

Debugging im Informatikunterricht

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)
eingereicht am
Fachbereich Mathematik und Informatik
der Freien Universität Berlin

vorgelegt von
Tilman Michaeli

Berlin, im August 2020

Erstgutachter: Prof. Dr. Ralf Romeike, Freie Universität Berlin
Zweitgutachterin: Prof. Dr. Yasmin Kafai, University of Pennsylvania

Tag der Disputation: 3.12.2020

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel und Hilfen verfasst habe. Weiterhin versichere ich, dass diese Arbeit nicht bereits in einem früheren Promotionsverfahren eingereicht wurde.

Berlin, 07.01.2021 *Tilman Michaeli*

Zusammenfassung

Schülerinnen und Schüler durch die Vermittlung von Programmierfähigkeiten zur aktiven und kreativen Gestaltung der sogenannten „digitalen Welt“ zu befähigen, ist zentrale Aufgabe des Informatikunterrichts. Dazu gehört neben dem Erlernen algorithmischer Konzepte und deren Umsetzung in einer konkreten Programmiersprache insbesondere auch der Umgang mit Programmierfehlern. Dabei nehmen solche „Bugs“ in der Informatik einen besonderen Stellenwert ein: In der professionellen Softwareentwicklung sind sie unvermeidbar und können, wenn sie unentdeckt bleiben, zu Programmabstürzen oder dem Versagen kritischer Infrastrukturen führen. Daher wenden Entwicklerinnen und Entwickler zwischen 20 und 40 Prozent ihrer Arbeitszeit dafür auf, „Bugs“ zu finden und zu beheben, also zu *debuggen*.

Im Informatikunterricht stellt diese zentrale Tätigkeit im Kontext der Programmierung für Schülerinnen und Schüler nicht nur ein erhebliches Hindernis beim Programmierenlernen dar, sondern ist zusätzlich eine große Quelle für Frustration. Aber auch die Lehrkräfte stehen vor der enormen Herausforderung, allen Schülerinnen und Schülern gleichzeitig gerecht zu werden. Oftmals eilen sie von PC zu PC und versuchen, alle Schülerinnen und Schüler individuell zu unterstützen – ein Phänomen, das umgangssprachlich auch als Turnschuhdidaktik bezeichnet wird. Dabei fehlt es Lehrkräften an Konzepten, Materialien und Best Practices für den Unterricht, um die Schülerinnen und Schüler zu einem selbstständigen Umgang mit Fehlern zu befähigen. In der Schulpraxis sind Lernende folglich oftmals gezwungen, sich selbst geeignete Strategien und Vorgehensweisen anzueignen. Erfahrungsgemäß stellt dies für einen Großteil der Schülerinnen und Schüler eine kaum zu bewältigende Herausforderung dar und kann als Schlüsselproblem des Informatikunterrichts angesehen werden. Auch in der informatikdidaktischen Forschung ist bisher nicht untersucht worden, wie Debugging zielführend in die unterrichtliche Praxis integriert werden kann.

Daher wird in dieser Arbeit gemäß dem Forschungsformat der didaktischen Rekonstruktion der Prozess Debugging aus informatikdidaktischer Sicht aufgearbeitet, um Konzepte und Materialien für den Informatikunterricht zu entwickeln und damit das Schlüsselproblem Debugging im Unterricht zu adressieren. Dazu werden zunächst im Zuge der *fachlichen Klärung* die Fähigkeiten identifiziert, die Schülerinnen und Schülern für selbstständiges Debugging benötigen. Weiterhin wird die *Perspektive der Lehrkräfte* untersucht und analysiert, mit welchen Herausforderungen die Lehrkräfte beim Debuggen im Unterricht konfrontiert sind und wie sie mit diesen umgehen. Darüber hinaus werden *gesellschaftliche Ansprüche* und der Beitrag zur Allgemeinbildung von Debugging als einer zentralen Herangehensweise des Computational Thinking untersucht. Daneben werden in der *Perspektive der Lernenden* Debugging-Lernvoraussetzungen von Schülerinnen und Schülern erhoben, die deren Debuggingvorgehen beeinflussen. Auf dieser Basis werden Gestaltungskriterien für Konzepte und Materialien für das Debugging im Unterricht entwickelt sowie ein konkretes integratives Unterrichtskonzept für den Informatikunterricht entworfen. Abschließend wird die Wirksamkeit der expliziten Vermittlung von Debugging anhand des Unterrichtskonzepts empirisch validiert und für eine tatsächliche Praxiswirksamkeit der Ergebnisse der Transfer in die Schulpraxis untersucht.

Damit wird in dieser Arbeit sowohl die theoretische Grundlage für die Vermittlung von Debugging im Unterricht gelegt als auch ein konkretes Unterrichtskonzept entwickelt und evaluiert und somit ein Beitrag zu einem Bereich geleistet, der in der informatikdidaktischen Forschung trotz seiner Bedeutung bisher weitestgehend vernachlässigt wurde. Auf Grundlage der Ergebnisse wird das Schlüsselproblem Debugging im Informatikunterricht erfolgreich adressiert und damit dazu beigetragen, Schülerinnen und Schüler zur aktiven und kreativen Gestaltung der „digitalen Welt“ zu befähigen.

Vorabveröffentlichung von Teilen dieser Arbeit

Teile dieser Arbeit wurden bereits vor Einreichung der vorliegenden Dissertation wörtlich oder sinngemäß veröffentlicht. Die entsprechenden Referenzen sind in der untenstehenden Tabelle dem jeweiligen Abschnitt der Arbeit zugeordnet.

Kapitel	vorab veröffentlicht
Kapitel 3	Michaeli, Tilman und Romeike, Ralf (2019a). „Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study“. In: <i>2019 IEEE Global Engineering Education Conference (EDUCON)</i> . Dubai, VAE: IEEE, S. 1030–1038
Kapitel 4	Michaeli, Tilman und Romeike, Ralf (2019a). „Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study“. In: <i>2019 IEEE Global Engineering Education Conference (EDUCON)</i> . Dubai, VAE: IEEE, S. 1030–1038
Kapitel 8	Michaeli, Tilman und Romeike, Ralf (2019c). „Improving Debugging Skills in the Classroom: The Effects of Teaching a Systematic Debugging Process“. In: <i>Proceedings of the 14th Workshop in Primary and Secondary Computing Education</i> . New York, NY, USA: ACM, S. 1–7 Michaeli, Tilman und Romeike, Ralf (2019b). „Debuggen im Unterricht—Ein systematisches Vorgehen macht den Unterschied“. In: <i>INFOS 2019, 18. GI-Fachtagung Informatik und Schule</i> . Bonn: Gesellschaft für Informatik, S. 129–138

Inhaltsverzeichnis

1	Einleitung und forschungsmethodische Einordnung	3
1.1	Motivation und Problemaufriss	3
1.2	Forschungslücke und Ziel	5
1.3	Forschungsmethodische Einordnung	6
1.3.1	Das Modell der didaktischen Rekonstruktion	7
1.3.2	Die didaktische Rekonstruktion in der Informatikdidaktik	10
1.3.3	Forschungsfragen	14
1.4	Struktur der Arbeit	14

Teil I: Debugging als Thema informatikdidaktischer Forschung

2	Forschungsstand	19
2.1	Debuggen, Testen und Programmieren	20
2.2	Debuggingprozess	22
2.2.1	Experten	23
2.2.2	Novizen	29
2.2.3	Experten vs. Novizen	36
2.2.4	Eigene vs. fremde Programme	38
2.2.5	Implikationen	39
2.3	Fehler	41
2.3.1	Fehlerarten	42
2.3.2	Bedeutung für Novizen	44
2.3.3	Implikationen	45
2.4	Lernende	46
2.4.1	Affektive Einstellungen der Lernenden	46
2.4.2	Lernvoraussetzungen	48
2.4.3	Implikationen	49
2.5	Unterricht	50
2.5.1	Debuggen in der Hochschullehre	50
2.5.2	Debuggen im Informatikunterricht	53

2.5.3	Werkzeugbasierte Ansätze	54
2.5.4	Implikationen	56
2.6	Transfer	56
2.7	Zusammenfassung und Fazit	59

Teil II: Zugrunde liegende Perspektiven

3	Fachliche Klärung	65
3.1	Ziele der Untersuchung	65
3.2	Methodik	66
3.3	Ergebnisse	68
3.3.1	Anwenden eines systematischen Debuggingvorgehens	68
3.3.2	Anwenden von Debuggingstrategien	70
3.3.3	Anwendung von Heuristiken und Mustern für typische Fehler	72
3.3.4	Verwenden von Werkzeugen	72
3.4	Interpretation	72
3.5	Fazit	77
4	Perspektive der Lehrkräfte	79
4.1	Ziele der Untersuchung	79
4.2	Methodik	80
4.3	Ergebnisse	81
4.3.1	Wie gehen Schülerinnen und Schüler sowie Lehrkräfte mit Programmierfehlern im Klassenzimmer um? (RQ2.1)	81
4.3.2	Wie wird Debugging im Unterricht vermittelt? (RQ2.2)	86
4.3.3	Aus welchen Gründen wird Debugging (nicht) zum Unterrichtsthema? (RQ2.3)	89
4.4	Interpretation	90
4.5	Fazit	94
5	Klärung gesellschaftlicher Ansprüche	97
5.1	Ziel	97
5.2	Debugging und Allgemeinbildung	98
5.3	Computational Thinking und Debugging	99
5.3.1	Problemlösen und Troubleshooten	99
5.3.2	Troubleshootingprozess	101
5.3.3	Troubleshootingfähigkeiten	102
5.3.4	Troubleshooting und Debugging	104
5.4	Fazit	106

6	Perspektive der Lernenden	107
6.1	Ziele der Untersuchung	107
6.2	Spezifischer Forschungsstand: Escape-Rooms als Unterrichts- bzw. wissenschaftliche Untersuchungsmethode	108
6.3	Erhebungsinstrument Escape-Room	110
6.3.1	Entwicklung	110
6.3.2	Resultierende Aufgaben	114
6.4	Methodik	116
6.5	Ergebnisse	120
6.5.1	Monitor	120
6.5.2	Kabelsalat	121
6.5.3	Telefon abhören	123
6.5.4	Tal der Könige	124
6.5.5	Mr. X	125
6.6	Interpretation	126
6.6.1	Generalisierung und Diskussion der Ergebnisse	126
6.6.2	Einordnung in der Forschungsstand	130
6.6.3	Limitationen und Güte	130
6.7	Fazit	132

Teil III: Didaktische Strukturierung

7	Entwicklung von Konzepten und Materialien für den Informatikunterricht	137
7.1	Synthese: Gestaltungskriterien	137
7.2	Exemplarische Umsetzung	141
7.2.1	Systematisches Vorgehen	141
7.2.2	Strategien und Werkzeuge	142
7.2.3	Heuristiken und Muster für typische Fehler	143
7.3	Fazit	144

Teil IV: Implementierung im Informatikunterricht

8	Der Effekt der expliziten Vermittlung von Debugging	151
8.1	Ziele der Untersuchung	151
8.2	Methodik	152
8.3	Ergebnisse	154

8.3.1	Hat die Vermittlung eines systematischen Vorgehens einen positiven Effekt auf die Selbstwirksamkeitserwartungen der Schülerinnen und Schüler? (RQ6.1)	154
8.3.2	Hat die Vermittlung eines systematischen Vorgehens einen positiven Effekt auf die Debuggingleistung der Schülerinnen und Schüler? (RQ6.2)	155
8.4	Interpretation	157
8.5	Fazit	159
9	Gestaltung und Evaluation einer Fortbildung zum Transfer in die Unterrichtspraxis	161
9.1	Fortbildung, professionelle Kompetenz und Professionswissen	161
9.2	Entwicklung einer Fortbildung für das Debugging im Unterricht	163
9.2.1	Professionelle Kompetenz und Professionswissen der Lehrkräfte für das Debugging	164
9.2.2	Konzeption	166
9.2.3	Durchführung	167
9.3	Transfer in die Praxis	170
9.3.1	Ziele der Untersuchung	170
9.3.2	Methodik	171
9.3.3	Ergebnisse	172
9.3.4	Interpretation	176
9.4	Fazit	177

Teil V: Abschluss

10	Fazit	181
10.1	Zusammenfassung	181
10.2	Beitrag und Ausblick	185

Verzeichnisse

Literaturverzeichnis	191
Abbildungsverzeichnis	207
Tabellenverzeichnis	209

Anhang

A Interviewleitfaden Perspektive der Lehrkräfte	213
B Unterrichtsmaterialien	215
C Poster „Debuggen leicht gemacht“	227
D Fragenbogen Schülerinnen und Schüler	229
E Personas	231
F Interviewleitfaden Fortbildungsevaluation	233

Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundreds of nitty frustrations drive away all but those who habitually focus on the end goal. Perhaps it is merely that computers are young, programmers are younger, and the young are always optimists. But however the selection process works, the result is indisputable: „This time it will surely run“ or „I just found the last bug.“

FRED BROOKS JR., THE MYTHICAL MAN-MONTH.

