoster* SourceForge.net which were (1) at least one year old and (2) in self-declared state alpha or beta, (3) had made at least one major release within the last two years and (4) at least three project members with commit access to the repository.

Response rate to our e-mail was low at nine reactions to 76 e-mails sent. Eight e-mails were lost because 20 of the 76 addressed mailing lists were moderated and their moderator did not let our message pass. We received six positive and three negative reactions to our idea. Of the positive answers, we received (a) three general signs of interest for the role, praise for the idea and minor suggestions for the manual, (b) one response by a project maintainer who praised the idea, but noted that with him being the only permanent participant it was hard to find somebody to pick up the role, (c) one response which noted that the project already had two people filling this role in their project, and (d) one project who had tried to conduct information management but had failed because of lack of time. Of the negative replies, one did devalue the use of information management but praised wikis in general, while the remaining two complained that our message was spam.

Despite the fact that we do not know whether some of the projects might have picked up on the idea of information management, we conclude that our goal to demonstrate external validity could not be achieved and consider four reasons likely: (1) The *communication channel* of an anonymous mass mailing is not a suitable way to introduce innovations such as information manager, (2) achieving introduction success without *personal involvement* of a dedicated innovator and in such an indirect manner is unlikely, (3) our *guide document* about the information manager is not communicating the ideas well enough to have them become picked up, or (4) the general *idea of information management* is flawed. Naturally, for our research we hoped to primarily discount the last reason and so conducted another study later on using three more personal contact strategies which succeeded in two of six cases (see Section 7.3). We also explored whether and how much personal involvement and engagement are necessary (see the next Section 7.2).

### 7.1.3 Using Information Management

Next, I wanted to answer the second question of what would happen beyond Schuster's engagement as information manager in the project GNU Classpath, how the reality of the information manager would pan out over a longer time and what can be learned about innovation introduction from this observation beyond the adoption period. To answer these questions I waited from the initial announcement of the information manager in January 2005 until September 2006 and downloaded all messages sent over the mailing list from the beginning of its archival up to then.<sup>173</sup> I extracted all e-mails related to our

<sup>172</sup>See [454, p.35–40] or online at <https://www.inf.fu-berlin.de/w/SE/ThesisFOSSIMMediationManual>.

<sup>173</sup>Unfortunately, actual usage statistics from the wiki were not available for analysis.

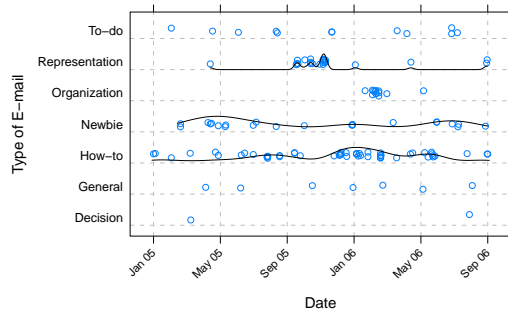


Figure 7.1: Distribution of information management information types over time. Each circle represents one message. The lines are gaussian-kernel density estimates.

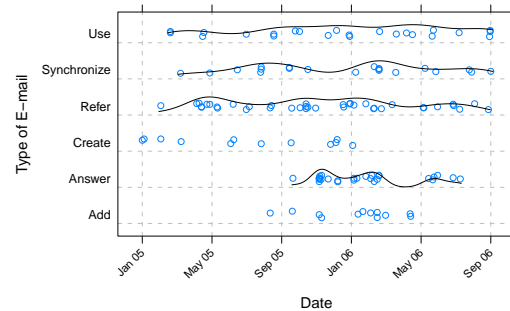


Figure 7.2: Distribution of information management action types over time. Each circle represents one message. The lines are gaussian-kernel density estimates.

information management effort using a keyword search, which resulted in 175 messages. Last, I coded all messages using a simple coding scheme for the activity described or conducted in the e-mail and, if applicable, the type of information managed therein.

On the question which kind of information is being managed in the wiki I assigned 106 messages to one of the following content categories:<sup>174</sup>

1. *How-tos*, i.e. instructions to accomplish tasks such as compiling the project, making a release, submitting a patch (42 of 106 messages).
2. Information for *new developers* (20).
3. Information in the wiki which *represents* the project to others or demonstrates the capabilities of GNU Classpath (16).
4. *To-do* and roadmap information (9).
5. *Organization* of face-to-face meetings [115] (9).
6. *General* references to the information management effort as part of releases being made (7).
7. Only 2 messages were about *decisions* to be managed in the wiki.

A temporal overview of when the messages containing these content references occurred can be seen in Figure 7.1.

These results provide insights about how information management is actually used in contrast to as envisioned by us. Primarily, the low number of decisions needs to be put into perspective. Here we found that GNU Classpath actively avoided making decisions. Two main cases of this could be distinguished: (1) If a decision was uncontroversial, it would get turned into a canonical description of steps to be execute, which can also explain the high number of how-to and new developer documents. (2) If a decision was controversial, an explicit decision was seen as inefficient on two counts. First, an explicit decision might offend the discussion party whose opinion is disregarded. We can hypothesize that the danger of driving members away prevents such non-integrative behavior from happening. Second, the project reported in a follow-up discussion to be afraid of constraining itself on future situations in which the same issue would come up again. This fear appears to originate in the technical merit that the alternative opinion must possess, since it found champions arguing in favor of it. Ruling such an alternative out for all future cases must appear inefficient, if technical excellence is highly valued. In a

<sup>174</sup>The remaining 69 messages were related to the introduction process or replies to the messages containing information management which had been found by the search due to quoting the e-mail they were a reply to.

way, GNU Classpath seemed determined to avoid falling for premature standardization on a sub-optimal choice<sup>175</sup>, but rather wanted to leave space for project members to rely on their common-sense judgment to choose an appropriate solution.

If we adapt this insight for an innovator of an arbitrary innovation, one conclusion may be to leave sufficient room for the technical discussion to be resolved to an extent that project participants are comfortable with the efficiency of the solution.<sup>176</sup> Whether this implies that, unless such a solution is found, an innovation decision should be held off, is not entirely clear. Rather, when choosing an innovation, one should keep in mind that the future might reveal more optimal solutions and that the existence of migration paths away from a to-be-adopted innovation should be considered.

**Strategy 11 (Go for Technical Excellence)** *The innovator should not force a discussion towards a decision and let the technical discussion evolve sufficiently to derive a good solution for the project.*<sup>177</sup>

A second surprising outcome was that the three categories *to-do*, *representation*, and *organization*, which had not been anticipated by the information management proposal in its initial formulation, totaled 35 messages (33% of 106).

If we consider this in conjunction with the misjudgment about the number of *decisions* to be managed, we can conclude that the innovator must be flexible regarding the actual uses of an innovation introduced by him (Rogers calls this *reinvention* by users [425][436, pp.180–88]), because these uses might actually constitute a large part of the benefit derived from an innovation by a project:

**Strategy 12 (Reinvention)** *The innovator should be flexible to accept reinvention of his innovation to maximize the benefit for the project.*

Next, I wanted to know whether it was worth to manage information in the wiki based on the mailing list data. To this end I categorized the e-mails with regard to the activity of the author. I found that 44 of the 106 messages could be assigned to report about information being created (13) or updated (12) in the wiki or inform about information being synchronized from the mailing list to the wiki (19), while 62 messages consumed information either by stating or quoting information from the wiki (23) or referring a discussion participant to the wiki (39). An overview of the temporal distribution of the different activity types is given in Figure 7.2. Since this ratio of roughly 2:3 in e-mails referring to the wiki about information being created to information being consumed does only include those incidents which were explicitly mentioned on the mailing list, we cannot be entirely sure whether this relationship holds in general. Yet, it seems reasonable to assume that people who consume already existing information would cause fewer reference to the wiki (for instance because they found the information using a search engine) than those who created new content and might want to announce it to the project.

**Summary** The information manager study at GNU Classpath was the first in this thesis to demonstrate the successful introduction of an innovation into an Open Source project. Introducing an innovation was found to provide detailed insights into the innovation and to help the innovator advance in the project. In contrast to our expectations we found little use of the information manager role for tracking decisions in the project and hypothesized that this was consciously done to avoid premature standardization. Rather, the project had reinvented our idea in parts to track additional types of non-controversial information, for instance as part of a showcase of achievements.

<sup>175</sup>*Premature Standardization* is a concept from path dependence research (see also Section 6.1). It states that while standardization provides benefits, it can also bring about inefficiencies: A sub-optimal solution might be committed on too early in the process, when the full viability of alternatives has not been sufficiently explored [131].

<sup>176</sup>Using a Garbage Can comes to mind (see Section 6.2).

<sup>177</sup>This is not to say that such an open discussion cannot benefit from an innovator who actively manages this discussion (see for instance Section 6.2).



## 7.2 Gift Giving in Open Source Projects

The second case study conducted by a student under my supervision originated from the difficulties to replicate the circumstances of the first one. Schuster had already been a project member in GNU Classpath in the information manager case study described above [454]. Since none of the other students available to my research project were members of any Open Source project at that point, and neither was I, the approach towards introducing an innovation had to change. From the perspective of a project outsider, Luis Quintela-García and I asked ourselves in summer 2006 (1) how would an Open Source project react to an innovation proposal brought forward by an external person and (2) which role do up-front investments such as code gifts [44] by the proposer play?

RQ1: How do Open Source projects react to external proposals?  
RQ2: What role does up-front investment play?

As a proposal to investigate these research questions with, we selected a usability improvement for the Unix shell command line interface (CLI). We proposed to add scaffolding to bridge the gap between a textual shell and a graphical user interface (GUI). The goal was to reduce the problems which users experience, if they are unaccustomed to the textual paradigm which dominates most Unix operating systems.<sup>178</sup> This proposition is at its core a usability improvement, but can also be understood in terms of a novel requirement posed for an existing terminal emulator or a process innovation for learning how to use the command line interface.

Methodologically, we conducted a field experiment with the two main Open Source desktop environments KDE<sup>179</sup> and GNOME [136], manipulating the amount of up-front investment as the independent variable. For GNOME we kept the description and idea in the abstract, but for KDE we added 150 hours of invested time to develop a prototype. The up-front investment in KDE consisted mainly of learning about the user interface toolkit QT<sup>180</sup>, understanding the implementation of the KDE terminal emulator named *Konsole*, and writing a prototype. We called this working prototype the *SkConsole*<sup>181</sup>. It was able to demonstrate the concept with three standard Unix commands such as *tar*. The source code of the *SkConsole* consisted of 1,250 commented lines.

We proposed the idea abstractly in the case of GNOME and included the prototype as a gift in the case of KDE. Reactions were collected using participant observation [272, 304]. Afterwards, we used content analysis [334] to conceptualize the reactions and derive results.

Results are based on a total of 42 statements (22 from KDE, 20 from GNOME) which we received upon our proposal. Of these there were more positive ones in the case of KDE with a gift being presented (12 positive statements to 10 negative ones), in contrast to the case of GNOME where no up-front investment was being made (2 to 18). Based on a self-devised conceptual model of acceptance and rejection of proposals, we identified 15 categories of reactions to our proposal, from which the following main results were gathered (for a full discussion refer to [413]). A graphical representation of the results is shown in Figure 7.3.

### 7.2.1 Socio-Hierarchy and Techno-Centricity

While the discussion is more focused on technical than social statements (30 vs. 12), we found that technical statements were often driven by socio-hierarchical factors [413, p.25] such as group thinking [47, pp.75f.]. For instance, we received technical remarks on the implementation only to discover that the underlying code had never been looked at. Thus, a focus on code quality was irrelevant for the contacting phase of a proposal.

<sup>178</sup>A detailed description of the proposal is given in [413].

<sup>179</sup><http://www.kde.org/>

<sup>180</sup><http://www.qtsoftware.com>

<sup>181</sup>*SkConsole* derives its name from "Skinning the Command-Line", as the first UI concept was called.

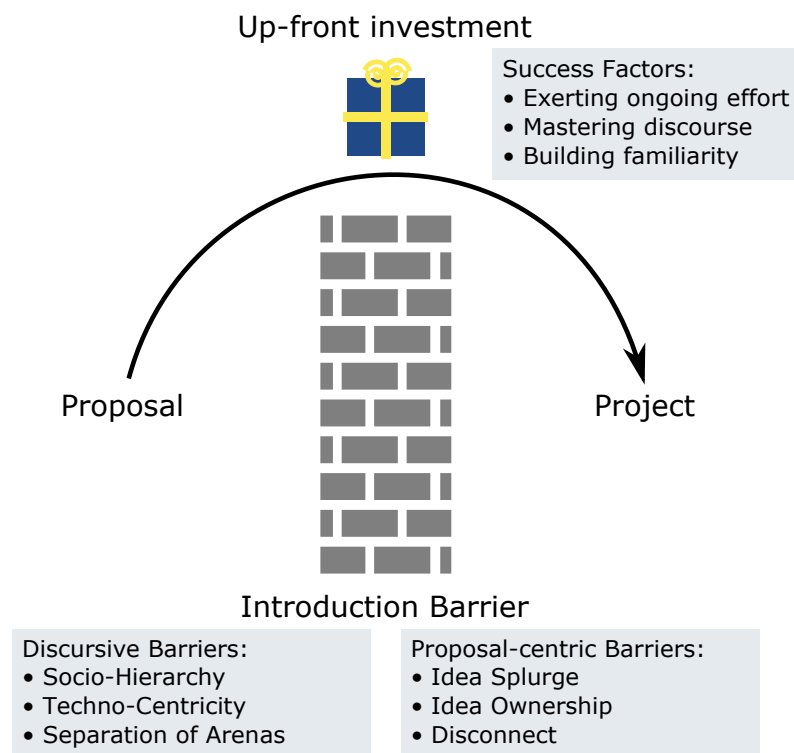


Figure 7.3: A graphical overview of the results of the up-front investment case study (Section 7.2) [413]. Our initial hypothesis was that an up-front investment such as a code gift could help to surmount the introduction barrier facing such a proposal. Yet, we found that even with a gift our proposal had to face the hurdles listed below the barrier for which we propose three counter measures.

### 7.2.2 Idea Splurge and Disconnect

The most common reaction to our proposal and at the same time a type of reaction we did not anticipate were references to existing applications supposedly already accomplishing the proposed idea. Looking at these ideas revealed that they were all related to our proposed idea, but had different goals and solution designs. In short, the referred applications did neither aim for nor accomplish what SkConsole was meant to do. Thus, even though our description of the idea was precise and supported by a website including screenshots, such misunderstandings did arise and drained much of the energy from the discussion, prompting us to conclude that the proponent must pay particular attention to prevent misconceptions:

**Strategy 13 (Counter the Idea Splurge)** *An accurate description of a proposed solution still needs to include a list of common misconceptions about what the solution is not. This list should be kept liberal to include even remotely similar ideas.*

To explain the reason for this “idea splurge” to occur, we can hypothesize that a lack of knowledge about the underlying issue might be a primary cause. This is because the proposal has at its heart complicated questions about usability and learning of new user interfaces. Similarly, Thiel observed in his case study of introducing a security innovation that a lack of in-depth understanding of security issues in web development made communication of his innovation much more difficult [501]. Also, episodes surrounding legal innovations can be named as additional examples in which project participants showed significant difficulty with mastering the subtleties surrounding copyright law and the use of the GPL, causing discussions permeated by explorations into how these legal instruments actually work and plain wrong information (see the *episode regarding GPL3 in the project GRUB* and the *licensing discussions on schematics* in the project *gEDA*).

**Hypothesis 11** *Open Source participants have less expertise in peripheral topics such as law, usability, or security compared to their technical and programming related skills.*

A second way to interpret the “idea splurge” is to consider the conceptual “distance” of a proposal to the immediate problems, worries, and day-to-day activities surrounding the development of an Open Source software solution. If a proposal is close to the daily tasks and goals, it is easier to understand it and its implications. If it is far removed or remote, a much larger conceptual gap needs to be bridged. In our case, what was proposed was a remote and disconnected idea. Krogh et al. have used the term “unsolicited ‘new’ ideas” [533, p.1227] to describe such propositions which are not yet present in the discourse of a project. An implication of such a distance between a proposal and the project’s immediate needs is that the term “gift” for an up-front investment becomes dubious. How can something be a gift, if it is not something perceived as useful to the project, but rather something which needs comprehension and integration first?

Similar problems arise in the context of bringing research results into industry [406] or transferring knowledge into the heads of students [101], and we might propose:

**Strategy 14 (One step at a time)** *To achieve the transfer of ideas contained in a proposal, the innovator needs to connect to existing ideas inside the project.*

Or as a hypothesis:

**Hypothesis 12** *The immediacy of proposals (in contrast to their remoteness) is positively correlated with acceptance.*

From an even more critical perspective: The gift—being motivated by latent self-interest of the giver—might incur indebtedness via reciprocity in the receiver and thereby limit the receiver’s independence [503, pp.14f.][463]. It remains the responsibility of the innovator as a giver to consider the ethical implications of his or her action.

### 7.2.3 Idea Ownership

One initial assumption when proposing the SkConsole was that agreement to our idea would cause the idea to be taken up by the project and incorporated into the goals of a future release. Yet, while we did receive a series of positive remarks (14 statements of 42), there were no signs that the idea would be picked up and realized by anybody. Rather, all positive remarks we received pointed back at us to devote energy and time to make the idea a reality. Thus, for an external innovator the implication is that he can not assume that a proposition alone will affect any change, but that rather he himself will have to bear the effort to execute it:

**Hypothesis 13** *Ideas and proposals are strongly owned by the people proposing them.*

**Hypothesis 14** *Achieving delegation for tasks is unlikely for externally proposed ideas.*

### 7.2.4 Gaining Familiarity

Discursive elements such as discussions revolving around definitions of terms (for instance “power-user” vs. “newbie”, “Open Source” vs. “Free Software” [47]) capture large portions of communication. This is not necessarily bad, as such discussions often define the goals, requirements, and ideals of a community [447]. Yet, the proponent—often unaccustomed to such arenas of discourse—must be cautious to use the right terminology. For example, Quintela-Garcia received negative remarks for his proposal because he had inadvertently confused the terms “beginner” (a user inexperienced with a particular program) and “newbie” (a user inexperienced with Linux in particular or computers in general). Since such preferences for particular terms are not formalized anywhere, a familiarity with the community and its norms is necessary to avoid such a mistake:

**Strategy 15 (Build Familiarity)** *Building a close familiarity with a community pays off.*

Thiel in his innovation introduction study provides us with another example of how familiarity might be beneficial [501]. When he proposed his security enhancing innovation, he first explored the code of the project to understand their mechanisms for securing the software. When he did not find state-of-the-art security mechanisms, he concluded that he could propose his innovation by referring to a lack thereof. Yet, it turned out that such a proposal was ill-received by the project community in at least one case because the members believed to have guarded against the alleged vulnerabilities using mechanisms which were non-obvious to Thiel. While Thiel could show that the mechanisms were unsuitable for establishing security, he might have been able to avoid raising tempers in the first place, if more familiarity of the code base of the project had been established.

On the question of how to best attain such familiarity, one interesting insight from Schuster’s study in the project GNU Classpath might be worth recalling (see the previous Section 7.1). Schuster found that the process of introducing an innovation including discussing and designing an innovation, executing it within the project and speeding the adoption has helped him gain exactly such close familiarity. In this way a virtuous cycle can be set in motion, where the keen interest of the innovator to improve the project raises his status and knowledge, thus enabling him to act more freely and efficiently.

### 7.2.5 Separation of Discourse Arenas

When proposing our idea initially to the Konsole mailing list of the KDE project, we received negative remarks and were referred to the KDE Quality mailing list as a forum more suitable for propositions as ours. While this forum was friendly to our proposal, we found that it also was effectively powerless to help us to accomplish the goal of getting the idea of the SkKonsole into the KDE project. The existence of such contradicting attributes puzzled us, until we resolved this issue by understanding that the arenas in which power about decisions is present are more highly guarded and shielded from open discussion (see above and compare with Section 6.2 on the Garbage Can Model).

We believe that it is this separation of discourse into inclusionist and exclusionist arenas which has led to the idea of Open Source projects being suitable for outsourcing implementation effort. In particular, targeting an inclusionist forum such as KDE Quality can result in overly positive reactions, giving the impression that the idea might be implemented by project members within weeks. From the results of this study, we believe that such free riding cannot be substantiated when considering commitment and meritocracy in the project.

### 7.2.6 Introduction Barrier

Taken together, idea ownership, lack of expertise with peripheral topics, conceptual distance of a proposed idea, unfamiliarity with the existing arenas of discourse, and idea splurge can be seen as causing a considerable introduction barrier between the proponent and his goal to see an idea adopted by a project. When analyzing the reasons for such resistance to change, we found three explanations:

1. Popular projects such as KDE and GNOME are constantly assaulted by new ideas from external parties. Unless barriers exist for filtering out ideas of questionable benefit, the projects would easily drown in numerous propositions. One sensible way to test any proposition is to ensure the proponents' commitment to the idea up to the point where the execution of the idea is put on the proponent himself.
2. Even if commitment is demonstrated (as in our case), the project and its leadership might put barriers in the way of a proposal due to the risk of destabilizing the project. Such instability can be introduced either in the code base, which needs to adapt to include the implementation of the idea potentially causing novel defects in the software, or in the community hierarchy due to the entry of a new project member.
3. As Thiel has found in his study when proposing the creation of a database abstraction layer, his preference for such an innovation was not shared by the project. The lack of hard knowledge about security by the project made the benefits appear far-fetched. Rather, the associated drawbacks of the proposal appeared much more substantial. The project feared the loss of direct expressiveness in SQL and cited associated learning costs and effort for plug-in developers as indicators of the superiority of the status quo. An interpretation could be that the proponent is easy to underestimate the impact of his proposal and the relative importance assigned to its implications.

### 7.2.7 Summary for the Innovator

In this case study of proposing a usability improvement to the projects KDE and Gnome we learned about several barriers to getting a proposal accepted, and we discussed strategies for overcoming them. Most importantly, (1) up-front investment is essential, but should be directed at gaining close familiarity with the project rather than implementing a solution, (2) on-going commitment by the proponent to his own idea will be necessary well into the process of making the idea a reality, and (3) mastering the discourse of an Open Source project represents a major hurdle in becoming an integrated participant who can contribute ideas. It is an open question after this study whether and how idea ownership can pass from the originator to other project members.

## 7.3 Contact Strategies for Open Source Projects

The previous study of up-front investment in proposing a usability improvement had allowed us to gain many insights, yet, the overall response to our proposal had been negative despite considerable effort. We thus still felt that a better understanding of approaching an Open Source project was necessary.

RQ: How  
should an  
Open Source  
project be  
approached?

The research question that Alexander Roßner and I thus set ourselves next in the context of external proposals being targeted at Open Source projects was the following [437]: (1) How should an Open Source project be approached? And in particular: (2) Who should be contacted?

From a survey into the literature we discovered Roger's typology of innovation decisions [436, p.403] and decided to explore it further: Rogers had categorized decisions to adopt an innovation inside an organization into (1) *optional innovation decisions*, which each member of the organization made individually and independently, (2) *collective innovation decisions*, which were made as a consensus within the organization, and (3) *authority innovation decisions*, which were made by a small influential group within the organization [436].

From this categorization, we hypothesized that the question of whom to contact should be answered in accordance to the expected innovation decision type (the details of this relationships are discussed in [377]):

1. If an innovation was expected to be adopted via optional innovation decisions, then selected individuals should be *targeted* using private e-mail to convince them of adoption (contact strategy TARGETED<sup>182</sup>).
2. If a consensus was expected to lead to adoption by the project, then a discussion should be *spread* as widely as possible by the innovator onto the mailing list (contact strategy SPREAD).
3. If an innovation decision was expected to be taken by the leadership of a project, then the innovator should focus on the *top* people in the project (contact strategy TOP).

To sum up the hypothesis for this study: For optional innovation decisions the innovator should contact individual developers, for collective innovation decisions the innovator should contact the mailing list, and for authority decisions the innovator should contact the maintainer or project leadership.

The obvious problem with this approach is that for many innovations there is no predefined associated type of innovation decision and the innovator does not know ex-ante for instance whether the project leadership will decide or let the project vote. Any clear mapping from innovation to innovation decision type is further confounded by innovations requiring many decisions, each of which can be of a different type. For example, the information manager as a light-weight role-based process innovation (see Section 7.1) includes aspects of all three decision types: If a wiki as a technical platform for the information manager is necessary to be established first, this might involve an authority innovation decision by the project leadership to provide the innovator with *hosting* for the wiki on the project infrastructure. Yet, the innovator can also use external wiki hosting to sidestep this decision and directly proceed to a discussion on the mailing list. Here the innovator needs a consensus on which information to collect or whether to collect information at all. Yet again, this discussion might be unilaterally decided by the maintainer or conversely by the individuals in the project who decide with their feet by entering information on their own into the wiki.

To explore how these theoretical relationships would play out in actual innovation cases, we decided to adopt the methodology of an exploratory field experiment with six medium-sized Open Source projects during Spring 2007 [236]. As an independent variable we used the three contact strategies TOP (1 project), SPREAD (2 projects), and TARGETED (3 projects). We then conducted an innovation introduction attempt in each project with the information management innovation from [454] and observed the influence on the dependent variable of *introduction success*. For this study we defined success as establishing a wiki filled with some initial content. Thus, this definition does not consider adoption by project members as readers and writer of the wiki but only takes one step beyond *execution*. Data was gathered during the experiment using participant observation [272, 304], but unlike the previous study [413], the data analysis was performed unstructuredly.

Results of these six innovation introductions stretching a total of 70 days are that (1) the TOP strategy

<sup>182</sup>In [437], the strategy TARGETED was called BOTTOM as a contrast to TOP. Yet, since BOTTOM has too much a connotation to the periphery of a project and not as intended the set of active core developers excluding the project leadership, it is not used here.

of contacting the maintainer was successful in the single given case, (2) the SPREAD strategy of discussing on the mailing list was successful in one of two cases, and (3) the TARGETED strategy failed in all three cases. In more detail:<sup>183</sup>

1. The TOP strategy was successful with a wiki being *set up* within six days and *announced* after another eleven days. The wiki then gathered 400 page hits in the first 16 days of *use*. In proposing to the maintainer, the innovator found somebody who was enthusiastic about the idea and had the capabilities to execute the necessary steps to realize the innovation. Thus, by stipulating the idea and supporting its execution by writing articles for the wiki, the idea was successfully established.<sup>184</sup> TOP
2. The SPREAD strategy of opening a discussion directly on the mailing list was successful in one of two cases. Yet, unfortunately the successful case did not lead to a discussion on the mailing list. Rather, the list was moderated, the project leader intercepted the message as not trustworthy, and tested the innovator for commitment to the idea first. He did so by first ignoring the innovator and then by doubting the resolve of the innovator. Effectively, this turned the introduction into a TOP approach. SPREAD

Unlike our first TOP case though, the project leader as moderator was not immediately convinced of our innovation. Thus, it became necessary to build trust and demonstrate commitment to overcome the moderator's doubts. This insight is a known fact in the community literature, well put by Pavlicek as "Forget about taking more than you give" [402, p.137]. This is not to say that Raymond's "Magic Cauldron" [418], into which you put a little bit and get much out, is not rewarding everybody's participation in a project. In fact the cauldron appears to work so well that 55% of respondents in [211] state that "I take more than I give", over only 9% which say the opposite is true. Rather, such rewarding or "taking" is working via software artifacts and code which can easily be duplicated as digital goods. Consuming more *man power and time* on the other hand—which cannot be easily replicated—than providing such, cannot work. Open Source projects must naturally guard against free riding to some degree or otherwise fear being overwhelmed by requests from the public. In our case demonstration of commitment came by persistently communicating with the moderator, until after six days the maintainer established a wiki. This wiki was then filled by us with content and linked to from the project homepage after 60 days by the maintainer.

The second SPREAD approach failed without gathering a response in the project forums. Thus unfortunately we were not able to start a discussion on the mailing list to analyze the responses to a SPREAD approach.

3. For the TARGETED approach we contacted three active developers in each of the three projects who were not in the leadership position in their respective projects. We expected to be able to discuss with each developer the proposal and possibly to convince them to become a champion. In the worst case we expected to have our e-mail forwarded to the list or being directed to contact somebody higher up. Yet, we did not gather any replies from the nine contact developers even after a reminder was sent and can conclude that the approach is unsuitable for an innovation such as the information manager. It appears that the innovation was perceived as within the scope of responsibilities of the project, yet outside the scope of legitimate actions to be performed by the individual. How and why the information manager innovation constitutes such a barrier and why this did not prompt the targeted developers to forward the e-mail to the list or suggest contacting the maintainer remains an open question. Most likely the implicated changes to the project hosting infrastructure, i.e. setting up a wiki server software such as MediaWiki or Foswiki, will have caused the reluctance. Whether it is possible to overcome this barrier posed by hosting in general seems doubtful, although innovations such as *distributed source code management tools* (for instance *Git*) have started to reduce the reliance on and importance of a centralized TARGETED

<sup>183</sup>Full results are available in [437].

<sup>184</sup>While successfully established, the wiki software fell prey to a hacker attack on the 50th day of the study and could not be reinstated before our observation period ended after 70 days.

hosting location (see Section 8.1.1).

4. In both cases in which we succeeded to introduce a wiki into the project, the project leaders established a filter between the innovator and the project. In the successful SPREAD approach this was obvious with the mailing list being moderated, but also notable when the work we did in the wiki was closely monitored by the maintainer. During the TOP approach, the maintainer did only announce the availability of the wiki to the project after it had been set up, primed with some content, and then found meeting the maintainer's quality standards.

The behavior of shielding the project from certain contributions has been called "gate keeping" (see Section 5.7). Projects would use it as a strategy to enforce the rules and principles of its community. Yet, in both cases here, not only was the contribution blocked but also the information about it. This conflicts with the general Open Source community values such as Open Collaboration and Open Process [66, p.126]. With so little data, we can only offer three interesting interpretations why such shielding of the project occurred:<sup>185</sup>

- (a) It is plausible to perceive the maintainer or moderator as taking on a paternalistic role<sup>186</sup> in this interaction, to decide what is best for the project without consulting it first. This might connect best to the notion of the "benevolent dictator", as project leaders such as Linus Torvalds of the Linux kernel project and Guido van Rossum of the Python project are frequently called because they claim for themselves the "right to unilaterally make decisions" about "their" projects [92, 420].
- (b) A second interpretation can be that the behavior of the maintainer and moderator is not in conflict with community norms but rather constitutes what Elliot and Scacchi call *bending* of "the rules [...] for development and documentation where appropriate and convenient" [162, p.9]. Under this interpretation there is no conflict until a counter position in the project comes up, which is set on enforcing the community norms more strictly. In the example given by Elliot and Scacchi, the screenshots displayed on the project homepage had been created with a non-Open Source software. This, while in violation of the community preference for using software which is Open Source (Brown and Booch call this Open Environment), did not raise a conflict until a project member raised it as an issue in an IRC chat [162, p.4].
- (c) As a last alternative one could conclude that Open Collaboration and

Open Process are not as much community values as one might think. Certainly many communities are built around the participation of volunteers from all over the world and many aspects of Open Source licensing lend themselves to using an open development model, yet even Richard M. Stallman as the father of the Free Software movement has famously demonstrated that cherishing freedom of software does not necessarily lead to a collaborative development process. The case alluded to is GNU Emacs being forked into XEmacs (Lucid Emacs initially) because the FSF and Lucid as a company using Emacs in a commercial offering could not agree to collaborate on version 19 of Emacs [348, 512].<sup>187</sup>

As a cautious implication for the innovator we might conclude:

**Strategy 16 (Stick to Community Norms)** *Adhere to community norms, but keep a backup plan, in case the project does not.*

5. Four smaller results are: (1) In one of the projects the communication infrastructure of the project used forums instead of a mailing list, which we found to be more difficult for the innovator, as forums' rely on project participants to come by and check for new posts reduces feelings of

<sup>185</sup>Too much spam or a flood of low-quality support requests being the less interesting, yet plausible causes.

<sup>186</sup>Paternalistic action of an initiator such as the maintainer towards a recipient such as a project member can be defined as action which is "(1) [...] primarily intended for the benefit of the recipient and (2) the recipient's consent or dissent is not a relevant consideration for the initiator" [251].

<sup>187</sup>As with many forking stories the details are somewhat disputed between the fractions on the opposing sides of the fork [560].



involvement. (2) A lack of replies to innovation proposals does not necessarily imply an immediate rejection of the ideas in the proposal. Technical reasons for failure such as the e-mail being categorized as spam or being overlooked by the recipient are common enough to always send a reminder in case of a missing reply. (3) We discovered again that introducing an innovation helps to gain a better understanding of the innovation [377]. In particular we found that the maintainer can naturally fill the role of managing information because of his close familiarity with the project and interest in its well-being. Yet, from a critical standpoint, a maintainer who takes on this additional job is doing little more than micro-managing. Rather, the community would benefit more if it is taken on by somebody else in the project. (4) Of a total of 214 messages which we received as part of this study, 96% were sent during a total of seven hours over the course of a day when expressed in local time of the sender. These hours were before work (7–8am, 28 messages), before/at lunch (11am–12pm, 75 messages), after work (4–5pm, 65 messages) and at night (8pm–12am, 37 messages). Despite the data set being small, we can infer that our sample must be primarily one of hobbyists, as professionals would also be able to respond during working hours [132]. We find that e-mail conversation during the weekend is markedly reduced as reported in the literature [510].

As conclusions for the innovator on the question of whom to contact, we have found that an innovation which affects project infrastructure such as the information manager was best targeted at the project leadership. Our results indicate that building a strong relationship with the project leadership is important and demonstrating persistent commitment is the best way of building such.

As a closing remark, this study is an exploratory field experiment with limited data set and validity needs to be discussed. Regarding internal validity, we primarily cannot be sure whether our interpretations of the observable phenomena in the projects are correct or whether important events occurred outside of the publicly observable communication. Regarding external validity, we did select a sample of six projects which fit our criteria for normal, active, mid-sized projects, but naturally this selection has biased our result. Finally, we should note that in particular we did not gain results regarding a successful SPREAD approach, so we cannot know how a discussion on the mailing list would have affected the innovation introduction.

## 7.4 Introducing Automated Testing

After both previous case studies had failed to observe more than the initial phase of approaching an Open Source project, I decided to perform an introduction myself in March 2007. This is still preceding the use of GTM and was a last attempt to see whether an active approach could lead to substantial results (see Section 3). The goal of this study was to demonstrate the feasibility of introducing an innovation as an outsider and gain some first strategic advice for an innovator to achieve a successful introduction. I chose automated regression testing [547, 35, 267, 275, 148, cf.] as an innovation because it represents a quality assurance practice well established in industry. I could therefore be more certain that the introduction would not just fail because the innovation was novel and first needed to proven useful.

The introduction was conducted in the Open Source game project FreeCol<sup>188</sup> in April and May 2007 using a four-stage activity model shown in Figure 7.4. During this time I became a project member, created 73 tests cases and collaborated with the project on testing. Yet, when I left the project, the test suite quickly decayed and since no testing activity could be observed, I considered the introduction a failure. As a last effort, I repaired the test suite in September 2007, but already oriented myself to the passive study of innovation introduction as discussed in Chapter 3 on GTM methodology.

When returning in August 2009, I was surprised that the test suite had grown to 277 tests, all tests were being passed, and coverage had increased to 23%, marking the introduction a success (compare

---

<sup>188</sup><http://www.freecol.org/>

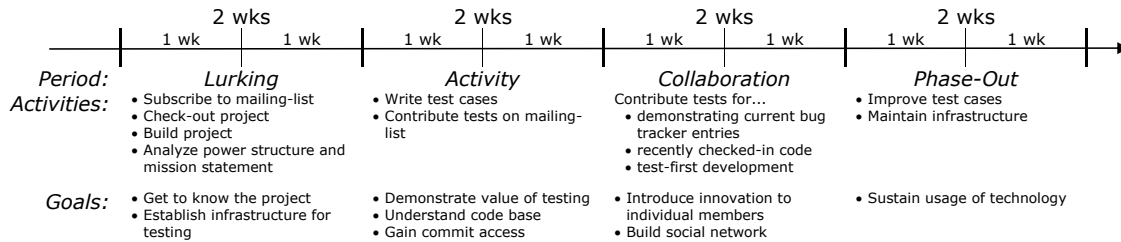


Figure 7.4: Phases in the introduction process of automated regression testing in the project FreeCol.

Figure 7.5). I then analyzed the mailing list and the source code repository both quantitatively and qualitatively, resulting in six insights about innovation introduction and testing and five strategies for the innovator given below. Details of testing method, methodology, the stage model, case selection, results, and validity considerations can be found in [374] and have been published in [375], from which some of the following paragraphs are excerpted.

First, it was uncovered how important it is to communicate to the project about important changes in the innovator's status. In the particular episode, the introduction model had suggested to phase out activity during week six to eight by slowly reducing activity. Analyzing this initial departure, it appeared that the project did not become aware of the innovator's reduction in activity and the associated fact that this reduction must be compensated by other project members to keep the test suite working as refactoring and code changes occur. The first point can be directly associated to the notion of a participant's presence in a project [300], which has been discussed in Section 6.2.2.3 on phenomena related to the Garbage Can Model. In FreeCol, the test suite was broken entirely within three months, at which point the maintainer contacted me and asked for a repair, highlighting how slowly awareness of absence in a project can spread.

To prevent a similar fate after fixing the test suite, I started a discussion about the importance and responsibility associated with testing, raising awareness that the innovator was about to depart again and telling the project that the test suite needed active maintenance. As this departure was much more successful, with the maintainer assuming the role of a test maintainer until another project member assumed responsibilities for the tests, we can deduce the following strategy.

**Strategy 17 (Signal your Engagement)** *An innovator should explicitly signal his engagement and disengagement in the project to support the creation of shared ownership of the innovation introduction. In particular this will be necessary if the innovation itself has no signaling mechanisms to show individual or general engagement or disengagement.*

This strategy is cast wider than just talking about starting and stopping on a task, but also includes being outspoken about those process steps one has taken off the mailing list. During the introduction of *Git at gEDA*, for instance, the innovators even used the incident of their demo system no longer working to talk about the innovation with the project [geda:2889].

The second part of the strategy needs some more elaboration: Automated regression testing does have a mechanism by which its users can detect disengagement, namely failed tests. As the case illustrates, this prompted the maintainer to contact me, yet since running the tests was not integrated into the process of committing code to the repository, did not prevent the test suite from breaking in the first place.