1 Einleitung und forschungsmethodische Einordnung

1.1 Motivation und Problemaufriss

Programmieren zu vermitteln – und damit eine wesentliche Grundlage für die aktive und kreative Gestaltung der sogenannten „digitalen Welt“ – ist zentrale Aufgabe des Informatikunterrichts. Oftmals stellt dies eine große Herausforderung dar. So müssen die Schülerinnen und Schüler nicht nur Algorithmik, Programmierkonzepte und deren Umsetzung in der gewählten Programmiersprache erlernen, sondern insbesondere auch dazu befähigt werden, Lösungen zu finden, wenn sie mit Programmierfehlern konfrontiert sind. Dafür müssen sie Vorgehensweisen entwickeln, die ihnen erlauben, Fehler zu finden und zu beheben.

Betrachtet man die Welt der professionellen Softwareentwicklung, stellt das systematische Finden und Beheben von Fehlern eine zentrale Kompetenz dar: Zwischen 20 und 40 Prozent ihrer Arbeitszeit wenden professionelle Entwicklerinnen und Entwickler dafür auf (Perscheid et al., 2017). Programmierfehler sind in der Praxis unvermeidbar und täglicher Begleiter. Auch deshalb pflegen Informatikerinnen und Informatiker einen besonderen Umgang mit (Programmier-)Fehlern. Diese sind oftmals von einer beinahe mystischen Aura umgeben, deren Bedeutung nur von anderen leidgeprüften Eingeweihten nachvollzogen werden kann. Das begründet sich schon im Mythos der Herkunftsgeschichte des – wiederum spezifisch informatischen – Begriffes des *Debugging*¹: 1947 testeten Grace Hopper und ihre Kolleginnen und Kollegen an der Harvard University in Cambridge den Rechner Mark II. Dabei sorgte eine Motte dafür, dass eines der Relais ausfiel. Nachdem ein Techniker die tote Motte gefunden hatte, klebte Grace Hopper sie in das Logbuch, versehen mit der Notiz „*first actual case of a bug being found*“. Ausgehend hiervon verbreitete sich der Begriff des *Debuggens*, den Hopper selbst wie folgt definierte: „*to remove a malfunction from a computer or an error from a routine*“ (Hopper, 1954).

Die Informatik prägt dabei nicht nur einen eigenen Begriff für Fehler und das Vorgehen, diese zu finden und zu beheben, der besondere Stellenwert von Fehlern zeigt sich auch anderweitig: Wichtiger Bestandteil der Infrastruktur eines jeden Softwareprojekts ist ein *bug tracker*, in dem Fehler festgehalten werden können – und oftmals auch von den Nutzerinnen und Nutzern der Software Bugs gemeldet werden können. Unternehmen schreiben gar *bug bounties* aus, um Personen finanziell zu belohnen, die gefundene Fehler melden. Und

¹Tatsächlich wurde der Begriff „Bug“ bereits Ende des 19. Jahrhunderts für die Bezeichnung von Fehlern und Defekten in Maschinen genutzt, vgl. Shapiro (1987).

keine andere Disziplin hat mit einer Plattform wie Stackoverflow.com eine ähnlich etablierte Anlaufstelle, die bei Fehlern Unterstützung bietet und damit als öffentliche Sammlung von Lösungen für gängige Bugs dient – insbesondere für professionelle Entwicklerinnen und Entwickler (*Stackoverflow, 2019*). Darüber hinaus pflegen einige professionelle Entwicklerinnen und Entwickler ein *developer log* – eine persönliche Sammlung von Fehlern und deren Lösung (*Perscheid et al., 2017*).

Aufgrund der Unvermeidbarkeit von Programmierfehlern nimmt der Umgang mit diesen in der professionellen Softwareentwicklung also einen großen und bis zu einem gewissen Grad einzigartigen Stellenwert ein. Dabei steht gerade das *Lernen aus vergangenen Fehlern* im Vordergrund, und Fehler werden damit eben auch als positiv konnotierte Chance zur Verbesserung des Produkts (z.B. durch *bug bounties*) oder als Lernanlass (z.B. durch *developer logs*) betrachtet. Doch wie gestaltet sich die Situation im Klassenzimmer?

Ziel des allgemeinbildenden Informatikunterrichts ist es natürlich nicht, professionelle Softwareentwicklerinnen und -entwickler aus- oder professionelle Softwareentwicklung nachzubilden. Allerdings machen Programmieranfängerinnen und -anfänger mehr Programmierfehler und verwenden ähnlich viel Zeit auf das Debuggen wie Experten (*Allwood und Björhag, 1990*). Die Behebung dieser Fehler stellt nicht nur ein erhebliches Hindernis beim Programmierenlernen dar (*Lahtinen, Ala-Mutka und Järvinen, 2005*), sondern ist zusätzlich eine große Quelle für Frustration (*Perkins et al., 1986*). Dies steht in einem starken Widerspruch zu einer Betrachtung von Fehlern als (positiv konnotierter) Lernanlass.

Dementsprechend stellt Debugging im Unterricht auch für die Lehrkräfte eine große Herausforderung dar: Oftmals eilen sie von PC zu PC und versuchen, allen Schülerinnen und Schülern möglichst gerecht zu werden – ein Phänomen, das umgangssprachlich auch als *Turnschuhdidaktik* bezeichnet wird. Einerseits sollen die Schülerinnen und Schüler nicht alleingelassen werden oder sich vernachlässigt fühlen, andererseits sollen sie eben auch selbstständig versuchen, ihr Programm zu debuggen, und nicht nur die Lösung von der Lehrkraft diktiert oder gar eingetippt bekommen. Gleichzeitig fehlt es aber auch an erprobten Konzepten, Best Practices und Formaten für das Thema Debugging im Informatikunterricht, auf die Lehrkräfte zurückgreifen könnten. Zudem haben Lehrkräfte – genauso wie professionelle Softwareentwicklerinnen und Entwickler (*Perscheid et al., 2017*) – Debugging oftmals selbst nicht systematisch gelernt. In der Schulpraxis werden Lernende daher häufig mit ihren Fehlern alleingelassen und sind folglich gezwungen, sich selbst geeignete Strategien und Vorgehensweisen anzueignen. Erfahrungsgemäß stellt dies für einen Großteil der Schülerinnen und Schüler eine kaum zu bewältigende Herausforderung dar (*Carver und Klahr, 1986*) und kann als Schlüsselproblem des Informatikunterrichts angesehen werden.

²In Ermangelung einer adäquaten Übersetzung des Terminus *troubleshooting* wird in dieser Arbeit durchgängig der englische Begriff verwendet.

Debugging spielt jedoch nicht nur in der Programmierung eine große Rolle, sondern ist auch in unserem Alltag allgegenwärtig: Funktioniert etwa „das Internet“ oder unser Fahrrad nicht mehr, versuchen wir den Fehler zu finden und zu beheben – wir *troubleshooten*². Debugging beschreibt Troubleshooting in der Domäne der Programmierung (*Katz und Anderson, 1987*) und ist damit ein Spezialfall allgemeinen Problemlösens (*Jonassen, 2000*). Bei der Fehlersuche im Alltag sind dabei ähnliche Fähigkeiten und Strategien wie in der Programmierung involviert (*Li et al., 2019*). Debugging stellt eine zentrale Herangehensweise des Computational Thinkings dar, einer von *Wing (2006)* geprägten Beschreibung informatischer Denk- und Herangehensweisen, die auch über die Informatik hinaus Anwendung finden können und damit einen Beitrag zur Allgemeinbildung leisten. Debugging hat insbesondere unter dieser Perspektive in den letzten Jahren an Aufmerksamkeit gewonnen (*Brennan und Resnick, 2012; Yadav et al., 2011; Kazimoglu et al., 2012*) und findet sich beispielsweise prominent in neueren Lehrplänen, die auf Computational Thinking aufbauen, wie z.B. dem britischen „Computing Curriculum“ (*Brown et al., 2014*).

Debugging ist also eine zentrale Tätigkeit – und Notwendigkeit – professioneller Softwareentwicklung, in der Fehler ob ihrer Unvermeidbarkeit einen besonderen Stellenwert einnehmen. Darüber hinaus ist Debugging eine Herangehensweise des Computational Thinking und trägt damit zur Allgemeinbildung bei. Nichtsdestotrotz stellt Debugging ein bisher ungelöstes Schlüsselproblem des Informatikunterrichts dar: Die Schülerinnen und Schüler müssen sich das systematische Finden und Beheben von Fehlern zumeist selbst aneignen und sehen sich damit einer großen Herausforderung und wiederkehrenden Quelle für Frustration beim Programmierenlernen ausgesetzt. Aber auch die Lehrkräfte stehen vor der enormen Herausforderung, allen Schülerinnen und Schülern gleichzeitig gerecht zu werden. Lösen ließe sich dieses Problem durch die adäquate Vermittlung von Debuggingfähigkeiten im Informatikunterricht, die die Selbstständigkeit der Schülerinnen und Schüler steigert und die Lehrkräfte gleichermaßen entlastet. Jedoch fehlt es an entsprechenden Konzepten und Materialien für den Informatikunterricht.

1.2 Forschungslücke und Ziel

Trotz der großen Bedeutung des Debuggings für den Informatikunterricht ist bisher ungeklärt, wie Debugging zielführend in die unterrichtliche Praxis integriert werden kann. Wie in Kapitel 2 dieser Arbeit ausführlich herausgearbeitet wird, ist die zentrale Frage „Wie kann Debugging vermittelt werden?“ in der informatikdidaktischen Forschung bisher unbeantwortet. Darüber hinaus fehlt es auch an der Klärung von Forschungsdesideraten, die die Grundlage für die Entwicklung entsprechender Konzepte und Materialien für den Informatikunterricht darstellen. Nimmt man beispielsweise die Perspektiven des didaktischen Dreiecks (vgl. z.B. *Bönsch (2006)*) als Ausgangspunkt, ist festzustellen, dass weder

bezüglich des *Stoffs* noch der *Lehrkräfte* oder der *Lernenden* notwendige Forschungsergebnisse vorliegen.