The second insight was deduced by analyzing the increase in coverage in the project (see Figure 7.5d), which shows two notable increases. The first was the expansion of coverage from 0.5% to 10% by me (as the innovator), and the second occurred in April and May of 2008, when one developer expanded coverage from 13% to 20% by writing test cases for the artificial intelligence module (see Figure 7.5e). Yet, beside these notable increases, which occurred over a total of only four months, coverage remained stable over the two years. This is unlike the number of test cases which constantly increased with a

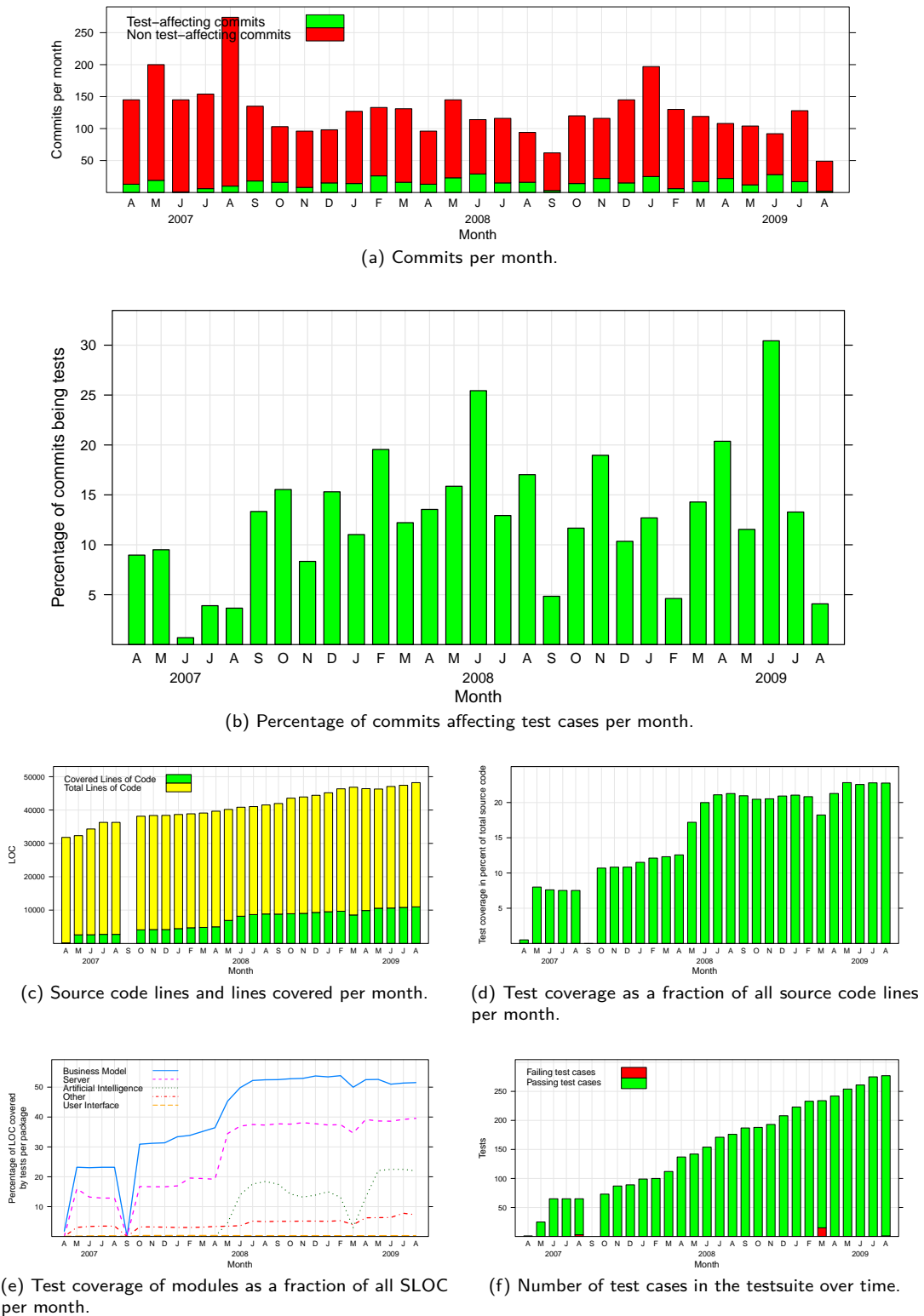


Figure 7.5: Six plots from the introduction of automated testing at FreeCol. The increasing number of test cases, increasing coverage, stable fraction of commits affecting test cases at above 10%, and a test suite being maintained to pass were taken as indicators that the introduction was successful. The gap in September 2007 is due to the test suite breaking entirely because of a refactoring.

remarkable rate of passing tests (see Figure 7.5f). On the mailing list a hint can be found that this is due to the difficulty of constructing scaffolding for new testing scenarios (see for instance the discussion following [freecol:4147] about the problems of expanding client-server testing) and thus indirectly with unfamiliarity with testing in the project. This thus poses a question to our understanding of Open Source projects: If—as studies consistently show—learning ranks highly among Open Source developers' priorities for participation [211, 238, 252], then why is it that coverage expansion was conducted by just two project participants? Even worse, the author as an innovator and the one developer both brought existing knowledge about testing into the project and that project participants' affinity for testing and their knowledge about it expanded only slowly.

**Hypothesis 15** *Knowledge about and affinity for innovations are primarily gathered outside of Open Source projects.*

This conjecture can be strengthened by results from Hahsler who studied adoption and use of design patterns by Open Source developers. He found that for most projects only one developer—independently of project maturity and size (very large projects being an exception)—used patterns [228, p.121], which should strike us as strange, if sharing of best practices and knowledge did occur substantially.<sup>189</sup> At least for knowledge- and skill-intensive innovations this argument then on the one hand can thus emphasize the importance of a Schumpeterian entrepreneur or innovator who pushes for radical changes:

**Hypothesis 16** *To expand the use and usefulness of an innovation radically, an innovator or highly skilled individual needs to be involved.*

On the other hand, if we put the innovator into a more strategic role to acquire such highly skilled and knowledgeable project participants with regard to an innovation, then we can deduce two different options: Either the innovator assumes an optimistic view and actively promotes learning in the project to train such highly skilled individuals, or pessimistically the innovator considers the abilities of the project members as mostly fixed and thus rather aims to increase the influx of knowledgeable participants into the project by lowering entry barriers. Formulated as strategies:

**Strategy 18 (Teach for Success)** *The innovator needs to actively promote learning about an innovation to achieve levels of comprehension necessary for radical improvements using this innovation.*

**Strategy 19 (Look for Experts)** *An innovator can strengthen innovation introductions by lowering entry barriers and recruit highly skilled individuals.*

Both strategic options have some arguments on their side. For instance, promoting learning in the project matches well with a perspective of Open Source projects as Communities of Practice [301, 541, 511] which should foster both re-experience and participatory learning [246, 238]. Lowering entry barriers on the other hand can be argued for well if high membership turnover would make any long term learning efforts to be likely in vain.

### 7.4.1 Testing-specific Results

The remaining insights are innovation-specific rather than about introducing innovations and are presented here primarily because they demonstrate the feasibility of using the introduction of an innovation in an Open Source project as a method to validate and understand a particular innovation [377].

First, it was found that automated regression testing in Open Source projects is used by the project participants as a way to bridge between soft contributions such as mailing list post and hard ones such as source code patches. In particular project members used test cases to communicate bug reports more precisely to each other [freecol:2606,2610,2640,2696,3983], codify their opinions about implementation alternatives as test cases [freecol:3276,3056] or as starting point for discussions about how FreeCol

<sup>189</sup>In personal communication Hahsler stated that to really confirm such a conjecture an in-depth study or interviews with the developers would be necessary (October 2009).

should behave [freecol:1935]. These uses were unexpected by the innovator and thus provide another case in which reinvention occurred (cf. Strategy 12).

Secondly, considerable difficulties were found with testing the network, user interface and artificial intelligence modules. These difficulties are mostly due to lack of scaffolding in the project for these specialized types of modules, but point to a lack of support by testing frameworks as well.

### 7.4.2 Summary

This case study has led to several new strategies for the innovator, notably signaling innovation-specific activities and state changes to the project participants, and two strategic options—promoting learning and lowering entry barriers—surrounding the surprising lack of learning achieved inside the project. This case study is an important part of this thesis, as it shows feasibility for an outside innovator (something which was not observed in the data analyzed using GTM) and that following a simple stage model of introduction can already lead to considerable adoption over time. It complements the innovation manager case study in Section 7.1, which was conducted by a project insider and with a documentation and role-centric innovation. From a methodological point of view, this study is important because it triggered the change in methods towards GTM because the effort and in particular the time necessary to achieve adoption was not commensurate to the insights gained.

## 7.5 Increasing Security by Annotation-guided Refactoring

This fifth case study, regarding security in Open Source projects, was conducted by Florian Thiel under the supervision of Martin Gruhn in the beginning of 2009. In contrast to the other case studies, this research was thus conducted after GTM has been adopted and most of the main results had already been gathered. This made it possible to condensed these results into the *Innovator's Guide* (see Section 8.3) to provide practical, accessible advice to Thiel and Gruhn, who used it successfully to this end [501, p.63].

The goal of the study was to design and evaluate a process innovation to prevent security vulnerabilities<sup>190</sup> from being included in an OSS software. This goal was to be realized in the specific context of popular Open Source web applications and in particular in Open Source Content Management systems developed in the programming language PHP. His study concentrated on the two most important classes of vulnerabilities—SQL Injection and Cross-site Scripting (XSS)—according to the well-respected vulnerability trend analysis by US government contractor MITRE<sup>191</sup> Corporation [83]. Both SQL Injection and XSS are vulnerabilities which commonly involve user input being mishandled by a software so that the input causes unintended behavior in an associated system such as a database or browser. SQL Injection and XSS are ultimately sub-types of a “failure to sanitize data into a different plane” [123].

To devise an innovation to prevent vulnerabilities to be included in the source code of an OSS project, Thiel first conducted an in-depth analysis of causes, mechanisms and mitigations for both types of vulnerabilities [501, p.9–44]. The result of this analysis was that technologies, processes, and concepts existed and were sufficiently understood by the web-application development community at large to prevent security-related defects if applied rigorously and correctly. For instance, on the technology side strongly-typed queries can prevent SQL injections by enforcing type constraints on parameters at the language level rather than causing unintended effects in the SQL interpreter [103]. As examples for processes, Defensive Design [336, pp.187] or Defense in Depth [220, p.48] both mandate to adopt a mindset in which system failure is common and design is centered around secure defaults. As a central

<sup>190</sup> A vulnerability is any structural deficiency of a software system (usually hailing from a defect in the source code) which increases the risk of a successful attack being made on or via the software system, thus reducing its security [501, p.7].

<sup>191</sup> MITRE is not an acronym.

concept, Thiel found that clear separation of concerns regarding sanitation and data handling simplifies keeping a software secured.

When looking at the state of the practice, he found that web development frameworks and components already used such technology and processes. For instance, the web development framework Django<sup>192</sup> can be cited as an exemplary project which uses data modeling and a clear separation of concerns for data sanitation to achieve security with regard to the two types of vulnerabilities. Yet, when turning to the state of the practice in popular Open Source web applications (WordPress, Mambo, Joomla, Zikula, Habari, phpBB<sup>193</sup>), little to no use of these possible solutions could be found.

The central contribution in Thiel's work then was the invention of a process innovation which would help projects to transition to such solutions. An indirect approach of devising a process innovation to help with the incremental introduction of security solutions was chosen instead of introducing any of such solutions directly, because the capacity for radical and direct change was seen as minimal (see Section 5.9 for a discussion on capacity for change in OSS projects).

Based on his analysis, Thiel created a process innovation based on the idea of establishing a standard for annotating source code locations which should be modified or refactored to improve security. Thus, the goal shifted from preventing vulnerabilities to be included to rather helping the project move away from existing problematic code which already contains vulnerabilities or is likely to induce them in the future. Using annotations has two alleged advantages: First, it provides a division of labor for detecting error-prone or vulnerable code locations and then resolving them, which should increase the ability to work incrementally. Second, the innovation provides an efficient and code-centric way of managing issues and information about them (see [376] for a discussion of the advantages of localizing such information in code in contrast to keeping them separate for instance in a bug tracking system). Annotations to designate problematic code locations of database access and HTML output were defined to consist of three parts for Thiel's study: (1) Problem and resolution type (for instance the identifier `method_exists` was used for a code location where raw SQL was used even though an appropriate method existed in the database module to encapsulate the given access), (2) an effort estimation to enable developers with a given amount of time for refactoring to chose an appropriate problem, and (3) a list of required changes to support and steer interdependent issue resolution.

Given this process innovation of using annotations to denote code locations requiring rework, it was decided to perform an innovation introduction to evaluate the innovation in the real world context of an OSS project (this research methodology of evaluating an innovation by introducing it to an Open Source project was published in [377]). From the list of examined Open Source web applications, WordPress was selected as the most promising candidate to benefit from the innovation. During this introduction, Mambo was suggested by a mailing list participant as potentially interested in the innovation and thus a second introduction was staged with this project as well.

Turning to the outcome of the introduction, the innovation introduction with WordPress was a *failure*, because the project members rejected the innovation during a mailing list discussion. The introduction at Mambo seemed successful at first in that the project members included annotations into their code base. Yet, the joy was short lived, after it was discovered that the project despite giving positive feedback to the effort was inactive and thus the chances for the issues represented by the annotations to be resolved slim.

Nevertheless, a series of important theoretical concepts regarding the state of Open Source web application security as well as regarding the introduction of a security related innovation could be deduced:

First, several concepts could be found to explain the status quo of Open Source web application security.

<sup>192</sup><http://www.djangoproject.com>

<sup>193</sup>Initially Thiel looked at six popular Open Source web applications written in PHP (WordPress, Joomla, phpBB, Zikula, Drupal, Typo3), in the process of which another three less popular projects were recommended to him or discovered by him (Mambo, Habari, Riotfamily). After a cursory examination, he focused on the six mentioned ones performing in-depth code reviews and an analysis of publicly available information for each.

The three primary reasons identified in source code for why SQL injection and XSS vulnerabilities are likely to occur were: (1) Non-uniform data base access, (2) inconsistent use of sanitation, and (3) sole reliance on input sanitation (for details see [501]). For the second concept Thiel found that lacking standards for how to perform sanitation of user input caused a bewildering array of different approaches, which must likely lead to slip-ups and thereby introduce possibilities for attacks. Searching for the root cause of these problems, Thiel found that the lack of a data model made consistent data handling hard or even directly facilitated attacks.

Second, when looking at concepts that explain why the projects did not leave the (probably insecure) status quo and in particular why WordPress rejected his proposition, he found five concepts preventing a transition and thereby limiting the chances for success of an innovator:

1. *Legacy constraints*, such as the reliance on a certain outdated version of PHP which is incompatible with modern technology, has been found in the case of WordPress to be a limiting factor. Abstractly, a legacy constraint causes prerequisite efforts before mitigating technology can be adopted. Since the completion of such required work does often not deliver any benefits in itself<sup>194</sup>, an adoption barrier results.
2. *Missing interfaces* for outside parties of the project caused a reverse situation from legacy constraints: Since no interface for accessing functionality in the software exists for plug-ins and extension modules, the authors of the software itself have no well-defined contract behind which they can change the implementation. The violation of David Parnas's *information hiding* principle [399] thus prevents changes to the core of the application, unless existing plug-ins and extensions would also be reworked. In the case of WordPress, project leadership strongly argued that even an intermediate deprecation of plug-ins and extensions would hurt the market share of WordPress and thus ultimately the project as a whole.
3. *Licensing issues* and in particular license incompatibilities have been found at least in the case of the project Habari to restrict the project's ability to move to desired technology. While the last years have brought some improvements to this issue with the introduction of the GPLv3 [209], the restrictions on the innovator for reusing existing code and technology must still be regarded.
4. A *loss of expressive power*<sup>195</sup> was argued as a strong reason in WordPress against the introduction of an architecture layer encapsulating all database access. The project members believed that such would reduce their ability to leverage the programming language to the fullest, possibly causing performance penalties.
5. Thiel noticed *structural conservatism* when proposing the introduction of a database abstraction at WordPress. This can be interpreted as both the normative power of the factual at work or a general skepticism against outside influences which any proponent of change must surpass to achieve adoption of an innovation.

Categorizing these, we find the first three (legacy constraints, licensing issues and missing interfaces) to fall into a category of product-centric or technically-motivated arguments. These certainly cause additional effort for settling them, but are more likely to be overcome by an innovator than the second type of concepts which is more psychological or socially induced (loss of expressive power and structural conservatism).

It must be noted that the original idea to use annotations as a process innovation to move to a more secure architecture in an evolutionary fashion did not prevent the project from abstracting from this proposal and discuss the implicated changes instead. Rather than discussing the viability of using annotations as a guide to aid a refactoring of the project, members discussed whether they were interested in the particular refactoring of introducing a database layer. As such, all five barrier concepts primarily occurred as opposition to the change of the architecture rather than the idea of annotations.

<sup>194</sup>In the case of WordPress the use of PHP5 over PHP4 was highly recommended because PHP4 is no longer maintained since August 2008.

<sup>195</sup>Thiel called this concept *fear of loss of power*, a term which connotes too much the fear of the leadership to lose control over the project, which is not implied.

This insight can be related to the discussion of enactment scopes (see Section 5.4), where I hypothesized that proposing an innovation to be applied in a limited set of situations would be more likely to be successful than in a general set of situations, but cautioned that the innovator still had to overcome adoption barriers regarding a single enactment. Similarly, I think that an innovation such as annotations for evolutionary change should be proposed with a specific change in mind rather than discussing the general ability of the innovation to aid in large-scale refactorings, but that this specific change will still be scrutinized intensely, as this study with WordPress showed.

**Hypothesis 17** *Process innovations with an impact on general project attributes should be proposed initially with specific but uncontroversial goals.*

As a second insight about process innovations, the episode at Mambo points to another difficulty with process-centric innovations: While Thiel wrote a set of annotations to facilitate the transition to a more secure code base and achieved the inclusion of these into the project code repository, the process innovation failed to attract developers which would then perform the actual changes. Such a problem might not exist for larger and more popular projects, where work—once designated—can be picked up by a large community of developers, but for a small or medium-sized project it might be difficult to find even a single developer interested in working on the proposed task. Thiel hypothesized that the innovator would likely have to invest time of his own to jump-start the work implied by the process innovation, but a more radical hypothesis can be given as well:

**Hypothesis 18** *The smaller a project is, the less likely is it that the effort in structuring and managing work via a process innovation is well invested.*

Thus, if Thiel would have foreseen that nobody was available to work on resolving annotated issues, he could have spent his effort immediately on resolving the issues instead of annotating them<sup>196</sup>. One possible counterpoint to this argument may be that the innovator is not so much interested in achieving results fast, but rather sees the inclusion of annotations in the code base as a step to increase the likelihood of the refactoring being achieved eventually. For instance, in the study of introducing automated regression testing at FreeCol (see Section 7.4), developers successfully created test cases to communicate issues in the code and increase the probability of them being fixed (see Section 7.4).

As a last point regarding the introduction at WordPress, an insight about the relationship of decision making on the mailing list to the decisions made by project leadership can be gathered. When Thiel first ran his proposal by several individual core developers and project leaders, he received a unanimous show of support for the idea. Yet, when proposing on the project's mailing list and receiving some criticism for the innovation, this support did not materialize in any way. One way of interpreting this discrepancy in reaction from a private to the public context is to assign an inclusionist attitude to the project leadership and an exclusionist one to the mailing list. In a popular project such as WordPress, the collective discussion on the mailing list appears to serve the role of filtering inappropriate ideas, while at the same time the project leadership can assume a motivating and encouraging stance towards novel developers and ideas. For the innovator, the consequence is another clarification of the suggested strategy to involve project leadership:

**Strategy 20 (Peel the Onion Inside Out)** *After attaining support from the project leadership, opinion leaders on the mailing list outside the leadership should be targeted next and individually.*

Also drawing from the episode at Mambo where the project leadership did encourage the innovator while lack of participants made a successful introduction unlikely, we might conclude that an inclusionist attitude of the project leadership should still be interpreted cautiously by the innovator, and thus arrive in combination at:

**Hypothesis 19** *Initial reactions by project leadership are likely to have an intention apart from assessing the technical merits of a proposal: Positive reactions mean to motivate, negative reactions mean to verify commitment and filter bad ideas.*

<sup>196</sup>He might have still done so to structure and manage the locations of all issues for him.



### 7.5.1 Summary

This study has contributed a second instance of an innovation episode which includes the two early phases of perceiving a problem and designing an innovation to solve this problem (the first being the information manager 7.1). In particular, the innovation was designed to transform a radical change (in this case the introduction of an architectural separation of concerns for database access and output sanitation) into a set of independently attainable tasks for a more evolutionary approach. The evaluation of this process innovation as a case study in two Open Source projects showed that (1) numerous barriers exist in the concrete case of introducing an architectural facade for database access such as *legacy constraints* or a perceived loss of expressive power and (2) that the possibility to recruit developers for participation in concrete tasks implied by an innovation must be critically analyzed for medium and small Open Source projects. For the innovator wishing to achieve a radical refactoring, this study suggests that a project consensus or substantial leadership support should first be acquired for implied changes of the refactoring. If successful, the potential for other developers to participate in the refactoring should be probed and if enough potential developers can be found, then a set of task annotations can be used to transform the radical change into well-manageable steps, solvable over time.



## Chapter 8

# Practical Advice

The goal of this chapter is to focus the gathered insights into practical advice for future innovators. This is done using three parts: First, the introduction of five common types of innovations will be discussed from an introduction-centric perspective (see Section 8.1). An innovator looking to convince a project to introduce a new source code management system (Section 8.1.1), to participate in the Google Summer of Code (Section 8.1.2), switch licenses (Section 8.1.3), adopt a new naming scheme for releases (Section 8.1.4) or better bug tracking processes (Section 8.1.5), will hopefully find these sections particularly helpful. Second, one discovered meta-principle of the origin for several innovations is presented (see Section 8.2). Third, results will be condensed into one accessible document for use in practice (see Section 8.3).

### 8.1 Common Innovations and their Introductions

The discussion up to this point has concentrated on understanding the introduction of innovation from a generic perspective. With this approach it was possible to explore individual facets of conducting an introduction deeply, but the understanding of any particular innovation remained shallow. In the following sections five common innovations will be presented holistically.

Methodologically these sections are in between the case studies presented in Chapter 7 and the main results in Chapter 5, because no in-depth Grounded Theory was developed on each innovation, yet combining the results of all studied episodes featuring a particular innovation provides more versatility and breadth in results.

#### 8.1.1 Source Code Management

New source code management (SCM) systems<sup>197</sup> represent the most prominent type of innovations introduced in the observed sample of projects. In particular the transition to the *decentralized version control system (DVCS) Git* [230] stands out as far-reaching for the projects which adopted it. Three of the thirteen analyzed projects switched to Git in 2007, one (*gEDA*) from CVS and two (*KVM*, *ROX*) from Subversion. One project (*U-Boot*) was already using Git, having switched in 2006, and another (*GRUB*) contemplated the migration to Git from CVS but was stopped by its maintainer [grub:4116] before eventually adopting the decentralized DVCS Bazaar in 2008. The project *FreeDOS* switched from CVS to Subversion. From the other projects in the sample, *Xfce* and *Request Tracker* both switched to Git after the observation period and *Bugzilla* moved to Bazaar in February 2010. This means that the

Which projects migrated?

<sup>197</sup>The terms revision control and version control are synonymous to SCM.

majority of projects are now using distributed version control and only two projects remain using CVS (*MonetDB*, *Bochs*), and three using Subversion (*FreeDOS*, *ArgoUML* and *Flyspray*).

SCM systems are important innovations for Open Source projects as they are the focal device for managing the source code of the software the project is producing. The capabilities of such a system and the workflows it enables can thus influence the collaboration and management to an important degree. The traditional use of a centralized SCM system such as CVS for instance has important implications on defining who constitutes the project and who wields power over this definition. In this centralized approach, a SCM system is run on a central server, accessible only to “committers” to the project for writing. The administrative ability to assign the role of a committer to a person and therefore grant write-access to the project’s source code represents the highest level of structural power in many Open Source projects [229].

The migration from centralized CVS to centralized *Subversion (SVN)* in the project *FreeDOS* gives some insights how capabilities can vary even between systems using the same paradigm. As proposed by one peripheral participant of the project, Subversion improves handling of platform depending line endings in text files and allows offline operations such as reverting local changes or creating a patch between local modification and working copy base version [freedos:4824]. While these advertised improvements over the status quo are not big and the influence of the proponent as measured by his yearly activity not large, the migration is soon executed [freedos:4837].

Looking to other discussions about Subversion and CVS, the following reasons could be extracted against the use of CVS or in favor of using SVN: (1) improved performance [462], (2) global revision numbers in SVN [freedos:4828], (3) ability to rename and move directories and files and retain version history [argouml:4820], (4) offline operations such as creating differences between base version and local modification and reverting local changes [freedos:4824], (5) atomic operations such as creating branches and committing related files together [geda:2997], (6) integration with bug trackers using metadata in systems such as Trac [geda:4231], and (7) ability to lock files during complex operations [geda:2997].

Had this thesis been conducted in 2005 or 2006, then migrations from CVS to SVN would have dominated the list of episodes as a common innovation introduction. So, how could Git as another innovation supersede SVN? The answer to this question lies in the paradigm shift offered by distributed SCM systems over their centralized counter-parts by offering the capability of replicating an existing repository locally. This gives the “cloned” copy the same technical capabilities as the original one and makes the designation of a central repository a purely social convention.

What are the  
benefits of  
distributed  
SCM systems?

This has several important implications and advantages: (1) It is no longer necessary to designate commit access to developers. Rather, anybody interested can clone from the project’s designated master repository or from other developers. Collaboration and sharing of contributions becomes thus easier for non-members without endangering the integrity of the project source [rox:9368]. Since the master repository, by social convention alone, contains the source from which the software of the project is built, the project leader is nevertheless in charge by controlling this central repository. At the same time a leader does not constitute a bottleneck, because social convention can be adapted without technical modifications [rox:9368]. (2) Since repositories are local, keeping changes private during a series of modification is easy and allows for many operations to execute much faster and while being offline than in the client-server model [geda:2889,4211]. This includes status commands, revert and diff, but also makes creating branches, switching to them [argouml:4772] and commits very fast. (3) As the existence of decentralized repositories is the norm, distributed SCMs include better support for the complex operations necessary to manage situations arising from parallel development. In particular merge tracking and tree conflict resolution, which were lacking in both CVS and SVN in 2007<sup>198</sup>, were implemented well in Git [470] improving the maintainers’ lives.

De Alwis and Sillito have recently discussed several cases in which Open Source projects were planning to move from centralized to decentralized source code management systems [135]. By looking at documents from wikis, website, discussion forums and in the case of Python extension proposals [24],

<sup>198</sup>Subversion has since added support for merge tracking and improved for some tree conflict cases.

the authors extracted a largely identical set of anticipated benefits of why projects were considering a migration.<sup>199</sup>

On the negative side of distributed SCM systems and in particular Git the following reasons are given: (1) Portability is reduced in comparison to CVS and SVN [grub:4132] for systems other than Linux. (2) The learning curve is steep [geda:5582], because of the tool's flexibility and by its decentralized nature, which allows for more complex relationships between repositories than the centralized client-server systems, (3) commit identifiers are no longer meaningful in the decentralized case, as hash values are used instead of increasing numbers [135].

Disadvantages  
of DVCSs?

It is hard to assess to which degree environmental effects influenced the popularity of Git. Project members for instance remarked that related Open Source projects had made the switch and that adopting Git would consolidate their tool set [geda:4211] rather than diversify it. Also in the case of KVM, which has some source code as part of the Linux kernel, the decision by Linus Torvalds to develop and use Git must have influenced their decision to adopt Git. Networking effects can thus be seen at work, adding to the technical advantages. Excitement about technical innovations is also involved [grub:4132], which might add some irrationality to the process of making a migration decision. Last, Git provides tools for forward and backward compatibility discussed in Section 5.6, which increase innovation attributes such as trialability and incrementality [rox:9373] of a migration. For instance, it is possible to set up a Git repository which tracks an existing SVN or CVS one [geda:2784], enabling project members to experiment and test Git without changing their existing infrastructure.

Other  
influences?

If we next consider the innovation decisions involved in introducing a DVCS into an Open Source project, we find that in all three cases in which Git was adopted, the maintainers and project leaders made the final decision of migrating in a unilateral way [rox:9368,geda:4123], in the case of KVM even post-hoc [kvm:1399]. Keith Packard of X.org discussed a reason why “tyrannical” selection of an SCM should be preferred and was in the case of X.org: He notes that the large amount of technical knowledge necessary to make an informed decision is likely missing in any discussion on the subject and that familiarity would thus trump technical capability unless project leadership steps up [396]. Only in gEDA is the introduction of Git the result of a long campaign of two core developers in improving their personal development work and eventually convincing their maintainer and the project to switch. They offered training [geda:3988], set up intermediate solutions for trials [geda:2799] and advertised the use of Git in many situations [geda:3068,3954].

How do  
projects  
decide?

In all three projects the migration of source code then is executed quickly and partially (see Section 5.3) with little problems. Conserving existing history, tags and branches is reported as the biggest challenge requiring either manual work or adapting import scripts [rox:9373]. De Alwis and Sillito identified one important additional challenge to a transition which did not appear in the episodes in this thesis: Legal implications of a switch must be considered, because a decentralized source code management system makes it much more difficult to comply for instance with the demand to remove certain source code globally.

How is the  
migration  
performed?

Following the technical migration, the adoption of the innovation and the adaption of the new system to existing processes and vice versa need to take place. Developers first need to become accustomed to new tools and processes, and relearn even simple operations such as creating a patch [geda:5017]. One mailing list participant estimated it to take in the order of “1 to 2 years” [geda:5426] to reach the same level of proficiency as with the tools he was currently accustomed to. To illustrate the complexity, consider the project *KVM*, which used Git to contribute to the Linux kernel following a two-stage process: First patches and modifications are developed, reviewed and aggregated inside the project KVM and then Linus Torvalds is asked to pull these into his kernel repository. This leads to increased

How is Git  
adopted?

<sup>199</sup>De Alwis and Sillito identified as motivation (1) to provide first-class access to all developers with the intent of making it easier for participants without commit privileges to work and removing “much of the politics” of having to assign them in the first place [135] (this is further discussed in Section 8.2), (2) to commit changes together atomically (a feature missing from CVS), (3) to improve the ability for automatic merging to encourage developers to stay closer to the trunk of developing and thus lessen the cost of merging branches back, (4) to reduce the amount of time necessary to open a branch so that developers can do extensive experimentation more easily, and (5) to support committing while being offline, which is trivial in decentralized source code management as the repository is local.

complexity for instance when figuring out which commits to the kernel have originated from the KVM project [kvm:4641] and proved challenging even for the maintainer of the project, who failed to correctly send pull-requests to Linus Torvalds in the beginning of using Git [kvm:1578,2414].

Taking a project-wide look though, only minor adoption problems could be found and most developers quickly become accustomed to basic Git usage. This is primarily because for many project members the processes associated with using the SCM remained identical. For instance, even with Git it is possible to create patches as peripheral developer and send them to the maintainer or continue using a central paradigm in which a master repository is committed<sup>200</sup> to by authorized project members [geda:4335]. Only if the project chooses to adapt its processes must new safeguards and rules be established. This is most visible in the project U-Boot, which switched to using Git prior to 2007 and following the Linux Kernel model, created the role of a custodian to take on responsibility for certain modules of the code base. Contributing to U-Boot then becomes a two-stage process, in which a contribution is first incorporated into the custodians' repository and only then accepted into U-Boot [uboot:25863].

Organizational processes, which become necessary, involve in particular the advertising of new repository location in transition times [geda:4747]. Confusion about which repositories to use can easily arise when old repositories are still online but no longer used to build software from. Also, the distributed nature makes it possible for developers to start keeping their own repositories and switching to a workflow where changes are pulled from them. Managing and sharing the URLs of such repositories makes up a large portion of initial communication related to Git, e.g. [rox:9519,9373,geda:4307]. Local repositories appear also to have the advantage of giving developers time to let modifications mature, because it relieves the pressure to commit changes before starting work on a next task [geda:5301]. As a downside, this might lead to modifications not being applied, because developers feel too little pressure to fight for inclusion in the master repository [geda:5302].

Other administrative processes are mostly reduced when using Git in a decentralized fashion, because it is no longer necessary to assign commit rights in the decentralized usage model. Projects which adopt centralized or hybrid models, such as gEDA and U-Boot of course retain some of these processes such as adding SSH keys for committers [uboot:29752,geda:4219].

To summarize, we find the following usage models of a DVCS:

- **Centralized**—As adopted by *gEDA*, in which the project continues to maintain a centralized repository with commit access for project members. This reduces operational overhead related to reviewing and pulling changes but at the same time reduces opportunities for code review [geda:4335]. Administrative processes for managing SSH keys remain necessary [geda:4219].

Jørgensen and Holck argue that centralized source code management can not be scaled indefinitely as the development branch becomes “overloaded” by concurrent operations with conflicting goals [274].

- **Decentralized**—As adopted by *ROX*, in which the project maintainer retains exclusive control over the master repository and only pulls contributions into his repository after he has been requested to do so. As the maintainer of the master repository constitutes a bottleneck in this model, it can become necessary to send reminders about pending changes [kvm:3534].

One important advantage of the decentralized model is that pull-requests constitute a natural opportunity to perform pre-commit patch reviews [kvm:4712].

- **Hybrid**—As used by *U-Boot*, in which the project uses a mixture of centralized and decentralized version control. Several “custodians” receive commit rights to the central repository, which requires setting up SSH keys on the server [uboot:29752], and are in charge of aggregating the decentralized development. Using such a hybrid model requires that “custodians” in the U-Boot slang or trusted lieutenants in Linux terms are easily identifiable by external parties, for instance by being listed on the project homepage, or otherwise confusion about who can commit directly and who will need assistance can arise [geda:5236].

<sup>200</sup>The correct term is “pushed” in DVCS lingo.

A detailed discussion of development and administrative processes resulting from distributed SCM can be found in [528].

Last, it should be noted that there is one project in the studied sample which achieved the benefit of providing “first-class” access to non-members without adopting a DVCS. In the project *ArgoUML* the participation in the *Google Summer of Code* program had shown that students could be assigned write-access to certain directories in the centralized repository, which would give them all capabilities of managing their code in version control and protect the project from low-quality commits at the same time. Since this model worked so well for the project, the maintainer decided after the summer to extend commit rights to this particular directory to all who would be interested by creating the explicit role of a contributor [argouml:5634].

Contributors at  
*ArgoUML*

#### 8.1.1.1 Two-tier version control

This paragraph presents one interesting aspect of using an adaptable SCM for the master repository. In the project *gEDA*, one core developer for instance used a Git repository in front of the master CVS repository of the project for working on a complex feature. Doing so, he could ensure that code committed to the branch in CVS designated to this feature was in working condition and at the same time make frequent commits via Git to track his own progress [geda:2918]. Such a combination of a “stable” repository and an “experimental” repository using different version control systems is also reported for the project FreeBSD, which is using CVS for their main repository and commercial Perforce for experimental work [319].

Such an officially encouraged use of two-tier version control can also help with differentiating access control: The core of the project is only accessible for writing to project members with commit rights via the centralized version control, while access to the second tier is given away freely to anybody interested. Thus, the project core is protected, while peripheral participants can use a sound work environment.

#### 8.1.1.2 Advice for the Innovator

For the innovator looking to successfully introduce a new SCM, I can recommend as the gist of the observed episodes the following four points: (1) Precisely define the goals of the introduction and chose the SCM and the usage model accordingly. For larger projects, hybrid models of combining central and decentralized processes seem to work best. (2) Focus on convincing the project leadership first, as all introductions of new SCM systems in this thesis show that the role of project leadership is central in achieving the execution of the migration. (3) Execute the migration as quickly as possible to prevent stalling the development process but at the same time (4) stretch the adoption period before and after the migration as possible, in particular by using adapters (see Section 5.6) and migrating partially (see Section 5.3), to achieve incremental learning and adoption matching the pace of the project’s members.

### 8.1.2 The Google Summer of Code

The Google Summer of Code (GSoC) “is a program which offers student developers stipends to write code for various open source projects”<sup>201</sup>. Google started the program in 2005 and named it after the Summer of Love of the sixties’ counter-culture in San Francisco [72]. From initially 400 students per year, the program has continued since on a yearly basis to reach 900 students in 2009. The student receives a stipend of 4,500 USD for the three months of participation and the mentoring project gets 500 USD.

Taking part in the Google Summer of Code proceeds roughly through four phases:

<sup>201</sup><http://code.google.com/opensource/gsoc/2008/faqs.html>

1. To introduce the GSoC as an innovation, a project needs to apply to Google with a set of project ideas. This means that the project needs to execute at least three things: (1) Find and write down ideas, (2) find mentors, and (3) write an application.
2. After being accepted (the GSoC is one of the rare innovations in which introduction success is not purely in the hands of the project itself), the project needs to attract applicants to these proposals. Students submit their applications to Google, which then asks projects to assess them. Based on this information Google will accept and assign students to projects.
3. When students have been accepted by Google and assigned to the project, the project needs to mentor them to complete the tasks over three months in the summer and most importantly integrate their work into the project code base.
4. After the summer, mentors in all projects participating in the GSoC are invited to the mentor summit to discuss the projects and share their experience [geda:4937,argouml:5511].

Six of the 13 projects studied in this thesis (*ArgoUML*, *Bugzilla*, *gEDA*, *KVM*, *GRUB*, and *Xfce*) discussed applying to the program and all four which decided to participate (*ArgoUML*, *Bugzilla*, *gEDA*, and *GRUB*) were accepted by Google. Most successful was the introduction for the project *ArgoUML*, which was *able to attract six students*.<sup>202</sup> *gEDA* was assigned two students, *GRUB* one, and *Bugzilla* none. *Bugzilla* even failed to attract a single student to apply for one of their proposals [bugzilla:6265], which one of the maintainers attributes to lack of advertising [bugzilla:6268].

Who participates? The nationality of the nine applicants in the observed episodes indicates that GSoC is in particular interesting for students from developing countries (Eastern European and BRIC nationals) and Spanish speaking ones. Comparing with the official statistics published by Google<sup>203</sup> reveals though that participants are predominantly from developed countries.

Why do projects participate? Looking at the reasons why each project participated it is surprising that most projects do not rationalize this on a high level. Beyond “last year was a success” [argouml:4846] there is little argumentation why to participate. Rather, projects jump right into assembling an application by gathering ideas and mentors [argouml:4846,bugzilla:6200,geda:3021,3026,3064,grub:2688]. This seems rather a short-sighted approach, because the basic question to ask is whether the project thinks about the GSoC as an opportunity to give new community members the chance to spend a considerable amount of time to get to learn about the project and contribute to it even after the GSoC or whether students are just seen as paid workers over the three summer months, who are likely to leave again after the project ends.<sup>204</sup>

Influence of being part of a Foundation? One noticeable condition which influences the decision to participate and the success chances in the GSoC was the affiliation of the project in an *Open Source foundation*. This is because the foundation can apply as a representative of a set of projects to Google, making the application particularly hard to resist, if the project belongs to one of the major foundations such as Mozilla, the GNU project, or Apache. This is in particular visible in the case of *GRUB*, which as part of the GNU project participates almost without any effort in 2007, since it only has to provide proposals for students to apply to, while the larger GNU project handled the application. At the same time, two drawbacks must be named for the participation as part of a foundation: (1) If the foundation does not participate, then the project is likely bound by this decision. This happened in the project *GRUB* in the first year of the GSoC. Google had required projects to be “Open Source” in their first iteration, violating the GNU projects preference for calling itself “Free Software”.<sup>205</sup> *GRUB* as part of the GNU project thus could not participate easily. (2) Participating as part of a larger foundation runs the risk of applications by students competing which each other. This happened in the project *Bugzilla* in 2007, which is part of the Mozilla Foundation. Despite *Bugzilla* being a well-known project, the other projects of the Mozilla Foundation such as *Firefox* and *Thunderbird* attracted all the student interest. The core developers were still discussing the

<sup>202</sup>The progress of these students was discussed in detail from a path dependence perspective in Section 6.1.2.2.

<sup>203</sup><http://spreadsheets.google.com/pub?key=p6DuoA21JToKmUzoSq6raZQ&output=html>

<sup>204</sup>In the project *gEDA* these considerations were made in their participation in the GSoC in 2008 ([http://www.geda.seul.org/wiki/best\\_practices](http://www.geda.seul.org/wiki/best_practices)).

<sup>205</sup><http://www.gnu.org/software/soc-projects/guidelines.html>



danger of this [bugzilla:6218], but the Mozilla Foundation had already entered the project under their banner [bugzilla:6246,6248].

Most projects are eager to participate and disappointed when their application fails [bugzilla:6265,xfce:14963]. Previous experiences also appears to influence the projects' decisions to participate: The maintainer in the project Xfce for example was able to halt all interest in participation when he remarked that he was unsure whether the project fit with Google's targeted audience [xfce:13244]. Positive experience equally seems to have a large influence, as for instance in the project ArgoUML, in which there was no discussion about whether to participate after previous years had already been successful [argouml:4846].

Considering the ability of the GSoC to serve as a method for attracting long-term developers to the project, results are mixed for the projects. On the positive side, many students earn their commit rights and are called core committers at the end of their GSoC term [argouml:5513]. But on the negative side, it is unclear whether these students really make ongoing contributions to the project, and an equal number of students fail to become members. For instance, in the project ArgoUML, six students won GSoC scholarships in the summer of 2007 [579], five of which finished their work [578], three of which received developer member status beyond the GSoC [argouml:5513], and only one showed sustained participation after the end of the summer.

Do students become members?

Looking next at the GSoC as a program which contributes three man-months of work to the Open Source projects and assessing the quality of the contributions, it can first be noted that Google reports over 80% of all cases to be successful in each year by their own definition.<sup>206</sup> Results in the observed projects are slightly lower. Only one of nine students was officially reported a failure and thus did not receive the full stipend. But looking at the projects' own assessment of the contribution, for instance based on whether it was possible to integrate the work of the student, the success statistics are reduced to six of nine. In gEDA, one of the mentors reported "great" results [geda:4844], while the other reported the project to have been "less successful" [geda:4848], indicating that the project was too ambitious for the student. Of the six students in ArgoUML, beside the one student who failed, another finished the GSoC, but his work never became integrated into the trunk of ArgoUML development. In the project GRUB, the single student made good progress [grub:3316,3529] on his assignment, but disappeared at the end of summer [grub:3545], no longer reacting to questions [grub:3900]. Only in February 2008, after more work had been done by other project members, could the task be integrated [grub:5293].

Are contributions valuable?

One possible explanation for these failures to integrate the work of the students could stem from the use of the American college calendar to place the Summer of Code, which collides with exam weeks or semesters in European universities [argouml:5178,5354].<sup>207</sup>

Participation in the GSoC interacts with other innovation introduction in particular based on the processes necessary for managing the students' work. In both the projects gEDA and ArgoUML this entailed proposing to designate a branch in the source code management system for each student. Students were then asked to use this branch for their work, which later was then to be integrated into the trunk of development.

Interaction with other Innovation Introductions

In both projects these proposals interacted with other introduction episodes. In the project gEDA, the proposal was made in the context of introducing *the decentralized source code management system Git* [geda:3068] and can be seen as another situation in which the innovators were able to demonstrate the versatility of their innovation. In the project ArgoUML, the proposal to use branches in Subversion for the students' work [argouml:4976] came two months after the use of branches had been rejected in a preceding *episode about the use of branches* between two core developers and the maintainer. The necessity to manage the students' work then overrode the previous objections and became the starting point for learning to use branches even for the opposing core developers.

<sup>206</sup><http://code.google.com/p/google-summer-of-code/wiki/ProgramStatistics>

<sup>207</sup>Collisions with vacation plans of mentors are rare and occur mostly in the beginning of the students' participation [argouml:4921].

### 8.1.2.1 Advice for the Innovator

For an innovator interested in introducing the GSoC in an Open Source project there are seven challenges to overcome: (1) The project needs to be convinced to participate. (2) Mentors and ideas must be gathered. (3) A successful application to Google must be written. (4) Students must be attracted to apply for the ideas. (5) Students need to be supported during the three months. (6) Their work needs to be integrated into the existing code base. (7) Students should be convinced to stay as project members active in the community. While the first four are necessary to achieve participation in the GSoC program, the last three are equally important to be successful at participating.

From the perspective of the innovator it is important to realize that of the four initial steps, all except the *organizational innovation decision* can be performed by the innovator (with some luck regarding the application at Google). Thus, if the innovator is prepared to write the application, collect ideas and motivate mentors (or be one himself), then there is little reason to believe the project would reject the participation. Relying on the maintainer to take care of all these aspects is unlikely to be successful.<sup>208</sup>

How to  
successfully  
apply?

On the question how to be successful with the application at Google, few hints emerged during the observed episodes, except that applications as part of a larger foundation are more likely to succeed. Once the application is a success, advertise the offered project proposals to students with the following in mind: (1) The number of students who can work on the project is mostly based on the number of successful applications and thus lots of information about them should be spread to attract many suitable candidates. (2) Students can only apply for proposals, and thus a large number of interesting and high-quality proposals should be written, so that there are no collisions of several capable students on the same tasks. Such occurred in the project ArgoUML, which had only six proposals, yet 42 applicants, most of which chose the same proposal [argouml:4975]. (3) Getting to know the students better before the GSoC or even letting project members apply who are students is given as one of the possibilities to increase the chances that they are going to stay in touch with the project beyond the program.

Supporting and mentoring the students during the three months has occurred mostly off-list in the studied projects and advice is thus difficult to give. In the project ArgoUML, the project leadership mandated weekly reports and the use of dedicated branches by the students, both of which are advisable yet are no panaceas for successful participation. Rather, these reports show that there is often little interest in the work of the students and no feedback for their activities. Deciding to participate in the GSoC should also include making an commitment to support students in case they win a GSoC scholarship. This implies reserving at least two and better four hours per week for performing patch reviews, explaining existing code and giving general advice.

The existing experiences show that the end of the three months often also mark the beginning of a new semester for the students, who then no longer have any time (and obligation under the program) to continue on their tasks. If integration of the work of the students is to be achieved, it is thus crucial to start this work at least two weeks before the end of the summer months.

Last, participation in the GSoC in one year should also include collecting and summarizing experiences with the program. Since participation is possible yearly, gathering best practices and standardizing procedures can make future participations easier and more successful. For instance, task proposals which have not been selected can be carried over to the next year, or branching schemes in the source code management system can be reused.

### 8.1.3 License Switching

Open Source licensing is a complex topic which stretches the boundaries of the knowledge of project participants and connects to little-known issues of copyright law and legal matters in general. For a

<sup>208</sup>Of the six projects discussing the participation, only in two was the application conducted by the maintainer, in two the maintainers did not have time and in two the maintainers did not have to apply because their foundations did.

basic overview refer to Section 2.3.3. In the studied sample of thirteen projects, five episodes on license switching could be found, three of which occurred in the context of *GPL version 3* being released in June 2007. These will be discussed first, before turning to the two other episodes and summarizing insights.

When the *GPL version 3* was released in 2007, projects using the GPL were faced with the question whether to update their license or remain using GPL version 2. *GPL version 3* had added better protection of free software for embedded devices and against patent holders and improved license compatibility notably with the Apache Software License. Yet, changing licenses requires gathering the consent of past contributors to change the license of the work they hold the copyright on. This can be “a nightmare” [argouml:5554] as contributors might “have dropped off of the face of the ‘net” [geda:4691] and each objection to the copyright change entails rewriting the code the person wrote [flyspray:5535]. This situation is made more difficult by the “or-later”-clause used in the GPL, which permits redistribution and modification of the software under the current or any later version of the GPL. In none of the projects could the question be resolved whether the “or-later” implies that individual source code files can be modified without the consent of the copyright holders to use the GPL v3 [geda:4691].

In the project *Xfce*, a long-term developer asked whether the project would want to change its license to “GPL version 2 only” by removing the “or-later”-clause [xfce:12625] to avoid the uncertainty involved with the new version. In a highly parallel discussion,<sup>209</sup> the difficulties of understanding the legal subtleties was apparent and the proposal is rejected in the end by most participants without the innovator resurfacing.

*GPLv2 only at Xfce*

In *gEDA*, a discussion was held to find out whether project participants felt a switch would be beneficial and possible for the project. Yet, the discussion quickly decayed into the legal aspects of switching and the implications of GPL v3 usage and lost focus. In the end several separate discussions remained with unresolved questions and the episode ended abandoned by its innovator.

*GPLv3 at gEDA*

Third and last, the maintainer of *GRUB* proposed to switch to GPLv3 shortly after it was officially released. Here again the discussion was highly parallel and uninformed. But this time, the innovator reappeared after one week, consolidating the opinion in his favor and unilaterally deciding to make the change to GPLv3 [grub:3380]. This episode may be seen as an indicator of the maintainer’s influence in a project, but then also reveals another difference to the other episodes: The project as part of the GNU project used copyright assignments to the FSF to keep copyright in the hands of a single legal entity [geda:4692]. This enabled the maintainer to execute the migration easily [grub:3380].

*GPLv3 at GRUB*

The adverse effects of a lack of a copyright assignment is illustrated in the next episode inspected: In the project *ArgoUML*, an outdated version of the BSD license was being used, a situation a core developer wanted to rectify as part of *ArgoUML* joining the *Free Software Conservancy*. Discussion was animated and involved several conversations about past implications of using the permissive BSD license rather than more restrictive licenses such as the GPL [argouml:5569]. In particular the fork of *ArgoUML* into a commercial product from which no contributions were ever returned stirred bad memories [argouml:5568]. Controversy then built between those who would prefer a GPL license to protect *ArgoUML* from forking by companies [argouml:5521] and those who would have preferred to stick to permissive licenses such as ASL or modified BSD [argouml:5582]. Finally, the project decided to switch to the weak-copyleft EPL in December 2007 as a middle ground [argouml:5726]. Yet, only in November 2009 could an incremental update path be decided on, in which newly created files get added using the EPL license and existing ones are modified to contain both the EPL and BSD license header.<sup>210</sup> This is despite the fact that one core developer argues that the use of the BSD license would permit any license change by its permissive nature [argouml:5555].

*License switch at ArgoUML*

The fifth and last episode of a licensing switch occurred in *gEDA*, when a developer criticized the unclear

*License for schemas at gEDA*

<sup>209</sup>In this case there were seven parallel replies forked off the initial proposal.

<sup>210</sup>See <http://argouml.tigris.org/wiki/License>.

legal situation when creating proprietary products using the software created by the project.<sup>211</sup> One core developer took charge of starting a discussion and achieved to open a Garbage Can (as discussed in Section 6.2.2.2), which similar to the previous episodes is lengthy and complex, with little satisfactory conclusion. Again the innovator refrained from participation in the discussion but, in contrast to the other episodes described above, skillfully summarized the discussion and extracted valid points to create a concrete proposal for resolving the issue. This proposal then was accepted by the project.

### 8.1.3.1 Advice for the Innovator

To summarize these five episodes, four important factors can be deduced for successful license migrations: (1) *Strong proponents* such as core developers and maintainers, who are willing to pursue the migration, are necessary. (2) Innovators must ensure the *ability to perform the migration*, which includes either the ability to acquire consent from all copyright holders, make use of a copyright assignment or using an incremental path for migrating the project. All variants incur a considerable amount of work [grub:3385,geda:3145] and it is thus most advisable to consider collecting copyright assignment early on in the history of an Open Source project. (3) *Overcoming the Garbage Can* is necessary because lots of confusion exists about the legal issues and discussion is quickly heated and personal. The innovators in the given episodes are noticeably refraining from engaged discussion (in GRUB and gEDA they are not involved at all), which might provide the foundation for a strategy to assume by the innovator: Keeping a low profile, providing solid information from external sources such as the FSF or the Free Software Law Center<sup>212</sup> to cool the discussion and refocusing the discussion by summarization and offering viable alternatives. (4) *Reusing existing legal documents* is highly encouraged, as the intricacies of legal wording is beyond the abilities of an Open Source project. In the project gEDA, the licensing switch for instance included replacing the word “font” with “symbol” in a given license text by the FSF [geda:3145]. Even such a small change raised the question whether the resulting text would have a different legal scope [geda:3372].

### 8.1.4 Version Naming

The inadequacy of the term Open Source “project” for a fluid collective of people working over years on software without a well-defined common goal and the limits of a budget and deadline has already been discussed in Section 2.3.2 on terminology. Yet, it appears that not being a project but rather an on-going endeavor with emergent goals and fluctuating activity has one interesting implication for *naming the different versions of software produced*.

First though, virtually all Open Source projects initially strive to attain version 1.0 as a typical milestone of achieving an envisioned goal. Such a goal can already be vague and implicit to the project, but often the project founder is able to focus the project on achieving a basic functionality with sufficient stability [geda:4072]. Beyond version 1.0, stability is commonly assumed in a release and thus completing a set of novel functionality, or a major internal refactoring is used as a criterion to measure the project’s advances.

Setting an internal criterion for a version increase and thus making the version number indicative of a notable increase in functionality or non-functional attributes is tempting [bugzilla:6580] and might be desirable to demonstrate progress and maturity [geda:4389], yet it turns out to be difficult for Open Source projects to achieve these within reasonable bounds of time. For instance, in Bugzilla, the transition from 2.0 to 3.0 took nine years [bugzilla:6568]. Similar delays can be found in the Linux Kernel with the 2.5 kernel series, which took two years to stabilize, or Debian when working on the 3.1 release [340, 4].

<sup>211</sup>The specific question was whether the inclusion of circuitry symbols from gEDA in a design for a printed circuitry board would turn this design to Free Software as stipulated by the GPL [geda:3108].

<sup>212</sup><http://www.softwarefreedom.org/>

One way to understand how functionality-bounded releases cause delays for the Open Source development model is to consider that with no formal task and project organization keeping to schedules is difficult. Additionally, new ideas will appear as a “never-ending stream of proposals” inflating the scope of a release [221], project members might reduce their engagement with the project at any time because of their volunteer nature and goals might have simply been set too ambitiously from the beginning.

This poses a problem to the Open Source quality assurance model: Since it relies on the users of a software to perform public testing, long periods between releases lead to a lack of testing opportunities and consequently to instability. Without “release early, release often”, there is nothing to look at for the “many-eyeballs” [417].

One possibility to counteract this effect is the introduction of intermediate releases to give users opportunity to test and thus maintain a stability focus. A versioning scheme in such a situation typically distinguishes stable and unstable versions, because such intermediate releases often include partially finished features and are destabilized by refactorings in progress.

Another solution to this problem was put in the public mind by the distribution project Ubuntu, which while based on Debian with a feature-based schedule, used *date-based releases* [529]. In this scheme, a fixed period of time is given for each release to accumulate new features followed by a stabilization period in which only bug fixing is allowed. If sufficient testing can be achieved in this stabilization period by “bleeding-edge” users working directly with the development version from version control, unstable releases become less important than in the feature-based schedule. This effect is also strengthened because less time is available to conduct radical changes (see Section 5.9), which might impair the quality of the software. If unstable releases are created in this model, then it is common to designate them as *milestones* (m1, m2,...) [argouml:4698].

In this model, between three to six months have been found to be a plausible length for the release cycle to balance development progress, users’ willingness to perform updates, overhead for the maintainer to execute releases and ability to gather user feedback [geda:4389]. This process based on time rather than features has also been adopted by Linux, which since the 2.6 series uses a *merge window approach*. Rather than using dates though, the 2.6 version prefix is still used, but has little meaning. In fact, with new releases of Linux roughly every three months [294], there have been complaints from distributors that releases are too fast and that a stable release should be produced on which to base long-term supported products. This has led to particular versions of the 2.6 series being picked up for long-term maintenance [kernel:939800].

The Eclipse project, as a second example, even made “milestones first” one central principle of their development process [201]. By strictly following a date-based process of stable milestones approximately six weeks to two months apart, the project has been able to simultaneously release a new Eclipse version each year in June across more than 30 independently organized sub-projects for the last five years.<sup>213</sup>

The use of date-based releases and *date-based version numbers* fits the Open Source development model due to two reasons: (1) Releases become more frequent, enabling greater participation of the public in quality assurance. (2) It accommodates the incremental nature of Open Source development, which provides a slow but steady stream of improvements without major changes to the software (compare Section 5.9).

Three episodes can be found in this thesis’s sample:

- In the project *U-Boot* the use of *date-based releases* was suggested, because of the level of maturity *U-Boot* had achieved, which should reduce the changes for “earth shattering discontinuities”, and to reflect the “continuous rolling improvement” [uboot:31353]. This suggestion from August 2007 is rejected by the maintainer as “stupid” [uboot:31363], but then adopted in October 2008 [uboot:46175]. Two plausible explanations of this reversal are that either date-based release schemes have become more popular in general or that *U-Boot* did mature as a project to reach the point where they felt date-based releases are more sensible. The latter explanation is supported

*Release naming  
at U-Boot*

<sup>213</sup>For an overview see [http://wiki.eclipse.org/Simultaneous\\_Release](http://wiki.eclipse.org/Simultaneous_Release).

by U-Boot having reached the point where the last five releases preceding the naming change were all in the micro-number range of 1.3.x.<sup>214</sup>

Release naming  
at Bugzilla

- In *Bugzilla* the release of version 3.0 after nine years of development prompts the maintainer to suggest that from now on minor-number version increases should be turned into major-number releases to designate a higher level of progress [bugzilla:6568]. In the ensuing discussion on the mailing list and IRC, the project rejects the change by identifying the problem of an overly ambitious goal for version 3.0 triggered by prototyping work setting the bar for a release too high [bugzilla:6576]. The project decides to maintain the version numbers using a major/minor scheme, yet aiming for faster increases in major number with the goal of release version 4.0 instead of moving to 3.4 [bugzilla:6578].<sup>215</sup>

Release naming  
at gEDA

- In *gEDA* the “first ever stable release” of the main library gEDA/gaf [geda:4360] triggers a discussion about the naming scheme for the following releases. In this discussion the project decides to adopt “major.minor.micro.date” as its new naming scheme [geda:4446], combining both numbered and *date-based versioning*.

Adopting a new version naming scheme has in principle little implications on the development processes in the project, but rather is symptomatic of changes to the release processes or perception of the projects development progress. In gEDA the change came due to the achievement of version 1.0 for the main library, while in Bugzilla it came as a response to a long major release cycle and an intention to speed it up.

A last and smaller point is the use of *nicknames for releases* as identifiers which are easier to remember and associate with. This again was popularized by the distribution Ubuntu, which uses alliterations of adjectives and animal names such as “Feisty Fawn” as release names.<sup>216</sup> In the project MonetDB, names of planets were proposed [monetdb:369], in U-Boot submarine names [uboot:46180] and colors [uboot:46183] were suggested.

#### 8.1.4.1 Advice for the Innovator

Adopting a date-based release scheme has been presented as both an indicator of and an additional step towards higher process maturity in Open Source development. This is because it emphasizes incremental improving over discontinuities and enhances the viability of the Open Source quality assurance model by providing frequent updates to users and reducing the reliance on individuals to complete their tasks. The following advice can be extracted from the episodes above: (1) Adopting a date-based versioning scheme should be easy if not for the personal preferences of a maintainer [uboot:31363] or the general perception of progress as still being too fast. In both cases the best strategy seems to be to wait until attitudes change and to suggest each time a release is overdue that date-based versioning could make the release process more reliable. (2) After having decided in favor of a change, distributions must be notified so they can incorporate the new naming convention to maintain their ordering of releases. In the Debian distribution for instance, the use of an “epoch” prefix to the version number is used when the new scheme results in numbers which are out of order with existing ones [geda:4363]. (3) Intended changes to the process such as increased regularity of a release might not happen automatically just by adopting a new naming scheme (though a slight correlation is plausible) and should be supported for instance by offering to take responsibility for packaging every other release.

#### 8.1.5 Bug Tracking Procedures

Bug tracking is pivotal in Open Source development since it is the primary process by which users and developers negotiate their expectations [445, cf.] towards the software leading to an informal

<sup>214</sup>See <http://www.denx.de/wiki/U-Boot/ReleaseCycle>.

<sup>215</sup>In February 2010 (two and a half years later) Bugzilla has almost reached version 3.6 instead of 4.0.

<sup>216</sup>See <https://wiki.ubuntu.com/DevelopmentCodeNames>.

specification and task assignment. In contrast to the mailing list and other systems for communication, the bug tracker is more structured by defining a bug's lifecycle, its boundaries<sup>217</sup> and stakeholders.

The importance of bug tracking in this role between users and developers can be derived from the "many-eyeballs" proposition of Raymond, which stresses the diversity of users' goals, platforms and skills as the sources of Open Source quality [417], but also has some more empirical backing. Zhao and Elbaum report 44% of projects stated that users found bugs which would have been hard to find otherwise [561]. Absolute numbers of bugs are less conclusive with almost 50% of projects reporting that users found at most 20% of their bugs [561], yet this likely includes bugs which the developers fix en passant while developing, which never make it into the bug trackers. Crowston and Scozzi found in analysis of four medium-sized projects that 78% to 100% of bug reports were made by external parties [118].

The common process for handling bugs is the following:<sup>218</sup> (1) A user *reports* an unexpected behavior. (2) A developers *confirms* the behavior in negotiation with the reporter as a defect or rejects it. (3) The bug is *assigned* to a developer.<sup>219</sup> (4) The defect is *repaired* by the developer. (5) The repair, commonly in the form of a patch to the mailing list, is *verified* to fix the bug. (6) The repair is *committed* to the source code management system. (7) The bug is marked *resolved*. But it is important to note that of these steps only the first (reporting), fourth (repair) and sixth (commit) are strictly necessary to see the bug repaired in the software, and the others might be omitted or forgotten.

Also, additional process steps might be beneficial for a project as can be illustrated by an example from the project *Bugzilla*. In this episode the maintainer proposed to add an optional step in the bug tracking process where a developer could ask for a *software design approval* to prevent spending time and effort on an implementation, which is later rejected on architectural grounds [bugzilla:6943].

*Design  
approvals at  
Bugzilla*

Having sane and efficient bug tracking procedures in place is important because it affects other parts of the project's daily work. For instance, in the project *MonetDB* it was decided that a new commit should be made to the stable branch instead of development branch if it related to a reported bug [monetdb:85]. This ensures that the stable branch will receive bug fixes and intermediate stable releases can be made while the development branch can be updated more easily by merging all changes from the stable branch periodically. Yet, to make this work the bug tracker needs to be clean and maintained or otherwise the decision whether a bug has already been reported is too time consuming.

Projects regularly struggle to keep their bug trackers maintained, often leading to take short-cuts or omit stages in the bug procedures. Crowston and Scozzi for instance report less than 10% of bugs having been assigned in the four medium-sized projects studied during the bug-fixing cycle [118]. This highlights the primary hurdle to success with many process innovations surrounding bug tracking: Getting project participants to be motivated to execute them is hard. The lack of enthusiasm for working on bug tracking leads to bug entries which are out-dated and makes the bug tracker even harder to use. Consider the following episodes:

In *ArgoUML*, it was for instance noticed that bugs would get closed to infrequently because another developer would have to verify the patch first. To improve this process, the *bug's reporter was given the right to verify* (and thus resolve) issues [argouml:4653].

*Bug  
verification at  
ArgoUML*

In *Xfce*, as another example, one peripheral developer proposed *to add milestone information to each bug*, i.e. assign to each bug the release this bug is intended to be fixed in. This is meant to reduce the number of bugs to look at when working on a single release. This proposition was not considered by the project, probably rightly so, since it would even add another attribute to be managed in the bug tracking procedures. Rather, a proposal within the next two weeks in the same project was more successful.

*Milestones for  
bugs at Xfce*

<sup>217</sup>Describing precisely how to replicate individual bugs and dealing with duplicate bugs are two important facets of creating and maintaining such boundaries.

<sup>218</sup>Crowston and Scozzi use a similar model of six elementary tasks: (1) submit, (2) assign, (3) analyze, (4) fix, (5) test and post, and (6) close [118].

<sup>219</sup>Because participants are volunteers, an assignment in a formal sense is not possible. Rather, an assignment is more a suggestion for a developer to look at a bug or in the case of a self-assignment an acknowledgment that one is intending to work on it.

*Cleaning the  
bug tracker at  
Xfce*

Here a peripheral developer, with even less influence, proposed to use the opportunity of a recent release of the software to clean the bug tracker of obsolete issues. This suggestion was accepted by a core developer [xfce:13128] and access rights are assigned to the innovator, which takes one week [xfce:13143] (the implications of such access restrictions is discussed in more detail in Section 8.2). It then took more than seven months until the volunteering developer started to clean the tracker [xfce:14195], highlighting how hard it is to find enthusiasm for such a managerial task, even for an innovator.

*Bug tracking  
observer at  
ArgoUML*

In the project *ArgoUML*, the search for a similar person had led to the innovation of defining an *observer role* akin to the information manager discussed in Section 7.1, who is explicitly in charge of managing the bug tracker and improving information flow between mailing list and bug tracker. Unfortunately the introduction failed because no volunteer responded to the call to fill the role [argouml:4752].

In the project Xfce on the other hand, the developer experienced a major set-back while cleaning the bug tracker: His strategy of commenting on pending bugs to ask whether the bug could be closed, disturbed the core developers' e-mail-based processes in which current bugs are identified by recent e-mail activity [xfce:14199,14202]. The innovator was essentially asked to close entries only which he himself can identify as already being fixed in the code or obsoleted by pertaining to an old version of the software. Bugs which cannot be sorted in this way are asked to be left alone.

*Reactivating  
the bug tracker  
at GRUB*

In all the projects we only find one episode in the project *GRUB* in which the bug tracking procedures have totally failed and thus need to be reinstated. The episode starts, when one core developer suggests that better tracking of tasks and requirements will be necessary to coordinate and speed up development of the upcoming version GRUB2 [grub:3236]. Similar to other situations the discussion on how to manage tasks gets quickly overwhelmed by concrete task proposals (see Section 5.4 on enactment scopes), which only then leads back to the original question how to manage them. A first discussion thread ponders the use of the bug tracker for tasks, which is rejected by the innovator as too inflexible for the requirements of grouping tasks, adding dependencies and generating reports on tasks progress [grub:3247]. In a second thread the topic of using the bug tracker comes up from a different route. Among the concrete tasks that one developer listed was the introduction of regression testing, which prompted one core developer to ask whether any actual bugs had been noted and to document them in the wiki [grub:3266]. The wiki is immediately rejected as inconvenient for such a task, which leads to the bug tracker being rediscovered as an option. Yet, the bug tracker is still filled with bugs for the legacy version of GRUB and finding an alternative solution is suggested. Here the maintainer forcefully steps in to defend the existing bug tracker but without offering a suggestion of how to separate it into different instances for GRUB legacy and GRUB2 [grub:3273]. One month later, another core developer asks whether and how to make the existing bug tracker usable for GRUB2, but is only referred to the maintainer's response, which offers no hint about how to proceed [grub:3391] and the initiative dies again. Almost four months later, a third attempt is made, now by a core developer involved in the initial discussion [grub:3902]. Yet this time, the maintainer again kills the discussion by stating his personal preference for maintaining bug reports in his personal mail box and tasks in the wiki [grub:3971]. Another three weeks later, the second core developer makes a finally successful attempt by proposing to clean the existing bug tracker of bugs for GRUB legacy and thus enabling it to be used for GRUB2 [grub:4047]. The key to success in this fourth proposal was that the core developer did not become intimidated by a lack of response to his first e-mail but rather took it as a sign of consent and thus only asked for objections a week later and then proceeded [grub:4071].

*Tracking  
developer  
activity at  
MonetDB*

The link between bug tracking and task tracking has appeared quickly in this episode at GRUB and should be clarified here. Each entry in a bug tracker first is associated with a deviation from expected behavior of the software, which gets translated into a defect if this expected behavior is confirmed to be part of the specification. Then a close connection can be made to an associated task of repairing the issue, which leads to the idea of organizing tasks in the bug tracker. In the project *MonetDB*, it was for instance suggested that developers should create bug reports before starting the task of fixing a defect they found to create a link to potential failures of the software and make the development process more transparent [monetdb:165]. The suggestion is not further discussed in MonetDB, but all projects except *Bugzilla*, which was strict about using its own bug tracker to structure the tasks in the development process, did chose more lightweight and unstructured means of task tracking (notably



wikis [grub:3732]).

### 8.1.5.1 Advice for the Innovator

For the innovator, bug tracking procedures are not difficult to change informally and even getting a change accepted in the process guidelines [argouml:4663] should not be too hard. But bug tracking innovations often conflict with the volunteer nature of participation and thus fail in the adoption phase as developers can not be made to feel enthusiastic about them. Often a balance needs to be struck between the advantages of additional process elements and the overhead which they incur, and an innovator should critically weigh and rather err in estimating too much overhead. Last, the innovator must consider that while the bug tracker represents a valuable memory of all those situations in which the software can be improved and in which a user was troubled by it, this memory rapidly loses value if the developers cannot understand and associate with these issues. Making a clean cut as in the project GRUB to restart a bug tracker might seem to be ignorant towards those who spent effort to report bugs, but might provide the best way for developers to become motivated to reconnect to it.

## 8.2 Open Access

The most important class of innovations introduced in the studied sample in 2007 was the new generation of source code management tools based on a distributed paradigm (see Section 8.1.1). In this new model the previously “walled server” [229] of a SCM is abolished and replaced by a socially defined hierarchy of technically equivalent contributors. Alwis and Sillito found in unison with the results in this thesis that providing “first-class access” to all developers and reduced maintenance effort were key reasons for switching to a DVCS [135]. Even in one project with a centralized infrastructure was the tendency found to *make access more open by offering commit access to a particular directory* in the SCM to everybody interested [argouml:5634]. A similar motivations can be put at the heart of using wikis:

Wikis as originally envisioned by Ward Cunningham in the mid-nineties were developed to provide a quick and unbureaucratic way for users to modify content on web-pages [309]. In contrast to traditional content management systems, wikis are commonly deployed without access control and thus similarly replace a “walled server” paradigm with open access<sup>220</sup> for anybody with control structure being purely social. As famously demonstrated in the case of popular and open Wikipedia vs. the walled and soon abandoned Nupedia [88], such open access can leverage a greater number of excellent contributions despite the possibility for low-quality contributions and spam than with a strict reviewing process.

This of course leads to a wider question: In which cases is protecting a repository, website or server really necessary and in which cases might open access and distribution enable usages with possibly unforeseen and positive benefits<sup>221</sup> to an Open Source project?

How about for instance for the bug tracker? In the project *Xfce*, it was proposed to clean the bug tracker of obsolete bugs and while this proposal was accepted by the project, there is a noticeable delay in starting with performing the clean-up because access rights needed to be assigned [xfce:13143]. Similarly, the episode of rejuvenating the bug tracker in *GRUB* might not have taken months and four individual attempts to success if no access rights would have been needed to make the necessary changes to the bug tracker (see Section 5.9). What would be the potential drawbacks if everybody was allowed to close bugs or set the status of bugs as only developers are usually allowed? The only

*Bug tracking  
at GRUB*

<sup>220</sup>The term used in this thesis should not be confused with the term Open Access as used in (scholarly) publishing, for publications which are available to readers at no cost.

<sup>221</sup>In *KVM*, for instance one of the benefits the project was hoping to achieve using a distributed SCM was that interested parties would start providing long-term support for certain versions independently of their official association with KVM [kvm:1399].

fear appears to be about vandalism. Yet, the wiki case shows that if efficient undo is possible, then malicious edits can easily be handled.

Taking a step back and looking at the whole Open Source infrastructure, the trend seems to be clear that with open communication channels such as mailing lists and IRC, open version control and open websites it is only a matter of time until we also see some open platforms in the remaining areas. Bug tracking is one example, but more interesting is release management. If an Open Source project uses a plug-in architecture in which developers implement given interfaces to extend the core as a way to maximize the benefits from loosely-coupled contributors, then not only offering them first-class source code management via a DVCS but also access to the release management platform might be a sensible approach.

For the innovator, particularly in the case in which a new innovation is devised, this section recommends a cautious stance towards access control schemes as part of an innovation, because the trend appears to be to find ways in which the negative externalities of access control such as in particular maintenance effort and unrealized loosely-coupled participation is counteracted.

## 8.3 The Innovator's Guide

The following innovator guide was written to be a stand-alone document and is included here as a summary of the strategies and hypotheses developed in this thesis. References to the sources of suggestions have been added in bold brackets for the scientific audience of this thesis.

### 8.3.1 Introduction

This guide should help everybody who is interested in achieving changes in an Open Source project by introducing new tools, processes or technologies. It is the result of a study on Open Source projects (which can be found at <https://www.inf.fu-berlin.de/w/SE/OSSInnovation>) and is particularly aimed at changing processes and tool usage in medium-sized project of 5 to 30 members.

This guide will guide you as an innovator through the stages of realizing a change in a project, starting with defining a goal and devising a solution, through proposing it to a project and seeing it adopted.

### 8.3.2 Getting Started

When setting out to change the Open Source project of your choice, you first need to make sure that you—the innovator—are aware of the following:

- Introducing an innovation is a time-consuming matter and can and should not be done without dedication and sufficient time in the next couple of months. [Compare with Section 5.8 on the role of time in innovation introductions and Strategy 10 “No Introduction without Commitment”.]
- Introducing an innovation can harm a project, cause tension, stress, and conflict. It is your obligation as an innovator to assure that you do not cause such harm unnecessarily. [Derived from the principles behind Action Research, such as the researcher client agreement. See Section 3.1.1 and also compare with the discussion on Open Source research ethics [373].]

And as a direct consequence: Be honest about your goals and your affiliation (in particular if you are paid to do the job), so that the project can judge you based on what you really are.

- Achieving change requires an open mind to the concerns, interests and attitudes of the other participants in the project. If you are too focused on your own private problems and goals, then you will likely fail to achieve anything positive. [See Strategy 12 “Reinvention” for the positive effects of being open for alternative usage of a proposed innovation.]

- Changing the hearts and minds as a stranger is difficult and probably dangerous as well, because you do not understand the reasons for the status quo. [See Strategy 15 “Building familiarity”.] Making a call for change without being part of the project is thus likely to fail. Instead, start by contributing to the project. If you are established and have proved that you are capable and committed, changes will be much easier. If you really want to achieve a change as a stranger, you need to get in contact with the maintainer or a high ranking person in the project and convince him explicitly before you proceed to convince the project.

### 8.3.3 Getting Your Goal Straight

Before you approach the project with your new idea or concern, you should first take a step back and make sure that you understand your own goal or the perceived problem. [For the origin of this idea see the separation of problems from solutions in the Garbage Can Model (Section 6.2).]

To this end, write down what you want to achieve (your goal) or what you believe the problem currently is. For instance, if you are fed up with the poor security track record of the project resulting in many vulnerabilities being published, you might want to write down as a goal:

“I want our software to be more secure!”

Or if you like to think more about the issue as a problem, you might say:

“The number of vulnerabilities found in our software is too high!”

Then take ten minutes and break your goal down in more manageable chunks, so that from one single sentence you get a more substantial division of the problem or goal at hand. To achieve this, write down the causes of the problems and what could achieve the goal.

In our example of achieving more security or dealing with too many vulnerabilities, we might conclude that the reasons for too many vulnerabilities are the result of using dangerous API functionality and lacking peer review, while ways to achieve the goal could be to use static type checkers or use a framework which validates input.

Iterate this approach by looking at each resulting cause or way to achieve a goal and look for its causes or ways to achieve it. This root cause analysis this will help you understanding the problem better and better.

Stop once you reach the point where splitting problems into causes and goals into sub-goals becomes too abstract. For instance, if you arrive at reasons that are bound to human strengths, weaknesses, or characteristics, such as discipline to sanitize inputs, motivation and time to perform peer review, or knowledge about security topics, then you have probably gone one step too far. These are not problems that can be fixed easily, but rather boundary conditions within which you have to operate.

At the end of this process, you should have a good sense for the problems at hand and a vision for what you want the project to be like.

If you get stuck in the process and feel that you do not understand the problem sufficiently, jump two sections down to Proposing.

### 8.3.4 Getting a Solution

During the last step, it is often tempting to go right ahead and get the discovered problems and goals solved or achieved. In the above example, when finding the problem of dangerous API access, one might want to dive right into the code and get rid of all of them. Instead, it is recommended to stay a little bit longer with the problems and goals to see that there are many, many possible ways to improve your project.

If you now start looking for solutions—i.e. concrete things that could be done to reach a goal or solve a problem—keep the following things in mind:

- Using proven existing solutions from other projects is much easier than inventing solutions yourself. [The case study of introducing the SkConsole to KDE and Gnome is a good show case for the difficulties of communicating a novel idea (see Section 7.2) in particular when comparing with the introduction of unit testing as an established idea (see Section 7.4).]
- The better you understand the solution, the more likely is it that you can be a good advocate for it. [See Strategy 13 “Counter the Idea Splurge” for how much precise knowledge about an innovation is often required to delineate it from other ideas.]
- The less work a solution causes for the project members, the more likely it is to be accepted by your project. A direct implication of this is that radical changes, which require a lot of heavy lifting before the first results can be seen, are less likely to succeed than anything which can be achieved incrementally by investing little effort but over a long time. [See Section 5.9 on radicality and innovation introductions.]

If you found a solution that is promising to you to achieve the goal or solve the problem, the next step should be to think quickly about the consequences of your ideas. Who is affected? Which existing processes need to be changed? How would these people feel? Are there situations where your innovation is inferior to the status quo or using common sense? [See Strategy 11 “Go for Technical Excellence”.] Are you prepared to convince them that the change is appropriate for the goal? Is the solution compatible with Open Source norms and ideologies (e.g. using proprietary software isn’t going to fly in most cases)? [See Strategy 16 and Section 5.10 on tool independence.]

At the end of this step, you should feel confident that your solution will help your project in getting the problem solved or the goal achieved.

If you get stuck in this step, you should proceed to proposing (see below).

### 8.3.5 Proposing

In the best case you can now approach the whole project with a good idea for a solution and a good sense of the problem that you are intending to solve. You might use an opportunity in which the problem is especially apparent (for instance when another vulnerability was found in your project) or when your project is discussing the future anyway (for instance after a release).

Since this is the first time that you approach the project with the new idea, it is important to stay open to the way the other project members feel about the problem and your solution. Do not force a decision. This is also the reason why you do not have to think too hard about the problems and possible solutions before proposing them and why it is not a big problem if you got stuck in the previous two sections. Use the discussion with your project to understand the problem and find even better solutions together with your peers.

There are many ways that such a discussion can evolve and it is not always easy to keep it within constructive boundaries. The following hints might help:

- Keep the discussion focused on the immediate future. If discussion becomes too abstract, many people quickly lose interest. This includes giving concrete descriptions what would be the implication of adopting an innovation. [See Section 5.4 on the enactment scope of an innovation, Strategy 2 “Keep the Enactment Scope Small” and Strategy 3 “Protect against Scope Expansion”.]
- If the discussion becomes overly complex, aggressive or incoherent and more and more participants do not follow the technical arguments, restart the discussion in a new thread by providing summaries of the proposed options and their pros and cons. [See Strategy 9 “Manage the Garbage Can”.]

- External validation from other projects about how an innovation worked out for them can provide strong arguments and good sources of technical expertise in a discussion.
- If the discussion becomes deadlocked between a number of alternatives, conducting a trial of the alternatives can help make the advantages of each option clear. [Section 5.3 on partial migrations and Section 6.2 on the Garbage Can contain discussion of running trials to assess and demonstrate the viability of an innovation.]

Once you have identified a promising candidate for a solution, the next step becomes to achieve a decision and agreement with your project peers that you and the others really want to make the proposed change.

This should be of interest to you even if you feel like you have sufficient influence in the project to just go ahead with the idea, because it strengthens your possibilities to point back to the decision and the legitimacy you got to make the change if you encounter resistance. Here are some hints:

- Getting the maintainer to be in favor of the innovation is in most cases the most important step. Thus, react to his objections, questions, and criticisms carefully. [More generally you may want to “Peel the Onion Inside Out” as the corresponding Strategy 20 is called ]
- It is not necessary to achieve a consensus with all project members, but rather you should strive to have a substantial set of supporting members and nobody who is fundamentally opposed to the change (think about how Apache projects vote—three +1s and no veto are sufficient—for a change to understand the basic motivation behind this<sup>222</sup>).
- If participation in making the decision is low, you can push the discussion by asking directly for objections and telling the project that you would start working on implementing the change in a couple of days or at the next weekend. [This strategy of forcing a reaction was demonstrated in the project GRUB during a *proposal to move to Git* [grub:4116].]
- If there are many proponents and a few very outspoken opponents, the discussion can become lengthy and tiring. In this case, calling for a vote can speed the decision process up considerably. This can often be initiated by a simple single line reply such as “I vote for the new [...] suggested by [...]”. [Refer to Section 5.7.1 on organizational decision making and in particular Hypotheses 5–7 on voting.]

If you fail at this step because people decide against your ideas, then do not give up, but rather have a look at the reasons why your ideas were rejected. Are there things that can be changed the next time you propose something or were your ideas incompatible with the realities of your project?

Experience shows that many complex ideas take three to six months to become understood in a project, in extreme cases repeated efforts for over a year can even be observed, before a change is achieved. [Episode *Git at gEDA* is the best example for a successful introduction requiring more than six months of dedicated effort; the *reinstatement of the bug tracker* in the project GRUB illustrates an introduction requiring four separate attempts for success.] Accepting the given reasons for rejection of a particular solution but being persistent about solving the underlying problem will eventually lead to success.

Change is hard, try smaller, more local, less abstract ideas next time if all your effort is to no avail.

### 8.3.6 Executing

If you achieve a decision at the organizational level, then you most probably will have now the possibility to realize your ideas. This can be a lot of work depending on your innovation and the resources available to you. Some suggestions:

- If you need to do some work on a server, it is best to ask for an account rather than splitting responsibilities. [See Section 5.5 and Strategy 4 “Prioritize Control”.] The time available to project participants usually causes too many synchronization issues.

<sup>222</sup>As an easy read from Roy Fielding <http://doi.acm.org/10.1145/299157.299167>

- Proceed timely with executing, unless you want your project to forget that they decided on making the change. [See Strategy 7 “Act on Multiple Time Scales”.]
- If the task is too big for a single weekend, split it into chunks which you can finish and show to others. [See Strategy 1 “Migrate Partially”.] For instance, if you work on a process guide, the chapters on how to submit bugs and how to create release branches can probably be written independently.
- Use all opportunities you get to tell people about what you are doing. [See Strategy 17 “Signal your Engagement”.] Write e-mails on how you installed the software, migrated the data, broke the server, switched system A offline, upgraded tool B, etc. Every chance you get to talk about the innovation and spread the word should be used to create awareness and let others understand the technology or process that will become available shortly.
- Involve everybody who is interested, so that people can start learning to use the new processes and technology and become your co-innovators. [See Strategy 19 “Look for Experts”.]