So ist bisher unbeantwortet, welche Fähigkeiten Programmieranfängerinnen und -anfänger überhaupt für selbstständiges und effektives Debuggen benötigen (vgl. Kapitel 2.2). Die Klärung des eigentlichen Unterrichtsgegenstandes und der entsprechenden zu vermittelnden Inhalte stellt dabei die Grundlage jedes fachdidaktischen und unterrichtlichen Handelns dar, ohne die sich Fragen nach dem methodischen Vorgehen oder der adäquaten Aufbereitung im Unterricht gar nicht erst stellen.

Genauso fehlt es an Erkenntnissen bezüglich der Perspektive der Lehrkräfte: Diese sind tagtäglich mit den Programmierfehlern der Schülerinnen und Schüler konfrontiert. Allerdings mangelt es an Untersuchungen, die sowohl die zumeist anekdotisch wiedergegebene Problematik der *Turnschuhdidaktik* als auch bereits eingesetzte Ansätze der Lehrkräfte beleuchten (vgl. Kapitel 2.7). Erst die Berücksichtigung der unterrichtspraktischen Realität ermöglicht die Entwicklung geeigneter Lehr-Lern-Konzepte für den Informatikunterricht.

Und auch Schülerinnen und Schüler und die Vorkenntnisse, die sie aus ihrem Alltag als Lernvoraussetzungen bereits mitbringen, wurden bisher nicht eingehend untersucht (vgl. Kapitel 2.4). Dabei sind solche Lernvoraussetzungen und Erfahrungen aus dem Alltag gemäß einer konstruktivistischen Auffassung von Lernen zentral für die Gestaltung von Konzepten und Materialien für den Unterricht.

Damit ist festzustellen, dass es an entsprechenden Grundlagen und Forschungsergebnissen für die Gestaltung von Konzepten und Materialien fehlt, um das Schlüsselproblem Debugging im Informatikunterricht angemessen zu adressieren. Dementsprechend sollen in dieser Arbeit Debugging im Informatikunterricht informatikdidaktisch aufgearbeitet und entsprechende Forschungsdesiderate aufgegriffen werden. Darauf aufbauend ist es das Ziel dieser Arbeit, durch die Entwicklung geeigneter Konzepte und Materialien für den Unterricht die Selbstständigkeit der Schülerinnen und Schüler in der Fehlerbehebung zu steigern und damit die Frustration beim Programmierenlernen im Unterricht zu senken sowie das Problem der *Turnschuhdidaktik* anzugehen.

1.3 Forschungsmethodische Einordnung

Wie lässt sich nun ein Forschungsprozess ausgestalten, um dieses unterrichtspraktische Problem zu adressieren? In der Informatikdidaktik bzw. den Didaktiken der Naturwissenschaften existieren verschiedene Forschungsformate, die insbesondere Veränderungen und die Weiterentwicklung der Unterrichtspraxis als zentrale Zieldimension haben und Orientierung für die Gestaltung eines Forschungsvorhabens bieten können. Im Rahmen von *Wirk-*

samkeitsforschung wird „vor der flächendeckenden Verbreitung von Innovationen im Bereich des Lehrens und Lernens sorgfältig [...] erforsch[t], welche Wirkungen und Nebenwirkungen mit Neuerungen verbunden sind“ (Gräsel, 2011), beispielsweise durch (quasi-)experimentelle Interventionsstudien, um damit auch „zur Weiterentwicklung fachdidaktischer Theorien bei[zu]tragen“ (Reiss und Ufer, 2009). Implementationsforschung geht der Frage nach, wie sich entsprechende fachdidaktische Innovationen in der Praxis umsetzen und verbreiten lassen (Gräsel und Parchmann, 2004). Voraussetzung für die Anwendung der beschriebenen Forschungsformate sind dabei existierende Innovationen, also beispielsweise theoretisch fundierte Interventionen oder Unterrichtskonzepte (Gräsel, 2011).

Entwicklungsforschung (oder auch *Design-Based Research*) zielt auf die Weiterentwicklung von Unterricht, indem oftmals partizipativ Konzepte entwickelt und in der unterrichtlichen Praxis iterativ erprobt und verbessert werden (Jahn, 2017). Damit werden Erkenntnisse zur Anreicherung bestehender Theorien über Lehr-Lern-Prozesse gewonnen. Auch hier liegt ein großer Fokus der wissenschaftlichen Arbeit auf der iterativen unterrichtspraktischen Erforschung.

Für das Themengebiet Debugging fehlt es im Informatikunterricht bisher allerdings sowohl an theoretisch fundierten Unterrichtskonzepten, die nun auf Wirksamkeit überprüft oder in der Unterrichtspraxis verbreitet werden könnten, als auch an einer informatikdidaktischen Aufarbeitung des Themas, welche die Grundlage für die Entwicklung und Erprobung entsprechender Unterrichtskonzepte darstellen könnte (vgl. Kapitel 2.7).

Im Gegensatz zu den gerade beschriebenen Forschungsformaten legt die *didaktische Rekonstruktion* zunächst einen Fokus auf die *didaktische Strukturierung* des Unterrichtsgegenstandes, basierend auf der Untersuchung verschiedener Perspektiven. Aufbauend darauf werden entsprechende Lernangebote und Lehr-Lern-Arrangements entwickelt und evaluiert.

1.3.1 Das Modell der didaktischen Rekonstruktion

Die didaktische Rekonstruktion, entwickelt von Kattmann *et al.* (1997), hat ihren Ursprung in der Didaktik der Naturwissenschaften. Ausgangspunkt ist dabei die Feststellung der Autoren, dass fachdidaktische Forschung oftmals stark von einer fachlichen Perspektive ausgeht, um Themen für den Unterricht aufzubereiten. Da sich fachliche Inhalte eben nicht unverändert in den Unterricht übertragen ließen, müsse dabei insbesondere die Perspektive der Lernenden auf den Unterrichtsgegenstand berücksichtigt werden.

Im Modell der didaktischen Rekonstruktion werden fachliche Perspektive und die Perspektive der Lernenden daher explizit gleichberechtigt dargestellt. Hierdurch sollen „die im Rahmen der Didaktischen Rekonstruktion durchgeführten Forschungsarbeiten lernförderlicher

werden als solche, die sich allein auf eine fachwissenschaftliche Sachstruktur oder lernpsychologische Prinzipien stützen könnten“ (Kattmann, 2007). Entsprechend dem fachdidaktischen Triplet (siehe Abbildung 1.1) bildet das Wechselspiel beider Perspektiven die Grundlage der didaktischen Strukturierung des Lerngegenstandes.

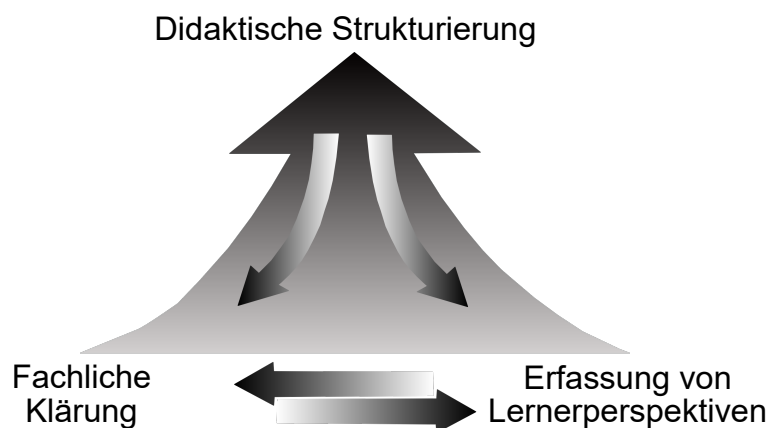


Abbildung 1.1: Fachdidaktisches Triplet nach Kattmann et al. (1997)

Typischerweise handelt es sich bei den Unterrichtsgegenständen, die mit Hilfe des Modells der didaktischen Rekonstruktion untersucht und aufbereitet werden, um *Themengebiete* (oder auch *Inhaltsbereiche*), wie etwa *Bestand und Änderung in der Analysis* (Hahn und Prediger, 2008), das *Stoff-Teilchen-Konzept* in der Chemie (Schmidt, 2011), *Daten* in der Informatik (Grillenberger, 2019) oder *Wasser* in der Biologie (Kattmann et al., 1997). Im Gegensatz dazu handelt es sich bei Debugging jedoch um einen *Prozess*. Nichtsdestotrotz erscheint die Anwendung des Modells der didaktischen Rekonstruktion auch für Debugging als Unterrichtsgegenstand geeignet. Dazu wird im Folgenden ausgeführt, welche Fragen sich für die jeweilige Perspektive des Modells für den *Prozess* Debugging stellen und inwieweit dabei teilweise ein anderer Schwerpunkt als für ein *Themengebiet* gelegt wird.

Fachliche Klärung. Ziel der *fachlichen Klärung* ist es, die zugrunde liegende Sachstruktur des Unterrichtsgegenstandes herauszuarbeiten, die die Grundlage für die unterrichtliche Vermittlung bildet. Dazu werden fachwissenschaftliche Vorstellungen, Theorien, Methoden und Termini aus einer fachdidaktischen Perspektive analysiert. Kattmann et al. (1997) schlagen dazu eine hermeneutisch-analytische Untersuchung geeigneter Quellentexte wie Lehrbücher vor, beispielsweise mit Hilfe der qualitativen Inhaltsanalyse nach Mayring.

Wie gestaltet sich die fachliche Klärung eines *Prozesses* wie Debugging? Auch hier müssen die entsprechenden Unterrichtsinhalte aus der wissenschaftliche Struktur herausgearbeitet

werden. Allerdings handelt es sich dabei weniger um die zugrunde liegende „Sachstruktur“, als um fachwissenschaftliche Theorien bezüglich des Ablaufs des Debuggingprozesses. Eine entsprechende fachdidaktische Analyse dieser Theorien ermöglicht es, die *Fähigkeiten* (im Gegensatz zu *Konzepten* eines *Themengebietes*, vgl. z.B. Grillenberger (2019)) zu konkretisieren, die Programmieranfängerinnen und -anfänger für das Debuggen benötigen und die dementsprechend im Unterricht vermittelt werden müssen. Damit ergibt sich für die fachliche Klärung des Unterrichtsgegenstandes Debugging die folgende Forschungsfrage:

RQ: *Was sind relevante Debuggingfähigkeiten für Programmieranfängerinnen und -anfänger?*

Erfassung von Lernerperspektiven. Im Zuge der *Erfassung von Lernerperspektiven* werden Vorstellungen der Lernenden zum Lerngegenstand erhoben. Gemäß der konstruktivistischen Annahmen des Modells der didaktischen Rekonstruktion ist Lernen ein konstanter und aktiver Prozess der Anpassung mentaler Modelle, die insbesondere durch Alltagserfahrungen geprägt sein können. Im Zuge einer empirischen Untersuchung, beispielsweise mittels offener Interviews oder Fragebögen, werden Kategorien gebildet, um entsprechende themenbezogenen Lernvoraussetzungen zu identifizieren.