The successful end of this stage should be reached with an e-mail to the project that announces the availability of the new system, process, document or solution.

### 8.3.7 Adoption and Sustaining

After the announcement has been made, the usage of the innovation can start, but your job as the innovator is still not over. Having written a process guide and having it put up on the web-site is only halfway there. Helping your peers to get used to the innovation and seeing it well adopted into the daily processes of the project can still be a challenge and might take several months. [See Strategy 18 “Teach for Success”.] Do not force anybody, but provide assistance to everybody struggling.

To ease adoption pressure for some innovation, one possibility is to maintain access to the old innovation via an adapter. For instance, when adopting a new version control system, an adapter might be used to continue offering CVS read access to those you have trouble adapting. [See Section 5.6 and Strategy 6 “Adapter”.]

Every couple of months you should take some extra time to check back with the innovation. First, take a step back and have a look at the innovation. Is it still used appropriately? Is it still appropriate at all? Did changes occur in the project that allow easier and better solutions? Think about this, and then, as a second step, let the project know about it. Write a short status report which takes note on the progress and successes of the new innovation, hands out praise, and encourages those who have trouble adjusting. Third, have a look whether there are any administrative tasks to be taken care of, such as cleaning-up stale bug tracker tickets or orphaned wiki-pages, and then apply updates and security patches.

### 8.3.8 Write an Episode Recap

No matter whether your innovation introduction failed or was successful, there is something to be learned from it for your next introduction or for others who want to accomplish similar things. Take the time to summarize your experience and make it available to your project.<sup>223</sup> It could help set a discussion in motion on where to improve the project next and become a valuable source for inspiration for others [29].

---

<sup>223</sup>I also would love to hear from you about your experiences with changing an Open Source project. Write an e-mail to [christopher.oezbek@fu-berlin.de](mailto:christopher.oezbek@fu-berlin.de).

## Chapter 9

# Conclusion

This thesis has explored the new research area of innovation introduction in Open Source projects using Grounded Theory Methodology, five case studies, and a comparison to four existing theories from the organizational and social sciences. The main contribution of this thesis is a set of theoretical concepts grounded in studying the mailing list communication of thirteen medium-sized Open Source projects. These concepts are the result of analyzing relevant phenomena that occurred in the 134 studied episodes. They illustrate the broad scope of innovation introduction as a research area and constitute a foundation for analyzing the forces and mechanisms at work during innovation introductions. Each concept has been discussed in depth resulting in strategies and hypotheses about innovation introduction and Open Source development in general.

These concepts are the building blocks of a theory of innovation introduction in Open Source projects and can be a starting point for both the necessary validation of the concepts and their integration and expansion into a holistic theory. This thesis has established the methodological tool-set including GmanDA as software support and the necessary vocabulary<sup>224</sup> to achieve the latter goal of expanding and integrating a theory of innovation introduction.

On the former goal of validating the results presented in this thesis, I believe new means must be devised. This is not to say that using the same methods of case studies, GTM, and possibly even the theory-driven approach of Chapter 6 could not lead to additional insights, theoretical saturation, and broader grounding in data. For instance, looking at forcing effects (see Section 5.7), it seems best to proceed qualitatively with additional episodes from other projects to consolidate the technical and social mechanisms by which individual adoption decisions are promoted. But, validation of concepts and strategies would at this point benefit more from quantitative assessments. For instance, the questions surrounding the strategies of enactment scopes (see Section 5.4) have reached a point where a statistical analysis is the best way to conclusively answer the question of how the success chances for differently scoped proposals vary.<sup>225</sup>

To summarize the resulting theory fragments:

- Hosting is the concept which consolidates the phenomena surrounding the provision of computing resources for innovations. Choosing among the various types of hosting was associated with the five concepts of effort, cost, capability, control, and identity. Each of these concepts was shown to be the primary motive in at least one decision situation why a project chose a particular kind of hosting (see Section 5.5). For the innovator, hosting was found to be both an interesting trigger of innovation episodes and a barrier to overcome, due to the control the leadership exerts over hosting resources.

---

<sup>224</sup>This includes both the codes given in the Glossary and the innovation lifecycle in Section 5.2.

<sup>225</sup>Since field experimentation requires too much effort, post-hoc case analysis or surveys are probably the best methods to gather data.

- The concepts of enactment scopes, partial migrations, and radicality were proposed to explain deviations from what had been assumed to be “normal” in both innovation proposals and their executions. Enactment scope, for instance, captures the difference between proposals which are cast widely to the project in general and those specifically cast to certain people and certain points in time (see Section 5.4). Success in innovation introduction could be linked to reduced scopes, partial executions, and incremental changes.
- Adapters for connecting two innovations were linked to successful innovation introductions based on the ability of an adapter to (1) increase the tool-independence of the innovators and of adopters in the late majority, (2) decrease enactment scopes and (3) enable partial migrations, in particular to new source code management systems (see Section 5.6).
- Decision making about innovations in Open Source projects was divided into organizational and individual innovation decisions and linked by the execution of the innovation in the project. The organizational decision making mechanisms discovered were voting, authority decisions, and representational collectives with the occasional normative “just do it” which reverses execution and decision making. After execution was achieved, the concepts most useful to describe individual decision making were (1) forcing effects originating from the innovation, for instance via a “code is law” mechanism, (2) enforcing strategies such as the gate keeper, and (3) social expectancy arising, for instance, from community norms (see Section 5.7).
- The role of time during innovation introductions in Open Source projects was found to be important, because of the high variability with which Open Source participants can engage and the intransparency of working globally distributed. High levels of activity were captured using the concept of a participation sprint. Sprints were found to cause overloading in the project, leading to decreased chances for successful innovation introduction. Signaling was offered as an important strategy to synchronize expectations about activity (see Section 5.8).
- The concept of tool-independence captures the phenomenon of few tools being proposed for the individual developer (less than five percent of all episodes deal with tools which are to be used on the client-side only). It was shown that projects had learned that defining a tool-set explicitly would increase maintenance effort and cause inflexibility in the future. Thus, projects avoided giving more than basic recommendations towards tool usage (see Section 5.10).

Next, these results were reviewed from the perspective of four theories from the social sciences (see Chapter 6). It was found that the theories could neither invalidate the concepts nor add any significant new insights, thus strengthening the integrity and saturation of the GTM-derived main results and the importance of having used GTM as a method. Nevertheless, practical contributions were derived from studying each theory in the context of selected episodes: (1) The Garbage Can Model was used to describe how a skilled innovator could handle difficult discussions (see Section 6.2). (2) Punctualization and heterogeneous networks from Actor-Network Theory were shown as a way to conceptualize and understand power relationships in a project prior to and during an introduction (see Section 6.4). (3) A Path Dependence analysis of possible lock-ins was suggested as a method for the innovator to early on assess possible introduction barriers (see Section 6.1). (4) Visualizing the communication of the studied projects using Social-Network Analysis highlighted the importance of the project core and brought the concept of maintainer might into focus (see Section 6.3).

The case studies described in Chapter 7 then complement and extend the main results from the perspective of the researcher as an innovator: First, the studies of introducing the information manager and unit testing demonstrated the viability of approaching an Open Source project with innovation proposals as both a project insider and outsider (see Section 7.1 and Section 7.4). Second, the gift culture study at KDE and Gnome revealed that making a proposal in conjunction with a considerable code gift altered the perception of the proposal. However, the change occurred not for the reason expected—demonstration of technical expertise—but rather as a show of commitment (see Section 7.2). Third, the study on contact strategies tested whether to send a proposal to the maintainer, the whole project, or individual high-ranking participants and showed that only the strategy via the maintainer was successful for organization-wide innovations such as an information manager role (see Section 7.3).



Fourth, the longitudinal case of introducing unit testing discussed in more depth the withdrawal of an innovator and the effects of adoption. It was found that learning opportunities need to be actively created by the innovator and that handing over responsibilities for sustaining an innovation does not occur automatically (see Section 7.4). Last, Thiel's case study of introducing a process innovation to improve the ability of Open Source projects to deal with security vulnerabilities was presented. It led to important concepts explaining the rejection of a proposal by the project members such as legacy constraints and structural conservatism (see Section 7.5).

Last, assuming a practical perspective in Chapter 8 resulted in a description of the most popular innovations occurring in the dataset in 2007 and led to advice for their introduction. The chapter includes the adoption of new source code management systems (see Section 8.1.1), license migrations surrounding the GPLv3 (see Section 8.1.3), modification to bug-tracking procedures (see Section 8.1.5), participation in the Google Summer of Code (see Section 8.1.2), and adoption of new version naming schemes (see Section 8.1.4).

The most practical contribution of this thesis is the Innovator Guide in Section 8.3. It consolidates the most important results and strategies into a short, accessible document.

## 9.1 Future Work

Much remains to be studied to integrate the theory fragments this thesis uncovered, to validate the underlying concepts, and to further explore the landscape of innovation introduction behavior:

**Analyze Tactical Innovator Behavior**—First, it would be beneficial to target the tactical dimension of innovation discussions, because this thesis has only explored this area using global-level concepts such as the Garbage Can. In particular, such research should assess the use of different argumentational and rhetorical devices in the context of Open Source projects and link these to the choreography of threaded discourse. The work of Barcellini et al. discussed in Section 4.3 is a starting point for this. The goal of additional research in this area should be a comprehensive theory of decision making and decision discussions in Open Source projects.

**Focus on Particular Innovations**—The introduction-specific focus of this thesis should be complemented with additional work putting innovations in the center of the discussion. In other words, the question of “how to introduce innovations” should be shifted to “which innovation to introduce”. This puts more emphasis on the individual innovation, its origin, design, and underlying problem. Section 8.1 on five specific innovations provides a starting point for such an innovation-centric perspective. From there, one should explore the families of innovations, the alternatives between them, their construction, design, and heritage. For instance, one might expand on the idea of open access to project infrastructure (see Section 8.2) and explore its consequences to other areas such as project leadership. Also, such work would probably classify and assess the motives for change in Open Source projects. To achieve this, is it necessary to deviate from the position of this thesis which assumes that the pursuit of any goal should be supported as long as an innovator deemed it beneficial.

**Understand Commercially-Dominated Projects**—Three commercially-driven Open Source projects (*KVM*, *U-Boot*, and *RequestTracker*) are included in the data set of this thesis, but no noticeable differences to their purely volunteer-driven counterparts in their innovation introduction behavior could be found. All three projects show strong authority-driven decision making by the project leaders, but the same holds for more than half of the volunteer projects. The absence of such differences might be a hint that the autonomous interaction by participants largely reduces the differences between commercially-motivated developers and volunteers. A more systematic exploration in commercially-dominated contexts might yield additional insights into changes driven from the top down (for instance a better understanding of the concept of *maintainer might*) and into the interaction of the community with the sponsoring company.

**Conceptualize Innovation Adoption as a Learning Process**—The abilities and processes in Open

Source projects by which knowledge is spread were not focused on in this thesis. If one argues that innovation introduction adoption is primarily a learning process of new technologies, rationales, and mechanisms, then this perspective is much more important.<sup>226</sup> There is existing research which has emphasized the possibilities offered by open processes and communication for external members to participate legitimately and thereby join into an apprentice/master relationship with the project [555, 301]. Yet, this thesis has described in Section 7.4 how difficult it is to acquire knowledge from outside the project.<sup>227</sup>

**Understand the Role of Informalisms**—One can argue that the perception of chaotic development in Open Source projects arises from its informal nature, which for an outsider makes processes appear unstructured, irrational, and ad-hoc. That development is able to succeed without explicitly defined processes might appear at odds with the conventional wisdom of software engineering and can therefore be the starting point for an exciting research area. One might first ask whether embedding of rules and processes into project infrastructure and architecture might strengthen the development paradigm (see Section 2.3.8). This brings up the larger question of how efficient this alternative is compared to formalizing processes in documentation. In one of the case studies, for instance, it was suggested that formalization produces inefficiencies because it reduces flexibility [379]. We could therefore hypothesize that it is ultimately never the formalization that improves software development, but rather the informal means for enforcing that participants abide it.

**Extract the Ideology of Innovation**—Looking at norms and ideology, the apparent discrepancy between technical enthusiasm of the Hacker culture [97] and the observed resistance to software engineering innovations need to be resolved. Most likely this will lead to an expanded view on the motives and mechanism that underlie participation in Open Source projects and structure people's behavior. It will also link the topic of innovation introduction more strongly to the socialization of new members, decision making, and trust formation.

Last, and as a methodological suggestion:

**Talk to Innovators**—Open Source practitioners should be asked for their experience and knowledge about introducing innovation in order to gather higher-level data than available in the raw mailing list discourse. This thesis tried to gather such data using a survey, but could not connect to the way the Open Source practitioners think and talk about these issues (see Section 3). Nevertheless, it would be highly beneficial to get abstracted data to complement the existing secondary approach [6]. In particular, interviews should be performed to collect first-person accounts. Furthermore, action research studies should be conducted in close communication with a project to increase the depth of results presented in this thesis.

## 9.2 Epilogue

I am confident that this thesis has enabled the reader to successfully pursue his or her own introduction of an innovation into an Open Source project and has shed some scientific light on process change in Open Source projects. For the computer scientists reading this thesis, I hope they found the choice of a qualitative method interesting. Coming from a field that emphasizes theoretical and quantitative empirical work, the use of Grounded Theory Methodology with its focus on theory construction rather than generalization might have required some adjustments.

It might help to consider the organizational scientist, who is probably more familiar with GTM, but who had to master all the computer science knowledge expected in this thesis such as on distributed version control and bug tracking procedures. I hope, both types of readers were able to accept that this thesis sits in between the fields and needs both a solid technological basis and a sociological approach.

<sup>226</sup>One interesting phenomenon observed in this area was that during *execution* of preparatory steps for an innovation participants often *narrated* their progress in detail on the mailing list. Such e-mails were seldom commented on, but appear to be artifacts of a distinct form of learning in the Open Source discourse.

<sup>227</sup>The same difficulties were observed when regarding knowledge from non-core knowledge domains such as legal issues.

Before closing this dissertation, I want to recommend (1) browsing the appendices, which contain the detailed glossary of the episodes and concepts from this thesis (see Appendix A.1 and Appendix A), and (2) looking into the electronic edition of this thesis<sup>228</sup> to explore the linked episodes yourself.

---

<sup>228</sup>Available from <https://www.inf.fu-berlin.de/w/SE.OSSInnovation>



## Appendix A

# Glossary and Codebook

This glossary contains descriptions and definitions of all episodes, innovations, codes and concepts referenced from the text using hyperlinks. The benefit for the reader is twofold (see the detailed discussion in Section 3.2.3): (1) The reader can conveniently access much detail information, without it cluttering the main text or having to be remembered by the reader. (2) The reader can use the intermediate level of descriptive summary [299, p.695] as a bridge between raw data in the e-mails and high-level conceptualization in the main chapters. For instance, a reader might want to read through the episode directory in Appendix A.1 first, to then be able to follow the conceptual discussion more fluently. For a methodologically inclined reader the glossary is furthermore valuable because it offers a glimpse into the intermediate substrate of performing GTM in this thesis.

Please note: (1) The total list of codes created during the research process and managed in GmanDA (see Section 3.4) is larger than the ones displayed here because the former also includes many codes which arose during conceptual development and which were later on discarded, subsumed by richer concepts or not further pursued in this thesis. (2) The descriptive summaries of the episodes are written following the best knowledge of what happened according to data. They are kept deliberately in a factual, short, and narrative style even though there is the possibility that real events might have been different. (3) Disambiguations were added for all pairs of concepts, which are hard to distinguish following their definitions alone. (4) Entries are sorted by code and not descriptive title.

## A.1 Episode Directory

An *episode* is the central means of aggregating messages associated with the introduction of one innovation or a related set of innovations into a project.

Because the unit of coding is a single e-mail and not the set of all messages associated with a certain introduction, I used the code `episode.<episode name>@<project name>` to aggregate those messages first.

Introduction episodes (and not innovations) are the primary mechanism of aggregation, because the episode will also contain the discussions about problems with existing innovations, discussions leading up to the idea for an innovation, the retiring of a superseded innovation, problems related to the use of the new innovation, etc.

Constructing episodes from the raw stream of messages is primarily a method to organize the data and serve as backdrop for a narrative strategy to understand event data [299].

- *Branch for Patches @ ArgoUML* (`episode.branch for patches@argouml`)  
An episode in which the maintainer and the top two developers in the project *ArgoUML* discussed

the possibility of using branches for collaborating on patches. The maintainer brought the topic up, when he noticed that the two developers were using attachments in the bug tracker to collaborate on a feature. Both developers *rejected* the given proposal by (1) a series of arguments such as inefficient operations in their development environment [argouml:4773] or the lack of an established tradition of using branches for experimental work [argouml:4784] and by (2) enlarging the *enactment scope* of the *proposal*: The maintainer had proposed the use of branches primarily in the given situation, but one of the developers brought up the implication of using branches in general as an effective argument against the proposition.

The episode was restarted two months later, when the maintainer proposed to use branches for the contributions of students as part of the *Google Summer of Code*. Taking this indirect approach, he achieved that within a month the use of branches had been adopted by the two previously opposing developers, and that over time even one of them explicitly suggested to a new developer to use branches for the development of a new feature [argouml:5681].

- *Branches for Contributors @ ArgoUML* (episode.branch for patches@argouml.contributor)  
In this episode the maintainer of the project *ArgoUML* proposed to define the role of a contributor to the project. This role—in contrast to a committer—is defined to hold only commit access to the branches directory of the *SCM* repository and not to the trunk of development. This role enabled the project to grant repository access to new participants without having to fear damages to the trunk. New participants on the other hand gained the advantages associated with using a *SCM* to manage their work.

This episode was successfully *executed* by the maintainer. It demonstrates that even with a centralized *SCM* it is possible to establish some of the benefits of open access to project infrastructure (see Section 8.2) which a *distributed SCM* provides.

- *Bug Tracking Procedure @ ArgoUML* (episode.bug tracking@argouml)  
This episode in the project *ArgoUML* is led by the maintainer and *triggered* by a comment to a bug entry. To increase the through-put of the quality assurance process, the reporter of an issue is granted the right to *close a bug directly from resolved (i.e. fixed) state*.

While this innovation is formalized as a rule in the process “cookbook” of the project [argouml:4663], there is no indication whether this additional process step was adopted by the project members and helped to increase the through-put as intended.

- *Checkstyle @ ArgoUML* (episode.checkstyle@argouml)  
One of the core developers in the project *ArgoUML* proposes to limit a new innovation (in this case the upgrade to *Java 5*) with regards to one aspect (auto-boxing) due to an alleged performance overhead which would be hard to detect by developers [argouml:4967]. The innovator directly notes the difficulty to *enforce* such a limitation due to the difficulty to detect a violation in the first place and thus proposes (which this episode is about) to use a *static code analyzer* to ensure the proposed avoidance of auto-boxing. The introduction fails, because (1) the innovator is not sufficiently *dedicated* to his proposal and (2) after the proposal is being made, the innovator has a period of low activity in the project (probably induced by some external restriction), which is discussed in Section 5.8 on time as an important aspect of innovation introduction.
- *Foundation @ ArgoUML* (episode.foundation@argouml)  
In this episode the project *ArgoUML* joins the *legal foundation* Software Freedom Conservancy with the goal to find a way to handle money and get assistance with a *licensing switch which was becoming necessary*. The episode is driven by the maintainer with little of the actual communication with the foundation being published on the list and while *ArgoUML* joins the SFC in 2007 no progress is visible on accomplishing the intended goals.
- *Google Summer of Code @ ArgoUML* (episode.gsoc@argouml)  
As with several other instances of participating in the *Google Summer of Code* (see Section 8.1.2 for details), the project *ArgoUML* undergoes several phases which closely mirror the process demanded by the program itself: (1) First, the project collects a list of ideas and mentors [argouml:4846,4899]

and settles questions about how to manage students [argouml:4975,4902]. (2) An application is then submitted to Google and fortunately accepted [argouml:4910]. (3) Students start to apply to Google and introduce themselves and their ideas on the list [argouml:4917,4926,4931,4936,4939]. One problematic situation arises in this step, because due to a large number of applicants (42) compared to the number of proposals (6), many students pick the same proposed task to work on [argouml:4975]. Since only one student can receive a scholarship per task, the project cannot profit from the interest by students to participate. (4) Google selects students and assigns them to ArgoUML, who in mid April start working for three months on their proposals [argouml:4987,4994,4987,4988,4990]. (5) After the summer period, the mentors assess the student's work and are invited by Google to meet at the *mentor summit* at the Google headquarters [argouml:5511]. (6) Three of the students are raised to full or core project member status and receive commit rights [argouml:5513].

- *Java 5.0 @ ArgoUML* (episode.java5@argouml)

The decision to migrate to Java 5.0 from Java 1.4 in the project *ArgoUML* precedes the observation period of 2007 and is included in this sample because the adoption period contains several interesting aspects. Most importantly the episode provides the backdrop for the episode of adopting *a static checker for enforcing compliance to a certain set of Java 5.0 features*.

The move to a new Java version is executed in the project at the very last moment as Sun is already declaring Java 1.4 to have reached the end-of-life phase in which no security updates are provided any more and at a point where Java 6 is already available. This is interesting because it strengthens the concept of a “legacy constraint” which was uncovered by Thiel in his case study of enhancing Open Source web application security (see Section 7.5) and indicates a belief in the “inertia of users” and a preference to keep satisfying such users even if nobody is personally known who is under a particular dependency constraint preventing an upgrade.

- *License Switch @ ArgoUML* (episode.license switch@argouml)

This episode is triggered as part of the move of *ArgoUML* to join the *Free Software Conservancy* when one core developer brings up the question of whether to *switch licenses* and replace the antiquated original BSD license with at least the revised BSD license or even EPL or LGPL [argouml:5067].

Discussion is animated and involves several conversations about past implications of using the permissive BSD license rather than more restrictive licenses such as the GPL [argouml:5569]. In particular the fork of ArgoUML into a commercial product from which no contributions were ever returned stirs bad memories [argouml:5568]. Controversy builds between those who would prefer a GPL-type license to protect ArgoUML from forking by companies [argouml:5521] and those who would stick to permissive licenses such as ASL or modified BSD [argouml:5582].

In the end, a middle ground is found in the use of the EPL as a weak-copyleft license, which mandates changes to software to be made available, while applications built on top of the EPL software can remain proprietary [argouml:5726].

Even though the decision to switch to the EPL was made in December 2007 [argouml:5726], it took until November 2009 until an incremental update path was decided on, in which newly created files get added using the EPL license and existing ones are modified to contain both the EPL and BSD license header.<sup>229</sup> This is despite the fact that one core developer argues that the use of the BSD license would permit any license change by its permissive nature [argouml:5555].

- *Bug Tracking Observer* (episode.observer@argouml)

The maintainer in the project *ArgoUML* announces in this episode the creation of a new role called “*observer*” to enhance mailing list communication. This role—similar to the *information manager* discussed in Section 7.1—is meant to monitor mailing list communication and help with answering recurring questions, identify duplicates in the bug tracker, etc. [argouml:4752]. The

<sup>229</sup>See <http://argouml.tigris.org/wiki/License>.

attempt to get this innovation successfully introduced in the project *ArgoUML* fails when *nobody volunteers to adopt* the role (in fact nobody even replies to the announcement).

- *Packaging @ ArgoUML* (episode.packaging@argouml)

This episode is triggered by a *preceding episode* in which a translator proposed to cooperate with distributions such as Debian to improve the translation of *ArgoUML*, but concluded that a prerequisite to such a contribution is the existence of up-to-date packages in the distributions. In this episode one of the core members explores the current state in the distributions and asks the project whether they care about the situation and *proposes* to discourage the distributions to use unstable releases for packaging. A second core member proposes to rather change the release numbers to clarify which versions are stable and which are not. A mailing list participant brings up arguments against both proposals and the episodes ends standing *rejected* and *abandoned*.

Interestingly, six months later a discussion arises about the outdated state of Debian and Ubuntu packages of *ArgoUML* and the maintainer of the Debian packages explains his reasons for not updating the package. As a reply the core member who proposed to discourage unstable releases is doing so *without an organizational decision* [argouml:5424]. Unfortunately for *ArgoUML* this discouraging of the Debian maintainer does not help the project, as far as I could determine.

- *Translate with Distributions @ ArgoUML* (episode.translation@argouml)

In this episode a translator proposes to cooperate with distributions such as Debian to improve the translation of *ArgoUML*, but concludes that a prerequisite to such a contribution is the existence of up-to-date packages in the distributions. This then triggers the separate episode *episode.packaging@argouml* in which this problem is discussed by a core developer. The original idea of cooperating with distributions is no longer discussed despite it having merit also without perfectly up-to-date packages. In fact, the following episode about packaging fails to find a recipe for how to encourage better adoption of new packages by distributions even though such a cooperation regarding translation could have been a starting point for helping each other.

- *Ask Smart Questions @ Bugzilla* (episode.ask smart questions@bugzilla)

An umbrella episode for the two associated episodes *episode.ask smart questions@bugzilla.write document* and *episode.ask smart questions@bugzilla.integrate document*.

- *Integrate Document on how to Ask Smart Question @ Bugzilla* (episode.ask smart questions@bugzilla.integrate document)

This is a follow-up episode to *episode.ask smart questions@bugzilla.write document*, where a core developer condensed a guide for how to ask good questions and is now *proposing* to include the resulting document into the process by which users can write support requests to the project.

This proposal is not discussed, but another core developer with *control* over the mailing list service directly *executes* it by adding a link to the document from the subscription page of the mailing list.

It is unclear from the data whether the innovation achieved its intended effect of reducing the number of low-quality support requests.

- *Write document on how to ask smart question @ Bugzilla* (episode.ask smart questions@bugzilla.write document)

One of the core developers in the project *Bugzilla* offers a condensed version of a guide by Eric S. Raymond on the question of how to ask smart questions.<sup>230</sup> The core developer *executed* this innovation before asking the project for feedback, which then is positive. When he offered the document, the innovator also proposed to integrate the document more closely into the process by which users come to write support requests to the project. This is covered in *episode.ask smart questions@bugzilla.integrate document*.

<sup>230</sup><http://www.catb.org/~esr/faqs/smart-questions.html>



- *Contribute @ Bugzilla* (episode.contribute@bugzilla)  
An umbrella episode started by one of the maintainers of the project *Bugzilla* to improve the amount of contributions the project receives. In the course of this episode the maintainer starts five separate sub-episodes with ideas dating back to the end of 2005 [bugzilla:6190]: (1) He first asks his fellow project members to *be nice to newbies*, (2) then *improves the contributor guide*, (3) *installs an IRC gateway*, (4) *adds recruiting information into Bugzilla itself*, and (5) starts to ask for *self introductions of new members*.

- *Be Nice to Newbies @ Bugzilla* (episode.contribute@bugzilla.be nice)  
The smallest of five episodes as part of one of the maintainers' campaign to *enhance contributions to Bugzilla*. In this episode, the maintainer tries to set a behavioral norm on treating new developers well, which he hopes to attract as an overall goal of the five episodes.

Since this is a purely *social* innovation, it is not surprising that *compliance enforcement* is difficult and primarily attempted using a *plea*.

- *Contributor Guide @ Bugzilla* (episode.contribute@bugzilla.contributor guide)  
This episode is part of one of the maintainers' campaign to *enhance contributions to Bugzilla*. In this episode, the maintainer improves the *contributor guide* to help external developers be more productive in producing patches which will be accepted. Again, it cannot be determined whether this innovation was *successfully* used to improve contributions to the project or whether a determined contributor who had achieved to produce a patch for an issue important to him would really fail with getting it across the process hurdles of reviews and submission when no document would exist.

- *IRC Gateway @ Bugzilla* (episode.contribute@bugzilla.irc gateway)  
This episode is part of one of the maintainers' campaign to *enhance contributions to Bugzilla*. In this episode, the maintainer sets up an *IRC gateway* to improve access to the IRC channel of the project for users who are behind firewalls or cannot install an IRC client. There is one user who applauds this move [bugzilla:6294], but it is unclear whether any benefits for participation are ever achieved using this innovation.

- *Link from Product @ Bugzilla* (episode.contribute@bugzilla.link from product)  
This episode is part of one of the maintainers' campaign to *enhance contributions to Bugzilla*. In this episode, the maintainer adds a call for contributors to the product Bugzilla based on the rationale that most contributors to the project Bugzilla in the past were administrators who ran the product Bugzilla. Adding a call for contributors and a hyperlink guiding them to the Bugzilla website and *contributor guide* is believed to increase the chance that those administrators will notice the need for new contributors to join the project. As with the other episodes targeted to enhance contribution it is very hard to assess the practical usefulness of this innovation.

- *Self Introductions @ Bugzilla* (episode.contribute@bugzilla.self intros)  
This episode is part of one of the maintainers' campaign to *enhance contributions to Bugzilla*. In this episode during July 2007, the maintainer introduces a new *social process innovation* to the project, asking *everybody to introduce themselves* on the mailing list so that people build trust and a community.

In contrast to the other episode aimed to improve contributions, this episode provides some hints to the success of the introduction. Around 15 people introduce themselves over the course of the next five months in the proposed way. The most interesting one of which is an employee from Yahoo who informs the project about Yahoo's Bugzilla installation, which with a database of around 1.5 million bugs and 2,500 products, could be the world's largest. Concerning the success of attracting new contributors though none of those who introduced themselves but were not members already did contribute a single patch in the observation period.

- *Design Approval @ Bugzilla* (episode.design approval@bugzilla)  
A small episode in which one of the maintainers in the project *Bugzilla* suggests an additional *process* element to the review process. To avoid being rejected with a patch because of faulty

design, the maintainer suggests an *optional design review phase* in which a reviewer would approve a design without this design already being implemented. The goal is to avoid superfluous effort on an implementation where the design was already faulty.

This episode was potentially *triggered* by a previous discussion between the maintainer and a mailing list participant on the difficulties of being a corporate contributor who cannot afford taking the risk of getting a patch rejected into which much effort has been invested [bugzilla:6774].

The introduction fails when the maintainer *abandons* the proposition after a critical question [bugzilla:6944]. To verify that this innovation has not been adopted, I manually searched for a design review flag on the bug tracker and searched the mailing list until 2008-09-25 for other occurrences of discussions related to such a design review, yet found no indication for its introduction.

- *Google Summer of Code @ Bugzilla* (episode.gsoc@bugzilla)  
This episode in the project *Bugzilla*, to apply to the *Google Summer of Code* is driven by two core developers and is never really challenged by anybody. As Bugzilla is part of the larger Mozilla *foundation*, the application process is much simplified for the project. The innovation introduction still *fails* because nobody applies for the offered projects. This highlights the danger of applying to the GSoC as part of a *foundation*: Mozilla's other projects could absorb the most capable students with more attractive projects (see Section 8.1.2).
- *Release Naming @ Bugzilla* (episode.release\_naming@bugzilla)  
The maintainer of the project *Bugzilla* proposes to more aggressively advance the major version number, because the public interpretation of the nine year delay between version 2.0 and 3.0 has been one of slow development at Bugzilla [bugzilla:6568].  
  
One core developer remarks on the psychological factor in equating version 3.0 with a certain set of features suggest by earlier prototype work [bugzilla:6576]. After a discussion on IRC, the project agrees to use more progressive version increases, yet keeps minor versions as well [bugzilla:6578].
- *Set Reply-to Headers to Sender @ Flyspray* (episode.mailinglist\_reply-to\_sender@flyspray)  
A co-developer proposes to switch the reply-to setting of the mailing list to target the sender of a mailing list post instead the mailing list itself because it caused him problems while replying privately. One of the maintainers *kills* this proposition by portraying the discussion as classic—which it is [190, p55-58]—and referring to the status quo. There is more heated discussion, but the initial two participants do not participate any more and no change is achieved (which indicates that the discussion has become a Garbage Can, see Section 6.2).
- *Plugin Architecture @ Flyspray* (episode.plugin@flyspray)  
This episode in the project *Flyspray* is not an innovation introduction episode but rather a design episode. It is included here because it was used in Section 4.3 to discuss a set of implications of software design on Open Source development.

In this episode a co-developer suggests that a feature proposed by the maintainer might be best implemented as a plug-in to the Flyspray software. This proposal is strongly *rejected* by the maintainer who disagrees that a plug-in architecture helps in solving the problem, because he feels that it only potentially attracts third parties who could implement the exact desired feature (he argues that such is very unlikely) and that the problem behind the feature still would first need solving.

- *Branching @ gEDA* (episode.branching@geda)  
This episode regarding the introduction of using *branches* into the development process of the project *gEDA* triggered the *episode of migrating to Git* [geda:4123] as a prerequisite of using branches.

It was successfully *executed* and the branching *scheme* frequently *used*, but the idea proposed initially to designate a stable maintainer who would be in charge of maintaining each stable branch created is not put into reality.

- *Git @ gEDA* (episode.git@geda)

This episode is about the introduction of the *decentralized source code management system Git* into the project *gEDA* (see Section 8.1.1 for a general overview of introducing a SCM). Because it is a very complex and long episode, continuing over more than one hundred messages, the following summary is naturally a drastic simplification of the real events, while including the essential parts of the episode:

The *successful* introduction of *Git* into the project *gEDA* can be traced back to a period of about six months, in which the two innovators—both core developers—continuously supported the introduction. Operating within their bounds of available resources and independently from the project itself, they set up an infrastructure centered around *Git* for working with the project source code and repeatedly promoted the use of *Git* for instance by linking to external opinion pieces on SCM choice [geda:3954].

With these activities they prepared the stage for the switch to *Git*, which occurred when the project *decided to adopt a new branching scheme*. The maintainer unilaterally makes the choice in favor of *Git* to support the new branching and release arrangement [geda:4123] and the actual conversion of the *gEDA* project repository takes place within two weeks.

There are numerous sub-episodes, for instance about the resulting processes of using a decentralized version control system or the role which individual tool preferences can play during an introduction (for instance *Cogito @ gEDA*).

- *Changelog for Git @ gEDA* (episode.git@geda.change log creation)

In this episode in the project *gEDA* one of the core developers proposes a change to the *scheme* for creating changelog messages to better accommodate the *upcoming migration to Git*. His proposal—which is successfully adopted—is an excellent example of using varied *enactment scopes* to convince fellow project members. First the proposal is described abstractly based on consequences of not adopting the innovation, and then made very concrete by giving detailed instructions on how to use the new scheme for the first time [geda:4330].

- *Cogito @ gEDA* (episode.git@geda.cogito)

This episode in the project *gEDA* is about the personal preference for the tool *Cogito* (see Section 5.10 on such tool preferences) by one of the innovators who introduced the *distributed SCM Git* into the project. *Cogito* is a set of helper scripts on top of *Git*, meant to simplify its usage. Using *Cogito* is not really required and its proposition thus by definition of the concept of tool-independence already a minor intrusion onto the territory of *optional innovation decisions*. The innovator in this episode thus is only suggesting its use by giving positive testimony and offering support to those who want to use it.

The introduction in this case *fails* ultimately when *Cogito* is discontinued by its developer, as *Git* had matured sufficiently to make the helper scripts redundant. Yet, even before this, reception was lukewarm at best. Unfortunately it could not be determined whether the lack of adoption was due to the narrowing gap of functionality that *Cogito* could provide or the preference for tool-independence that makes the personal development set-ups so drastically different that sharing and proposing tools is futile.

- *Stacked Git @ gEDA* (episode.git@geda.stacked git)

In this episode in the project *gEDA* one of the core developers suggests the usage of *Stacked Git* (*StGit*)—a set of scripts on top of *distributed SCM Git* for managing patches. In contrast to the suggestion to use *Cogito* in a separate episode, this proposal successfully achieves some adoption with less advertisement. This hints at the superiority of the *tool StGit* to provide a benefit to the developers in comparison to *Cogito*.

- *GPLv3 @ gEDA* (episode.gpl3@geda)

A core developer probes the introduction of *GPLv3*. The ensuing discussion is first serial in nature and then quickly becomes side-tracked into the technical/legal aspects of the proposed switch (see Section 6.2 on the Garbage Can). The innovator himself is not active in this discussion

and merely put the proposition out there (in his words: “I want to see what the community thinks” [geda:4674]).

Since no decision is being made and the innovator himself does not restart the discussion, this episode fails *abandoned*.

- *Google Summer of Code @ gEDA* (episode.gsoc@geda)

In 2007 the project *gEDA* successfully participates in the *Google Summer of Code* program. Proposed by a mailing list participant initially, this introduction is quickly executed by several core developers who are successful in applying to Google and getting two students to achieve a scholarship with the program. Participation of the students is not visible because they work in close contact with their students and partially successful at the end [geda:4844,4848].

Apart from being thus a smooth iteration of participating in the *GSoC*, there is one interesting interaction with another innovation introduction. Just as in other *GSoC* episodes the problem arises how to manage the work of the students, and just as in other projects it is suggested to provide access for the students to the repository. Similarly to the *GSoC episode in the project ArgoUML* does this present an opportunity which a skilled innovator can use for an introduction. In the case of *gEDA* the innovators working on the *long-term introduction* of the *distributed SCM Git* into the project use the opportunity to enumerate the advantages of letting students manage their work via Git and the dangers of not doing so [geda:3068]. This proposition still fails at this point because objections are raised, but however one can see the potential of the *GSoC* as a trigger for innovation introductions.

- *Licensing for Schema Files @ gEDA* (episode.license for schemas@geda)

This episode is triggered by an outspoken developer who is confused about the license used for *gEDA* symbols. Fairly quickly a core developer jumps into the discussion and helps clarifying the text used on the web-page by pointing to the FSF's handling of fonts. This core developer as the innovator then opens a thread for “comment/flame” [geda:3122] (discussed as a potential Garbage Can in Section 6.2), which indeed gets heated and brings up one alternative solution. The innovator himself strategically refrains from the discussion, asking only one clarifying question [geda:3123]. The maintainer of the project also stays away but rather makes his own comments on the issue in a new top-level thread.

In the end, the chosen solution is to stick to a revised version of the Font license exemption by the FSF [geda:3145]. *Execution* is quick and simple, because the number of copyright holders on the symbols is small [geda:3164].

- *Release Naming @ gEDA* (episode.release naming@geda)

When *gEDA/gaf* has its first official release at version 1.0, the question arises how to continue version numbers. The project settles on using “major.minor.maintenance.release date” and assigning odd minor numbers to unstable releases and even ones to stable releases.

- *Reinstate the Bug Tracker @ GRUB* (episode.bug tracking@grub)

In this episode in the project *GRUB* it takes four separate attempts to *successfully* clean the bug tracker of the project, which had been filled with bugs from the legacy version of *GRUB*. The episode is discussed in detail in Section 8.1.5 on bug tracking procedures in Open Source projects.

- *Changelog Format @ GRUB* (episode.changelog@grub)

In this episode a developer proposes to adopt a new changelog format, which contains more information about the purpose of a change, after a frustrating experience with the existing format, which lacked such information.

This episode fails because the innovator is mostly thinking problem-based [grub:4347] and thus he (1) loses important momentum towards his goal of reducing the problem and (2) does not arrive at a precisely defined solution to his problem (what he really needs is to have a place in the changelog to describe the overall effect and purpose of a patch).

First he does not provide a solution, which requires an additional exchange with a core developer asking for clarification. Then his solution (using the Linux kernel changelog format) does not make the benefit apparent that he wants to achieve. The maintainer can thus attack the proposal for aspects that are not of interest to the core developer who would otherwise have likely supported the change (the core developer wants a place for high-level descriptions, but the maintainer can attack the lack of detail in the Linux format).

This whole episode thus *fails* to achieve a change, except for the suggestion to be more disciplined with writing detailed descriptions. Such generic character or behavior-based solutions can not be deemed successful introductions.

- *Git @ GRUB* (episode.git@grub)

This episode in the project *GRUB* is a major example of the concept of *maintainer might*, in this case directed at preventing the introduction of a new *SCM* into the project. Initially proposed by a core developer and soon finding a broad base of support in the project, the idea to switch from antiquated CVS to a new SCM results in distributed *Git* being selected for execution in the project. It is only at this point that the maintainer interjects by *rejecting* *Git* on technical grounds and by reopening the discussion about finding an appropriate SCM implementation. While the maintainer lists several options and demonstrates his technical knowledge about the topic, it is visible that he is not interested in making a switch and the innovators become intimidated sufficiently to let the proposal die. Only in 2008 is the project successful in adopting the *decentralized SCM* Bazaar.

- *GPLv3 @ GRUB* (episode.gp13@grub)

The maintainer *proposes* switching to *GPLv3* and invites discussion about it. The resulting discussion centers on the question what to do with GRUB legacy, which cannot move to version 3 since it includes GPLv2-only code.

Some discussion focuses on the maintainer's wrong argument that maintaining GRUB legacy would become more difficult after making the switch, as patches would not be able to be backported (this is not really true since the copyright lies with the FSF and the project can just dual-license the patches for backporting). There is one hard-line supporter of the switch and one critical voice, asking for more thought and expertise in the discussion.

One week after the proposition, the maintainer unilaterally makes a decision (see the concept of *maintainer might*), bypassing all discussion and not replying to any concerns raised in the preceding discussion. Another week later, the migration is announced as complete.

As with the other *license switches*, we see that there is little legal expertise in the projects and many uninformed opinions exist. This makes discussion difficult. In contrast to the *license switching* episodes at *Xfce*, *gEDA* and *ArgoUML*, this episode ends in a *successful* execution, because of two major reasons: (1) The maintainer of the project is pushing for the migration. (2) GRUB does have a simple copyright situation as all legally significant contributors have assigned their copyright to the FSF. Even though the "or-later" clause allows the copyright switch to *GPLv3*, the problems of switching copyright without agreement by all contributors is raised in all other unsuccessful discussions.

- *Google Summer of Code @ GRUB* (episode.gsoc@grub)

*GRUB* participated in 2007 in the *Google Summer of Code* with minimal effort by (1) reusing the ideas of the previous year, (2) making the student from the previous GSoC a mentor and (3) letting the GNU Project handle the application procedure. The application of the GNU Project was successful (demonstrating the advantages of participating as part of a *foundation*) and a student was assigned to GRUB with the task of implementing booting from CD-ROM drives. The student's participation shows the typical problem of integrating the work of the student into the project [grub:3900].

- *Task Tracking @ GRUB* (episode.task tracking@grub)

This starts a long and winding episode about the question how to track outstanding tasks in

the project *GRUB* (this episode is related to the `episode.bug_tracking@grub`, which seems to display similar problems with lacking power to make decisions).

The episode consists of three discussions:

1. A core developer starts this episode by stating his goal to find a good way to store tasks. Here the discussion crosses the *bug tracking episode* as the proposal is made to use the bug tracker for tracking tasks, which the core developer rejects (without giving a convincing reason) [`grub:3247`].
2. The issue of task tracking is reraised by another core developer when he discovers the TODO file in the repository to be outdated and decides to clean it up [`grub:3707`].

Both discussions grind to a *dead end* as both core developers (as the innovators) seem to let themselves be side-tracked into marginal issues (the first developer falls for a funny situation [`grub:3250`] and the other for a minor rejection of his idea [`grub:3734`]).

3. On the third attempt in this episode, the second core developer gets an open and supportive reply from the maintainer regarding the question of whether to update or remove the TODO file in the repository. This causes him to come up with a piece of text to put into the TODO file which would refer to the wiki. Yet, instead of completing the execution directly (he does have the access rights to commit it), he still seeks the confirmation of the project members. This almost led to another failed thread, when it takes a week for a third core developer to reply supportively to the proposal.

Even though several times the idea of using a task tracker like Trac appears, the project never reaches enough decision power or structural power to accomplish an introduction of such a system.

- *Work Groups @ GRUB* (`episode.work_groups@grub`)

In this episode one core developer proposed to his fellow project member to team up in *work groups* to tackle long-standing issues such as the graphical boot menu. The introduction attempt fails, because the other project members jump too quickly to discuss possible work groups without having built a project-wide consensus in particular among the senior project members. After the initial enthusiasm fizzles out, the attempt thus fails.

- *Git @ KVM* (`episode.git@kvm`)

Unilaterally proposed by the maintaining company, the project KVM moved to Git in February 2007. This was an unspectacular innovation introduction, with only two interesting bits:

1. There was interaction with Linus Torvalds who was and is the pull-request recipient of this project. Especially in the beginning of adopting Git, the maintainer Avi made a couple of mistakes that highlight some of the issues that are complicated with Git (for instance which branch to pull from)
2. After a month it became visible how difficult it is to maintain the code in two repositories: One repository was used for the code that is part of the kernel and another for the user-space KVM parts that are independent of the kernel. A large discussion ensued about unifying these repositories without this discussion being resolved in the observed period.

- *Google Summer of Code @ KVM* (`episode.gsoc@kvm`)

A mailing list participant raises the question whether *KVM* will be participating in the *Google Summer of Code* with the hope of applying himself as a student. The maintainer responds vaguely by stating that it is difficult to predict what will be needed in the summer. There is no further post to the thread, thus I would conclude that this was pessimistic enough a reply to *kill the proposal* by a participant unlegitimized to take the innovation into his own hands.

- *Wiki @ KVM* (`episode.wiki@kvm`)

Only part of this episode is visible because the initial discussion on the mailing list precedes the observation period. A peripheral mailing list participant (who only appears in this episode on the list) initiates this discussion by reminding the project about previous promises to introduce a *wiki*

for the project. He offers a solution with little set-up costs on the servers of the *affiliated* Kernel.org website. Despite this solution being readily usable and perfectly functional, the proposition is ignored and the company backing KVM brings up their own official wiki within a week. This episode prompted the investigation into the concept of *hosting* which led to the development of the concept of *control* as an important aspect easy to miss (see Section 5.5 for all five concepts which could be related to hosting).

- *Bug tracking @ MonetDB* (episode.bug\_tracking@monetdb)  
One core developer from the project *MonetDB* *proposes* for developers to create bug reports for defects they are about to fix in the software even if no user ever reported the bug. The goal is to make the development process more transparent [monetdb:165], which would be achieved under the proposal in particular by e-mails being sent from the bug tracker to the mailing lists on any status change. The proposition is not replied to by anybody and thus must be assumed to have *failed*. The core developer himself addresses the major hurdle this innovation would entail, namely slowing down the developer by additional process steps, but seems to believe that this downside is worth the additional quality assurance received.
- *Git @ ROX* (episode.git@rox)  
In the project *ROX* a *partial migration* from existing *Subversion* to *Git* was *proposed* in June 2007. The migration was partial, because the maintainer only proposed to migrate the ROX-Filer, which is the central component of *ROX* to *Git* (for details see Section 5.3). The migration is *executed* with in twenty-four hours of the proposal being made and thus highlights both (1) the influence a maintainer can have on the introduction of innovations in a project, and (2) the possibility to affect wide-reaching changes in relatively short stretches of time.
- *Merge Conflicts @ U-Boot* (episode.git@uboot.merge\_conflicts)  
This episode is about changing the format of the build-file *MAKEALL* to be more resistant regarding merge conflicts by placing each item in the file on an individual line.  
  
This episode is started by a developer who sends a patch to the list performing the change to the new format and is referring back to a suggestion by another developer. Since I could not trace back this initial suggestion, one is left to wonder if the first developer as the innovator possibly uses an interesting tactic to overcome potential resistance.  
  
The maintainer of the project is against the change and simply *rejects* it. In contrast to other episodes where such a decision by the maintainer often marks the end of a discussion, one of the core developers opts for the change and starts to *call for a vote* (see Section 5.7). This immediately brings up two votes in favor of the change, but the discussion becomes side-tracked into another possible modification to the make files (which is *rejected*). Only by a third vote (out of more than twenty active developers on the list) one day later is the topic revived and the maintainer's resistance then overcome.
- *Personalized Builds @ U-Boot* (episode.personalized\_builds@uboot)  
As part of the episode to make the build-file in the project *U-Boot* more robust against merge conflicts, a developer is proposing to make it also easier for developers to customize their build environment for local changes that are not part of the public distribution. This is rejected by the maintainer and a core developer on the basis that they prefer that people do not exclude custom code from the U-Boot repository. In this sense, the proposal is too much in violation of the Open Source principles and norms to be accepted.
- *Release Naming @ U-Boot* (episode.release\_naming@uboot)  
Small episode in the project *U-Boot* in which *year/month-based release names* are proposed [uboot:31353], but directly rejected by the project maintainer [uboot:31363]. Interestingly, one year afterwards the project adopts the very date-based releases it rejected [uboot:46175]. In Section 8.1.4 it is argued that adoption of a *date-based release scheme* is correlated with project maturity.
- *White Space Tooling @ U-Boot* (episode.white\_space@uboot.tools)

In this episode one of the *custodians* (developer responsible for managing contributions to a certain module) offers to the project a script which can clean up whitespaces violating the project's *U-Boot coding conventions*.

With other *tool innovations* similar to this script I have seen that it is common to include them into the repository of the project. This does not occur in this episode, rather first one suggestion is made to make the source code of the tool more readable. Then two suggestions for how to solve the problem of trailing whitespace in different ways are made by project members. One of these makes use of an option in the *source code management system* used by the project and is integrated into the project's wiki page for custodians. Thus instead of using the proposed innovation, the project seems to have found a preferred solution in the discussion and formalized it into a *convention*.

We can understand this episode as an example of (1) how execution and discussion can be reversed (this innovation is first executed and then discussed) and combined to solve a problem or achieve a goal, and (2) how innovations can relate to knowledge about innovations.

- *Milestones for Bugs @ Xfce* (episode.bug milestone@xfce)  
In this episode one developer proposed to assign *milestones to bugs in the bug tracker* to enhance the visibility of progress measures for users and developers. This episode ended in a *dead end* failure when the innovator gathered no replies to his proposal. Interestingly enough, a couple of days later a similar proposal was accepted (see Episode *bug tracking cleaning@xfce*). The discrepancies between both episodes triggered the development of the concept of *enactment scopes* (see Section 5.4).
- *Cleaning the Bug Tracker @ Xfce* (episode.bug tracking cleaning@xfce)  
*Triggered* by the 4.4 release of *Xfce*, a peripheral developer asks whether this would be a good time to clean up the bug tracker. One core developer asks him back whether he would volunteer for the job. The peripheral developer is willing to do so but asks for rules by which to do the cleaning. He receives several heuristics and is granted the associated *access rights* to edit bugs in the bug tracker.  
  
When he starts the clean-up 7 months (!) later [xfce:14195], he immediately gets into conflict with two core developers, because he did not proceed with the heuristics initially given, which caused a lot of e-mails to be sent to the two of them. The harsh reactions this evokes potentially cause some harm, as the peripheral developer finishes his last e-mail on the topic with a somewhat disillusioned “Just trying to do my bit to support the projec [sic]” [xfce:14204].
- *Commit Script @ Xfce* (episode.commit script@xfce)  
A peripheral developer offers a small script as a *tool innovation* to the project, which can assemble a commit message for *Subversion* from “changelog” files. Even though the innovator receives commit *rights* to the repository because of this proposal [xfce:13131], it is not clear whether the innovation was adopted by anybody.
- *GPLv2 only @ Xfce* (episode.gpl2only@xfce)  
A long-time contributor proposes the *legal innovation* of switching the license of the project to “GPLv2 only” by removing the update-clause in the GPL. The main reason for proposing is based on the wrong assumption that the release of the *GPLv3* would cause an automatic licensing upgrade and the proposition is dismissed by 5 rejections to 2 supports. The proponent—faced with such opposition—*abandons* the discussion and the introduction attempt has thus failed.

Two points stand out in the discussion: (1) The discussion is highly parallel, with seven replies to the original posting. (2) The lack of knowledge about the ‘technical’ aspects of the innovation (in this case the legal details) are very visible and are the likely cause for the first point as well: With much opinion about the innovation but little well-informed knowledge on the technical domain to counteract the opinion of others the discussion necessarily fans out in many directions which are not replied to. One could easily imagine that if there had been a bit more knowledge on the issue, then a heated Garbage Can could have easily emerged from this discussion (compare Section 6.2).



- *Google Summer of Code @ Xfce* (episode.gsoc@xfce)

This episode in the project *Xfce* about a proposal to participate in the *Google Summer of Code* in 2007 has to be seen within the context of *Xfce* having been rejected in 2006. When the application deadline in 2007 approaches and a student voices his interest of working as a GSoC fellow this summer, the maintainer chooses not to get involved with applying a second time stating “I’m not sure *Xfce* would fit in Google’s interested projects” [xfce:13244] despite there being some enthusiasm for generating ideas [xfce:13243].

The failure of the innovator to get the project to participate can be explained using three plausible alternatives:

- The innovator shows lacking commitment or *dedication* to follow through with the proposal, essentially *abandoning* the episode.
- The maintainer answers too negatively to the proposal, thereby withdrawing too much legitimation from the proposal, effectively *killing* it.
- The co-developer supporting the proposal causes the proposal to be side-tracked too much into details without considering the proposal as such; an outcome common for a figurative Garbage Can (see Section 6.2).

When looking into 2008, one of the core developers picks up the idea to get involved in the Summer of Code [xfce:14850]. While he achieves an application to Google with many project ideas [580], the application is not successful [xfce:14962], which is discussed even outside of the project as a notable rejection in the GSoC program [574]. On the one hand this second rejection strengthens the maintainer’s suggestion that Google might not be interested in a project such as *Xfce* [xfce:13244]. On the other hand, a blogger remarks that the ideas submitted might have been of inferior quality: “all the other ideas look terribly uninteresting and lacking any vision!” [569].

## A.2 Innovations

An innovation is a means for changing the development process in an Open Source project. The term is used as an umbrella to unify different *innovation types* (see Section A.8) such as *processes*, *tools* and *services*.

Alternative terms which could have been used include “technology” [82] or “invention” [140], which carry their own connotations.

**Disambiguation:** One common alternative interpretation of innovation is to associate the term with the delta of change. For instance when switching from technology A to technology B, the distance between B and A according to some dimension could be seen as the innovation [82]. In this thesis though, A and B are innovations.

- *Allow Reporter to Verify* (innovation.bug tracking.allow reporter to verify)  
This innovation defines a bug tracking process element by which the person who reported a bug is allowed to verify the proposed solution to resolve the issue. Alternatively, bug tracking procedures often mandate that different persons report, fix and verify an issue.
- *Evaluate Patches in Branches* (innovation.bug tracking.evaluate patches in branches)  
A third possibility besides evaluating the patches on the mailing list (i.e. classical *peer review*) or as bug tracker attachments is this innovation: Take a patch and put it into a separate branch. While discussion still takes place on the mailing list, modification and collaboration is much simplified.

In particular the advent of *distributed SCM systems* which allow private branches with little effort makes this process element attractive.

- *Integrated Bug Tracker* (innovation.bug tracking.integrated bug tracker)  
An integrated bug tracking system such as Trac<sup>231</sup> combines a bug tracker and a *SCM* and possibly other project infrastructure services such as a *wiki* or a *continuous integration* (CI) system. This enables more efficient operations such as closing a bug by committing to the repository or conversely seeing all commits which affected a certain bug.
- *Assigning Milestones to Bugs* (innovation.bug tracking.milestoning)  
This innovation involves the idea to mark or tag each bug in the bug tracker with a target milestone or release. The proposed advantage is increased overview for users and developers about the status of an upcoming release.
- *Online Developer Demo System* (innovation.build.online devel system)  
An online developer demo system tries to solve the problem of projects that produce server applications which are difficult to set up and *host*. For developers and users to report bugs against the unstable version thus becomes difficult since it requires a large time-investment. Instead, this innovation mandates that the project itself should provide an up-to-date unstable demo system against which users can report problems.
- *Coding Standards* (innovation.coding standards)  
A coding standard is a recommendation or *convention* on how to write source code. This commonly includes the layout of the source as defined for instance by indentation rules, when to break lines, or which type of whitespace to use, but also suggestions for how to devise names for variables and classes, and best practices for error handling.
- *coding standards.whitespace tool* (innovation.coding standards.whitespace tool)  
A tool that deals with adherence to the white space coding standard.
- *Internet Relay Chat (IRC)* (innovation.communication.irc)  
Internet Relay Chat (IRC) is an open standard for server-based real-time chat communication. It has become popular in Open Source projects because of its mode of operating a channel on a centralized server, which enables project members to drop into a discussion during their time working on the project and leave again at any point. In this way it combines synchronous and asynchronous communication in a practical way for globally distributed people in a project.
- *IRC Gateway* (innovation.communication.irc.irc gateway)  
An Internet Relay Chat (IRC) gateway is a web application by which people can participate in IRC discussions of the project.  
  
Project participants can use the gateway if they cannot or do not want to install an IRC client or if they are blocked because of network issues. An IRC gateway can be seen as an adapter from web to IRC technology (see Section 5.6).
- *Task Tracking* (innovation.coordination.task tracking)  
A task list (also often to-do list) is a document in which the project members collect tasks to perform such as feature requests, bug fixes, architecture refactorings, etc. Similar to prioritization in a bug or feature tracker, a task list is a method for enhancing coordination in the project by highlighting tasks to project members.  
  
Task lists have been suggested to improve recruiting of new project participants [494, p.98], but choosing a task granularity matching the abilities of newbies is then important [533].  
  
Keeping task lists current is part of the suggested responsibilities for the *information manager*.
- *Work Groups* (innovation.coordination.work groups)  
A work group is a way to coordinate tasks in a project by creating groups of people who work together on a specific issue.

---

<sup>231</sup><http://trac.edgewall.org/>

Since such close collaboration is in natural opposition to the loosely coupled work ethos in Open Source projects, it is unclear whether work groups are really beneficial.

- *Design Approval* (`innovation.design.approval`)  
A design approval is a *process innovation* by which people can request their design ideas to be scrutinized before implementation. This additional process step can prevent effort to be wasted on producing a patch, which is rejected on design reasons later on.

- *API Docs* (`innovation.documentation.api docs`)  
An *API documentation tool* (or inline documentation tool) such as Doxygen<sup>232</sup> is an example of a *tool innovation*. Such a tool first defines a set of keywords to be used inside of source code comments. Given source code annotated with such keywords, the inline documentation tool is then able to generate documentation of the application programming interface (API) of the annotated software.

Note that the introduction of an inline documentation tool which can be run by each developer individually to generate documentation (thus constituting an *optional innovation decision*) is in many cases combined with *rules and conventions* that determine when and how to add annotations (thereby making the innovation decision an *expected one*).

- *Contributor Guide* (`innovation.documentation.contributor guide`)  
A *contributor guide* is a *document* which explains and summarizes how to participate in the project with a particular focus on the technical aspects such as checking out the source code, compiling the project, making a patch or report a bug.
- *Google Summer of Code (GSoC)* (`innovation.external.GSoC`)  
“The Google Summer of Code is a program which offers student developers stipends to write code for various open source projects.”<sup>233</sup>

Participating in the GSoC is discussed in detail in Section 8.1.2.

- *Information Manager* (`innovation.information manager`)  
A person in charge of managing all user-driven sources of information such as mailing lists, bug trackers and wikis. In particular the person is in charge of looking for duplicated or incorrect information, create FAQ entries of repeatedly asked questions, welcome new developers and help them get accustomed to the project, take note of decisions and information relevant for project management and put them into appropriate places.

This role-based innovation is extensively defined and discussed in Section 7.1.

- *Self Introductions* (`innovation.join script.self introduction`)  
A *process innovation* targeted at community building. Persons interested in participating in the project are asked to introduce themselves to the mailing list with several sentences about their professional interest in the project and their personal background.
- *Foundation* (`innovation.legal.foundation`)  
To create or join a foundation which can be used as an entity representing the project in legal, representational, or monetary matters.

Well-known examples include the Apache Software Foundation (ASF)<sup>234</sup> and the Free Software Foundation (FSF)<sup>235</sup> which are powerful representatives of the Open Source cause. Lesser-known examples such as the Software Freedom Conservancy<sup>236</sup> provide more specialized services such as legal protection or money handling for Open Source projects.

<sup>232</sup><http://www.doxygen.org>

<sup>233</sup><http://code.google.com/opensource/gsoc/2008/faqs.html>

<sup>234</sup><http://www.apache.org>

<sup>235</sup><http://www.fsf.org>

<sup>236</sup><http://conservancy.softwarefreedom.org/>

The primary positive implication of being a member project in a foundation is that the member project can act for some matters under the label of the foundation. For instance when considering the *Google Summer of Code*, a foundation can apply as a single organization with a large number of mentors, project ideas and backup administrators, thereby vastly increasing the chances of being accepted by Google (see Section 8.1.2).

Benefits, rationales and consequences of founding or joining a foundation have been well discussed by O'Mahony [387].

- *GNU GPLv3* (`innovation.legal.gplv3`)

The GNU General Public License Version 3 [196] was created in 2007 by the Free Software Foundation (FSF) to pay tribute to several changes in the technological and legal landscape such as the increased use of digital rights management, use of Open source software in embedded devices (tivoization) and application of patents to software [480]. The key-principle of the GPL as a strong copyleft license remains unchanged though.

- *License Switch* (`innovation.legal.switch license`)  
To change the licensing terms of the project or part of it.

Introducing a new license into an Open Source project is usually difficult because all copyright holders need to agree to the switch for the files they worked on. Only the larger, well-organized projects such as the GNU project or the Apache Foundation seem to be able to handle the logistics of either gathering copyright assignments from all individuals right from the beginning [190, cf. pp.241f.] or contact them individually about the licensing switch.

- *Peer Review* (`innovation.peer review`)

Peer Review (also Code Review) is one of the primary quality assurance mechanisms in Open Source development and entails the review of submitted code by other participants in the project. This can be conducted prior to commit by sending patches for review to the mailing list (*Review-Then-Commit*, RTC) or post-hoc when a commit triggers a commit message to be sent to a mailing list for review (*Commit-Then Review*).

Performing peer review is said to increase the spread of knowledge about the code base in the project [190, pp.39f.] and has been studied in-depth by Stark [483].

- *Continuous Integration (CI)* (`innovation.qa.continuous integration`)

Continuous Integration (CI) is a quality assurance technique based on automating the process of integrating the components of a project and running it continuously to monitor the status of the code base [195]. Typically, to integrate components one has to make a full build of a project and run automated tests. If problems occur during build or test, the developers are notified and can react quickly.

- *Static Checkers* (`innovation.qa.static checker`)

A static checker is a *tool innovation* which verifies properties of the source code such as adherence to *coding standards* without running the program (hence “static” checking). Popular examples of static checkers for the Java programming language include CheckStyle<sup>237</sup> or FindBugs<sup>238</sup> [13].

- *Real World Meetings* (`innovation.real world meeting`)

Since Open Source development is mostly done using electronic means in the “virtual world” of the Internet, it represents usually a next stage of evolution for an Open Source project to have meetings in the “real world”. Most common are such meetings as part of Open Source community events such as the Free/Open Source Developer European Meeting (FOSDEM)<sup>239</sup> [xfce:13819] or the Ottawa Linux Symposium (OLS)<sup>240</sup> [uboot:29717].

<sup>237</sup><http://checkstyle.sourceforge.net/>

<sup>238</sup><http://findbugs.sourceforge.net/>

<sup>239</sup><http://www.fosdem.org/>

<sup>240</sup><http://www.linuxsymposium.org>

The mentor summit of the *Google Summer of Code* (see Section 8.1.2) has also been mentioned as a good opportunity for senior project members to meet [argouml:5512].

- *Merge Window* (innovation.release process.merge window)  
The use of a *merge window* is a *process innovation* to increase the quality and regularity of releases. After each release the maintainer allows for new features to be merged into trunk during a period which is referred to as the merge window. After the merge window closes, commits are often restricted to bug-fixes and localizations with the intent to encourage testing and debugging of the new features (this intention is not necessarily achieved, as short merge windows make it often necessary to directly start working on new features to be included during the next merge window).

A merge window is often combined with a fixed interval release *scheme*, *date-based release naming* and a single maintainer or *custodian* who performs the role of a *gate keeper* to enforce the merge window.

- *Date-Based Versioning* (innovation.release process.naming of version.date based)  
A *date-based naming scheme* emphasizes the “continuous[ly] rolling improvement” in Open Source development over the “earth shattering discontinuities” [uboot:31353] one would associate with an increase in the major number of a software version. Date-based releases in combination with a *merge window* to ensure quality have become more and more popular as many Open Source projects have matured and thus found it much harder to coordinate an ever-growing number of volunteers to achieve a list of self-set goals for a release. Lacking any authority structure to enforce timely completion of tasks has often caused releases to be delayed and thus led eventually to the adoption of date-based releases, in which the list of features is variable while the release date is not.
- *Milestone Versioning* (innovation.release process.naming of version.milestones)  
After a stable release in a milestone *naming scheme* for releases (sometimes also called alpha-beta scheme) there are consecutive milestone releases m1, m2, m3,...until a sufficient amount of new features has been added or enough time passed since the last stable release. The project then releases a new stable release based on such a milestone and repeats. To assure quality, it is common to add “release candidate”-milestones prior to a release, in which the number and size of new features is largely reduced.
- *Nickname Versioning* (innovation.release process.naming of version.nicknames)  
A *naming scheme* using “nicknames” as identifiers for releases can make complicate version numbers easier to remember. In the project *MonetDB*, the names of planets were proposed [monetdb:369], while the distribution Ubuntu uses animal names preceded by an alliterating adjective such as “Feisty Fawn”.<sup>241</sup>
- *Stable/Unstable Versioning* (innovation.release process.naming of version.stable unstable)  
A stable/unstable release *naming scheme* alternates between stable and unstable releases by designating them as even (typically stable) and odd (typically unstable). Most famously the Linux kernel used to follow this naming scheme until the stable kernel 2.6 was released after a stretch of three years in unstable 2.5. Since these three years were perceived as too long for in-between stable releases, the scheme was abandoned starting with 2.6. The Linux kernel is now being developed using a *merge window* of two weeks followed by two months of stabilization [kernel:706594]. As a result, the naming scheme is now *milestone-based* with 2.6 being the unchanged prefix since December 2003.
- *Source Code Management (SCM) system* (innovation.scm)  
A source code management system (SCM) is one of the most central innovations for the use by an Open Source project because it enables the efficient cooperation of project participants on the project files. One typically distinguishes between centralized SCMs such as *Subversion* and *decentralized SCMs* such as *Git*, which describes how the system stores the managed data.

<sup>241</sup>See <https://wiki.ubuntu.com/DevelopmentCodeNames>.

SCMs are discussed in detail in Section 8.1.1.

- *SCM Adapters* (`innovation.scm.adapter`)  
An adapter for a *source code management system* is a tool which adapts an existing *SCM* to be used by a particular client tool. Examples include *git-svn*<sup>242</sup> and *tailor*<sup>243</sup>.

Adapters in general are discussed in Section 5.6.

- *Default Branching Scheme* (`innovation.scm.branching`)  
The standard *convention* of using branches of a *source code management system* in an Open Source project is to develop in the trunk and to create branches in which a release is stabilized. This standard way is usually adopted early on in the history of an Open Source project when the number of participants increases and some want to continue developing new features during the stabilization of a release.
- *Custodian* (`innovation.scm.custodian`)  
A custodian is a role-based innovation used in particular with decentralized repositories in which maintainers are needed to coordinate contributions to sub-projects.

This tree-like model of development draws directly from the way the Linux kernel is developed, with Linus Torvalds at the top who communicates with a set of “trusted lieutenants” [462] who in turn communicate with people they trust, etc. This model is particularly suited when development can be split along well-defined lines such as module boundaries.

- *Distributed Version Control System (DVCS)* (`innovation.scm.dvcs`)  
A distributed version control system (DVCS) is a *source code management system* which does not require a central repository to *host* the revision control data. Rather, this data is replicated by each developer who is participating in development. A DVCS commonly allows each user who has obtained such a replica (also called clone) to work independently from the original master copy. Since the replica is truly identical, what defines *the official repository* is rather a social convention. For instance in the Linux kernel this naturally is Linus Torvalds’, which is being tracked by others.

This principle of *primus inter pares* (first among equals) has many important implications on power relations which existed with centralized *SCMs* because of the need to assign commit-*rights* to the single central repository.

Distributed version control is discussed in detail in Section 8.1.1.

- *Git* (`innovation.scm.git`)  
Git is a *distributed version control system* developed by Linus Torvalds<sup>244</sup> as a *SCM* for managing the source code of the Linux kernel [230]. Git has gained widespread popularity with Open Source projects in the year 2007 (see Section 8.1.1).

Like other distributed *SCMs*, Git allows a “pull-based” mode of operation which does not require anybody to be granted commit access while retaining versioning capabilities for all developers involved.

Git’s advantage over other distributed versioning tools (such as Bazaar<sup>245</sup> or Mercurial<sup>246</sup>) is primarily its performance with the daily tasks of Open Source project maintainers.

- *Gitweb* (`innovation.scm.git.gitweb`)  
Gitweb is a *web application* for browsing *Git* repositories. It is similar to ViewVC<sup>247</sup> for CVS or ViewSVN<sup>248</sup> for *Subversion*.

<sup>242</sup><http://www.kernel.org/pub/software/scm/git/docs/git-svn.html>

<sup>243</sup><http://progetti.arstecnica.it/tailor>

<sup>244</sup>A good overview by Linus Torvalds himself can be found in the following video lecture Torvalds delivered at Google <http://www.youtube.com/watch?v=4XpnKHJAok8>.

<sup>245</sup><http://bazaar.canonical.com/>

<sup>246</sup><http://mercurial.selenic.com/>

<sup>247</sup><http://viewvc.org/>

<sup>248</sup><http://viewsvn.berlios.de/>

- *Subversion (SVN)* (`innovation.scm.svn`)  
Subversion (abbreviated as SVN) is the most popular successor to the centralized Concurrent Versioning System (CVS). Subversion improves on several issues with CVS such as renaming and moving files, while keeping the workflow as compatible as possible. Subversion is criticized by proponents of *distributed SCMs* for its slow speed of branching and lack of off-line commit capabilities.
- *Wiki* (`innovation.wiki`)  
A WikiWiki (often only wiki) system (from the Hawaiian word ‘wiki’ meaning ‘fast’) is a content management system which is especially tailored towards collaborative and easy editing [309].

Due to their simple, collaborative nature wikis are often used for *information management* tasks in Open Source projects such as *task tracking* or coordinating *real-world meetings*.

## A.3 Activities

An activity is the conceptualization of the things the author of a message has done and is reporting about or is now doing by writing an e-mail.

An example of a type of activity typically reported about is *the execution of an innovation*: The innovator usually works in the background on setting an innovation up, and after he is finished he will report about the result of his work. Conversely, *proposing* an innovation typically is an activity that occurs within the e-mail.

- *announce* (`activity.announce`)  
The announcement of an innovation is defined as the moment during or after *execution*, when the innovation is declared as officially ready to be *used*. This can happen explicitly [`bugzilla:6190`] or implicitly [`uboot:29736`].
- *execute* (`activity.execute`)  
An activity which is concerned with making an innovation usable for within a project. Such an activity might include installing software, migrating and converting data, or writing documentation.

**Disambiguation:** It is not always possible to distinguish activities that are executing the innovation from activities of *using the innovation* (`activity.use`). To illustrate the difficulty to disambiguate *using* and *executing*, consider for example that setting up (and thus *executing*) the *source code management system Git* involves both installation of a Git server, which is clearly *executing*, but then also often includes the migration of existing data, which involves *using* the system as well.

- *narrate execution* (`activity.narrate execution`)  
Giving a detailed account of the steps involved in *executing* the innovation for *use* in the project.

**Disambiguation:** This activity is meant to distinguish messages informing that an innovation was executed (`activity.execute`) from those that explicate how and following which steps this was done (`activity.narrate execution`). Narration seems to be such a distinctive literary form employed in e-mail-based communication that it warranted its own code. Yet, often these two aspects of whether and how become mixed so that the following way to disambiguate can be used: A narration involves at least a minimal sequence of steps that could be used to reexecute the setting-up of an innovation.

- *offer* (`activity.offer`)  
To offer an innovation is to *propose* an innovation that is ready for *usage* by the target audience. As with a proposition the intention of an offer is to seek the opinion of the project.

**Disambiguation:** Contrasting *offering* to *announcing*, the latter is about officially releasing an innovation to the target audience for usage rather than proposing it for discussion.

- *propose* (activity.propose)  
To propose an innovation is the activity to start a discussion about an innovation. Important properties of a proposition are its intention (which might not always be possible to figure out) or for instance which kind of perspective on innovation is being taken.
- *sustain* (activity.sustain)  
An activity aimed at supporting an innovation which is already in use. Such sustaining can include performing software upgrades to a service innovation [flyspray:4417], clarifying the capabilities of an innovation in a document [kvm:1372], granting commit rights to new developers [flyspray:5320], deleting log-files [uboot:31112], etc.
- *use* (activity.use)  
Activity of using an innovation (for instance posting a patch for peer review) or describing an innovation use (for instance telling others that a patch has been committed to the new SCM).

## A.4 Concepts

A concept is an abstraction of phenomena observed in data. Concepts are constructions of the researcher, who moves from labeling phenomena (coding) to description and understanding and finally to theoretical analysis. A rich concept should have its origins and implications explored, be well-defined and connected to other concepts.

- *control* (concept.control)  
The degree to which the potential capabilities of an innovation and activities related to it can be actually used or performed by the project.

For example, in the project *Flyspray* the password to the mailing list software had been lost so that the project participants could not perform any *sustaining* administrative operations [flyspray:5411]. This lack of *control* prompted them to migrate to a new mailing list service.

**Disambiguation:** Note that the *concept of control* does not cover the question of who can actually perform a particular task. This question is covered by the concept of (*access*) *rights*.

- *effort* (concept.effort)  
The amount of time, physical or mental energy necessary to do something. Effort is a key component to understand how anything gets done in a project.
- *enactment scope* (concept.enactment scope)  
The set of situations in which the activities mandated by a process should be executed (see Section 5.4). Such situations can stretch out over time (for instance *performing a peer review* every time a new patch is submitted), people (for instance *performing a peer review* only for authors without commit rights), or over other dimensions such as software components (for instance *performing a peer review* only for components in the core of the application).
- *forcing effect* (concept.forcing effect)  
A property or mechanism of an innovation that promotes the *use* of this innovation (see Section 5.7).
- *hosting* (concept.hosting)  
Provision of computing resources such as bandwidth, storage space and processing capacity, etc. for the use by an innovation.

I identified five concepts that seem to influence most of the decisions people make towards hosting: These are (1) *effort*, (2) *control*, (3) *identification*, (4) *cost*, and (5) *capability* (see Section 5.5).

Hosting is an aspect of innovation that applies especially to *service innovations*, since they need a centralized system to work, which in turn requires hosting. Yet, one should not forget that



also other types of innovation might need some minor hosting (for instance, *process innovations* are often enacted using software or formalized into *documents* both of which need to be hosted somewhere).

- *identification* (`concept.identification`)  
The degree to which innovations (or more generally behavior, situations or artifacts) are in accordance with the norms and identity of the particular project and general Open Source community. Identification might for instance be violated by using proprietary tools, not giving credit to contributors or *hosting* on private servers.
- *maintainer might* (`concept.maintainer might`)  
Concept highlighting the salient role of the maintainer during many innovation introductions. The maintainer's power for instance arises from *control* over *hosting* resources, opinion leadership in discussion or legitimization for unilateral action.
- *partial migration* (`concept.partial migration`)  
An innovation introduction in which only parts of an existing innovation are replaced by a novel one and thus two innovations co-exist side-by-side (see Section 5.3).
- *participation sprint* (`concept.participation sprint`)  
A period of highly intensive activity by one or several project members as indicated for instance by a large number of written e-mails or code submissions (see Section 5.8). Participation prints are certainly highly productive but hold danger of overloading the capacity of the project.
- *rights* (`concept.rights`)  
Rights (or access control) refer to the degree to which a project participant can use an innovation. For example, a project might grant anonymous read access to its *source code management system* but restrict write access to those who have contributed several patches. Rights and the rights to assign rights are one of the primary sources of structural power in Open Source projects [229] and are often connected to *control* over *hosted* systems.
- *time* (`concept.time`)  
A very broad concept covering the influence of time on the introduction of innovations (see Section 5.8). This concept for instance manifests itself in phenomena such as participants having difficulties of setting a time for a meeting on *IRC*, there being a surge in innovation activity after a release has been shipped or a core developer being absent from the project for two weeks without this being noticed by others.

## A.5 Compliance Enforcement Strategies

*Compliance enforcement* is an attribute of an innovation introduction related to the question how its *usage* or the quality of its usage is ensured. For instance, we have seen a maintainer *plea* with other members of the project to improve their changelogs [xfce:13533].

Lawrence Lessig's book "*Code and other Laws of Cyberspace*" is a good source for ways to enforce compliance (in his terms 'to regulate freedom'): Law, Norms, Market, Code [307].

- *code* (`compliance enforcement.code`)  
An innovation using *code to enforce compliance* is one where a software or hardware system does enforce a certain way of usage. For instance, a mailing list server might check that all posts are made by subscribers of the list [bochs:7274].
- *forced undo* (`compliance enforcement.forced undo`)  
Forcing an undo of modifications by a violator of norms and guidelines is a post-hoc compliance enforcement strategy. By removing a contribution from the project's repository, the effort spent

by the violator becomes invalidated. Consequently, the larger the contribution and the smaller the effort to perform the undo, the more effective the strategy is of threatening a *forced undo*.

- *gate keeper* (compliance enforcement.gate keeper)

A *gate keeper enforcement strategy* is one where an outside and inside area can be distinguished and a dedicated mechanism exists by which access to the inside area is granted. For many innovations, quality might be assured primarily by this model of a diligent maintainer as gate keeper to the *repository* of the project.

We might further distinguish gate keeping by contributor and gate keeping by contribution as for instance with centralized and decentralized source code management tools.

- *social.plea* (compliance enforcement.social.plea)

A plea in compliance enforcement is a weak form of moral (i.e. social) enforcement. For instance, one maintainer asked of his fellow project members to “Please be kind to your release manager(s), think of the NEWS file” [xfce:13533].

## A.6 Episode Outcomes

An episode outcome is an abstraction of the result of an attempt to introduce an innovation into an Open Source project. The episode outcome thus is the central measure by which the innovator would assess the success of his or her efforts.

- *failed* (outcome.failed)

An episode fails if the innovator cannot achieve the intended goal, solve the associated problem or convince project members to use the proposed innovation.

Such failure might occur at many different points during the introduction of an innovation: The innovator can fail to gather an *organizational innovation decision* in favor of the innovation (i.e. the innovation is *rejected*), *fail to execute* the innovation or *fail to achieve adoption* of the innovation.

- *abandoned* (outcome.failed.abandoned)

An *abandoned episode* is one in which there is no apparent reason why the episode did not continue except for a failure of the innovator to continue to engage the rest of the project or execute the innovation himself.

A common post-hoc explanation for an episode to end in abandonment is reduced engagement or low *dedication* of the innovator, for instance at the end of a *participation sprint*.

**Disambiguation:** An abandoned episode contrasts best with a *rejected* episode in that the latter contains active opposition to the proposed innovation, where an abandoned episode might even contain support for the innovation.

- *dead end* (outcome.failed.deadend)

A *failed* innovation episode is called a dead end, if the last person to write an e-mail is the innovator. Thus the innovator was unable to draw any response from the project. Such episodes might be used to understand the tactical mistakes that can be made.

**Disambiguation:** A *dead end* most naturally is always also an *abandoned* episode, because the innovator could have always come back and have restarted the discussion by new e-mail. So, to disambiguate we always code a dead end if the innovator is the last person to write a message with regards to the innovation.

- *no adoption* (outcome.failed.no adoption)

The introduction of the innovation got to the point that it was *executed* and *announced*, but no adoption or usage beyond the innovator himself can be seen.

- *failed.no execute* (outcome.failed.no execute)  
An introduction episode failing in this way did successfully decide to adopt an innovation but has not *put into reality* the necessary preparatory steps such as setting up software, devising rules, or assigning roles prior to starting individual adoption.
- *failed.rejected* (outcome.failed.rejected)  
An episode ends rejected if the primary reason for the end is the opposition in the project to the new idea.
- *killed* (outcome.failed.rejected.killed)  
A killed innovation introduction is one where a high ranking member *rejects* the innovation and this has a substantial impact on the episode ending in *failure*.  
  
A *killed* innovation introduction was introduced as a special type of *rejection* to connect such failures to the concept of *maintainer might*.  
  
**Disambiguation:** In case of difficulties to distinguish a substantial and a non-substantial impact on the episode outcome, code rather a *plain rejected* than a *killed* outcome.
- *postponed* (outcome.failed.rejected.postponed)  
An episode ends postponed if the project rejects the innovation but says that the proposal will be revisited sometime later.
- *success* (outcome.success)  
An innovation is successfully introduced when it is used on a routine basis and has solved the problem it was designed to solve or attained the goal it was designed to attain (also compare with Section 2.3.10 on how to define success in the Open Source world).
- *unknown* (outcome.unknown)  
Used if it was not possible to determine by looking at the available data whether the innovation was *successfully* introduced in the enclosing episode.
- *unknown adoption* (outcome.unknown.adoption)  
An episode outcome is unknown with regards to adoption if by looking at the mailing list and other publicly available sources of information it is not obvious whether the innovation was adopted by its target audience or not.
- *unknown.success* (outcome.unknown.success)  
Used for episodes in which it was not possible to determine whether the intended goal of the innovation was achieved or not.