Auch für das Debugging stellt die Erfassung der Lernerperspektive gemäß konstruktivistischer Lerntheorie die Voraussetzung für die Entwicklung von Konzepten und Materialien für den Unterricht dar. Dabei beeinflussen insbesondere *Troubleshooting*-Erfahrungen der Lernenden aus ihrem Alltag den Debuggingprozess und stellen damit im konstruktivistischen Sinne die Basis des Lernprozesses dar. Dementsprechend leitet sich gemäß dem Modell der didaktischen Rekonstruktion folgende Forschungsfrage ab:

RQ: *Welche debuggingspezifischen Lernvoraussetzungen bringen Schülerinnen und Schüler aus ihrem Alltag mit?*

Didaktische Strukturierung. In der *didaktischen Strukturierung* werden, basierend auf der fachlichen Klärung und der Erfassung der Lernerperspektiven, konkrete Lernangebote entwickelt und empirisch auf ihre Wirksamkeit überprüft. Ergebnis der didaktischen Strukturierung können dabei konkrete Aufgaben, die Formulierung von Leitlinien und Prinzipien oder konkrete Unterrichtselemente sein.

Für das Debugging müssen ebenfalls, aufbauend auf der fachlichen Klärung und Lernerperspektive, theoretisch-fundierte konkrete Konzepte für die Unterrichtspraxis entwickelt werden. Damit ergibt sich zunächst die folgende Forschungsfrage:

RQ: *Wie sollten Konzepte und Materialien für die Vermittlung von Debuggingfähigkeiten im Unterricht gestaltet werden?*

Diese Konzepte müssen dann weiterhin in der Schulpraxis empirisch überprüft und die geeignete Implementierung in der Unterrichtspraxis untersucht werden. Vor dem Hintergrund des Ziels, die Selbstständigkeit der Schülerinnen und Schüler zu steigern, stellen dabei die Selbstwirksamkeitserwartungen, aber auch die eigentliche Debuggingleistung geeignete Zieldimensionen dar. Damit leitet sich die folgende Forschungsfrage zur Evaluation aus dem Modell ab:

RQ: *Welchen Effekt hat die explizite Vermittlung von Debuggingfähigkeiten auf Selbstwirksamkeit und Debuggingleistung?*

Damit derart entwickelte Konzepte und Materialien für den Unterricht auch in der Schulpraxis verbreitet und angewendet werden, stellt sich außerdem die Frage, wie diese geeignet für Lehrkräfte aufgearbeitet werden können. Daher muss für eine tatsächliche Praxiswirksamkeit der Ergebnisse der Transfer und die Veränderung der Unterrichtspraxis untersucht werden. Zu diesem Zweck wird eine Fortbildung konzipiert und evaluiert:

RQ: *Wie ändert sich der Unterricht der an einer Fortbildung beteiligten Lehrkräfte im Bezug auf Debugging?*

1.3.2 Die didaktische Rekonstruktion in der Informatikdidaktik

Diethelm et al. (2011) entwickelten eine Variante des Modells der didaktischen Rekonstruktion speziell für die Informatikdidaktik. Sie erweiterten das Modell dabei um drei Bereiche (siehe Abbildung 1.2): Insbesondere aufgrund der jungen Tradition des Schulfaches Informatik müsse der Allgemeinbildungsanspruch des jeweiligen Bildungsinhaltes unter dem Aspekt der *Klärung gesellschaftlicher Ansprüche ans Fach* untersucht werden. Da gerade in der Informatik Lehrkräfte oftmals auch ohne dedizierten fachlichen oder pädagogischen Hintergrund nachqualifiziert oder weitergebildet werden oder sich entsprechende Inhalte selbst angeeignet haben, sei für die Akzeptanz und Verbreitung entwickelter Unterrichtskonzepte die Untersuchung der *Perspektive der Lehrkräfte* ebenso essentiell³. Darüber hinaus betonen die Autorinnen und Autoren die *Auswahl informatischer Phänomene* aus der Lebenswelt der Schülerinnen und Schüler und ordnen sich damit in die Tradition einer naturwissenschaftlichen phänomenorientierten Herangehensweise an Unterricht ein.

Klärung gesellschaftlicher Ansprüche ans Fach. Für die *Klärung gesellschaftlicher Ansprüche* wird der Frage nachgegangen, „*welchen allgemeinbildenden Gehalt der zu Rede stehende Aspekt der Informatik hat*“ (Diethelm et al., 2011). Hierdurch können für die Schülerinnen und

³Bereits Duit, Gropengießer und Kattmann (2005) nehmen die Perspektive der Lehrkräfte in das Modell der didaktischen Rekonstruktion mit auf und erweitern die Erfassung der Lernerperspektive hin zur *Research on teaching and learning*, die insbesondere die Einstellungen der Lehrkräfte inkludiert.

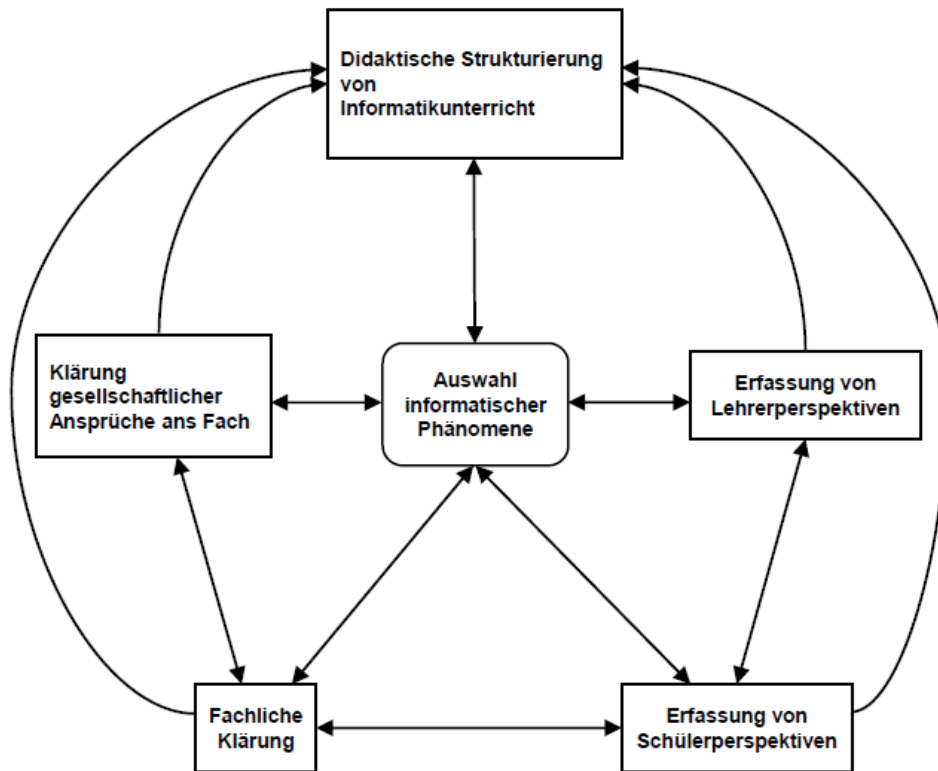


Abbildung 1.2: Didaktische Rekonstruktion für die Informatik nach Diethelm et al. (2011)

Schüler relevante Kontexte und Phänomene identifiziert werden. Basis für diese Klärung können zum Beispiel existierende Standards wie die Bildungsstandards der Gesellschaft für Informatik und deren Argumentation sein.

Debugging ist eine zentrale Herangehensweise des Computational Thinking und damit auch über den Informatikunterricht hinaus relevant für den Alltag der Schülerinnen und Schüler. Um Debugging nachhaltig zu unterrichten, sollten daher der Beitrag für die Allgemeinbildung und Konsequenzen für die Ausgestaltung entsprechender Unterrichtskonzepte herausgearbeitet werden. Damit ergibt sich die folgende Forschungsfrage:

RQ *Welchen Beitrag können Debuggingfähigkeiten zur Allgemeinbildung leisten?*

Erfassung von Lehrerperspektiven. Bei der *Erfassung der Perspektive der Lehrkräfte* werden einerseits die Vorstellungen und Erklärmuster der Lehrkräfte vom und für das Thema berücksichtigt, andererseits auch ihre Ansichten bezüglich des Lernprozesses der Schü-

lerinnen und Schüler oder der Strukturierung des Lerngegenstandes für den Unterricht untersucht. Als Beispiele für eine mögliche Methodik führen *Diethelm, Hubwieser und Klaus (2012)* eine Befragung von Lehrkräften via Online-Fragebögen oder durch semistrukturierte Interviews an.

Lehrkräfte sind tagtäglich mit den Programmierfehlern der Schülerinnen und Schüler konfrontiert und damit Experten für die unterrichtspraktischen Anforderungen von Konzepten und Materialien für das Debugging im Informatikunterricht. Darüber hinaus bieten bisherige Best Practices, Methoden und die Strukturierung des Unterrichtsgegenstandes Debugging durch die Lehrkräfte Ansatzpunkte für die didaktische Strukturierung. Damit ergibt sich gemäß dem erweiterten Modell der didaktischen Rekonstruktion die folgende Forschungsfrage:

RQ *Wie gehen Lehrkräfte mit Programmierfehlern im Unterricht um und wie vermitteln sie Debugging?*

Auswahl informatischer Phänomene. Um die Motivation der Schülerinnen und Schüler zu erhöhen und eine direkte Anwendung des Gelernten zu ermöglichen, werden geeignete Kontexte und Phänomene aus der Lebenswelt der Schülerinnen und Schüler identifiziert. Damit wird die Relevanz und Bedeutung der Informatik für diese alltäglichen Phänomene herausgestellt.

Auch das Finden und Beheben von Fehlern ist ein Phänomen des Alltags: Dort *troubleshooten* die Schülerinnen und Schüler, beispielsweise wenn „das Internet“ nicht mehr funktioniert oder das Fahrrad kaputt ist. Allerdings steht dabei für den *Prozess* Debugging im Gegensatz zu einem *Themengebiet* (wie etwa des Physical Computing) nicht die Erklärung von Phänomenen (wie etwa „Wie ist es möglich, dass Autos Verkehrszeichen lesen und verstehen können?“, vgl. *Przybylla (2018)*) mit Hilfe von Methoden und Konzepten des Faches im Vordergrund, sondern die tatsächliche Anwendung entsprechender Fähigkeiten im Sinne des Computational Thinking. Damit ist die Auswahl solcher Phänomene für das Debugging aber bereits in der *Klärung gesellschaftlicher Ansprüche ans Fach* subsumiert und kann nicht mit der ursprünglichen Intention des Modells angewandt werden. Daher ergibt sich für diese Arbeit keine eigenständige Forschungsfrage für diesen Teil des Modells der didaktischen Rekonstruktion.

Grillenberger, Przybylla und Romeike (2016) haben dieses Modell der didaktischen Rekonstruktion nach Diethelm et al. in ein Prozessmodell überführt und adaptiert (siehe Abbildung 1.3). Dabei bezeichnen sie die Untersuchung der Perspektiven der Lehrkräfte, Schülerinnen und Schüler, gesellschaftlicher Ansprüche und fachlicher Klärung als *underlying perspectives*. Der Auswahl der informatischen Phänomene wird als *educational content preparation* die Auswahl von geeigneten Kontexten und informatischen Konzepten vorangestellt. Außer-

dem wird die didaktische Strukturierung explizit in die von Kattmann et al. formulierten Zieldimensionen wie die Formulierung von Leitlinien und Prinzipien oder die Gestaltung konkreter Unterrichtselemente unterteilt. Diese prozessorientierte Darstellung des Modells ermöglicht es, den Forschungsprozess zu strukturieren und die einzelnen Teiluntersuchungen in eine Reihenfolge zu bringen. Dies erlaubt es, die Forschungsfragen entsprechend anzuordnen: So werden in dieser Arbeit zunächst die zugrunde liegenden Perspektiven untersucht, um darauf aufbauend die didaktische Strukturierung vorzunehmen.