**Disambiguation:** This code is used in contrast to *outcome.unknown.adoption*, when the idea of ‘success as adoption’ does not seem fitting. For instance, we have seen an innovation which does not aim for adoption as regular use, but rather aims to increase the likelihood of users becoming developers by putting links into strategic places [bugzilla:6190]. This innovation cannot be adopted, but rather we would like to call it successful, if it achieves additional developers to join the project.

## A.7 Hosting Types

A hosting type refers to the virtual and physical location a *service innovation* is being hosted at. Using the definition of *hosting* as the provision of computing resources, one could also say that the hosting type describes the origin of these resources.

- *affiliated project* (hosting type.affiliated project)  
Hosting at an affiliated project refers to the usage of hosting resources from another, thematically related Open Source project.

**Disambiguation:** The distinction to *federated hosting* is twofold: First, an affiliated hosting is occurring between thematically associated projects, while a federation is based on associated people. Second, the federation is usually operated together, while in affiliated hosting one project supports another by providing resources to it.

- *federated hosting* (hosting type.federated hosting)  
Federated hosting occurs if thematically unrelated projects (compare with *affiliated hosting*) share common hosting resources. Such is often based on particular individuals who share their resources with friends [geda:2899].
- *forge* (hosting type.forge)  
A Forge (or project hoster) is a website for hosting a large number of common, thematically unrelated Open Source projects. Common examples include SourceForge.Net, Savannah, and Launchpad.
- *foundation* (hosting type.foundation)  
A host shared with other projects because of belonging to the same *foundation* such as for instance the Apache foundation or the GNU project.
- *private pc* (hosting type.private pc)  
Privately owned computers which do not have a permanent Internet connection.
- *private server* (hosting type.private server)  
A server on the internet owned or paid for by a person privately.

**Disambiguation:** In contrast to a *private server*, a *private pc* is not constantly connected to the Internet and usually has more limited bandwidth.

- *service host* (hosting type.service host)  
A service host provides a very specific type of *service innovation* such as mailing lists or repositories of a particular SCM system for others to use. For instance, <http://repo.or.cz> offers *Git* hosting for Open Source projects and individuals.

**Disambiguation:** One could not run a project based on the service of a single such service hoster, otherwise it would be called a *Forge*.

- *university server* (hosting type.university server)  
Computing resources offered by a university to students or staff who then use them to operate systems for use in an Open Source project.

## A.8 Innovation Types

Innovation types provide a top-level distinction between various *innovations*.

- *documentation* (innovation type.documentation)  
The artifact form of the innovation is primarily a document.
- *legal* (innovation type.legal)  
A legal innovation uses legal mechanisms such as copyright, patent or civil law to structure an aspect of the project such as the social relationships or development processes. Most importantly in Open Source projects such innovations are represented by software licensing schemes such as the *GNU General Public License* (yet other types of legal innovations exist such as incorporation as a *foundation*).
- *process* (innovation type.process)  
Innovations of this kind consist primarily of a generalizable course of action intended to achieve a result. In contrast to *tools* or *services* such innovations focus on repeatable sequences of steps to be enacted rather than technology.

- *convention* (`innovation type.process.conventions`)

An innovation of the type *convention* is used to define rules or guidelines for certain activities in the project or steps in a *process*. Commonly such innovations are associated with an advantage which can be drawn from performing these activities consistently across the project.

A typical example is the use of a *coding convention*, which leads to the following stages following the terminology in this thesis:

1. *Execution* of a coding *convention* consists of changing existing artifacts to conform to the new standard and formalize convention in a *document*.
2. The innovation is *announced* to be in effect, which implies that *compliance* is now expected.
3. A coding convention is successfully *adopted* when contributions by individuals conform to the convention.

- *naming convention* (`innovation type.process.conventions.naming`)

A naming *convention* is a *process innovation* that defines the process steps to assign a name to project artifacts such as files, releases, or variables.

Version naming and numbering for releases has been discussed in-depth in Section 8.1.4.

- *service* (`innovation type.service`)

A service innovation relies on a centralized or dedicated host on which a software is installed that can be accessed by projects members (commonly using a client-side *tool*).

- *social* (`innovation type.social`)

A social innovation is a *process innovation* that uses a change in the social interaction of the project members to achieve a certain goal. For instance, a yearly meeting at an Open Source conference uses the social ties that can be created by having met in real life to strengthen trust and cooperation between participants in a project.

- *tool* (`innovation type.tool`)

Tools are innovations that each developer runs locally on his private computer. Thus a *tool innovation* will always require an individual adoption decision.

Some tools do have dependencies or requirements that still make them require *organizational adoption decisions* though (for instance when using a certain build-tool, then build-files need to be created for the whole project).

## A.9 Innovation Decision Types

These types categorize how decisions during an innovation introduction are made. The types include both organizational mechanisms (see Section 5.7.1) and individual ones (see Section 5.7.2).

- *authority* (`innovation decision.authority`)

An authority innovation decision is one where a small group of people or even a single person (most probably a *powerful maintainer*) makes the decision to adopt an innovation.

- *expected* (`innovation decision.expected`)

Each member is expected to *adopt/use* the innovation, yet there are no structural attributes of the innovation that would force people to adopt it, i.e. there is no build-in *compliance enforcement* or *forcing effect* in the innovation.

- *just do it* (`innovation decision.just do it`)

This innovation introduction was *executed* with no prior decision by the organization. We call such also a skipped decision.

- *optional* (innovation decision.optional)  
An innovation decision is *optional* if each project member can individually and independently make a decision to adopt, and the influence of other members is negligible. The innovation might still profit from networking effects (see Section 6.1), which increase the benefit of adopting as more and more project members sign up.

- *representational collective* (innovation decision.representational collective)  
An representational collective innovation decision is one where the decision to adopt or reject an innovation is made by the participants in the discussion by implying that the project members present in the discussion represent the whole project. A typical set-up of such a representational collective is a discussion of the maintainer with a set of developers who agree or reject an innovation. The full consensus of the project then often is not invoked.

A representational collective might call upon other project members for opinions. This is typically between high-ranking project members.

The representational collective is another strategy for achieving legitimation for one's action. This also points to the weakness of the collective: If legitimation fails to be achieved, the collective will not work [grub:4116].

- *vote* (innovation decision.vote)  
This decision against or in favor of an innovation is made by project participants by casting a vote. I have found voting to be highly informal without rules regarding who can participate, how long the polls are open, and typically even without counting the resulting votes. Typical voting methods such as Apache's "three votes in favor with no votes against" [186] point towards the interpretation of voting in Open Source systems as a means to condense discussions into a decision.

**Disambiguation:** Similar to a *representational collective* a vote most often does not involve the whole project but rather only those participants, which are participating in a discussion. In this sense, voting can be a sub-type of a representational collective with a distinct form for making the decision.

## A.10 Uncategorized Codes

This section contains the codes which were not yet included in more specific sections above.

- *argumentation* (argumentation)  
Argumentation is the base concept for the means by which innovation discussion is pursued. It includes in particular the "how" of discussion such as for instance the use of humor, questions, lists of *capabilities*, etc.

Despite the fact that over 200 argumentation codes were created, the analysis of this aspect was not achieved for this dissertation. If argumentation should be in the focus of future work then I would advise to either devote a complete thesis to the development of a theory of Open Source argumentation or—probably the better approach—accept that argumentation in Open Source projects is likely no different from discussion in other fields and reuse an existing general theory of debate or rhetoric and instantiate this theory with more specific questions in mind.

**Disambiguation:** See *capability*.

- *capability* (capability)  
Capability is a concept used to collect reasons brought up during an innovation discussion which can be tied to the innovation. Capabilities are thus properties of an innovation that make it attractive or unattractive to the discussants.

This concept has not been explored much in this thesis, because capabilities are highly contextual to a particular innovation and do not generalize to all introductions.

**Disambiguation:** During coding *capability* was separated from the concept of *argumentation* (a concept meant to include the means by which reasons were brought up in discussion such as rhetorical devices, humor or ignoring of particular aspects of posts). This distinction is not always easy, because using clever argumentation each individual capability might be distorted to the point where it is not longer clear whether the innovation really holds the capability or not.

- *dedication* (*dedication*)

The degree to which the innovator is prepared to invest *time* and *energy* to achieve the introduction of the proposed innovation.

A typical outcome due to lack of dedication is an *abandonment*.

- *force* (*force*)

A force is something that importantly influences the success or failure of an innovation introduction. By this open definition it is of course a large, situational and relative concept, but represents the starting point for a concept to be of interest for this thesis.

For instance, when the concept of *hosting* was developed, an initially identified force was the available bandwidth of a server, because in one episode it became overloaded by a large number of users adopting a novel innovation, which in turn stirred discussion about the viability of the innovation as a whole. As more and more important influence factors were discovered, this force was later subsumed in the concept of *capability*.

- *forcing effects.code is law* (*forcing effects.code is law*)

A “code is law”-*forcing effect* is a mechanism by which a software system is causing adoption of an innovation. Labeled following the book by Lawrence Lessig [307], this effect relies on rules being embedded in the source code from which the system has been created.

**Disambiguation:** Source code can be both a forcing effect (which forces adoption) and a compliance enforcement method (which forces correct use), which highlights again the difficulty to distinguish adoption and usage.

- *forcing effects.data dependence* (*forcing effects.data dependence*)

A data dependence is a *forcing effect* which derives its power from defining which data will be used to create the software product of the project from. Since project participants have to interact with this data, the innovation can force certain changes in behavior upon them.

- *forcing effects.legal* (*forcing effects.legal*)

A legal forcing effect is an associated part of an innovation which uses a legal mechanism such as copyright law or patent law to further adoption. The most prominent example is the GNU Public License, which is viral to ensure its continued usage.

- *role.translator* (*role.translator*)

A project participant who is primarily working on translating the project to one or more languages. Since translators are often only involved before releases, their influence and presence in the project is usually not very large.

**Disambiguation:** A person who is both translating and contributing code is more appropriately marked as a developer, if the contributed code exceeds minimal amounts.

- *trigger* (*trigger*)

A trigger is something that causes a proposal to be made. For instance a roadmap discussion calling for future directions for development might trigger a proposal.








## Appendix B

# Innovation Questionnaire

Freie Universität Berlin

Home | Impressum

### Changing the Development Process in F/OSS projects

Your experience as developers in Free and Open Source Software projects is needed. I am researching what happens if a developer attempts to change the way a project works (in contrast to changing the software being developed) and would like to ask for your stories of successful change that you achieved and attempts that did not work out.

More information about this survey can be found on the [pages of the F/OSS group at Freie Universität Berlin](#).

1. For instance tell me about...


- What was your goal (and what was the motivation for that?)?
- How did you approach it (what worked?, what didn't?, why?)?
- What kind of project was this with?
- Which people and tools were important for making this a success/failure?
- How did your relationship with the project change?
- What advice would you give other community members with similar goals?

2. If you want to be notified of the results, please leave your email here.

Submit Survey

© FU Berlin

Figure B.1: A short version of the Innovation Introduction Activities Survey send in August 2007 (see Chapter 3 for a discussion on the survey).



Freie Universität  
Berlin

[Home](#) | [Impressum](#)

## Changing the Development Process in F/OSS Projects

This survey tries to explore what happens if a developer in a Free Software or Open Source project attempts to change the way a project works (i.e., it affects the other developers in their work).  
 To participate pick your favorite episode of changing part of the development process and answer the questions below.  
 More information about this survey can be found on the [pages of the F/OSS group at Freie Universität Berlin](#).  
 Thank you very much.

1. What goal did you want to achieve? Why?
2. Which project did you try to change? (If you want the project to remain anonymous, please give a short description of what the software of the project does.)
3. What did you do to achieve your goal? (Please be specific enough that somebody with a similar goal could follow your example or avoid your mistakes.)
4. What did you achieve in the end?

(a) Page 1

Figure B.2: Long version of the Innovation Introduction Activities Survey send in August 2007 (see Chapter 3 for a discussion on the survey).

5. What (or who) have been barriers to your goal that you had to overcome? (These can be technical, social, legal, resource problems, etc...)

6. Who or what was involved in making your attempt a success?

7. In total, would you say that your attempt was a success?

☐ Full success  
☐ Partial success  
☐ Full failure  
☐ Other:

8. In which roles did you participate in the project at the beginning of your attempt to change the project? (Check all that apply.)

☐ User  
☐ Newbie  
☐ Bug reporter  
☐ Patch writer  
☐ Developer  
☐ Core developer  
☐ Leader of project  
☐ Founder  
☐ Retired Member  
☐ Other:   
☐ Other:

9. How did your relationship to other project members change during the period? (Check all that apply.)

☐ I became more well known  
☐ The project started to ignore me  
☐ I became more involved with the project  
☐ I became less involved with the project  
☐ I became a core developer / maintainer / leader

(b) Page 2

Figure B.2: Long version of the Innovation Introduction Activities Survey send in August 2007 (see Chapter 3 for a discussion on the survey).

☐ People became more friendly towards me  
☐ People became more unfriendly towards me  
☐ I left the project  
☐ I got banned from the project  
☐ My actions did not affect my standing in the project  
☐ I received commit rights  
☐ I got my commit rights revoked  
☐ Other:   
☐ Other:   
☐ Other:

10. How long in total did it take from deciding to change something in the project to having finished with the change? (Approximately in weeks or months. If you are still on it, put the time you have spent so far and an estimate how much longer you will need.)

11. How many people were active in the project including yourself at the beginning of your attempt to change the project?

12. If you have comments or suggestions about this survey, please let us know what you think.

13. If you like to be notified about the results, you can also provide an email address. Alternatively check back on the [survey homepage](#).

© FU Berlin

(c) Page 3

Figure B.2: Long version of the Innovation Introduction Activities Survey send in August 2007 (see Chapter 3 for a discussion on the survey).



## Appendix C

# Zusammenfassung

Diese Dissertation widmet sich der Fragestellung, wie Technologien, Werkzeuge und Prozesse der Softwaretechnik in Open-Source-Projekte eingeführt werden können. Zur Beantwortung dieser Frage werden 13 mittelgroße Open-Source-Projekte mit Hilfe der Grounded Theory Methodology untersucht und die gewonnenen Erkenntnisse in acht Kernkategorien konzeptualisiert. Betrachtet werden unter anderem die Bereitstellung von Infrastrukturressourcen, Entscheidungsmechanismen, Adaptertechnologien, soziale und technische Maßnahmen zur Einhaltung von Normen und Standards, die teilweise Einführung von Technologien in fragmentierten Innovationslandschaften und die Rolle von Zeit innerhalb von Innovationseinführungen. Diese Theoriebausteine werden einerseits durch fünf Fallstudien in den Bezug zur praktischen Durchführung von Innovationseinführung gestellt und andererseits durch einen Vergleich mit existierenden Theorien aus der Organisations- und Sozialwissenschaft wie z.B. Actor-Network Theory, Social Network Analysis, oder das Garbage Can Modell auf einer theoretischen Ebene betrachtet. Die Arbeit beschreibt Innovationseinführung als breitgefächert zu verstehendes Phänomen, welches sich nicht auf einzelne, konzeptuelle Zusammenhänge reduzieren lässt. Die Arbeit schließt mit praktischen Handlungsanleitungen mit besonderem Fokus auf häufig auftretende Innovationsklassen, wie Versionskontrollsysteme, Softwarelizenzen und Defektverwaltungssysteme.

# Christopher Özbek

✉ [Christopher@Ozbek.org](mailto:Christopher@Ozbek.org)

Due to privacy concerns the CV contained in the printed version  
of this dissertation is not available online.



---

## Publications

Publications marked with an asterisk (\*) originated from this dissertation.

- 2010 Edna Rosen, Stephan Salinger, **Christopher Oezbek**. Project Kick-off with Distributed Pair Programming. Psychology of Programming Interest Group 2010, 19-21 September, Madrid, Spain. Accepted for publication.
- \*Lutz Prechelt, **Christopher Oezbek**. The search for a research method for studying OSS process innovation. Special Issue on Qualitative Research in Software Engineering of the Empirical Software Engineering Journal. Submitted, February 2010.
- \***Christopher Oezbek**. Introducing automated regression testing in Open Source projects. P. Ågerfalk et al. (Eds.): Proceedings of the OSS 2010, Notre Dame, IL, IFIP AICT 319, pp. 361–366. IFIP International Federation for Information Processing, June 2010.
- \***Christopher Oezbek**, Lutz Prechelt, and Florian Thiel. The onion has cancer: Some social network analysis visualizations of Open Source project communication. In *Proceedings of the 2010 ICSE Workshop on Free, Libre and Open Source Software*, May 2010.
- Stephan Salinger, **Christopher Oezbek**, Karl Beecher, and Julia Schenk. Saros: An Eclipse plug-in for distributed party programming. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. ACM, 2010.
- \***Christopher Oezbek** and Florian Thiel. Radicality and the Open Source Development Model. In *Proceedings of the 2010 FLOSS Workshop, Jena, Germany*. 2010.
- 2008 \***Christopher Oezbek**. Research ethics for studying Open Source projects. In *4th Research Room FOSDEM: Libre software communities meet research community*, February 2008.
- \***Christopher Oezbek**, Robert Schuster, and Lutz Prechelt. Information management as an explicit role in OSS projects: A case study. Technical Report TR-B-08-05, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, April 2008.
- 2007 Riad Djemili, **Christopher Oezbek**, and Stephan Salinger. Saros: Eine Eclipse-Erweiterung zur verteilten Paarprogrammierung. In *Software Engineering 2007 - Beiträge zu den Workshops*, Hamburg, Germany, March 2007. Gesellschaft für Informatik.
- Sebastian Jekutsch, **Christopher Oezbek**, and Stephan Salinger. Selbstbestimmung oder Anleitung: Erfahrungen mit einem Softwaretechnikpraktikum im Bereich Qualitätssicherung. In *SEUH 2007 - Software Engineering im Unterricht der Hochschulen*, Hochschule für Technik, Stuttgart, Germany, 22.-23. Februar 2007.
- Christopher Oezbek** and Lutz Prechelt. JTourBus: Simplifying program understanding by documentation that provides tours through the source code. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*. IEEE Computer Society, 2007.

- \***Christopher Oezbek** and Lutz Prechelt. On understanding how to introduce an innovation to an Open Source project. In *Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW '07)*, Washington, DC, USA, 2007. IEEE Computer Society. reprinted in UPGRADE, The European Journal for the Informatics Professional 8(6):40-44, December 2007.
- 2005 Steven Dow, Jaemin Lee, **Christopher Oezbek**, Blair MacIntyre, Jay David Bolter, and Maribeth Gandy. Wizard of oz interfaces for mixed reality applications. In *CHI '05 extended abstracts on Human factors in computing systems*, pages 1339–1342, New York, NY, USA, 2005. ACM.
- Steven Dow, Jaemin Lee, **Christopher Oezbek**, Blair MacIntyre, Jay David Bolter, and Maribeth Gandy. Exploring spatial narratives and mixed reality experiences in oakland cemetery. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 51–60, New York, NY, USA, 2005. ACM.
- Steven Dow, Blair MacIntyre, Jaemin Lee, **Christopher Oezbek**, Jay David Bolter, and Maribeth Gandy. Wizard of oz support throughout an iterative design process. *IEEE Pervasive Computing*, 4(4):18–26, 2005.
- 2004 **Christopher Oezbek**, Björn Giesler, and Rüdiger Dillmann. Jedi training: playful evaluation of head-mounted augmented reality display systems. In *Proceedings of the Conference on Stereoscopic Displays and Virtual Reality Systems XI*, volume 5291 of *Proc. SPIE*, pages 454–463, San Diego, USA, May 2004.
- 2003 **Christopher Oezbek**. Spielerische Evaluierung eines Augmented Reality Systems. Studienarbeit, Universität Karlsruhe (TH), 2003.

# Bibliography

- [1] Eric Abrahamson and Lori Rosenkopf. Social network effects on the extent of innovation diffusion: A computer simulation. *Organization Science*, 8(3):289–309, May 1997.
- [2] Howard Aldrich. *Organizations evolving*. Sage Publications, London, 1999.
- [3] Vamshi Ambati and S. P. Kishore. How can academic software research and Open Source Software development help each other? *IEE Seminar Digests*, 2004(908):5–8, 2004.
- [4] Juan-José Amor, Jesús M. González-Barahona, Gregorio Robles-Martinez, and Israel Herráiz-Tabernero. Measuring libre software using Debian 3.1 (Sarge) as a case study: Preliminary results. *UPGRADE*, 6(3):13–16, June 2005.
- [5] Chris Anderson. The long tail. *Wired Magazine*, 12(10), October 2004.
- [6] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Vancouver, Canada*, pages 298–308, Washington, DC, USA, 2009. IEEE Computer Society.
- [7] Aleks Aris, Ben Shneiderman, Catherine Plaisant, Galit Shmueli, and Wolfgang Jank. Representing unevenly-spaced time series data for visualization and interactive exploration. In *Human-Computer Interaction - INTERACT 2005*, Lecture Notes in Computer Science, pages 835–846. Springer, 2005.
- [8] W. Brian Arthur. Competing technologies, increasing returns, and lock-in by historical events. *The Economic Journal*, 99(394):116–131, March 1989.
- [9] W. Brian Arthur. *Increasing Returns and Path Dependence in the Economy*. University of Michigan Press, 1994.
- [10] Chad Ata, Veronica Gasca, John Georgas, Kelvin Lam, and Michele Rousseau. Open Source software development processes in the Apache Software Foundation. Final Report for ICS 225 - Software Process - Spring '02. <http://www.ics.uci.edu/~michele/SP/final.doc>, June 2002. Accessed 2009-11-28.
- [11] David E. Avison, Francis Lau, Michael D. Myers, and Peter Axel Nielsen. Action research. *Commun. ACM*, 42(1):94–97, 1999.
- [12] Robert M. Axelrod and Michael D. Cohen. *Harnessing complexity: organizational implications of a scientific frontier*. Free Press, New York, 1999.
- [13] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using FindBugs on production software. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 805–806. ACM, 2007.
- [14] Paula M. Bach, Robert DeLine, and John M. Carroll. Designers wanted: participation and the

- user experience in Open Source software development. In *CHI '09: Proceedings of the 27th International Conference on Human Factors in Computing Systems*, pages 985–994. ACM, 2009.
- [15] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.
  - [16] Ragnar Bade, Stefan Schlechtweg, and Silvia Miksch. Connecting time-oriented data and information to a coherent interactive visualization. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 105–112. ACM, 2004.
  - [17] Joshua D. Baer and Grant Neufeld. The use of URLs as meta-syntax for core mail list commands and their transport through message header fields. Request for Comments 2369, Internet Engineering Task Force, July 1998.
  - [18] Carliss Y. Baldwin and Kim B. Clark. The architecture of participation: Does code architecture mitigate free riding in the Open Source development model? *Management Science*, 52(7):1116–1127, July 2006.
  - [19] Kerstin Balka, Christina Raasch, and Cornelius Herstatt. Open source enters the world of atoms: A statistical analysis of open design. *First Monday*, 14(11), November 2009.
  - [20] Stephen P. Banks, Esther Louie, and Martha Einerson. Constructing personal identities in holiday letters. *Journal of Social and Personal Relationships*, 17(3):299–327, 2000.
  - [21] Flore Barcellini, Françoise Détienne, and Jean Marie Burkhardt. Cross-participants: fostering design-use mediation in an Open Source software community. In *ECCE '07: Proceedings of the 14th European conference on Cognitive ergonomics*, pages 57–64. ACM, 2007.
  - [22] Flore Barcellini, Françoise Détienne, and Jean-Marie Burkhardt. User and developer mediation in an Open Source Software community: Boundary spanning through cross participation in online discussions. *International Journal of Human-Computer Studies*, 66(7):558–570, 2008.
  - [23] Flore Barcellini, Françoise Détienne, Jean-Marie Burkhardt, and Warren Sack. Thematic coherence and quotation practices in OSS design-oriented online discussions. In *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 177–186. ACM, 2005.
  - [24] Flore Barcellini, Françoise Détienne, Jean-Marie Burkhardt, and Warren Sack. A socio-cognitive analysis of online design discussions in an Open Source Software community. *Interacting with Computers*, 20(1):141–165, 2008.
  - [25] Flore Barcellini, Françoise Détienne, Jean-Marie Burkhardt, and Warren Sack. A study of online discussions in an Open-Source software community: Reconstructing thematic coherence and argumentation from quotation practices. In Peter van den Besselaar, Giorgio de Michelis, Jenny Preece, and Carla Simone, editors, *Second Communities and Technologies Conference, Milano 2005*, pages 121–140. Springer, May 2005.
  - [26] Flore Barcellini, Françoise Détienne, and Jean-Marie Burkhardt. Users' participation to the design process in an Open Source software online community. In P. Romero, J. Good, S. Bryant, and E. A. Chaparro, editors, *18th Annual Workshop on Psychology of Programming Interest Group PPIG'05*, pages 99–114, 2005.
  - [27] J. A. Barnes. Class and committees in a Norwegian island parish. *Human Relations*, 7(1):39–58, 1954.
  - [28] Christine Barry. Choosing qualitative data analysis software: Atlas/ti and nudist compared. *Sociological Research Online*, 3(3), September 1998.

- [29] Victor Basili, Gianluigi Caldiera, Frank McGarry, Rose Pajerski, Gerald Page, and Sharon Waligora. The software engineering laboratory: an operational software experience factory. In *ICSE '92: Proceedings of the 14th International Conference on Software Engineering*, pages 370–381. ACM, 1992.
- [30] Victor R. Basili. Software modeling and measurement: the goal/question/metric paradigm. UMIACS TR-92-96, University of Maryland at College Park, College Park, MD, USA, 1992.
- [31] Richard L. Baskerville. Investigating information systems with action research. *Communications of the Association for Information Systems*, 2(3es):4, 1999.
- [32] Michael Baur. *Visone – Software for the Analysis and Visualization of Social Networks*. Disseration, Fakultät für Informatik, Universität Karlsruhe (TH), Karlsruhe, November 2008.
- [33] Lina Böcker. Die GPLv3 - ein Schutzschild gegen das Damoklesschwert der Softwarepatente? In Bernd Lutterbeck, Matthias Bärwolff, and Robert A. Gehring, editors, *Open Source Jahrbuch 2007 – Zwischen freier Software und Gesellschaftsmodell*. Lehmanns Media, Berlin, 2007.
- [34] James Bearden, William Atwood, Peter Freitag, Carol Hendricks, Beth Mintz, and Michael Schwartz. The nature and extent of bank centrality of corporate networks. Unpublished paper submitted to the American Sociological Association. Reprinted in Scott J. Eds. *Social networks: critical concepts in sociology*, Volume 3. Taylor & Francis, 2002., 1975.
- [35] Kent Beck and Erich Gamma. Test-infected: Programmers love writing tests. In *More Java gems*, pages 357–376. Cambridge University Press, New York, NY, USA, 2000.
- [36] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Softw. Eng.*, 30(8):535–546, 2004.
- [37] Benjamin B. Bederson, Jon Meyer, and Lance Good. Jazz: An extensible zoomable user interface graphics toolkit in Java. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 171–180. ACM, 2000.
- [38] Brian Behlendorf. Open Source as a business strategy. In DiBona et al. [144], page 280.
- [39] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML ain't markup language (YAML®) version 1.2. Technical report, The YAML Project, October 2009.
- [40] Jonathan Bendor, Terry M. Moe, and Kenneth W. Shotts. Recycling the garbage can: An assessment of the research program. *The American Political Science Review*, 95(1):169–190, March 2001.
- [41] Yochai Benkler. Coase's penguin, or, Linux and The Nature of the Firm. *Yale Law Review*, 112(3):369–446, December 2002.
- [42] Calum Benson, Matthias Müller-Prove, and Jiri Mzourek. Professional usability in open source projects: GNOME, OpenOffice.org, NetBeans. In *CHI '04 extended abstracts on Human factors in computing systems*, pages 1083–1084. ACM, 2004.
- [43] Evangelia Berdou. *Managing the bazaar: Commercialization and peripheral participation in mature, community-led F/OS software projects*. Doctoral dissertation, London School of Economics and Political Science, Department of Media and Communications, 2007.
- [44] Magnus Bergquist and Jan Ljungberg. The power of gifts: Organizing social relationships in Open Source communities. *Information Systems Journal*, 11(4):305–320, 2001.
- [45] Jean-Marc Bernard. Analysis of local and asymmetric dependencies in contingency tables using the imprecise Dirichlet model. In Jean-Marc Bernard, Teddy Seidenfeld, and Marco Zaffalon, editors, *ISIPTA '03, Proceedings of the Third International Symposium on Imprecise Probabilities and Their Applications, Lugano, Switzerland, July 14–17, 2003*, volume 18 of *Proceedings in Informatics*, pages 46–61. Carleton Scientific, 2003.

- [46] Jean-Marc Bernard. An introduction to the imprecise Dirichlet model for multinomial data. *International Journal of Approximate Reasoning*, 39(2–3):123–150, June 2005. Imprecise Probabilities and Their Applications.
- [47] David M. Berry. The contestation of code. *Critical Discourse Studies*, 1:65–89, April 2004.
- [48] David M. Berry. Internet research: privacy, ethics and alienation: an Open Source approach. *Internet Research: Electronic Networking Applications and Policy*, 14(4):323–332, September 2004.
- [49] Nikolai Bezroukov. Open Source development as a special type of academic research (critique of vulgar Raymondism). *First Monday*, 4(10), October 1999.
- [50] Nikolai Bezroukov. A second look at the cathedral and bazaar. *First Monday*, 4(12), December 1999.
- [51] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143. ACM, 2006.
- [52] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in Open Source projects. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 24–35. ACM, 2008.
- [53] Jürgen Bitzer, Wolfram Schrettl, and Philipp J. H. Schröder. Intrinsic motivation in Open Source software development. *Journal of Comparative Economics*, 35(1):160–169, March 2007.
- [54] Christopher Boehm. Egalitarian behavior and reverse dominance hierarchy. *Current Anthropology*, 34(3):227–254, 1993.
- [55] Andrea Bonaccorsi, Silvia Giannangeli, and Cristina Rossi. Entry strategies under competing standards: Hybrid business models in the Open Source software industry. *Management Science*, 52(7):1085–1098, July 2006.
- [56] Andrea Bonaccorsi and Cristina Rossi. Why Open Source software can succeed. *Research Policy*, 32(7):1243–1258, 2003.
- [57] Andrea Bonaccorsi and Cristina Rossi. Comparing motivations of individual programmers and firms to take part in the Open Source movement: From community to business. *Knowledge, Technology & Policy*, 18(4):40–64, December 2006.
- [58] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.
- [59] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2006.
- [60] Peter A. Boncz and Martin L. Kersten. Monet: An impressionist sketch of an advanced database system. In *Proceedings Basque International Workshop on Information Technology, San Sebastian, Spain*, July 1995.
- [61] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 555–563. ACM, 1999.
- [62] Scott Bradner. The Internet Engineering Task Force(ietf). In DiBona et al. [144], pages 47–52.

- [63] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Marshall. GraphML progress report structural layer proposal. In *Graph Drawing*, volume 2265/2002 of *Lecture Notes in Computer Science*, pages 109–112. Springer, January 2002.
- [64] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, April 1998. Proceedings of the Seventh International World Wide Web Conference.
- [65] Frederick P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley, Reading, MA, 1975.
- [66] Alan W. Brown and Grady Booch. Reusing Open-Source Software and practices: The impact of Open-Source on commercial vendors. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 123–136. Springer-Verlag, 2002.
- [67] Nick Brown. ROX founder: Why I brought RISC OS to Unix. <http://www.drobe.co.uk/riscos/artifact2002.html>, July 2007. Accessed 2009-10-26.
- [68] Christopher B. Browne. Linux and decentralized development. *First Monday*, 3(2), March 1998.
- [69] Larry D. Browning, Janice M. Beyer, and Judy C. Shetler. Building cooperation in a competitive industry: SEMATECH and the semiconductor industry. *The Academy of Management Journal*, 38(1):113–151, February 1995.
- [70] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, pages 213–222. ACM, 2009.
- [71] Tony Buzan and Barry Buzan. *The Mind Map book*. BBC Books, London, 1993.
- [72] Bruce Byfield. Google's summer of code concludes. Linux.com article. <http://www.linux.com/archive/articles/48232>, September 2005. Accessed 2010-02-03.
- [73] Michel Callon. Some elements of a sociology of translation: Domestication of the scallops and the fishermen of Saint Brieu Bay. In John Law, editor, *Power, action and belief: A new sociology of knowledge?*, Sociological Review Monograph, pages 196–233. Routledge, Boston, 1986.
- [74] Donald T. Campbell. Blind variation and selective retention in creative thought as in other knowledge processes. *Psychol Rev*, 67:380–400, Nov 1960.
- [75] Andrea Capiluppi and Karl Beecher. Structural complexity and decay in FLOSS systems: An inter-repository study. In *European Conference on Software Maintenance and Reengineering CSMR'09*, pages 169–178. IEEE Computer Society, 2009.
- [76] Andrea Capiluppi and Martin Michlmayr. From the cathedral to the bazaar: An empirical study of the lifecycle of volunteer community projects. In *Open Source Development, Adoption and Innovation*, volume 234/2007 of *IFIP International Federation for Information Processing*, pages 31–44. Springer, Boston, MA, 2007.
- [77] Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International Journal of Human-Computer Interaction*, 17(3):333–356, September 2004.
- [78] Kathy Charmaz. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. Sage Publications Ltd, 1st edition, January 2006.
- [79] David J. Cheal. *The gift economy*. Routledge New York, London, 1988.
- [80] Giorgos Cheliotis. From Open Source to open content: Organization, licensing and decision processes in open cultural production. *Decision Support Systems*, 47(3):229–244, June 2009.

- [81] Michelene T. H. Chi. Quantifying qualitative analyses of verbal data: A practical guide. *The Journal of the Learning Sciences*, 6(3):271–315, 1997.
- [82] Clayton M. Christensen. *The innovator's dilemma: when new technologies cause great firms to fail*. Harvard Business School Press, Boston, MA, 1997.
- [83] Steve Christey and Robert A. Martin. Vulnerability Type Distributions in CVE (1.1). <http://cwe.mitre.org/documents/vuln-trends/index.html>, May 2007. Accessed 2009-10-12.
- [84] Scott Christley and Greg Madey. An algorithm for temporal analysis of social positions. In *Proceedings of the North American Association for Computational Social and Organizational Science Conference (NAACSOS) 2005*, June 2005.
- [85] Chromatic. Myths open source developers tell ourselves. ONLamp.com. <http://www.onlamp.com/pub/a/onlamp/2003/12/11/myths.html>, November 2003. Accessed 2009-11-30.
- [86] M. Chui and Dillon A. Speed and accuracy using four boolean query systems. In *Tenth Midwest Artificial Intelligence and Cognitive Science Conference, Bloomington, Indiana, USA*, pages 36–42. The AAAI Press, April 1999.
- [87] Wingyan Chung, Hsinchun Chen, Luis G. Chaboya, Christopher D. O'Toole, and Homa Atabakhsh. Evaluating event visualization: a usability study of COPLINK spatio-temporal visualizer. *International Journal of Human-Computer Studies*, 62(1):127–157, 2005.
- [88] Andrea Cifollilli. Phantom authority, self-selective recruitment and retention of members in virtual communities: The case of Wikipedia. *First Monday*, 8(12), December 2003.
- [89] Marcus Ciolkowski and Martín Soto. Towards a comprehensive approach for assessing Open Source projects. In *Software Process and Product Measurement*, volume 5338/2008 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Berlin / Heidelberg, 2008.
- [90] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70(6):066111, December 2004.
- [91] Mark J. Clayton. Delphi: a technique to harness expert opinion for critical decision-making tasks in education. *Educational Psychology: An International Journal of Experimental Educational Psychology*, 17(4):373–386, December 1997.
- [92] Jill Coffin. An analysis of Open Source principles in diverse collaborative communities. *First Monday*, 11(6), June 2006.
- [93] Michael D. Cohen and James G. March. *Leadership and ambiguity*. Harvard Business Press, 1974.
- [94] Michael D. Cohen, James G. March, and Johan P. Olsen. A garbage can model of organizational choice. *Administrative Science Quarterly*, 17(1):1–25, 1972.
- [95] Susan G. Cohen and Diane E. Bailey. What makes teams work: Group effectiveness research from the shop floor to the executive suite. *Journal of Management*, 23(3):239–290, June 1997.
- [96] Jorge Colazo and Yulin Fang. Impact of license choice on Open Source software development activity. *Journal of the American Society for Information Science and Technology*, 60(5):997–1011, 2009.
- [97] E. Gabriella Coleman. *The Social Construction of Freedom in Free and Open Source Software: Hackers Ethics, and the Liberal Tradition*. PhD thesis, University of Chicago, August 2005.
- [98] E. Gabriella Coleman. Three ethical moments in Debian. [97], chapter 6.
- [99] James J. Collins and Carson C. Chow. It's a small world. *Nature*, 393:409–410, June 1998.
- [100] Stefano Comino, Fabio M. Manenti, and Maria Laura Parisi. From planning to mature: On the success of Open Source projects. *Research Policy*, 36(10):1575–1586, December 2007.



- [101] Committee on Developments in the Science of Learning with additional material from the Committee on Learning Research and Educational Practice and National Research Council. Learning and transfer. In John D. Bransford, Ann L. Brown, and Rodney R. Cocking, editors, *How People Learn: Brain, Mind, Experience, and School*, chapter 3, pages 51–78. National Academies Press, Washington, D.C, expanded edition edition, September 2003.
- [102] Melvin E. Conway. How do committees invent? *Datamation*, 14(4):28–31, April 1968.
- [103] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 97–106. ACM, 2005.
- [104] Juliet M. Corbin and Anselm Strauss. Grounded theory research: Procedures, canons and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, March 1990.
- [105] Juliet M. Corbin and Anselm L. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 3rd edition, 2008.
- [106] Joseph A. Cottam and Andrew Lumsdaine. Extended assortitivity and the structure in the Open Source development community. In *Proceedings of the International Sunbelt Social Network Conference*. International Network for Social Network Analysis, January 2008.
- [107] Robin Cowan. Nuclear power reactors: A study in technological lock-in. *The Journal of Economic History*, 50(3):541–567, September 1990.
- [108] Robin Cowan. Tortoises and hares: Choice among technologies of unknown merit. *The Economic Journal*, 101(407):801–814, July 1991.
- [109] Andrew Cox. What are communities of practice? a comparative review of four seminal works. *Journal of Information Science*, 31(6):527–540, 2005.
- [110] Rob Cross, Stephen P. Borgatti, and Andrew Parker. Making invisible work visible: Using social network analysis to support strategic collaboration. *California Management Review*, 44(2):25–46, 2002.
- [111] Colin Crouch and Henry Farrell. Breaking the path of institutional development? Alternatives to the new determinism. *Rationality and Society*, 16(1):5–43, 2004.
- [112] Kevin Crowston, Hala Annabi, James Howison, and Chengetai Masango. Towards a portfolio of FLOSS project success measures. In *Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering, International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, May 25, 2004.
- [113] Kevin Crowston and James Howison. The social structure of Free and Open Source software development. *First Monday*, 10(2), 2005.
- [114] Kevin Crowston, James Howison, and Hala Annabi. Information systems success in Free and Open source software development: theory and measures. *Software Process: Improvement and Practice*, 11(2):123–148, 2006.
- [115] Kevin Crowston, James Howison, Chengetai Masango, and U. Yeliz Eseryel. Face-to-face interactions in self-organizing distributed teams. In *Proceedings of the OCIS division, Academy of Management Conference, Honolulu, Hawaii, USA*, August 2005.
- [116] Kevin Crowston, Qing Li, Kangning Wei, U. Yeliz Eseryel, and James Howison. Self-organization of teams for free/libre Open Source software development. *Information and Software Technology*, 49(6):564–575, June 2007.
- [117] Kevin Crowston and Barbara Scozzi. Open Source software projects as virtual organisations: competency rallying for software development. *IEE Proceedings — Software*, 149(1):3–17, February 2002.