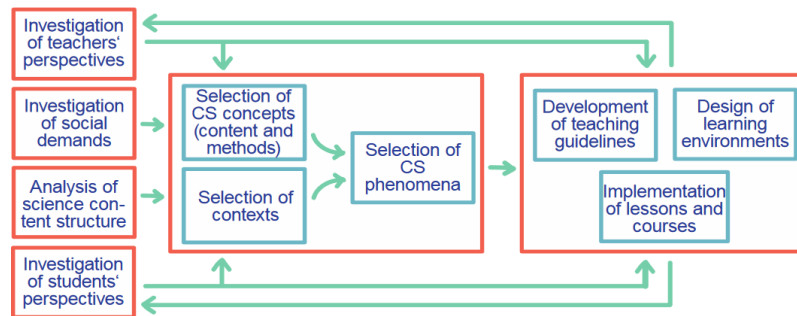


Abbildung 1.3: Adaptiertes Modell der Didaktischen Rekonstruktion für die Informatik (Grillenberger, Przybylla und Romeike, 2016)

Fazit

Zusammenfassend zeigt sich, dass das Forschungsformat der didaktischen Rekonstruktion für die fachdidaktische Aufbereitung von Debugging für den Informatikunterricht geeignet erscheint. Gerade der Fokus auf die Untersuchung der zugrunde liegenden Perspektiven, auf deren Basis die didaktische Strukturierung vorgenommen wird, eignet sich vor dem Hintergrund des Forschungsstandes und der entsprechenden ungeklärten Fragen. Auch die Anwendung des Forschungsformats auf einen *Prozess* im Gegensatz zu den üblichen *Themengebieten* erscheint zielführend. Für manche Aspekte des Modells, wie beispielsweise die *fachliche Klärung*, ergeben sich damit adaptierte Zielsetzungen (*Fähigkeiten* statt *Konzepte*), die nichtsdestotrotz im Sinne des Modells die jeweilige Fragestellung untersuchen (fachdidaktische Analyse der wissenschaftlichen Struktur des Unterrichtsgegenstandes). Lediglich die *Auswahl informatischer Phänomene* mit dem Ziel der Steigerung der Motivation und der direkten Anwendung des Gelernten durch die Schülerinnen und Schüler kann nur eingeschränkt für den Prozess Debugging angewandt werden. Ob der direkten und unmittelbaren Anwendung eines Prozesses (und insbesondere des Debugging im Kontext der Programmierung, die, wie bereits dargelegt, unvermeidbar ist) erscheint diese Einschränkung vernachlässigbar.

1.3.3 Forschungsfragen

Im vorherigen Abschnitt wurde das Modell der didaktische Rekonstruktion und dessen Anwendung auf den Prozess Debugging diskutiert. Im Folgenden werden nun die Forschungsfragen dieser Arbeit, die aus dem Forschungsformat abgeleitet wurden, gemäß dem Prozessmodell von *Grillenberger, Przybylla und Romeike (2016)* zusammenfassend angeordnet und dargestellt.

Zunächst werden dabei die *zugrunde liegenden Perspektiven* untersucht:

- **(RQ1)** Was sind relevante Debuggingfähigkeiten für Programmieranfängerinnen und -anfänger? (*Fachliche Klärung*)
- **(RQ2)** Wie gehen Lehrkräfte mit Programmierfehlern im Unterricht um und wie vermitteln sie Debugging? (*Perspektive der Lehrkräfte*)
- **(RQ3)** Welchen Beitrag können Debuggingfähigkeiten zur Allgemeinbildung leisten? (*Klärung gesellschaftlicher Ansprüche*)
- **(RQ4)** Welche debuggingspezifischen Lernvoraussetzungen bringen Schülerinnen und Schüler aus ihrem Alltag mit? (*Perspektive der Lernenden*)

Aufbauend auf der Klärung dieser zugrunde liegenden Perspektiven auf das Thema Debugging können dann im Sinne der didaktischen Strukturierung theoretisch-fundierte konkrete Konzepte für die Unterrichtspraxis und den Transfer entwickelt und evaluiert werden.

- **(RQ5)** Wie sollten Konzepte und Materialien für die Vermittlung von Debuggingfähigkeiten im Unterricht gestaltet werden?
- **(RQ6)** Welchen Effekt hat die explizite Vermittlung von Debuggingfähigkeiten auf Selbstwirksamkeit und Debuggingleistung?
- **(RQ7)** Wie ändert sich der Unterricht der an einer Fortbildung beteiligten Lehrkräfte im Bezug auf Debugging?

1.4 Struktur der Arbeit

Auch die Struktur dieser Arbeit orientiert sich am Modell der didaktischen Rekonstruktion und den entsprechenden Forschungsfragen (vgl. Abbildung 1.4). Zunächst wird jedoch im zweiten Teil dieser Arbeit der relevante Forschungsstand des Themas Debugging in der informatikdidaktischen Forschung dargestellt. Dazu wird *Debugging* zunächst begrifflich von der Programmierung und dem Testen abgegrenzt. Darüber hinaus werden Erkenntnisse

bezüglich des Debuggingprozesses von Debugging-Novizen und -Experten dargelegt. Außerdem werden existierende Forschungsergebnisse zur Perspektive der Lernenden sowie zur expliziten Vermittlung von Debuggingfähigkeiten ausgeführt. Abschließend werden, basierend auf dem Forschungsstand relevante Erkenntnisse und ihr Einfluss auf den Forschungsprozess zusammengefasst.

In Teil II der Arbeit werden die *zugrunde liegenden Perspektiven* auf das Thema untersucht. In Kapitel 3 wird dazu zunächst die fachliche Klärung vorgenommen (RQ1). Entsprechend der vorgeschlagenen Methodik nach *Kattmann et al. (1997)* werden dazu relevante Dokumente hermeneutisch-analytisch ausgewertet und vier wesentliche Debuggingfähigkeiten identifiziert. In Kapitel 4 wird darauf aufbauend die Perspektive der Lehrkräfte untersucht (RQ2). Dazu werden mit Hilfe einer Interviewstudie der Umgang mit Programmierfehlern im Klassenzimmer, die vermittelten Debuggingfähigkeiten sowie die Begründungen für die entsprechende Auswahl analysiert. Die Perspektive des Computational Thinking wird in Kapitel 5 aufgearbeitet und der Beitrag von Debuggingfähigkeiten für die Allgemeinbildung untersucht (RQ3). Zuletzt wird die Perspektive der Schülerinnen und Schüler (RQ4) in Kapitel 6 genauer betrachtet: Mit Hilfe der Auswertung von Videodaten aus einem Debugging-Escaperoom werden debuggingspezifische Lernvoraussetzungen aus dem Alltag identifiziert und Konsequenzen für die Vermittlung von Debuggingfähigkeiten für die Programmierung diskutiert.

In Teil III der Arbeit werden, basierend auf den Untersuchungen der zugrunde liegenden Perspektiven, im Sinne der didaktischen Strukturierung Gestaltungskriterien für Debugging im Informatikunterricht dargelegt und anschließend wird ein integratives Konzept für den Informatikunterricht entworfen (RQ5).

Dieses wird in Teil IV in der Praxis implementiert und in Kapitel 8 im Zuge einer Interventionsstudie auf seine Wirksamkeit untersucht (RQ6). Zusätzlich wird in Kapitel 9 der Transfer der Forschungsergebnisse in die Unterrichtspraxis analysiert (RQ7).

Abschließend werden in Kapitel 10 die Ergebnisse der Arbeit zusammengefasst und es wird ein Fazit gezogen.

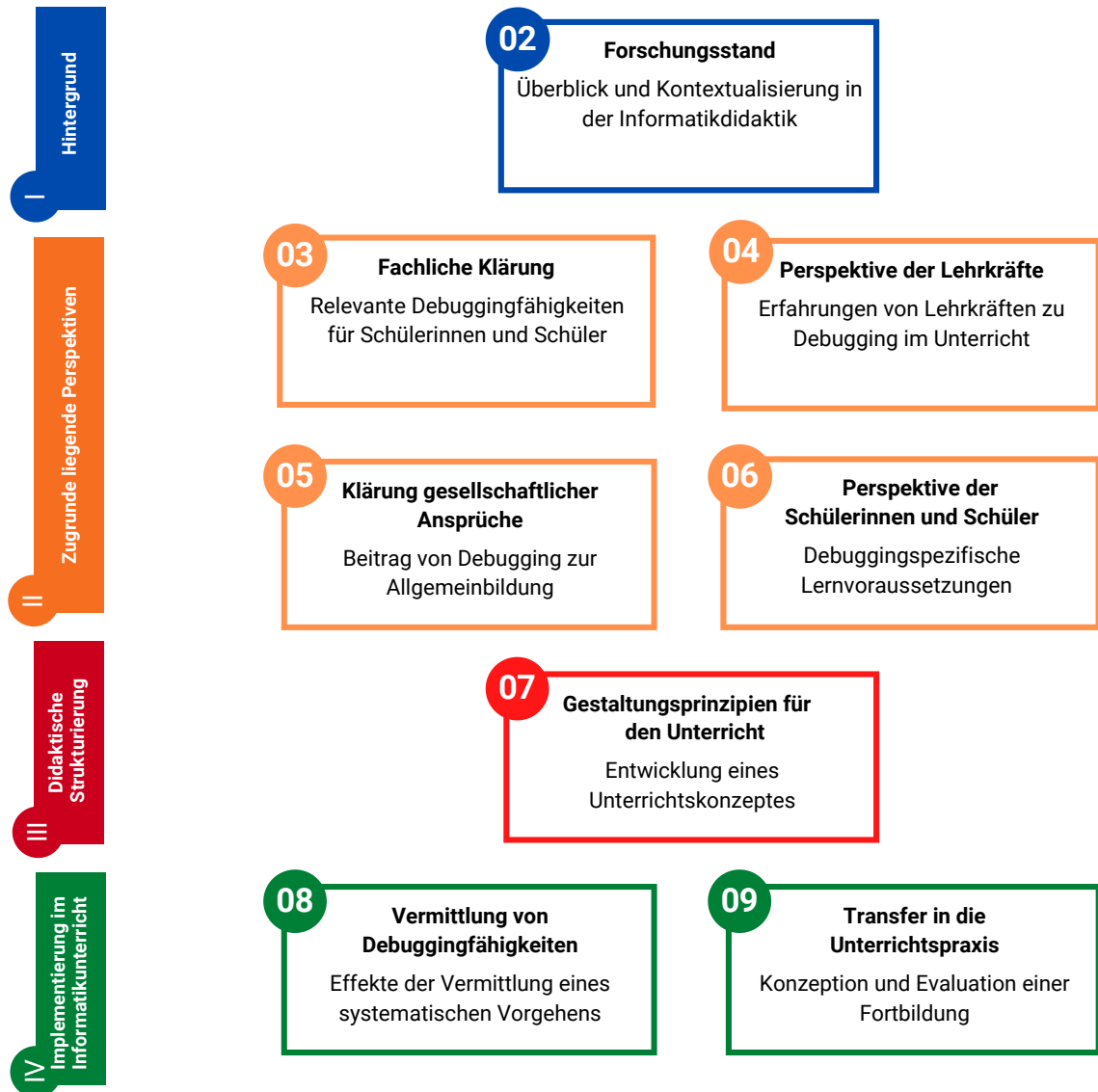


Abbildung 1.4: Struktur dieser Arbeit nach dem Modell der didaktischen Rekonstruktion

Teil I:

**Debugging als Thema
informatikdidaktischer Forschung**

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

MAURICE WILKES, 1949

2 Forschungsstand

Im Folgenden soll der Forschungsstand für das Thema Debugging aus informatikdidaktischer Perspektive beschrieben werden. Dabei werden in Ermangelung von Studien und Forschungsergebnissen spezifisch für den schulischen Informatikunterricht auch hochschuldidaktische Forschungsarbeiten berücksichtigt. Der Forschungsstand ermöglicht dabei einerseits die ausführliche Herausarbeitung der Forschungslücke, die in Kapitel 1.2 bereits knapp zusammenfasst wurde. Andererseits stellen die existierenden Ergebnisse die Basis für die Untersuchung der zugrunde liegenden Perspektiven als auch für die didaktische Strukturierung von Debugging im Informatikunterricht dar. Die Beschreibung des Forschungsstandes wird dabei durch die folgenden Fragen strukturiert:

Zunächst ist es für diese Arbeit notwendig, den Prozess *Debugging* zu definieren und vom Programmieren und Testen abzugrenzen:

1. Was unterscheidet Debugging von **Testen und Programmieren**?

Aufbauend darauf wird der Forschungsstand bezüglich des Ablaufs des Debuggingprozesses ausgeführt. Dabei wird geklärt, welches Vorgehen einen „guten Debugger“ (in Form von Experten) ausmacht, und welche Unterschiede zu Novizen existieren. Gemäß dem Experten-Novizen-Paradigma (*Gruber und Stöger, 2011*) können für diese Arbeit damit diejenigen Aspekte identifiziert werden, die für erfolgreiches Debugging essentiell sind:

2. Wie läuft der **Debuggingprozess** von Experten und Novizen ab?

Einen großen Einfluss auf den Debuggingprozess haben dabei die unterschiedlichen Fehlerarten, deren Kategorisierung und Bedeutung für Novizen dargestellt werden:

3. Welche Rolle spielen unterschiedliche **Fehlerarten** für den Debuggingprozess?

Darüber hinaus werden existierende Erkenntnisse bezüglich der affektiven Einstellung und der Lernvoraussetzungen von Lernenden in Bezug auf Debugging ausgeführt. Damit wird die Grundlage für die Untersuchung der Perspektive der Lernenden gelegt:

4. Welche Einstellungen und Lernvoraussetzungen haben **Lernende** bezüglich Debugging?

Weiterhin bilden existierende Ansätze und empirische Untersuchungen zur Vermittlung von Debuggingfähigkeiten das Fundament der didaktischen Strukturierung des Prozesses Debugging. Daher werden existierende Konzepte und Studien insbesondere bezüglich der methodische Ansätze und vermittelten Inhalte untersucht:

5. Welche Ansätze für das **Unterrichten** von Debugging gibt es?

Daneben werden existierende Belege für einen Transfer von Debuggingfähigkeiten aus der Programmierdomäne hinaus als empirisches Fundament für den Beitrag von Debugging zur Allgemeinbildung dargestellt:

6. Findet ein **Transfer** von Debuggingfähigkeiten über die Programmierdomäne hinaus statt?

2.1 Debuggen, Testen und Programmieren

Programmierfehler – und deren Behebung – sind so alt wie die Programmierung selbst. Code, der eben nicht frei von Programmierfehlern ist, bedeutet dabei in der Praxis stets einen hohen – zumeist finanziellen – Aufwand und hat teilweise gar verheerende Konsequenzen, wie die vielzitierten Geschichten des arithmetischen Überlaufs der Ariane V⁴, die Einschränkungen des Datumsformates des Y2K-Bugs⁵ oder die Zerstörung des Mars Climate Orbiters durch einen Einheitenfehler⁶ eindrucksvoll belegen. Doch wie kommt es zum Fehlverhalten eines solchen Programms?

Zeller (2009) beschreibt das wie folgt: In der Entwicklung wird durch die Programmiererin bzw. den Programmierer ein *Fehler* oder *Bug* im Programmcode eingefügt. Nicht immer ist das dabei die Schuld der Programmiererin bzw. des Programmierers, beispielsweise könnte eine von vornherein fehlerhafte Spezifikation korrekt umgesetzt worden sein. Wird das Programm nun ausgeführt, kann dieser Fehler zu einem fehlerhaften internen Programmzustand führen. In Konsequenz verhält sich das Programm anders als intendiert. Dieses *Fehlverhalten* wird nun also von außen beobachtbar.

Debugging ist nun der Prozess „to detect, locate, and correct faults in a computer program“ (ISO, 2010): Ausgehend vom Fehlverhalten des Programms muss der zugrunde liegende Fehler (*fault*) lokalisiert und behoben werden⁷. Damit befindet sich Debugging sowohl im Kontext der Programmierung (schließlich wird hier der Fehler gemacht und behoben) als auch der

⁴Ursache des Ausfalls der Steuerungseinheit war ein arithmetischer Überlauf bei der Konvertierung einer 64-Bit-Gleitkommazahl in eine 16-Bit-Ganzzahl, da das Modul vom deutlich langsameren Vorgänger Ariane 4 übernommen wurde, vgl. Lions et al. (1996).

⁵Zur Speicherung von Jahreszahlen wurden, um teuren Speicherplatz zu sparen, nur die letzten beiden Ziffern verwendet, was ab dem Jahr 2000 große Probleme bedeutet hätte, vgl. BSI (1997).

⁶Die fehlende Umrechnung des Impulses vom imperialen ins metrische System führte zur falschen Berechnung des Kurses, vgl. Board (1999).

⁷Gerade in der englischsprachigen Literatur finden sich durchaus unterschiedliche Verwendungen der Begriffe *fault*, *error*, *infection*, *mistake*, *failure* und *defect*.

Qualitätssicherung (hier wird der Fehler entdeckt). Im Folgenden soll Debugging daher zunächst innerhalb dieser beiden Kontexte genauer eingeordnet und abgegrenzt werden.

Betrachtet man den Kontext der Qualitätssicherung, zeigt sich, dass die Begriffe Debugging und Testen oftmals nicht klar voneinander abgegrenzt werden. Metzger (2004) beschreibt den Unterschied zwischen dem Testen und dem Debuggen eines Programmes wie folgt:

Testing is the process of determining whether a given set of inputs causes an unacceptable behavior in a program.

Debugging is the process of determining why a given set of inputs causes an unacceptable behavior in a program and what must be changed to cause the behavior to be acceptable.

Dieser Kontrast zeigt sich auch daran, dass die entsprechenden Tätigkeiten in der Praxis häufig von unterschiedlichen Personen übernommen werden: Es ist die Aufgabe der Testerin bzw. des Testers, das Programm systematisch zu überprüfen, um *fehlerhaftes Verhalten* zu identifizieren und entsprechende Testfälle zur Verfügung zu stellen. Dabei muss er oder sie nicht notwendigerweise auch Einblick in die innere Funktionsweise des Programms haben. Es ist dann Aufgabe der Entwicklerin bzw. des Entwicklers, die Ursache des Fehlverhaltens im Code zu identifizieren und zu beheben. Allerdings werden die meisten Debuggingprozesse in der professionellen Softwareentwicklung nicht durch fehlgeschlagene automatisierte Testfälle ausgelöst, sondern bereits während des Lesens oder Änderns von Code durch die Programmiererin bzw. den Programmierer, also während der eigentlichen Entwicklung (Beller et al., 2018): Die Überprüfung des Codes – fernab von der Systematik eines formalen Softwaretests und der entsprechenden Prozesse –, zumindest mit einem in der Spezifikation enthaltenen Anwendungsfall, ist Teil des Programmierprozesses und deckt bereits eine Vielzahl an Fehlern auf.

Nichtsdestotrotz unterstützen das Testen und gerade automatisierte Testfälle den Debuggingprozess: Zunächst kann so der Fehler reproduziert und das Fehlverhalten genauer analysiert werden. Darüber hinaus kann der Fehler auf Basis der erfolgreichen bzw. fehlgeschlagenen Testfälle genauer eingegrenzt werden, wie beispielsweise auf ein Modul, das nur in den fehlschlagenden Testfällen aufgerufen wird. Außerdem können die Korrekturversuche mit Hilfe automatisierter Testfälle überprüft werden (Beller et al., 2018).

Damit sind Testen und Debuggen eng miteinander verbunden: Einerseits kann durch das Testen eines Programms ein Debuggingprozess gestartet werden. Andererseits ist Testen oftmals Bestandteil des Debuggingprozesses, insbesondere am Anfang (Fehler reproduzieren) und Ende (Korrektur überprüfen). Nichtsdestotrotz sind es unterschiedliche Prozesse, die unterschiedliche Fähigkeiten erfordern und oftmals von unterschiedlichen Personen

übernommen werden. Debugging ist dabei vorwiegend Aufgabe der Entwicklerin bzw. des Entwicklers.

Doch Debuggingfähigkeiten unterscheiden sich auch von Programmierfähigkeiten. *Ahmadzadeh, Elliman und Higgins (2005)* untersuchten diesen Zusammenhang explizit. Dazu erhoben sie einerseits die Programmierfähigkeiten von Informatikstudierenden (Programmieranfängerinnen bzw. -anfänger) über die Leistung in Programmieraufgaben, andererseits die Debuggingfähigkeiten mittels einer Debuggingaufgabe, also einem mit Fehlern versehenen Programm. Dabei stellten sie fest, dass gute Programmierer nicht notwendigerweise auch gute Debugger⁸ sind: Etwa $\frac{2}{3}$ der guten Debugger ihrer Untersuchung waren auch gute Programmierer, während nur etwa 40% der guten Programmierer auch gute Debugger waren. Als Probleme der guten Programmierer, aber schwachen Debugger identifizierten die Autoren fehlende oder falsch angewandte Debuggingstrategien sowie ein mangelndes Verständnis für das „fremde“, also nicht selbstverfasste, zu debuggende Programm.

Fitzgerald et al. (2008) bestätigten die Ergebnisse von *Ahmadzadeh, Elliman und Higgins (2005)*: In ihrer Untersuchung von Informatikstudierenden (ebenfalls Programmieranfängerinnen und -anfänger) lösten die Studierenden zunächst eine Programmieraufgabe. Im Anschluss erhielten sie den Auftrag, ein vorgegebenes Programm für dasselbe Problem zu debuggen, welches mit Fehlern versehen wurde. Auch hier zeigte sich, dass gute Debugger oftmals auch gute Programmierer sind, gute Programmierer hingegen teilweise große Probleme mit dem Debugging hatten.

In beiden Studien erzielten die schwachen Programmierer mehrheitlich schlechte Ergebnisse beim Debuggen. Gewisse Programmierfähigkeiten können dementsprechend als Voraussetzungen für effizientes Debugging angesehen werden.