- [118] Kevin Crowston and Barbara Scozzi. Bug fixing practices within Free/Libre Open Source software development teams. *Journal of Database Management*, 19(2):1–30, 2008.
- [119] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/libre open source software development: What we know and what we do not know. *ACM Computing Surveys*, May 2010. In press.
- [120] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. Core and periphery in Free/Libre and Open Source software team communications. In *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, page 118ff. IEEE Computer Society, 2006.
- [121] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, New York, 1st edition, March 1991.
- [122] Linus Dahlander and Mats G. Magnusson. Relationships between Open Source software companies and communities: Observations from Nordic firms. *Research Policy*, 34(4):481–493, May 2005.
- [123] Eric Dalci, Veracode, Inc., and CWE Content Team. CWE-74: failure to sanitize data into a different plane (aka ‘injection’) (1.3). <http://cwe.mitre.org/data/definitions/74.html>, March 2009. Accessed 2009-10-12.
- [124] Norman Dalkey and Olaf Helmer. An experimental application of the Delphi method to the use of experts. *Management Science*, 9(3):458–467, April 1963.
- [125] Jean-Michel Dalle and Nicolas Jullien. ‘Libre’ software: turning fads into institutions? *Research Policy*, 32(1):1–11, January 2003.
- [126] Patrick D’Astous, Françoise Détienne, Willemien Visser, and Pierre N. Robillard. Changing our view on design evaluation meetings methodology: a study of software technical review meetings. *Design Studies*, 25(6):625–655, November 2004.
- [127] Patrick D’Astous, Pierre N. Robillard, Françoise Détienne, and Willemien Visser. Quantitative measurements of the influence of participant roles during peer review meetings. *Empirical Software Engineering*, 6(2):143–159, June 2001.
- [128] Datamonitor. Software: Global Industry Guide. Global Industry Guide series DO-4959, Datamonitor, April 2009.
- [129] Paul A. David. Clio and the economics of QWERTY. *The American Economic Review*, 75(2):332–337, May 1985.
- [130] Paul A. David. Path Dependence and the quest for historical economics: One more chorus of ballad of QWERTY. Oxford University Economic and Social History Series 020, Economics Group, Nuffield College, University of Oxford, November 1997.
- [131] Paul A. David and Shane Greenstein. The economics of compatibility standards: An introduction to recent research. *Economics of Innovation and New Technology*, 1(1):3–41, 1990.
- [132] Paul A. David, Andrew Waterman, and Seema Arora. FLOSS-US: The Free/Libre/Open Source software survey for 2003. SIEPR-Nostra Project Working Paper. <http://www.stanford.edu/group/floss-us/report/FLOSS-US-Report.pdf>, September 2003. Accessed 2009-08-11.
- [133] Scott Davidson. Open-source hardware. *IEEE Design & Test*, 21(5):456, September 2004.
- [134] Robert Davison, Maris G. Martinsons, and Ned Kock. Principles of canonical action research. *Information Systems Journal*, 14(1):65–86, January 2004.
- [135] Brian de Alwis and Jonathan Sillito. Why are software projects moving from centralized to decentralized version control systems? In *CHASE '09: Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, Vancouver, Canada, pages 36–39. IEEE Computer Society, 2009.

- [136] Miguel De Icaza, Elliot Lee, Federico Mena, and Tromeo Tromeo. The GNOME desktop environment. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 38–38. USENIX Association Berkeley, CA, USA, 1998.
- [137] Stefano De Paoli, Maurizio Teli, and Vincenzo D'Andrea. Free and open source licenses in community life: Two empirical cases. *First Monday*, 13(10), September 2008.
- [138] Cleidson de Souza, Jon Froehlich, and Paul Dourish. Seeking the source: Software source code as a social and technical artifact. In *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 197–206. ACM, 2005.
- [139] Jason Dedrick and Joel West. Movement ideology vs. user user pragmatism in the organizational adoption of Open Source software. In Kenneth L. Kraemer and Margaret S. Elliott, editors, *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, ASIST monograph, chapter 16, pages 427–454. Information Today, Medford, NJ, 2008.
- [140] Peter J. Denning and Robert Dunham. Innovation as language action. *Commun. ACM*, 49(5):47–52, 2006.
- [141] Jean-Christophe Deprez and Simon Alexandre. Comparing assessment methodologies for Free/Open Source Software: OpenBRR and QSOS. In *Product-Focused Software Process Improvement*, volume 5089/2009 of *Lecture Notes in Computer Science*, pages 189–203. Springer, Berlin / Heidelberg, 2009.
- [142] Mark Dery, editor. *Flame Wars: The Discourse of Cyberculture*. Duke University Press, Durham, NC, 2nd edition, 1994.
- [143] Amit Deshpande and Dirk Riehle. The total growth of Open Source. In *Open Source Development, Communities and Quality*, IFIP International Federation for Information Processing, pages 197–209. Springer, Boston, 2008.
- [144] Chris DiBona, Sam Ockman, and Mark Stone, editors. *Open Sources: Voices from the Open Source Revolution*. O'Reilly, Sebastopol, CA, USA, 1st edition, January 1999.
- [145] Oliver Diedrich. Trendstudie Open Source. Heise Open. <http://www.heise.de/open/artikel/Trendstudie-Open-Source-221696.html>, February 2009. Accessed 2010-04-22.
- [146] Stefan Dietze. Metamodellbasierte Fallstudien der Entwicklungsprozesse repräsentativer Open Source Software Projekte. ISST-Berichte 68/03, Fraunhofer-Institut für Software- und Systemtechnik ISST, Berlin, October 2003.
- [147] Stefan Dietze. *Modell und Optimierungsansatz für Open Source Softwareentwicklungsprozesse*. Doktorarbeit, Universität Potsdam, 2004.
- [148] Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, August 1970.
- [149] Trung T. Dinh-Trong and James M. Bieman. The FreeBSD project: A replication case study of Open Source development. *IEEE Trans. Softw. Eng.*, 31(6):481–494, 2005.
- [150] Leonhard Dobusch. Migration discourse structures: Escaping Microsoft's desktop path. In *Open Source Development, Communities and Quality*, volume 275/2008 of *IFIP International Federation for Information Processing*, pages 223–235. Springer, Boston, 2008.
- [151] Judith S. Donath. Identity and deception in the virtual community. In P. Kollock and M. Smith, editors, *Communities in cyberspace*, pages 29–59. University of California Press, Berkeley, 1999.
- [152] Nicolas Ducheneaut. *The reproduction of Open Source software programming communities*. PhD thesis, University of California at Berkeley, Berkeley, CA, July 2003.
- [153] Nicolas Ducheneaut. Socialization in an Open Source Software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, V14(4):323–368, August 2005.

- [154] Frans-Willem Duijnhouwer and Chris Widdows. Open Source Maturity Model. Capgemini Expert Letter. [http://kb.cospa-project.org/retrieve/1097/GB\\_Expert\\_Letter\\_Open\\_Source\\_Maturity\\_Model\\_1.5.31.pdf](http://kb.cospa-project.org/retrieve/1097/GB_Expert_Letter_Open_Source_Maturity_Model_1.5.31.pdf), August 2003. Accessed 2009-11-26.
- [155] William N. Dunn and Fredric W. Swierczek. Planned Organizational Change: Toward Grounded Theory. *Journal of Applied Behavioral Science*, 13(2):135–157, 1977.
- [156] Tore Dybå and Torgeir Dingsøy. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, August 2008.
- [157] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, chapter 11, pages 285–311. Springer, London, 2008.
- [158] Kerstin Severinson Eklundh and Henry Rodriguez. Coherence and interactivity in text-based group discussions around web documents. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 4*, page 40108.3. IEEE Computer Society, 2004.
- [159] Margaret S. Elliott. Examining the success of computerization movements in the ubiquitous computing era: Free and Open Source software movements. In Kenneth L. Kraemer and Margaret S. Elliott, editors, *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, ASIST monograph, chapter 13, pages 359–380. Information Today, Medford, NJ, 2008.
- [160] Margaret S. Elliott. The virtual organizational culture of a Free Software development community. In Feller et al. [178], pages 45–50.
- [161] Margaret S. Elliott and Walt Scacchi. Free Software developers as an occupational community: Resolving conflicts and fostering collaboration. In *GROUP '03: Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pages 21–30. ACM, 2003.
- [162] Margaret S. Elliott and Walt Scacchi. Communicating and mitigating conflict in Open Source software development projects. *Projects & Profits*, IV(10):25–41, October 2004.
- [163] Margaret S. Elliott and Walt Scacchi. Mobilization of software developers: The Free Software movement. *Information Technology & People*, 21(1):4–33, 2008.
- [164] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007.
- [165] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz: Open Source graph drawing tools. In *Graph Drawing*, volume 2265/2002 of *Lecture Notes in Computer Science*, pages 594–597. Springer, 2002.
- [166] Robert English and Charles M. Schweik. Identifying success and abandonment of Free/Libre and Open Source (FLOSS) commons: A classification of Sourceforge.net projects. *UPGRADE*, VIII(6):54–59, December 2007.
- [167] Robert M. Entman. Framing: Toward clarification of a fractured paradigm. *The Journal of Communication*, 43(4):51–58, 1993.
- [168] Justin R. Erenkrantz. Release management within Open Source projects. In Feller et al. [178], pages 51–55.
- [169] Kouichirou Eto, Satoru Takabayashi, and Toshiyuki Masui. qwikWeb: Integrating mailing list and WikiWikiWeb for group communication. In *WikiSym '05: Proceedings of the 2005 International Symposium on Wikis*, pages 17–23. ACM Press, 2005.

- [170] Steffen Evers. An introduction to Open Source software development. Diplomarbeit, Technische Universität Berlin, Berlin, August 2000.
- [171] Steffen Evers. *Ein Modell der Open-Source-Entwicklung*. Dissertation, Technische Universität Berlin, Berlin, Germany, November 2008.
- [172] Julien Fang and Derrick Neufeld. Understanding sustained participation in Open Source software projects. *Journal of Management Information Systems*, 25(5):9–50, 2009.
- [173] Paul Farrand, Fearzana Hussain, and Enid Hennessy. The efficacy of the ‘mind map’ study technique. *Medical Education*, 36(5):426–431, May 2002.
- [174] Michael Fauscette. Worldwide Open Source software 2009–2013 forecast. Market Analysis 219260, IDC, July 2009.
- [175] Peter H. Feiler and Watts S. Humphrey. Software process development and enactment: Concepts and definitions. Technical Report CMU/SEI-92-TR-04, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., September 1992.
- [176] Joseph Feller and Brian Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley, London, UK, 2002.
- [177] Joseph Feller, Brian Fitzgerald, Frank Hecker, Scott Hissam, Karim Lakhani, and André van der Hoek, editors. *Meeting Challenges and Surviving Success: The 2nd Workshop on Open Source Software Engineering*. ACM, 2002.
- [178] Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors. *Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, 2003. IEEE Computer Society.
- [179] Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors. *Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering, International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, May 25*, Washington, DC, USA, 2004. IEEE Computer Society.
- [180] Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani, editors. *Perspectives on Free and Open Source Software*. The MIT Press Ltd., Cambridge, MA, July 2005.
- [181] Joseph Feller, Brian Fitzgerald, and André van der Hoek, editors. *Making Sense of the Bazaar: 1st Workshop on Open Source Software Engineering*. ACM, 2001.
- [182] Robert G. Fichman. Information technology diffusion: A review of empirical research. In *ICIS '92: Proceedings of the Thirteenth International Conference on Information Systems, Dallas, Texas, USA*, pages 195–206, Minneapolis, MN, USA, 1992. University of Minnesota.
- [183] Robert G. Fichman. *The assimilation and diffusion of software process innovations*. PhD thesis, Massachusetts Institute of Technology, Sloan School of Management, 1995.
- [184] Robert G. Fichman and Chris F. Kemerer. Toward a theory of the adoption and diffusion of software process innovations. In *Proceedings of the IFIP TC8 Working Conference on Diffusion, Transfer and Implementation of Information Technology*, pages 23–30. Elsevier, 1994.
- [185] Robert G. Fichman and Chris F. Kemerer. The Illusory Diffusion of Innovation: An Examination of Assimilation Gaps. *Information Systems Research*, 10(3):255–275, September 1999.
- [186] Roy T. Fielding. Shared leadership in the Apache project. *Commun. ACM*, 42(4):42–43, 1999.
- [187] Brian Fitzgerald. The transformation of Open Source software. *MIS Quarterly*, 30(3):587–598, September 2006.
- [188] Timo Fleischfresser. Evaluation von Open Source Projekten: Ein GQM-basierter Ansatz. Diplomarbeit, Institut für Informatik, Freie Universität Berlin, Berlin, April 2007.

- [189] Lee Fleming and David M. Waguespack. Brokerage, boundary spanning, and leadership in open innovation communities. *Organization Science*, 18(2):165–180, March 2007.
- [190] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly, Sebastopol, CA, USA, 1st edition, October 2005.
- [191] Richard Fontana, Bradley M. Kuhn, Eben Moglen, Matthew Norwood, Daniel B. Ravicher, Karen Sandler, James Vasile, and Aaron Williamson. A legal issues primer for Open Source and Free Software projects. version 1.5.1. Technical report, Software Freedom Law Center, New York, March 2008. <http://www.softwarefreedom.org/resources/2008/foss-primer.html>. Accessed 2009-11-09.
- [192] Michel Foucault. *L'ordre du discours; leçon inaugurale au Collège de France prononcée le 2 décembre 1970*. Gallimard, Paris, 1971.
- [193] Martin Fowler. Inversion of control containers and the dependency injection pattern. <http://www.martinfowler.com/articles/injection.html>, January 2004.
- [194] Martin Fowler. Module assembly. *IEEE Software*, 21(2):65–67, 2004.
- [195] Martin Fowler. Continuous Integration. <http://www.martinfowler.com/articles/continuousIntegration.html>, May 2006. Accessed 2006-11-16.
- [196] Free Software Foundation. Gnu general public license version 3. <http://www.gnu.org/licenses/gpl.html>, June 2007. Accessed 2009-10-16.
- [197] Free Software Foundation. Gnu lesser general public license version 3. <http://www.gnu.org/copyleft/lesser.html>, June 2007. Accessed 2009-11-09.
- [198] Linton C. Freeman. Visualizing social networks. *Journal of Social Structure*, 1(1), 2000.
- [199] Linton C. Freeman. *The Development Of Social Network Analysis— A Study In The Sociology Of Science*. Empirical Press, Vancouver, BC, Canada, 2004.
- [200] Cristina Gacek and Budi Arief. The many meanings of Open Source. *IEEE Software*, 21(1):34–40, January/February 2004.
- [201] Erich Gamma. Agile, Open Source, distributed, and on-time: inside the Eclipse development process. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 4–4. ACM, 2005. Slides of keynote talk available at <http://icse-conferences.org/2005/ConferenceProgram/InvitedTalks/GammaKeynote.pdf>. Accessed 2010-04-25.
- [202] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [203] Mike Gancarz. *Unix philosophy*. Digital Press, New Jersey, 1993.
- [204] Mike Gancarz. *Linux and the Unix philosophy*. Digital Press, Amsterdam; Boston, 2003.
- [205] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.
- [206] Raghu Garud and Peter Karnøe. Path creation as a process of mindful deviation. In *Path Dependency and Creation*, chapter 1, pages 1–38. Lawrence Earlbaum Associates, 2001.
- [207] Daniel M. German. The GNOME project: a case study of Open Source, global software development. *Software Process: Improvement and Practice*, 8(4):201–215, 2003.
- [208] Daniel M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, September 2006.
- [209] Daniel M. German and Jesús M. González-Barahona. An empirical study of the reuse of software licensed under the GNU General Public License. In *Open Source Ecosystems: Diverse Communities*

- Interacting*, volume 299/2009 of *IFIP Advances in Information and Communication Technology*, pages 185–198. Springer, Boston, June 2009.
- [210] Rishab Aiyer Ghosh. Understanding Free Software developers: Findings from the FLOSS study. In Feller et al. [180], pages 23–46.
- [211] Rishab Aiyer Ghosh, Ruediger Glott, Bernhard Krieger, and Gregorio Robles. Free/Libre and Open Source Software: Survey and study – FLOSS – Part 4: Survey of developers. Final Report, International Institute of Infonomics University of Maastricht, The Netherlands; Berlecon Research GmbH Berlin, Germany, June 2002.
- [212] Anthony Giddens. *The constitution of society: outline of the theory of structuration*. University of California Press, Berkeley, 1984.
- [213] David G. Glance. Release criteria for the Linux kernel. *First Monday*, 9(4), April 2004.
- [214] Barney G. Glaser. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory*. Sociology Press, Mill Valley, CA, 1978.
- [215] Bernard Golden. Making Open Source ready for the enterprise: The Open Source Maturity Model. Whitepaper, Navica, 2005.
- [216] Bernard Golden. *Succeeding with Open Source*. Addison-Wesley, August 2005.
- [217] Jesús M. Gonzalez-Barahona, Gregorio Robles, Roberto Andradás-Izquierdo, and Rishab Aiyer Ghosh. Geographic origin of libre software developers. *Information Economics and Policy*, 20(4):356–363, December 2008. Empirical Issues in Open Source Software.
- [218] Sigi Goode. Something for nothing: management rejection of Open Source software in Australia's top firms. *Information & Management*, 42(5):669–681, July 2005.
- [219] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Amsterdam, 3rd edition, June 2005.
- [220] Mark G. Graff and Kenneth R. van Wyk. *Secure Coding: Principles & Practices*. O'Reilly, Sebastopol, CA, USA, 1st edition, June 2003.
- [221] James Gray. Interview with Guido van Rossum. *Linux Journal*, 174, October 2008.
- [222] T.R.G. Green and M. Petre. When visual programs are harder to read than textual programs. In G.C. van der Veer, M.J. Tauber, S. Bagnarola, and M. Antalovits, editors, *ECCE-6: Proceedings of the 6th European Conference on Cognitive Ergonomics, Lake Balaton, Hungary*, pages 167–180, 1992.
- [223] Rajdeep Grewal, Gary L. Lilien, and Girish Mallapragada. Location, location, location: How network embeddedness affects project success in Open Source systems. *Management Science*, 52(7):1043–1056, July 2006.
- [224] Ido Guy, Michal Jacovi, Noga Meshulam, Inbal Ronen, and Elad Shahrar. Public vs. private: comparing public social network information with email. In *CSCW '08: Proceedings of the ACM 2008 conference on Computer supported cooperative work*, pages 393–402. ACM, 2008.
- [225] Jürgen Habermas. *Theorie des kommunikativen Handelns*. Suhrkamp, Frankfurt am Main, 1st edition, 1981.
- [226] Stefan Haefliger, Georg von Krogh, and Sebastian Spaeth. Code reuse in Open Source software. *Management Science*, 54(1):180–193, January 2008.
- [227] Lou Hafer and Arthur E. Kirkpatrick. Assessing Open Source software as a scholarly contribution. *Commun. ACM*, 52(12):126–129, December 2009.

- [228] Michael Hahsler. A quantitative study of the adoption of design patterns by Open Source software developers. In Stefan Koch, editor, *Free/Open Source Software Development*, chapter 5, pages 103–123. Idea Group Publishing, 2005.
- [229] T. J. Halloran and William L. Scherlis. High quality and Open Source software practices. In Feller et al. [177], pages 26–28.
- [230] Julio C. Hamano. GIT — a stupid content tracker. In *Proceedings of the 2008 Linux Symposium, Ottawa, Canada*, July 2006.
- [231] Jim Hamerly, Tom Paquin, and Susan Walton. Freeing the source: The story of Mozilla. In DiBona et al. [144], pages 197–206.
- [232] Jeffrey S. Hammond, Mary Gerush, and Justinas Sileikis. Open Source software goes mainstream. Technical Report 54205, Forrester, June 2009.
- [233] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input. In *Proceedings of the International Conference on Automated Software Engineering*, pages 332–343. IEEE Computer Society, 2009.
- [234] Il-Horn Hann, Jeff Roberts, and Sandra Slaughter. Why developers participate in Open Source software projects: An empirical investigation. In *International Conference on Information Systems (ICIS), December 12-15, Washington, DC*, pages 821–830. Association for Information Systems, 2004.
- [235] Il-Horn Hann, Jeff Roberts, Sandra Slaughter, and Roy Fielding. An empirical analysis of economic returns to Open Source participation. Working Paper Series 2006-E5, Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA, 2006.
- [236] Glenn W. Harrison and John A. List. Field experiments. *Journal of Economic Literature*, 42(4):1009–1055, December 2004.
- [237] Warren Harrison. Editorial: Open Source and empirical software engineering. *Empirical Software Engineering*, 6(3):193–194, September 2001.
- [238] Alexander Hars and Shaosong Ou. Working for free? – motivations of participating in Open Source projects. In *The 34th Hawaii International Conference on System Sciences*, 2001.
- [239] Erik Hatcher and Otis Gospodnetic. *Lucene in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.
- [240] Jürgen Hauschildt. *Innovationsmanagement*. Verlag Vahlen, München, Germany, 3rd edition, June 2004.
- [241] Frank Hecker. Setting up shop: The business of Open-Source software. *IEEE Software*, 16(1):45–51, 1999.
- [242] Robert Heckman, Kevin Crowston, U. Yeliz Eseryel, James Howison, Eileen Allen, and Qing Li. Emergent decision-making practices in Free/Libre Open Source Software (FLOSS) development teams. In *Open Source Development, Adoption and Innovation*, volume 234/2007 of *IFIP International Federation for Information Processing*, pages 71–84. Springer, Boston, MA, 2007.
- [243] Robert Heckman, Kevin Crowston, and Nora Misiolek. A structural perspective on leadership in virtual teams. In *Virtuality and Virtualization*, volume 236/2007 of *IFIP International Federation for Information Processing*, pages 151–168. Springer, Boston, 2007.
- [244] Gunnar Hedlund. The hypermodern MNC - a heterarchy? *Human Resource Management*, 25(1):9–35, 1986.
- [245] Jeffrey Heer and Danah Boyd. Vizster: Visualizing online social networks. In *INFOVIS '05: Proceedings of the 2005 IEEE Symposium on Information Visualization*, pages 32–39. IEEE Computer Society, 2005.



- [246] Andrea Hemetsberger and Christian Reinhardt. Sharing and creating knowledge in Open-Source communities: The case of KDE. In *Proceedings of the Fifth European Conference on Organizational Knowledge, Learning and Capabilities (OKLC)*, April 2004.
- [247] Joachim Henkel. Champions of revealing — the role of Open Source developers in commercial firms. *Industrial and Corporate Change*, 18(3):435–471, 2009.
- [248] Mark Henley and Richard Kemp. Open Source software: An introduction. *Computer Law & Security Report*, 24(1):77–85, 2008.
- [249] James D. Herbsleb and Rebecca E. Grinter. Splitting the organization and integrating the code: Conway’s law revisited. In *ICSE ’99: Proceedings of the 21st International Conference on Software Engineering*, pages 85–95. IEEE Computer Society, 1999.
- [250] Susan C. Herring. Interactional coherence in CMC. In *HICSS ’99: Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences-Volume 2*, page 2022. IEEE Computer Society, 1999.
- [251] Paul Turner Hershey. A definition for paternalism. *Journal of Medicine & Philosophy*, 10(2):171–182, 1985.
- [252] Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of software developers in Open Source projects: an internet-based survey of contributors to the Linux kernel. *Research Policy*, 32(7):1159–1177, July 2003.
- [253] James Hewitt. Beyond threaded discourse. *International Journal of Educational Telecommunications*, 7(3):207–221, 2001.
- [254] Francis Heylighen. Why is Open Access development so successful? stigmergic organization and the economics of information. In B. Lutterbeck, M. Baerwolff, and R. A. Gehring, editors, *Open Source Jahrbuch 2007*, pages 165–180. Lehmanns Media, 2007.
- [255] Henrik Holum and Svein Erik Reknes Løvland. Joining in Apache Derby: Removing the obstacles. Master’s thesis, Norwegian University of Science and Technology NTNU, June 2008.
- [256] Jeff Howe. The rise of crowdsourcing. *Wired Magazine*, 14(06), June 2006.
- [257] James Howison and Kevin Crowston. The perils and pitfalls of mining SourceForge. *IEE Seminar Digests*, 2004(917):7–11, 2004.
- [258] Julian P. Höppner. The GPL prevails: An analysis of the first-ever court decision on the validity and effectivity of the GPL. *SCRIPT-ed*, 1(4):628–635, December 2004.
- [259] Christopher L. Huntley. Organizational learning in Open Source software projects: an analysis of debugging data. *IEEE Transactions on Engineering Management*, 50(4):485–493, November 2003.
- [260] Federico Iannacci and Eve Mitleton-Kelly. Beyond markets and firms: The emergence of Open Source networks. *First Monday*, 10(5), 2005.
- [261] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
- [262] International Organization for Standardization. ISO/IEC IS 8601:2004: Data elements and interchange formats – information exchange – representation of dates and times. [http://www.iso.org/iso/catalogue\\_detail?csnumber=40874](http://www.iso.org/iso/catalogue_detail?csnumber=40874), 2004. Accessed 2009-10-17.
- [263] International Phonetic Association. *Handbook of the International Phonetic Association: A guide to the use of the International Phonetic Alphabet*. Cambridge University Press, Cambridge, U.K., 1999.
- [264] Lynn A. Isabella. Evolving interpretations as a change unfolds: How managers construe key organizational events. *The Academy of Management Journal*, 33(1):7–41, March 1990.

- [265] Letizia Jaccheri and Thomas Østerlie. Open Source software: A source of possibilities for software engineering education and empirical software engineering. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, page 5. IEEE Computer Society, 2007.
- [266] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting experiments in software engineering. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.
- [267] Ron Jeffries and Grigori Melnik. TDD – The art of fearless programming. *IEEE Softw.*, 24(3):24–30, May 2007.
- [268] Chris Jensen and Walt Scacchi. Process modeling across the web information infrastructure. *Software Process: Improvement and Practice*, 10(3):255–272, 2005.
- [269] Chris Jensen and Walt Scacchi. Role migration and advancement processes in OSSD projects: A comparative case study. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2007.
- [270] Justin Pappas Johnson. Open Source software: Private provision of a public good. *Journal of Economics & Management Strategy*, 11(4):637–662, 2002.
- [271] Kim Johnson. A descriptive process model for Open-Source software development. Masters thesis, University of Calgary, Department of Computer Science, June 2001. Accessed 2005-08-02.
- [272] Danny L. Jorgensen. *Participant observation: a methodology for human studies*, volume 15 of *Applied social research methods series*. Sage, Newbury Park, CA, 1989.
- [273] Niels Jørgensen. Putting it all in the trunk: Incremental software development in the FreeBSD Open Source project. *Information Systems Journal*, 11(4):321–336, 2001.
- [274] Niels Jørgensen and Jesper Holck. Overloading the development branch? A view of motivation and incremental development in FreeBSD. In Brian Fitzgerald and David L. Parnas, editors, *Workshop "Making Free/Open-Source Software (F/OSS) Work Better"*, XP2003 Conference, pages 17–18, May 2003.
- [275] Paul C. Jorgensen and Carl Erickson. Object-oriented integration testing. *Commun. ACM*, 37(9):30–38, September 1994.
- [276] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.
- [277] Michael L. Katz and Carl Shapiro. Network externalities, competition, and compatibility. *The American Economic Review*, 75(3):424–440, June 1985.
- [278] Federico Kereki. Xfce: the third man. *Linux Journal*, 2009(179), March 2009.
- [279] Bernard Kerr. THREAD ARCS: An email thread visualization. In *2003 IEEE Symposium on Information Visualization*, page 27. Citeseer, 2003.
- [280] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, August 2002.
- [281] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium, Ottawa, Canada*, pages 225–230, July 2007.
- [282] Jon Kleinberg. The convergence of social and technological networks. *Commun. ACM*, 51(11):66–72, 2008.

- [283] A. S. Klov Dahl. Social networks and the spread of infectious diseases: the AIDS example. *Social Science and Medicine*, 21(11):1203–1216, 1985.
- [284] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 344–353. IEEE Computer Society, 2007.
- [285] Stefan Koch. Software evolution in Open Source projects—a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(6):361–382, June 2007.
- [286] Stefan Koch. Exploring the effects of SourceForge.net coordination and communication tools on the efficiency of Open Source projects using data envelopment analysis. *Empirical Software Engineering*, 14:397–417, 2009.
- [287] Stefan Koch and Georg Schneider. Effort, co-operation and co-ordination in an Open Source software project: GNOME. *Information Systems Journal*, 12(1):27–42, 2002.
- [288] John Koenig. Seven Open Source business strategies for competitive advantage. *IT Manager's Journal*, 14, May 2004.
- [289] Bruce Kogut and Anca Metiu. Open-Source Software development and distributed innovation. *Oxford Review of Economic Policy*, 17:248–264, June 2001.
- [290] Günes Koru, Khaled El Emam, Angelica Neisa, and Medha Umarji. A survey of quality assurance practices in biomedical Open Source software projects. *Journal of Medical Internet Research*, 9(2):e8, May 2007.
- [291] Gueorgi Kossinets and Duncan J. Watts. Empirical Analysis of an Evolving Social Network. *Science*, 311(5757):88–90, January 2006.
- [292] Ben Kovitz. How to converse deeply on a Wiki. Why Clublet, February 2001. Accessed 2007-06-21.
- [293] Martin F. Krafft. *A Delphi study of the influences on innovation adoption and process evolution in a large open-source project—the case of Debian*. PhD thesis, University of Limerick, Ireland, April 2010.
- [294] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson. Linux kernel development—how fast it is going, who is doing it, what they are doing, and who is sponsoring it. The Linux Foundation Publication <http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php>, April 2008. Accessed 2010-02-22.
- [295] Ko Kuwabara. Linux: A bazaar at the edge of chaos. *First Monday*, 5(3), March 2000.
- [296] Tae H. Kwon and Robert W. Zmud. Unifying the fragmented models of information systems implementation. In Richard J. Boland and Rudy A. Hirschheim, editors, *Critical Issues in Information Systems Research*, chapter 10, pages 227–251. Wiley, 1987.
- [297] Karim R. Lakhani and Eric von Hippel. How Open Source software works: “free” user-to-user assistance. *Research Policy*, 32(6):923–943, June 2003.
- [298] Karim R. Lakhani and Robert G. Wolf. Why hackers do what they do: Understanding motivation and effort in Free/Open Source Software projects. In Feller et al. [180], pages 3–22.
- [299] Ann Langley. Strategies for theorizing from process data. *Academy of Management Review*, 24(4):691–710, 1999.
- [300] Giovan Francesco Lanzara and Michèle Morner. Artifacts rule! How organizing happens in Open Source software projects. In *Actor-Network Theory and Organizing*, pages 67–90. Copenhagen Business School Press, 1st edition, June 2005.
- [301] Jean Lave and Etienne Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, September 1991.

- [302] Anna Law, Yvonne Freer, Jim Hunter, Robert Logie, Neil McIntosh, and John Quinn. A comparison of graphical and textual presentations of time series data to support medical decision making in the neonatal intensive care unit. *Journal of Clinical Monitoring and Computing*, 19(3):183–194, June 2005.
- [303] John Law. Notes on the theory of the actor-network: Ordering, strategy and heterogeneity. *Systems Practice*, 5(4):379–393, 1992.
- [304] Christian Lüders. Beobachten im Feld und Ethnographie. In Uwe Flick, Ernst von Kardorff, and Ines Steinke, editors, *Qualitative Forschung*, pages 384–401. Reinbek: Rowohlt, 2000.
- [305] Josh Lerner and Jean Tirole. Some simple economics of open source. *The Journal of Industrial Economics*, 50(2):197–234, June 2002.
- [306] Josh Lerner and Jean Tirole. The scope of Open Source licensing. *Journal of Law, Economics, and Organization*, 21(1):20–56, 2005.
- [307] Lawrence Lessig. *Code and Other Laws of Cyberspace*. Basic Books, New York, July 2000.
- [308] Lawrence Lessig. *Free Culture: The Nature and Future of Creativity*. Penguin Press, February 2005.
- [309] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [310] Barbara Levitt and Clifford Nass. The lid on the garbage can: Institutional constraints on decision making in the technical core of college-text publishers. *Administrative Science Quarterly*, 34(2):190–207, June 1989.
- [311] Steven Levy. *Hackers: heroes of the computer revolution*. Anchor Press/Doubleday, Garden City, N.Y., 1984.
- [312] Kurt Lewin. Action research and minority problems. *Journal of Social Issues*, 2(4):34–46, November 1946.
- [313] Qing Li, Robert Heckman, Kevin Crowston, James Howison, Eileen Allen, and U. Yeliz Eseryel. Decision making paths in self-organizing technology-mediated distributed teams. In *Proceedings of the International Conference on Information Systems (ICIS) 2008*. Association for Information Systems, 2008.
- [314] S. J. Liebowitz and Stephen E. Margolis. Path dependence, lock-in, and history. *Journal of Law, Economics and Organization*, 11(1):205–226, April 1995.
- [315] Yu-Wei Lin and Enrico Zini. Free/libre Open Source software implementation in schools: Evidence from the field and implications for the future. *Computers & Education*, 50(3):1092–1102, 2008.
- [316] Yuwei Lin. The future of sociology of FLOSS. *First Monday*, Special Issue #2, October 2005.
- [317] Juho Lindman, Matti Rossi, and Pentti Marttiin. Applying Open Source development practices inside a company. In *Open Source Development, Communities and Quality*, volume 275/2008 of *IFIP International Federation for Information Processing*, pages 381–387. Springer, Boston, July 2008.
- [318] Jacques Lonchamp. Open Source Software development process modeling. In *Software Process Modeling*, volume 10 of *International Series in Software Engineering*, chapter 1, pages 29–64. Springer, Heidelberg, 2005.
- [319] Scott Long. Perforce in FreeBSD development. Available at <http://www.freebsd.org/doc/en/articles/p4-primer/article.html>, August 2008. Accessed 2010-02-15.
- [320] Luis López-Fernández, Gregorio Robles, Jesús Gonzalez-Barahona, and Israel Herráiz. Applying social network analysis techniques to community-driven Libre Software projects. *International Journal of Information Technology and Web Engineering*, 1(3):27–48, 2006.

- [321] Luis López-Fernández, Gregorio Robles, and Jesús M. Gonzalez-Barahona. Applying social network analysis to the information in CVS repositories. *IEE Seminar Digests*, 2004(917):101–105, 2004.
- [322] Niklas Luhmann. *Soziale Systeme. Grundriß einer allgemeinen Theorie*. Suhrkamp, Frankfurt am Main, January 1984.
- [323] Benno Luthiger. Fun and software development. In Marco Scotto and Giancarlo Succi, editors, *First International Conference on Open Source Systems, Genova, Italy*, pages 273–278, Genova, July 2005.
- [324] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the structure of complex software designs: An empirical study of Open Source and proprietary code. *Management Science*, 52(7):1015–1030, July 2006.
- [325] Conor MacNeill. The early history of ant development. Personal Blog. <http://codefeed.com/blog/?p=98>, August 2005. Accessed 2010-03-01.
- [326] Greg Madey, Vincent Freeh, and Renee Tynan. The Open Source software development phenomenon: An analysis based on social network theory. In *8th Americas Conference on Information Systems (AMCIS2002)*, Dallas, TX, pages 1806–1813, 2002.
- [327] Gregory Madey, Vincent Freeh, and Renee Tynan. Modeling the F/OSS community: A quantitative investigation. In Stefan Koch, editor, *Free/Open Source Software Development*, chapter 9, pages 203–220. Idea Group Publishing, 2005.
- [328] Dilan Mahendran. Serpents & primitives: An ethnographic excursion into an Open Source community. Unpublished master's final project, University of California at Berkley, School of Information, Berkley, CA, 2002.
- [329] James Mahoney. Path dependence in historical sociology. *Theory and Society*, 29(4):507–548, August 2000.
- [330] Kevin Makice. Politicwiki: exploring communal politics. In *WikiSym '06: Proceedings of the 2006 international symposium on Wikis*, pages 105–118. ACM Press, 2006.
- [331] Mary L. Manns and Linda Rising. *Fearless Change: Patterns for Introducing New Ideas*. Addison-Wesley, September 2004.
- [332] Vincent Massol and Timothy M. O'Brien. *Maven: A Developer's Notebook*. O'Reilly, June 2005.
- [333] Marcel Mauss. *The gift: the forms and functions of exchange in archaic societies*. Cohen and West, London, 1954.
- [334] Philipp Mayring. *Einführung in die qualitative Sozialforschung*. Beltz, Weinheim, February 2002.
- [335] Steve McConnell. Open-Source methodology: Ready for prime time? *IEEE Software*, 16(4):6–8, 1999.
- [336] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, Washington, 2nd edition, July 2004.
- [337] Warren S. McCulloch. A heterarchy of values determined by the topology of nervous nets. *Bulletin of Mathematical Biology*, 7(2):89–93, June 1945.
- [338] Daniel McGowan. Legal implications of Open-Source software. *The University of Illinois Law Review*, 2001(1):241ff, 2001.
- [339] Martin Michlmayr and Benjamin Mako Hill. Quality and the reliance on individuals in Free Software projects. In Feller et al. [178], pages 105–109.
- [340] Martin Michlmayr, Francis Hunt, and David Probert. Release management in Free Software projects: Practices and problems. In *Open Source Development, Adoption and Innovation*, volume

- 234/2007 of IFIP International Federation for Information Processing, pages 295–300. Springer, Boston, 2007.
- [341] Stanley Milgram. The small-world problem. *Psychology Today*, 1(1):61–67, May 1967.
  - [342] Henry Mintzberg. An emerging strategy of “direct” research. *Administrative Science Quarterly*, 24(4):582–589, December 1979.
  - [343] Henry Mintzberg and Alexandra McHugh. Strategy formation in an adhocracy. *Administrative Science Quarterly*, 30(2):160–197, June 1985.
  - [344] Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In Michael W. Godfrey and Jim Whitehead, editors, *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009*, pages 11–20. IEE, 2009.
  - [345] Audris Mockus. Succession: Measuring transfer of code and developer productivity. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 67–77. IEEE Computer Society, 2009.
  - [346] Audris Mockus, Roy T. Fielding, and James Herbsleb. A case study of Open Source software development: the Apache server. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 263–272. ACM, 2000.
  - [347] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two case studies of Open Source Software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.
  - [348] Rick Moen. Fear of forking. [http://linuxmafia.com/faq/Licensing\\_and\\_Law/forking.html](http://linuxmafia.com/faq/Licensing_and_Law/forking.html), November 1999. Accessed 2009-08-22.
  - [349] Eben Moglen. Anarchism triumphant: Free Software and the death of copyright. *First Monday*, 4(8), August 1999.
  - [350] Glyn Moody. *Rebel Code: Linux and the Open Source Revolution*. Basic Books, New York, new edition, June 2002.
  - [351] Jae Yun Moon and Lee Sproull. Essence of distributed work: The case of the Linux kernel. *First Monday*, 5(11), November 2000.
  - [352] K. Morgan, R. L. Morris, and S. Gibbs. When does a mouse become a rat? or ...comparing performance and preferences in direct manipulation and command line environment. *The Computer Journal*, 34(3):265–271, 1991.
  - [353] Håvard Mork. Leadership in hybrid commercial-open source software development. Directed study, Norwegian University of Science and Technology, December 2005.
  - [354] Håvard Mork. Documentation practices in Open Source — a study of Apache Derby. Master's thesis, Norwegian University of Science and Technology, 2006.
  - [355] Mozilla Foundation. Mozilla public license version 1.1. <http://www.mozilla.org/MPL/MPL-1.1.html>. Accessed 2009-10-25.
  - [356] Moreno Muffatto and Matteo Faldani. Open Source as a complex adaptive system. *Emergence*, 5(3):83–100, 2003.
  - [357] Thomas Muhr. Atlas/ti — a prototype for the support of text interpretation. *Qualitative Sociology*, 14(4):349–371, December 1991.
  - [358] Yefim V. Natis, George J. Weiss, Mark Driver, Nicholas Gall, Daniel Sholler, and Brian Prentice. The state of Open Source, 2008. Technical Report G00156659, Gartner, April 2008.