Während ausgeprägte Debuggingfähigkeiten also in der Regel auf entsprechende Programmierfähigkeiten schließen lassen, gilt die Umkehrung der Aussage nicht unbedingt: Gute Programmierer sind nicht notwendigerweise auch gute Debugger. Debuggingfähigkeiten unterscheiden sich also von Programmierfähigkeiten. Dies wirft die Frage auf, was einen „guten“ Debugger ausmacht?

2.2 Debuggingprozess

Gemäß dem Novizen-Experten-Paradigma können durch den Vergleich von erfahrenen „guten“ Debuggern (Experten) mit weniger guten Anfängerinnen und -anfängern (Novizen)

⁸Der Begriff *Debugger* wird hier als Bezeichnung für eine debuggende Person verwendet und nicht für das Werkzeug.

diejenigen Aspekte identifiziert werden, die für das erfolgreiche Ausführen einer Tätigkeit, in diesem Fall Debugging, essentiell sind (*Gruber und Stöger, 2011*). Diese müssen anschließend im Zuge der Entwicklung von Konzepten und Materialien für den Unterricht gezielt vermittelt werden. Daher sollen solche Unterschiede im Folgenden untersucht werden. Dazu wird anhand des Forschungsstandes zunächst der Debuggingprozess von Experten dargestellt. In einem zweiten Schritt werden existierende Untersuchungen zu Novizen und deren Vorgehensweisen und Problemen dargelegt. Daraufhin werden solcherart identifizierte Unterschiede anhand von Studien, die Experten und Novizen – gemäß klassischer Expertiseforschung – direkt miteinander verglichen, überprüft und abschließend mögliche Einschränkungen dieser Ergebnisse diskutiert.

2.2.1 Experten

Der Frage nach dem „guten“ Debugger kann auf unterschiedliche Art und Weise nachgegangen werden. Die meisten der im Folgenden vorgestellten Ansätze eint, dass sie ein induktives Vorgehen verfolgen: Debugging ist eine praktische *Notwendigkeit* bei der Entwicklung von Software, eine Tätigkeit, die ohne eine zugrunde liegende Theorie seit den Anfängen der Programmierung ausgeübt wird. Auch heutzutage haben die wenigsten professionellen Softwareentwicklerinnen und -entwickler eine formale oder theoriegeleitete Ausbildung für das Debugging erhalten, sondern lernen Debuggen vor allem durch Erfahrung (*Perscheid et al., 2017*). Um nun den Debuggingprozess von „guten“ Debuggern charakterisieren zu können, wird daher vorwiegend der Debuggingprozess von Experten der Programmierung beobachtet. Die jeweiligen Untersuchungen und deren relevante Ergebnisse werden im Folgenden chronologisch dargestellt, da diese oftmals auf vorherigen Erkenntnissen aufbauen.

Gould (1975) beobachtete professionelle Programmierinnen und -programmierer beim Bearbeiten von Debuggingaufgaben in FORTRAN. Das Vorgehen generalisiert der Autor dabei in folgendem ersten Modell des Debuggingprozesses (vgl. Abbildung 2.1): Zunächst wird eine *Debuggingtaktik* gewählt, wie beispielsweise die Ausgabe des Programms zu analysieren, Zeile für Zeile den Code zu untersuchen oder den Kontrollfluss nachzuvollziehen. Dies führt zu Indizien über die Ursache des Fehlverhaltens des Programms⁹, die möglicherweise bereits ausreichen, um den Fehler zu lokalisieren. Wenn nicht, können auf Basis der gewonnenen Informationen im nächsten Schritt *Hypothesen* formuliert werden, wobei dazu auch bisherige Erfahrungen aus anderen Projekten und verwandten Fehlern mit einbezogen werden. Auf Basis der Hypothese wird daraufhin erneut eine adäquate Debuggingtaktik ausgewählt und der Prozess so oft durchlaufen, bis der Fehler gefunden

⁹Auch wenn die Anwendung einer Taktik kein direktes Indiz als Ergebnis hat, stellt dies weitere Informationen bereit, die beim Aufstellen einer Hypothese unterstützen.

ist. Da in diesem Experiment die Probandinnen und Probanden lediglich den Fehler finden, aber ihn nicht korrigieren mussten, fehlt dieser Schritt im resultierenden Modell.

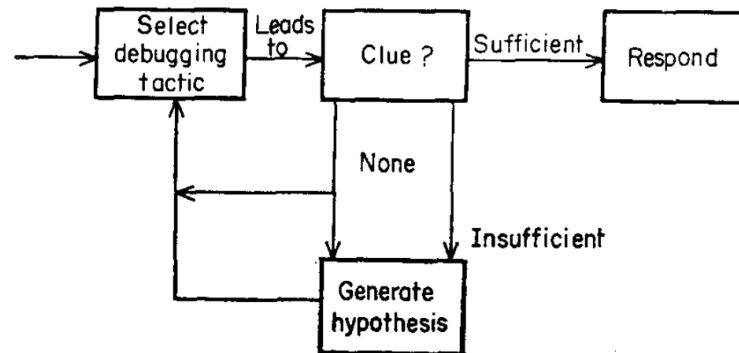


Abbildung 2.1: Modell des Debuggingprozesses nach Gould (1975)

Eine explizite Unterteilung von Debugging in verschiedene Subprozesse nahmen *Katz und Anderson (1987)* vor: Zunächst muss das Programm getestet und ein Fehlverhalten identifiziert werden. Im zweiten Schritt muss nun der Fehler lokalisiert werden. Zuletzt wird der Fehler korrigiert. Diese Dreiteilung entspricht auch der Definition der ISO/IEEE, in der Debugging in die Schritte „detect“, „locate“ und „correct faults“ eingeteilt wird (*ISO, 2010*). Zusätzlich merken Katz und Anderson an, dass, wenn das Programm von einer fremden Person oder bereits vor längerer Zeit geschrieben wurde, ein weiterer Teilschritt hinzukommt: Bevor das Programm getestet und ein Fehlverhalten identifiziert werden kann, muss es zunächst verstanden werden, d.h. die Programmiererin bzw. der Programmierer muss sich mit Funktion und Struktur des Programms vertraut machen.

Diese explizite Trennung und Reihenfolge der Schritte *Programmverständnis* und *Debugging* kritisierte *Gilmore (1991)*. Aus seiner Untersuchung des Vorgehens von Pascal- und BASIC-Entwicklerinnen und -Entwicklern schlussfolgert er, dass Programmverständnis und Debugging kombiniert stattfinden, also während des Schrittes des Verstehens bereits Fehler gefunden und korrigiert werden. Das Lokalisieren eines Fehlers findet dabei durch das Erkennen eines *mismatches* zwischen der mentalen Repräsentation des Programmausschnitts und der mentalen Repräsentation des Problems statt, das durch das Programm adressiert wird (vgl. Abbildung 2.2).

Einen anderen Ansatz wählten *Ducasse und Emde (1988)*: Statt Experten beim Debuggen zu untersuchen, analysierten sie deren in Form von automatisierten Debuggingssystemen abgebildetes Wissen. Dabei identifizierten sie die folgenden notwendigen Wissensbereiche:

- Wissen über das intendierte Programm (I/O, Verhalten, Implementierung)

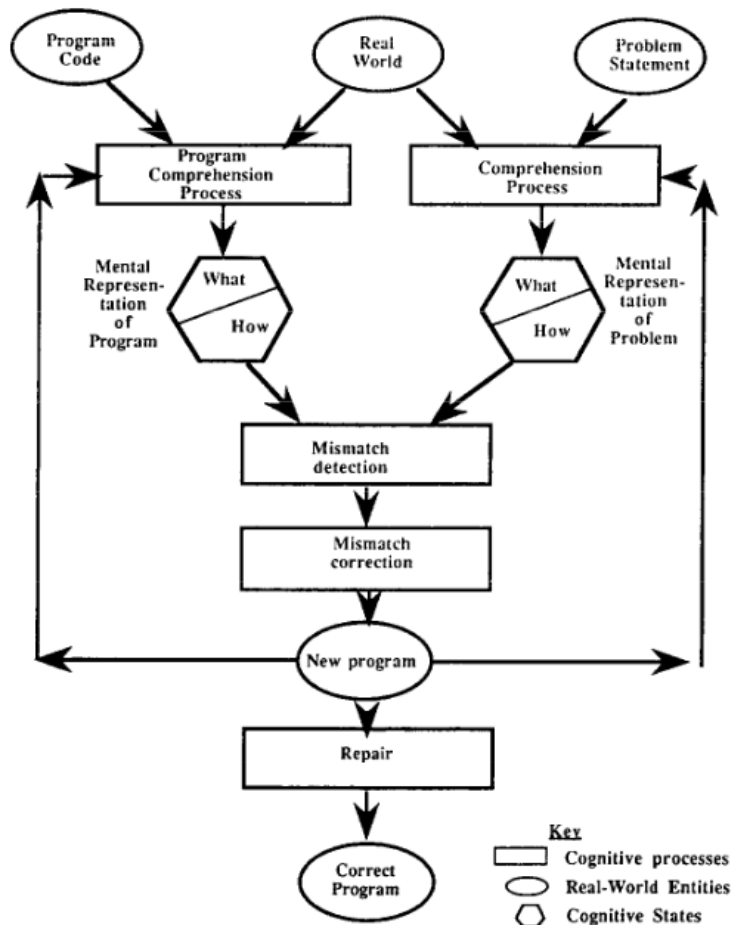


Abbildung 2.2: Modell des Debuggingprozesses nach Gilmore (1991)

- Wissen über das tatsächliche Programm (I/O, Verhalten, Implementierung)
- Verständnis der Programmiersprache
- Allgemeine Programmierexpertise
- Wissen über die Anwendungsdomäne
- Wissen über Bugs
- Wissen über Debuggingmethoden

Dabei werden nicht in jeder Debuggingssituation alle Wissenstypen benötigt. Auch hier zeigt sich also, dass Programmierfähigkeiten Voraussetzung für effektives Debugging darstellen,

darüber hinaus aber dezidiertes Wissen und spezielle Fähigkeiten für das Debugging notwendig sind. Konkrete *Debuggingmethoden* (bei Gould (1975) als Taktiken bezeichnet) der automatisierten Systeme, die das Debugging unterstützen, sind dabei einerseits *Filtering-Strategien*, bei der ausgewählte Programmdurchläufe nachvollzogen werden. Andererseits wird die *computational equivalence* des gewünschten und tatsächlichen Programms überprüft, beispielsweise mit Hilfe von *assertions*. Darüber hinaus wird das Programm auf typische Fehler geprüft.