- [359] M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 98(2):404–409, January 2001.
- [360] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
- [361] M. E. J. Newman. Analysis of weighted networks. *Physical Review E*, 70(5):056131, November 2004.
- [362] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences (PNAS)*, 103(23):8577–8582, June 2006.
- [363] David M. Nichols and Michael B. Twidale. The usability of Open Source software. *First Monday*, 8(1), January 2003.
- [364] G. Nicolis and Ilya Prigogine. *Self-organization in nonequilibrium systems: from dissipative structures to order through fluctuations*. Wiley, New York, 1977.
- [365] Friedrich Wilhelm Nietzsche. *Menschliches, Allzumenschliches: Ein Buch für freie Geister*. Schmeitzner, Chemnitz, May 1878.
- [366] Blair Nonnecke and Jenny Preece. Why lurkers lurk. In *Americas Conference on Information Systems*, June 2001.
- [367] Joseph Donald Novak and Alberto J. Cañas. The theory underlying concept maps and how to construct them. IHMC CmapTools 2006-01, Florida Institute for Human and Machine Cognition, January 2006.
- [368] Joseph Donald Novak and D. Bob Gowin. *Learning how to learn*. Cambridge University Press, New York, 1984.
- [369] Object Management Group. Object Constraint Language (OCL) version 2.0. Technical Report formal/06-05-01, Object Management Group, May 2006.
- [370] Object Management Group. Software Process Engineering Meta-Model (SPEM), version 2.0. Technical Report formal/2008-04-01, Object Management Group, 2008.
- [371] Object Management Group. OMG Unified Modeling Language (OMG UML) Superstructure Specification Version 2.2. Technical Report formal/09-02-02, Object Management Group, 2009.
- [372] Christopher Oezbek. Introduction of Innovation M1. Technical report, Freie Universität Berlin, April 2008.
- [373] Christopher Oezbek. Research ethics for studying Open Source projects. In *4th Research Room FOSDEM: Libre software communities meet research community*, February 2008.
- [374] Christopher Oezbek. Introducing automated regression testing in Open Source projects. Technical Report TR-B-10-01, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, January 2010.
- [375] Christopher Oezbek. Introducing automated regression testing in Open Source projects. In *Proceedings of the OSS Conference 2010, South Bend, USA*, May 2010.
- [376] Christopher Oezbek and Lutz Prechelt. JTourBus: Simplifying program understanding by documentation that provides tours through the source code. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*. IEEE Computer Society, 2007.
- [377] Christopher Oezbek and Lutz Prechelt. On understanding how to introduce an innovation to an Open Source project. In *Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW '07)*, Washington, DC, USA, 2007. IEEE Computer Society. reprinted in *UPGRADE, The European Journal for the Informatics Professional* 8(6):40-44, December 2007.

- [378] Christopher Oezbek, Lutz Prechelt, and Florian Thiel. The onion has cancer: Some social network analysis visualizations of Open Source project communication. In *Proceedings of the 2010 ICSE Workshop on Free, Libre and Open Source Software*, January 2010.
- [379] Christopher Oezbek, Robert Schuster, and Lutz Prechelt. Information management as an explicit role in OSS projects: A case study. Technical Report TR-B-08-05, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, April 2008.
- [380] Christopher Oezbek and Florian Thiel. Radicality and the open source development model. In *Proceedings of the FLOSS Workshop 2010, 1–2 July*, Jena, Germany., July 2010.
- [381] Michael Ogawa, Kwan-Liu Ma, Christian Bird, Premkumar Devanbu, and Alex Gourley. Visualizing social interaction in Open Source software projects. In *6th International Asia-Pacific Symposium on Visualization*, pages 25–32. IEEE Computer Society, 2007.
- [382] Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Ken-Ichi Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5. ACM, 2005.
- [383] Chitu Okoli and Suzanne D. Pawlowski. The Delphi method as a research tool: an example, design considerations and applications. *Information & Management*, 42(1):15–29, December 2004.
- [384] Johan P. Olsen. Garbage cans, new institutionalism, and the study of politics. *The American Political Science Review*, 95(1):191–198, March 2001.
- [385] Timothy G. Olson, Watts S. Humphrey, and Dave Kitson. Conducting SEI-assisted software process assessments. Technical Report CMU/SEI-89-TR-7, Software Engineering Institute, Pittsburgh, February 1989.
- [386] Siobhán O'Mahony. Guarding the commons: how community managed software projects protect their work. *Research Policy*, 32(7):1179–1198, July 2003.
- [387] Siobhán O'Mahony. Nonprofit foundations and their role in community-firm software collaboration. In Feller et al. [180], chapter 20, pages 393–414.
- [388] Siobhán O'Mahony. The governance of Open Source initiatives: what does it mean to be community managed? *Journal of Management & Governance*, 11(2):139–150, May 2007.
- [389] Wanda J. Orlikowski and Jack J. Baroudi. Studying information technology in organizations: Research approaches and assumptions. *Information Systems Research*, 2(1):1–28, 1991.
- [390] Stanisław Osiński and Dawid Weiss. Introducing usability practices to OSS: The insiders' experience. In *Proceedings of the 3rd International Conference on Open Source Systems, OSS2007, Limerick, UK*, 2007.
- [391] Thomas Østerlie. In the network: Distributed control in Gentoo Linux. In *Collaboration, Conflict and Control: Proceedings of the 4th Workshop on Open Source Software Engineering W8S Workshop - 26th International Conference on Software Engineering*, Edinburgh, Scotland, UK, pages 76–81, May 2004.
- [392] Thomas Østerlie and Letizia Jaccheri. Balancing technological and community interest: The case of changing a large Open Source Software system. In Tarja Tiainen, Hannakaisa Isomäki, Mikko Korpela, and Anja Mursu, editors, *Proc. 30th Information Systems Research Conference (IRIS'30)*, number D-2007-9 in D-Net Publications, pages 66–80, Finland, August 2007. Department of Computer Sciences, University of Tampere.
- [393] Thomas Østerlie and Alf Inge Wang. Establishing maintainability in systems integration: Ambiguity, negotiations, and infrastructure. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, Philadelphia, Pennsylvania, USA, pages 186–196, Washington, DC, USA, 2006. IEEE Computer Society.



- [394] Thomas Østerlie and Alf Inge Wang. Debugging integrated systems: An ethnographic study of debugging practice. In *23rd IEEE International Conference on Software Maintenance (ICSM)*, Paris, France, pages 305–314. IEEE Computer Society, October 2007.
- [395] Päivi Ovaska, Matti Rossi, and Pentti Marttiin. Architecture as a coordination tool in multi-site software development. *Software Process: Improvement and Practice*, 8(4):233–247, 2003.
- [396] Keith Packard. Tyrannical SCM selection. [http://keithp.com/blogs/Tyrannical\\_SCM\\_selection/](http://keithp.com/blogs/Tyrannical_SCM_selection/), September 2007. Accessed 2010-02-16.
- [397] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, June 2005.
- [398] Cyril Northcote Parkinson. *Parkinson's law: and other studies in administration*. The Riverside Press, Cambridge, Mass., 1957.
- [399] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [400] Terence J. Parr and R. W. Quong. ANTLR: a predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
- [401] James W. Paulson, Giancarlo Succi, and Armin Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30(4):246–256, 2004.
- [402] Russell Pavlicek. *Embracing Insanity: Open Source Software Development*. Sams, Indianapolis, IN, USA, 2000.
- [403] Christian Pentzold and Sebastian Seidenglanz. Foucault@Wiki: First steps towards a conceptual framework for the analysis of Wiki discourses. In *WikiSym '06: Proceedings of the 2006 International Symposium on Wikis*, pages 59–68. ACM Press, 2006.
- [404] Bruce Perens. The Open Source definition. In DiBona et al. [144], pages 171–188.
- [405] Etel Petrinja, Ranga Nambakam, and Alberto Sillitti. Introducing the OpenSource Maturity Model. In *FLOSS '09: Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 37–41. IEEE Computer Society, 2009.
- [406] Shari Lawrence Pfleeger. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software*, 47(2–3):111–124, 1999.
- [407] Paul Pierson. Increasing returns, path dependence, and the study of politics. *The American Political Science Review*, 94(2):251–267, June 2000.
- [408] Karl Popper. *Logik der Forschung: zur Erkenntnistheorie der Modernen Naturwissenschaft*, volume 9 of *Schriften zur Wissenschaftlichen Weltauffassung*. Julius Springer, Wien, 1934.
- [409] Mason A. Porter, Jukka-Pekka Onnela, and Peter J. Mucha. Communities in networks. *Notices of the AMS*, 56(9):1082–1097, October 2009.
- [410] Lutz Prechelt and Christopher Oezbek. The search for a research method for studying OSS process innovation. Submitted to *Empirical Software Engineering*, January 2010.
- [411] Jenny Preece, Blair Nonnecke, and Dorine Andrews. The top five reasons for lurking: Improving community experiences for everyone. *Computers in Human Behavior*, 20(2):201–223, 2004. The Compass of Human-Computer Interaction.
- [412] Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. Project Management Institute, Newton Square, PA, 2000 edition, 2000.
- [413] Luis Quintela García. Die Kontaktaufnahme mit Open Source Software-Projekten. Eine Fallstudie. Bachelor thesis, Freie Universität Berlin, 2006.

- [414] Eric S. Raymond. Email quotes and inclusion conventions. In *The new hacker's dictionary*, pages 20–22. MIT Press, Cambridge, MA, USA, 3rd edition, 1996. <http://www.ccil.org/jargon/>. Accessed 2009-06-30.
- [415] Eric S. Raymond. The cathedral and the bazaar. *First Monday*, 3(3), 1998.
- [416] Eric S. Raymond. Homesteading the Noosphere. *First Monday*, 3(10), 1998.
- [417] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Sebastopol, CA, USA, 1999.
- [418] Eric S. Raymond. The magic cauldron. In *The Cathedral and the Bazaar* [417], chapter 4, pages 113–166.
- [419] Eric Steven Raymond. How to become a hacker. <http://catb.org/~esr/faqs/hacker-howto.html>, 2001. Accessed 2010-07-24.
- [420] Joseph M. Reagle, Jr. Do as I do: authorial leadership in Wikipedia. In *WikiSym '07: Proceedings of the 2007 International Symposium on Wikis*, pages 143–156. ACM, 2007.
- [421] Red Hat Inc. Red Hat Annual Report 2009. [http://files.shareholder.com/downloads/RHAT/898346664x0x304106/3145E646-AE56-4FE1-9C59-B79F4491C4C5/FY09\\_Annual\\_Report\\_on\\_Form\\_10-K.pdf](http://files.shareholder.com/downloads/RHAT/898346664x0x304106/3145E646-AE56-4FE1-9C59-B79F4491C4C5/FY09_Annual_Report_on_Form_10-K.pdf), 2009. Accessed 2010-04-22.
- [422] Christian Robottom Reis and Renata Pontin de Mattos Fortes. An overview of the software engineering process and tools in the Mozilla project. In Cristina Gacek and Budi Arief, editors, *Workshop on Open Source Software Development, Newcastle, United Kingdom*, pages 155–175. University of Newcastle upon Tyne, February 2002.
- [423] Paul Resnick, Derek Hansen, John Riedl, Loren Terveen, and Mark Ackerman. Beyond threaded conversation. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 2138–2139. ACM Press, 2005.
- [424] Peter W. Resnick. Internet message format. Request for Comments 2822, Internet Engineering Task Force, April 2001.
- [425] Ronald E. Rice and Everett M. Rogers. Reinvention in the innovation process. *Science Communication*, 1(4):499–514, 1980.
- [426] Dirk Riehle. The economic case for Open Source foundations. *Computer*, 43(1):86–90, January 2010.
- [427] E. Sean Rintel and Jeffery Pittam. Strangers in a strange land – interaction management on Internet Relay Chat. *Human Communication Research*, 23(4):507–534, 1997.
- [428] Brian D. Ripley. The R project in statistical computing. *MSOR Connections. The newsletter of the LTSN Maths, Stats & OR Network.*, 1(1):23–25, February 2001.
- [429] Gabriel Ripoché and Jean-Paul Sansonnet. Experiences in automating the analysis of linguistic interactions for the study of distributed collectives. *Computer Supported Cooperative Work (CSCW)*, 15(2):149–183, June 2006.
- [430] Jason E. Robbins. Adopting Open Source software engineering (OSSE) practices by adopting OSSE tools. In Feller et al. [180], pages 245–264.
- [431] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. Argo: a design environment for evolving software architectures. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 600–601. ACM, 1997.
- [432] Jason Elliot Robbins. *Cognitive support features for software development tools*. PhD thesis, University of California, Irvine, 1999.

- [433] Jason Elliot Robbins and David F. Redmiles. Cognitive support, UML adherence, and XML interchange in Argo/UML. *Information and Software Technology*, 42(2):79–89, January 2000.
- [434] Daniel Robey and M. Lynne Markus. Beyond rigor and relevance: producing consumable research about information systems. *Information Resources Management Journal*, 11(1):7–15, 1998.
- [435] Gregorio Robles, Jesús M. Gonzales-Barahona, and Martin Michlmayr. Evolution of volunteer participation in Libre Software projects: Evidence from Debian. In Scotto and Succi [458], pages 100–107.
- [436] Everett M. Rogers. *Diffusion of Innovations*. Free Press, New York, 5th edition, August 2003.
- [437] Alexander Roßner. Empirisch-qualitative Exploration verschiedener Kontaktstrategien am Beispiel der Einführung von Informationsmanagement in OSS-Projekten. Bachelor thesis, Freie Universität Berlin, May 2007.
- [438] Lawrence Rosen. The unreasonable fear of infection. <http://www.rosenlaw.com/html/GPL.PDF>, 2001. Accessed 2009-11-09.
- [439] Lawrence Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, Upper Saddle River, NJ, USA, July 2004.
- [440] Maria Alessandra Rossi. Decoding the “Free/Open Source (F/OSS) software puzzle” a survey of theoretical and empirical contributions. Quaderni 424, Università degli Studi di Siena, Dipartimento Di Economia Politica, April 2004.
- [441] Michal Przemyslaw Rudzki and Fredrik Jonson. Identifying and analyzing knowledge management aspects of practices in Open Source software development. Master thesis, School of Engineering, Blekinge Institute of Technology, Ronneby, Schweden, August 2004.
- [442] Warren Sack, Françoise Détienne, Nicolas Ducheneaut, Jean-Marie Burkhardt, Dilan Mahendran, and Flore Barcellini. A methodological framework for socio-cognitive analyses of collaborative design of Open Source Software. *Computer Supported Cooperative Work*, 15(2-3):229–250, 2006.
- [443] Stephan Salinger, Laura Plonka, and Lutz Prechelt. A coding scheme development methodology using Grounded Theory for qualitative analysis of Pair Programming. In *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group (PPIG '07)*, pages 144–157, Joensuu, Finland, July 2007. [www.ppig.org](http://www.ppig.org), a polished version appeared in: *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 4(1):9-25, May 2008.
- [444] Stephan Salinger and Lutz Prechelt. What happens during pair programming? In *Proceedings of the 20th Annual Workshop of the Psychology of Programming Interest Group (PPIG '08)*, Lancaster, England, September 2008. [www.ppig.org](http://www.ppig.org).
- [445] Robert J. Sandusky and Les Gasser. Negotiation and the coordination of information and activity in distributed software problem management. In *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 187–196. ACM, 2005.
- [446] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 23–33. IEEE Computer Society, 2009.
- [447] Walt Scacchi. Understanding the requirements for developing open source software systems. *IEE Proceedings – Software*, 149(1):24–39, 2002.
- [448] Walt Scacchi. Free/Open source software development: recent research results and emerging opportunities. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 459–468. ACM, 2007.

- [449] Walt Scacchi. Free/open source software development: Recent research results and methods. In Marvin V. Zelkowitz, editor, *Advances in Computers: Architectural Issues*, volume 69 of *Advances in Computers*, pages 243–295. Academic Press, Boston, 2007.
- [450] Thomas Crombie Schelling. *The strategy of conflict*. Harvard University Press, Cambridge, 1960.
- [451] Roy Schmidt, Kalle Lyytinen, Mark Keil, and Paul Cule. Identifying software project risks: An international delphi study. *Journal of Management Information Systems*, 17(4):5–36, 2001.
- [452] Georg Schreyögg, Jörg Sydow, and Jochen Koch. Organisatorische Pfade — Von der Pfadabhängigkeit zur Pfadkreation. In Georg Schreyögg and Jörg Sydow, editors, *Strategische Prozesse und Pfade*, volume 13 of *Managementforschung*, pages 257–294. Gabler, 2003.
- [453] Joseph A. Schumpeter. *Capitalism, Socialism, and Democracy*. Harper & Row, New York, 1st edition, 1942.
- [454] Robert Schuster. Effizienzsteigerung freier Softwareprojekte durch Informationsmanagement. Studienarbeit, Freie Universität Berlin, September 2005.
- [455] Charles M. Schweik, Robert English, and Sandra Haire. Open Source software collaboration: Foundational concepts and an empirical analysis. National Center for Digital Government Working Paper Series 2, University of Massachusetts Amherst, 2008.
- [456] John Scott. Social Network Analysis. *Sociology*, 22(1):109–127, 1988.
- [457] Peter Scott. Path dependence and Britain’s “coal wagon problem”. *Explorations in Economic History*, 38(3):366–385, July 2001.
- [458] Marco Scotto and Giancarlo Succi, editors. *The First International Conference on Open Source Systems*, Genova, July 2005.
- [459] Raphaël Semeteys, Olivier Pilot, Laurent Baudrillard, Gonéri Le Boudier, and Wolfgang Pinkhardt. Method for qualification and selection of Open Source software (QSOS) version 1.6. Technical report, Atos Origin, April 2006.
- [460] Jr. Sewell, William H. A theory of structure: Duality, agency, and transformation. *The American Journal of Sociology*, 98(1):1–29, July 1992.
- [461] Sonali K. Shah. Motivation, governance, and the viability of hybrid forms in Open Source software development. *Management Science*, 52(7):1000–1014, July 2006.
- [462] Maha Shaikh and Tony Cornford. Version management tools: CVS to BK in the Linux kernel. In Feller et al. [178], pages 127–132.
- [463] John F. Sherry, Jr. Gift giving in anthropological perspective. *The Journal of Consumer Research*, 10(2):157–168, September 1983.
- [464] Dag I.K. Sjøberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Trans. on Software Engineering*, 31(9):733–753, 2005.
- [465] Gregory J. Skulmoski, Francis T. Hartman, and Jennifer Krahn. The Delphi method for graduate research. *Journal of Information Technology Education*, 6:1–21, 2007.
- [466] Slashdot.org. Getting development group to adopt new practices? <http://ask.slashdot.org/article.pl?sid=06/11/15/0028238>, November 2006. Accessed 2010-07-24.
- [467] Diane H. Sonnenwald. Communication roles that support collaboration during the design process. *Design Studies*, 17(3):277–301, July 1996.
- [468] Sebastian Spaeth. Decision-making in Open Source projects. Vorstudie, University of St. Gallen, St. Gallen, Switzerland, April 2003.

- [469] Sebastian Spaeth. *Coordination in Open Source Projects – A Social Network Analysis using CVS data*. Dissertation, Universität St. Gallen, St. Gallen, Switzerland, October 2005.
- [470] Stefan Sperling. Investigation of tree conflict handling in selected version control systems. Bachelor thesis, Freie Universität Berlin, Berlin, 2008.
- [471] SpikeSource, Center for Open Source Investigation at Carnegie Mellon West, and Intel Corporation. Business Readiness Rating for Open Source. Whitepaper RFC 1, OpenBRR.org, 2005.
- [472] Dorit Spiller and Thorsten Wichmann. Free/Libre and Open Source Software: Survey and study – FLOSS – Part 4: Basics of Open Source software markets and business models. Final Report, International Institute of Infonomics University of Maastricht, The Netherlands; Berlecon Research GmbH Berlin, Germany, Berlin, July 2002.
- [473] Andrew M. St. Laurent. *Understanding Open Source and Free Software Licensing*. O'Reilly, 2004.
- [474] Matthew T. Stahl. Open-Source software: not quite endsville. *Drug Discovery Today*, 10(3):219–222, February 2005.
- [475] Richard M. Stallman. The GNU operating system and the Free Software movement. In DiBona et al. [144], pages 53–70.
- [476] Richard M. Stallman. On “Free Hardware”. *Linux Today*. [http://features.linuxtoday.com/news\\_story.php3?ltsn=1999-06-22-005-05-NW-LF](http://features.linuxtoday.com/news_story.php3?ltsn=1999-06-22-005-05-NW-LF), June 1999. Accessed 2009-11-24.
- [477] Richard M. Stallman. Why you shouldn't use the Library GPL for your next library. Usenet: GNU.Announce, January 1999.
- [478] Richard M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. GNU Press, October 2002.
- [479] Richard M. Stallman. Free but shackled - the Java trap. <http://www.gnu.org/philosophy/java-trap.html>, April 2004. Accessed 2007-06-19.
- [480] Richard M. Stallman. Why upgrade to GPLv3. <http://www.gnu.org/licenses/rms-why-gplv3.html>, 2007. Accessed 2010-04-06.
- [481] Richard M. Stallman. The Free Software definition. Version 1.77. <http://www.fsf.org/licensing/essays/free-sw.html#History>, April 2009. Accessed 2009-05-13.
- [482] David Stark. Ambiguous assets for uncertain environments: Heterarchy in postsocialist firms. In Paul DiMaggio, editor, *The Twenty-First-Century Firm: Changing Economic Organization in International Perspective*, pages 69–104. Princeton University Press, 2001.
- [483] Jacqueline Stark. Peer reviews as a quality management technique in Open-Source software development projects. In *ECSQ '02: Proceedings of the 7th International Conference on Software Quality*, pages 340–350. Springer-Verlag, 2002.
- [484] Katherine J. Stewart. OSS project success: from internal dynamics to external impact. In *Collaboration, Conflict and Control: 4th Workshop on Open Source Software Engineering - W8S Workshop - 26th International Conference on Software Engineering*, number 908 in IEE Seminar Digests, pages 92–96. IEE, 2004.
- [485] Katherine J. Stewart, Anthony P. Ammeter, and Likoebe M. Maruping. A preliminary analysis of the influences of licensing and organizational sponsorship on success in Open Source projects. *Hawaii International Conference on System Sciences*, 7:197–207, 2005.
- [486] Katherine J. Stewart and Sanjay Gosain. An exploratory study of ideology and trust in Open Source development groups. In Veda C. Storey, Sumit Sarkar, and Janice I. DeGross, editors, *Proceedings of the International Conference on Information Systems, (ICIS) 2001, December 16-19, New Orleans, Louisiana, USA*, pages 507–512. Association for Information Systems, 2001.

- [487] Katherine J. Stewart and Sanjay Gosain. The impact of ideology on effectiveness in Open Source software development teams. *MIS Quarterly*, 30(2):291–314, June 2006.
- [488] Katherine J. Stewart and Sanjay Gosain. The moderating role of development stage in Free/Open Source software project performance. *Software Process: Improvement and Practice*, 11(2):177–191, 2006.
- [489] Klaas-Jan Stol and Muhammad Ali Babar. Reporting empirical research in Open Source software: The state of practice. In *Open Source Ecosystems: Diverse Communities Interacting*, number 299 in IFIP Advances in Information and Communication Technology, pages 156–169. Springer, Boston, 2009.
- [490] Klaas-Jan Stol, Muhammad Ali Babar, Barbara Russo, and Brian Fitzgerald. The use of empirical methods in Open Source software research: Facts, trends and future directions. In *FLOSS '09: Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 19–24. IEEE Computer Society, 2009.
- [491] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.
- [492] Anselm L. Strauss and Juliet M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. SAGE, 1990.
- [493] Anselm L. Strauss and Juliet M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 2nd edition, September 1998.
- [494] Matthias Stürmer. Open Source community building. Licentiate, University of Bern, 2005.
- [495] B. Stvilia, M. B. Twidale, L. Gasser, and L. C. Smith. Information quality discussions in Wikipedia. Technical Report ISRN UIUCLIS–2005/2+CSCW., University of Illinois at Urbana-Champaign, 2005.
- [496] Roy Suddaby. From the editors: What Grounded Theory is not. *Academy of Management Journal*, 49(4):633–642, August 2006.
- [497] James Surowiecki. *The wisdom of crowds: why the many are smarter than the few and how collective wisdom shapes business, economies, societies, and nations*. Doubleday, New York, 2004.
- [498] Gerald I. Susman and Roger D. Evered. An assessment of the scientific merits of action research. *Administrative Science Quarterly*, 23(4):582–603, December 1978.
- [499] Jörg Sydow, Arnold Windeler, Guido Möllering, and Cornelius Schubert. Path-creating networks: The role of consortia in processes of path extension and creation. In *21st EGOS Colloquium, Berlin, Germany*, 2005.
- [500] Andrew Tannenbaum, Linus Torvalds, and many others. The Tanenbaum-Torvalds debate. In DiBona et al. [144], chapter Appendix A, pages 221–251.
- [501] Florian Thiel. Process innovations for security vulnerability prevention in Open Source web applications. Diplomarbeit, Institut für Informatik, Freie Universität Berlin, Germany, April 2009.
- [502] Craig Thomas. Improving verification, validation, and test of the Linux kernel: the Linux Stabilization Project. In Feller et al. [178], pages 133–136.
- [503] Nicholas Thomas. *Entangled objects: exchange, material culture and colonialism in the pacific*. Harvard University Press, Cambridge, MA, USA, 1991.
- [504] Michael Tieman. How Linux will revolutionize the embedded market. Linux Devices. <http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/>

- Tiemann-How-Linux-will-Revolutionize-the-Embedded-Market/, May 2002. Accessed 2009-11-13.
- [505] Sigurd Tjøstheim and Morten Tokle. Acceptance of new developers in OSS projects. Master's thesis, Norwegian University of Science and Technology, 2003. [http://www.idi.ntnu.no/grupper/su/su-diploma-2003/Tjostheim\\_FTokleaFOSS\\_acceptance.pdf](http://www.idi.ntnu.no/grupper/su/su-diploma-2003/Tjostheim_FTokleaFOSS_acceptance.pdf). Accessed 2006-11-17.
  - [506] Bruce Tognazzini. *Tog on interface*. Addison-Wesley, Reading, Mass., 1992.
  - [507] Linus Torvalds. The Linux edge. *Communications of the ACM*, 42(4):38–39, April 1999.
  - [508] Linus Torvalds and David Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperCollins, May 2001.
  - [509] Harrison Miller Trice and Janice M. Beyer. *The cultures of work organizations*. Prentice Hall Englewood Cliffs, NJ, 1993.
  - [510] Masateru Tsunoda, Akito Monden, Takeshi Kakimoto, Yasutaka Kamei, and Ken-ichi Matsumoto. Analyzing OSS developers' working time using mailing lists archives. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 181–182. ACM, 2006.
  - [511] Ilkka Tuomi. Internet, innovation, and Open Source: Actors in the network. *First Monday*, 6(1):1, January 2001.
  - [512] Stephen S. Turnbull. XEmacs vs. GNU Emacs. <http://www.xemacs.org/About/XEmacsVsGNUemacs.html>, January 2001. Accessed 2009-08-22.
  - [513] Murray Turoff and Starr Roxanne Hiltz. Computer-based Delphi processes. In Michael Adler and Erio Ziglio, editors, *Gazing into the oracle: the Delphi method and its application to social policy and public health*, chapter 3, pages 56–85. Jessica Kingsley Publishers, London, 1996.
  - [514] Michael B. Twidale and David M. Nichols. Exploring usability discussions in Open Source development. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, page 198.3. IEEE Computer Society, 2005.
  - [515] Joshua R. Tyler and John C. Tang. When can i expect an email response? a study of rhythms in email usage. In *ECSCW'03: Proceedings of the Eighth Conference on European Conference on Computer Supported Cooperative Work, Helsinki, Finland*, pages 239–258. Kluwer Academic Publishers, 2003.
  - [516] University of California, Berkeley. Berkeley Software Distribution (BSD) license. <http://www.opensource.org/licenses/bsd-license.php>, July 1999. Accessed 2009-10-26.
  - [517] Vinod Valloppillil. Halloween Document 1: Open Source software—a (new?) development methodology. Microsoft internal strategy memorandum leaked to Eric S. Raymond. <http://catb.org/~esr/halloween/halloween1.html>, August 1999. Accessed 2009-11-28.
  - [518] Frank van der Linden, Bjorn Lundell, and Pentti Marttiin. Commodification of industrial software: A case for Open Source. *IEEE Software*, 26(4):77–83, July 2009.
  - [519] Marian van der Meulen, Robert H. Logie, Yvonne Freer, Cindy Sykes, Neil McIntosh, and Jim Hunter. When a graph is poorer than 100 words: A comparison of computerised natural language generation, human generated descriptions and graphical displays in neonatal intensive care. *Applied Cognitive Psychology*, Early View, December 2008.
  - [520] Guido van Rossum. Python reference manual. Technical Report CS-R9525, Department of Algorithmics and Architecture, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, NL, May 1995.
  - [521] Mark Van Vugt, Robert Hogan, and Robert B. Kaiser. Leadership, followership, and evolution: some lessons from the past. *American Psychologist*, 63(3):182–196, April 2008.

- [522] Kris Ven, Jan Verelst, and Herwig Mannaert. Should you adopt Open Source software? *IEEE Software*, 25(3):54–59, 2008.
- [523] Gina Danielle Venolia and Carman Neustaedter. Understanding sequence and reply relationships within email conversations: a mixed-model visualization. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 361–368. ACM, 2003.
- [524] Fernanda B. Viégas, Scott Golder, and Judith Donath. Visualizing email content: portraying relationships from conversational histories. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 979–988. ACM, 2006.
- [525] Fernanda B. Viégas and Marc Smith. Newsgroup crowds and AuthorLines: Visualizing the activity of individuals in conversational cyberspaces. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 4*, page 40109.2. IEEE Computer Society, 2004.
- [526] Paul Vixie. Software engineering. In DiBona et al. [144], chapter 7, pages 91–100.
- [527] Mikko Välimäki. Dual licensing in Open Source software industry. *Systemes d'Information et Management*, 8(1):63–75, 2003.
- [528] Jana von dem Berge. Zentrale und dezentrale Versionsverwaltungswerkzeuge in Open Source Projekten. Diplomarbeit, Institut für Informatik, Freie Universität Berlin, Germany, April 2009.
- [529] William von Hagen. *Ubuntu Linux Bible*. Wiley, New York, NY, USA, 1st edition, January 2007.
- [530] Eric von Hippel. Innovation by user communities: Learning from Open-Source software. *MIT Sloan Management Review*, 42(4):82–86, January 2001.
- [531] Eric von Hippel and Karim Lakhani. Telephone interview with Stephen Blackheath (developer of the Freenet Project), December 2000.
- [532] Eric von Hippel and Georg von Krogh. Open Source software and the "private-collective" innovation model: Issues for organization science. *Organization Science*, 14(2):209–223, 2003.
- [533] Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. Community, joining, and specialization in Open Source Software innovation: A case study. *Research Policy*, 32:1217–1241(25), July 2003.
- [534] Taowei David Wang, Catherine Plaisant, Alexander J. Quinn, Roman Stanchak, Shawn Murphy, and Ben Shneiderman. Aligning temporal data by sentinel events: Discovering patterns in electronic health records. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 457–466. ACM, 2008.
- [535] Barry Warsaw, Jeremy Hylton, and David Goodger. PEP purpose and guidelines. PEP 1. <http://www.python.org/dev/peps/pep-0001/>, June 2000. Accessed 2009-11-28.
- [536] Anthony Wasserman, Murugan Pal, and Christopher Chan. The Business Readiness Rating model: an evaluation framework for Open Source. In *Workshop on Evaluation Framework for Open Source Software (EFOSS)*, June 2006.
- [537] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.
- [538] Steven Weber. The political economy of Open Source software. UCAIS Berkeley Roundtable on the International Economy, Working Paper Series 1011, UCAIS Berkeley Roundtable on the International Economy, UC Berkeley, June 2000.
- [539] Mark Weinem. pkgsrc and the concepts of package management 1997–2007. <http://www.netbsd.org/gallery/pkgsrc-interviews.html>, October 2007. Accessed 2009-10-26.



- [540] Dawid Weiss. Quantitative analysis of Open Source projects on SourceForge. In Marco Scotto and Giancarlo Succi, editors, *First International Conference on Open Source Systems (OSS 2005)*, Genova, Italy, pages 140–147, 2005.
- [541] Etienne Wenger. *Communities of Practice: Learning, Meaning, and Identity*. Cambridge University Press, December 1999.
- [542] Joel West. How open is open enough?: Melding proprietary and Open Source platform strategies. *Research Policy*, 32(7):1259–1285, July 2003.
- [543] Joel West and Siobhán O'Mahony. Contrasting community building in sponsored and community founded Open Source projects. In *38th Annual Hawaii International Conference on System Sciences*, volume 7, page 196c. IEEE Computer Society, 2005.
- [544] Joel West and Siobhán O'Mahony. The role of participation architecture in growing sponsored Open Source communities. *Industry & Innovation*, 15(2):145–168, April 2008.
- [545] S. J. Westerman. Individual differences in the use of command line and menu computer interfaces. *International Journal of Human-Computer Interaction*, 9(2):183–198, 1997.
- [546] David A. Wheeler. Why Open Source software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the numbers! [http://www.dwheeler.com/oss\\_fs\\_why.html](http://www.dwheeler.com/oss_fs_why.html), April 2007. Accessed 2010-04-22.
- [547] James A. Whittaker. What is software testing? And why is it so hard? *IEEE Software*, 17(1):70–79, 2000.
- [548] Andrea Wiggins, James Howison, and Kevin Crowston. Heartbeat: Measuring active user base and potential user interest in FLOSS projects. In *Open Source Ecosystems: Diverse Communities Interacting*, volume 299/2009 of *IFIP Advances in Information and Communication Technology*, pages 94–104. Springer, Boston, 2009.
- [549] Arnold Windeler. *Unternehmensnetzwerke: Konstitution und Strukturierung*. VS Verlag, 2001.
- [550] Ulrich Witt. “Lock-in” vs. “critical masses” – industrial change under network externalities. *International Journal of Industrial Organization*, 15(6):753–773, October 1997.
- [551] Ludwig Wittgenstein. *Tractatus logico-philosophicus*. Routledge and Kegan Paul, London, revised edition, 1974.
- [552] Chong-Guang Wu, James H. Gerlach, and Clifford E. Young. An empirical analysis of Open Source software developers' motivations and continuance intentions. *Information & Management*, 44(3):253–262, April 2007.
- [553] Jin Xu, Yongqin Gao, Scott Christley, and Gregory Madey. A topological analysis of the Open Source software development community. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, page 198ff. IEEE Computer Society, 2005.
- [554] Yutaka Yamauchi, Makoto Yokozawa, Takeshi Shinohara, and Toru Ishida. Collaboration with lean media: how open-source software succeeds. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 329–338. ACM, 2000.
- [555] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of Open Source Software developers. In *Proceedings of the of the 25th International Conference on Software-Engineering (Portland, Oregon)*, 2003.
- [556] Ka-Ping Yee. Zest: Discussion mapping for mailing lists. CSCW 2002 (demo) <http://zesty.ca/pubs/cscw-2002-zest.pdf>, 2002. Accessed 2009-11-26.
- [557] Robert K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. Sage Publications, Inc., 1st edition, 1989.

- [558] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.
- [559] Jamie W. Zawinski. Nomo zilla – resignation and postmortem. <http://www.jwz.org/gruntle/nomo.html>, March 1999. Accessed 2009-11-20.
- [560] Jamie W. Zawinski. The Lemacs/FSFmacs schism. <http://www.jwz.org/doc/lemacs.html>, February 2000. Accessed 2009-08-22.
- [561] Luyin Zhao and Sebastian Elbaum. Quality assurance under the Open Source development model. *Journal of Systems and Software*, 66(1):65–75, 2003.
- [562] Minghui Zhou, Audris Mockus, and David Weiss. Learning in offshore and legacy software projects: How product structure shapes organization. In *Proceedings of the ICSE Workshop on Socio-Technical Congruence, Vancouver, Canada, May 19th, 2009*.
- [563] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

## Additional Non-e-mail Documents

Occasionally in this thesis, data sources other than e-mails such as documents from the projects' websites were used. Since these could not be linked using hyperlinks into Gmane, these primary documents are given in bibliographic format below.

- [564] Jeremy Andrews. Interview: Avi Kivity. Kernel Trap <http://kerneltrap.org/node/8088>, April 2007. Accessed 2009-10-25.
- [565] Best Practical Solutions LLC. About Best Practical. <http://bestpractical.com/about/us.html>, 2008. Accessed 2009-10-26.
- [566] Raphael Bossek, Wolfgang Denk, and The PPCBoot Project. Embedded PowerPC Linux Boot Project. <http://ppcboot.sourceforge.net>, October 2002. Accessed 2009-10-26.
- [567] Wolfgang Denk. U-Bootdoc: 1.2. history. <http://www.denx.de/wiki/view/U-Bootdoc/History>. Revision 1.4, June 2004. Accessed 2009-10-26.
- [568] Gael Duval. MandrakeSoft buys Bochs for Linux and commits it to Open Source. <http://lwn.net/2000/0323/a/mand-bochs.html>, March 2000. Accessed 2009-10-19.
- [569] Radu-Cristian Fotescu. Google Summer of Code, Xfce and the lack of a vision. <http://beranger.org/index.php?page=diary&2008/03/18/21/38/34-google-summer-of-code-xfce-and-t>, March 2008. Accessed 2009-04-23.
- [570] Free Software Foundation, Inc. 1.2 History of GRUB. [http://www.gnu.org/software/grub/manual/html\\_node/History.html](http://www.gnu.org/software/grub/manual/html_node/History.html), 2006. Accessed 2009-10-25.
- [571] Jim Hall and the FreeDOS project. About us. <http://www.freedos.org/freedos/about/>. Accessed 2009-10-25.
- [572] Thomas Leonard. Re: ROX Desktop. comp.sys.acorn.misc, <http://groups.google.co.uk/group/comp.sys.acorn.misc/msg/bdb27a8da23af4e6>, October 1999. Accessed 2009-10-27.
- [573] Linux Devices. CEO interview: Wolfgang Denk of Denx Software Engineering. <http://www.linuxfordevices.com/c/a/News/CEO-Interview-Wolfgang-Denk-of-Denx-Software-Engineering/>, April 2004. Accessed 2009-10-26.
- [574] Phil Manchester. Reality crashes Google hippie code fest. The Register. [http://www.theregister.co.uk/2008/03/27/google\\_summer\\_code\\_debian\\_losers/print.html](http://www.theregister.co.uk/2008/03/27/google_summer_code_debian_losers/print.html), March 2008. Accessed 2009-04-21.
- [575] Tom Morris and The ArgoUML Project. Frequently asked questions for ArgoUML. <http://argouml.tigris.org/faqs/users.html>, November 2008. Accessed 2009-10-24.
- [576] Volker Ruppert and The Bochs Project. Bochs News. <http://bochs.sourceforge.net/news.html>, April 2009. Accessed 2009-10-24.

- [577] The ArgoUML Project. ArgoUML in Google's Summer of Code 2007. <http://argouml.tigris.org/googlessoc2007.html>, March 2007. Accessed 2009-04-24.
- [578] The ArgoUML Project. Code samples from students working with ArgoUML for Google Summer of Code 2007. <http://code.google.com/p/google-summer-of-code-2007-argouml/>, September 2007. Accessed 2009-04-21.
- [579] The ArgoUML Project. ArgoUML history. <http://argouml.tigris.org/history.html>, November 2008. Accessed 2009-04-21.
- [580] The ArgoUML Project. Google Summer of Code 2008 ideas. <http://wiki.xfce.org/gsoc/2008/ideas>, March 2008. Accessed 2009-04-21.
- [581] The Bugzilla Project. Brief history. <http://www.bugzilla.org/status/roadmap.html#history>, March 2009. Accessed 2009-10-25.
- [582] The Bugzilla Project. Current developers. <http://www.bugzilla.org/developers/profiles.html>, March 2009. Accessed 2009-10-25.
- [583] The Flyspray Project. Flyspray development team. <http://flyspray.org/team>, March 2009. Accessed 2009-10-25.
- [584] The gEDA Project. gEDA FAQ. <http://www.geda.seul.org/wiki/geda:faq>, February 2009. Accessed 2009-10-25.
- [585] The GNU Project. GNU guidelines for Summer of Code projects. <http://www.gnu.org/software/soc-projects/guidelines.html>, February 2009. Accessed 2009-04-21.
- [586] The MonetDB Project. Contributors — hall of fame. <http://monetdb.cwi.nl/projects/monetdb/Development/Credits/Contributors/index.html>. Accessed 2009-10-25.
- [587] The MonetDB Project. Credits — powered by MonetDB. <http://monetdb.cwi.nl/projects/monetdb/Development/Credits/index.html>. Accessed 2009-10-25.
- [588] The MonetDB Project. Synopsis of MonetDB 4.4 and MonetDB/SQL 2.4 releases. <http://monetdb.cwi.nl/projects/monetdb/Development/Releases/Version4.4/index.html>, October 2004. Accessed 2009-10-25.
- [589] The Perl Review. Interview with Jesse Vincent. <http://www.theperlreview.com/Interviews/jesse-vincent-200708.html>, August 2007. Accessed 2009-10-26.