Yoon und Garcia (1996) und Yoon und Garcia (1998) unterschieden in ihrer Untersuchung professioneller Entwicklerinnen und Entwickler zwischen der *comprehension-* und der *isolation-*Strategie (vgl. Abbildung 2.3 bzw. 2.4) für die Lokalisierung eines Fehlers. Bei der *isolation-*Strategie werden auf Basis von Symptomen, wie dem Fehlerverhalten des Programms oder Fehlermeldungen, Indizien gesammelt, damit eine Hypothese über die Ursache des Bugs generiert werden kann. Diese wird experimentell verifiziert, und gegebenenfalls mehrfach angepasst. Gemäß der *comprehension-*Strategie wird der Code schrittweise analysiert und „verstanden“, um das tatsächliche Programm mit dem erwarteten zu vergleichen. Durch die Erkennung einer Abweichung (*discrepancy*) kann der Bug lokalisiert werden. Dabei handelt es sich also um eine *statische Analyse* zur Kompilierzeit (wie z.B. ein Schreibtschlauf) im Gegensatz zur vorwiegend *dynamischen Analyse* des Programmverhaltens der *isolation-*Strategie zur Laufzeit. Oftmals werden diese beiden Ansätze kombiniert: Zunächst wird mit Hilfe der *isolation-*Strategie grob der Bereich im Code identifiziert, in dem sich der Fehler befindet (beispielsweise eine konkrete Funktion). Mit Hilfe der *comprehension-*Strategie wird dieser Bereich dann genauer analysiert, indem das Programm sequentiell nachvollzogen wird. Damit werden hier das hypothesengeleitete Vorgehen nach Gould (1975) bzw. Katz und Anderson (1987) (*isolation*) und der verständnisorientierte Ansatz nach Gilmore (1991) (*comprehension*), der insbesondere für *fremde* Programme angewandt wird, nebeneinandergestellt. Dabei beschreiben diese beiden Strategien das Vorgehen beim Debuggen auf einem höheren Abstraktionsniveau als die angeführten *Taktiken* (Gould, 1975) oder *Methoden* (Ducasse und Emde, 1988), die dieses allgemeine Vorgehen zu unterstützen scheinen, beispielsweise, in dem sie das Aufstellen und Überprüfen von Hypothesen unterstützen.

Zeller (2009) bezeichnet die *isolation-*Strategie als *scientific debugging* oder auch als *wissenschaftliche Methode*. Ähnlich wie in den Naturwissenschaften wird dabei der Fehler wie ein „Naturphänomen“ betrachtet: „If a program fails, this behavior is initially just as surprising and inexplicable as any newly discovered aspect of the universe. Having a program fail also means that our abstraction fails. We can no longer rely on our model of the program, but rather must explore the program independently from the model.“ Entsprechend der naturwissenschaftlichen Vorgehensweise werden also, ausgehend von der Beobachtung des Fehlverhaltens, Hypothesen formuliert, experimentell überprüft, gegebenenfalls abgelehnt oder verfeinert, bis die Ursache des Fehlers gefunden ist. Darüber hinaus beschreibt Zeller das Vorgehen beim

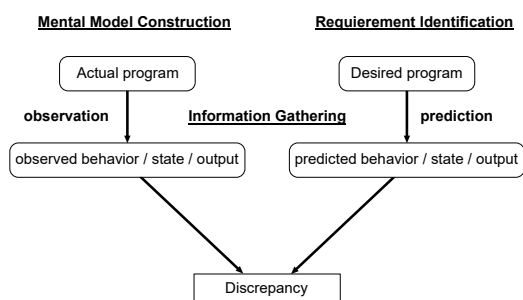


Abbildung 2.3: Comprehension-Strategie nach Yoon und Garcia (1998)

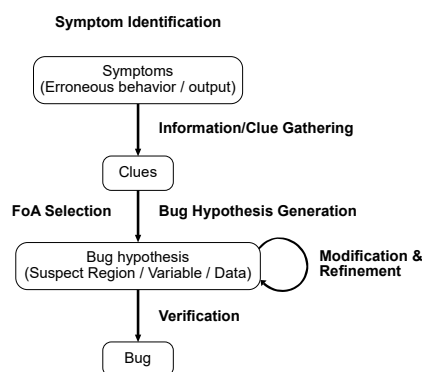


Abbildung 2.4: Isolation-Strategie nach Yoon und Garcia (1998)

Debuggen als Teil des professionellen Softwareentwicklungsprozesses: Zunächst wird das Problem dabei in einer Datenbank (wie einem *Bugtracker*) festgehalten. Anschließend wird das Fehlverhalten reproduziert und ein minimaler automatisierter Testfall dafür geschrieben. Nun wird mit Hilfe der wissenschaftlichen Methode die Ursache gesucht und isoliert, und abschließend wird der Fehler behoben.

Analog dazu beschreiben sowohl Araki, Furukawa und Cheng (1991) als auch Spinellis (2018) den Debuggingprozess in der Softwareentwicklung als Anwendung dieses wiederholten Formulierens von Hypothesen (vgl. Abbildungen 2.5 und 2.6), wiederum nicht auf einer empirischen Basis. Tatsächlich geben in einer Studie zum Status quo des Debugging in der Industrie alle beteiligten Entwicklerinnen und Entwickler ihre Vorgehensweise als ähnlich zu diesem Ansatz an (Perscheid et al., 2017). Außerdem wurden dort für das Debugging eingesetzte Werkzeuge erhoben, die für die Überprüfung und Verfeinerung der Hypothesen verwendet werden. Dabei werden insbesondere Werkzeuge für das *Tracen* oder auch *Loggen* des Programmablaufs wie etwa *print*-Anweisungen, *assertions*, interaktive Debugger, *back-in-time*-Debugger, die es erlauben, nicht nur vorwärts, sondern auch rückwärts die Ausführung des Programms zu verfolgen, oder *Slicing*¹⁰ genannt. Werkzeuge zur automatisierten Erkennung von Fehlern spielen in der Softwarepraxis hingegen noch keine größere Rolle. Manche dieser Vorgehensweisen, wie etwa *assertions* oder *Logging* finden sich dabei auch bei Gould (1975) sowie Ducasse und Emde (1988) und werden dort als Taktiken bzw. Methoden bezeichnet. Es scheinen hier also keine eindeutigen Begrifflichkeiten und trennscharfe Abgrenzungen zu existieren – was möglicherweise wiederum auf den praktischen Charakter von Debugging zurückzuführen ist.

¹⁰Um die Fehlersuche zu unterstützen, wird beim *Slicing* das Programm systematisch vereinfacht, indem ein *slice* erstellt wird, der nur noch aus den Anweisungen besteht, die die relevanten Aspekte (wie beispielsweise den Wert einer konkreten Variable) beeinflussen (Weiser, 1984).

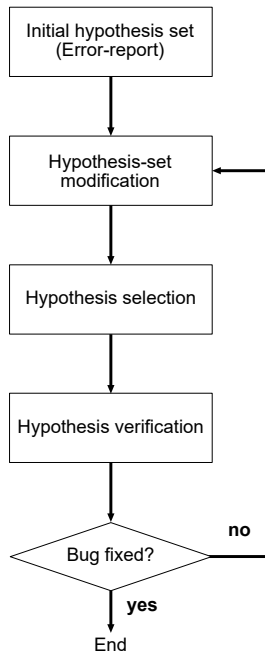


Abbildung 2.5: Modell des Debuggingprozesses nach Araki, Furukawa und Cheng (1991)

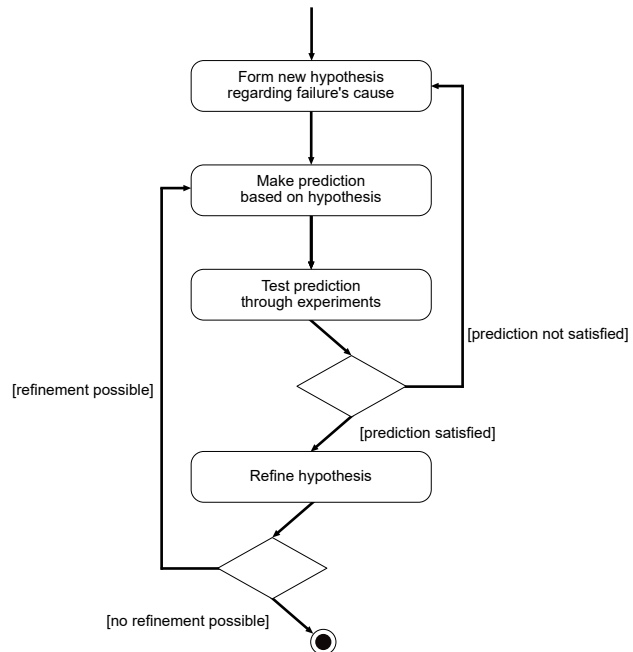


Abbildung 2.6: Modell des Debuggingprozesses nach Spinellis (2018) (Ausschnitt)

Zwischenfazit

Die verschiedenen betrachteten Vorgehensmodelle und Untersuchungen für Experten lassen sich damit wie folgt zusammenfassen: Es existieren zwei Ansätze für den allgemeinen Debuggingprozess, deren Einsatz insbesondere davon abhängig ist, ob es sich um ein *eigenes* Programm handelt, mit dessen Struktur und Aufbau die Programmiererin bzw. der Programmierer zumindest grundlegend vertraut ist, oder um ein *fremdes* Programm, das zunächst verstanden werden muss. Bei der *comprehension*-Strategie wird dazu der Code sequentiell händisch im Sinne eines Schreibtischlaufes nachvollzogen und dabei ein mentales Modell des *tatsächlichen* Programms gebildet und gemäß einer statischen Analyse mit dem spezifizierten gewünschten Programm verglichen. Im Gegensatz dazu geht die *isolation*-Strategie (bzw. *wissenschaftliche Methode*) vom Fehlverhalten des Programms zur Laufzeit aus: Anhand zur Verfügung stehender Indizien, wie der fehlerhaften Ausgabe oder der

Fehlermeldung des Programms, werden *Hypothesen* generiert. Diese werden schrittweise experimentell überprüft und dementsprechend verfeinert oder verworfen, bis der Fehler lokalisiert ist. Beide Ansätze werden dabei auch kombiniert genutzt. Daneben existieren eine Reihe von Taktiken, Methoden oder auch Werkzeugen, wie *print*-Ausgaben, *assertions*, der Debugger, usw., die dieses allgemeine Vorgehen jeweils unterstützen können.

2.2.2 Novizen

Nachdem nun das Vorgehen von „guten“ Debuggern geklärt ist, stellt sich die Frage, inwiefern sich Novizen davon unterscheiden. Verschiedene Autoren beobachteten dazu das Debuggingverhalten von Novizen, um typische Probleme zu identifizieren und damit Schlüsse über effiziente und weniger effiziente Vorgehensweisen zu ziehen. Bezüglich der Einteilung als *Novizen* orientiert sich diese Darstellung an der jeweiligen Einordnung der Autorinnen und Autoren. Zumeist waren die Probandinnen und Probanden dabei Studierende, die über nicht mehr als ein Jahr Programmiererfahrung verfügen. Lediglich in zwei der folgenden Studien waren tatsächlich Schülerinnen und Schüler Gegenstand der Untersuchung.

Carver und Klahr (1986) entwickelten ein Modell eines „guten“ Debuggingvorgehens speziell für Programmieranfängerinnen und -anfänger, um damit den Erwerb von Debuggingfähigkeiten für die Programmiersprache LOGO messen zu können. Ihr resultierendes Modell (vgl. Abbildung 2.7) besteht aus vier bzw. in einer später erweiterten Form aus fünf Schritten¹¹: Zunächst wird im Schritt der *program evaluation* das Programm ausgeführt und ein Fehlverhalten festgestellt. Aufbauend auf dem Unterschied zwischen tatsächlichem und erwartetem Verhalten wird eine Vermutung über die Art des Fehlers angestellt (*bug identification*). Daraufhin wird im Schritt der *program representation* die Struktur des Programms analysiert und auf Basis der bisherigen Indizien der Fehler lokalisiert (*bug location*). Abschließend wird der Fehler korrigiert (*bug correction*) und das Programm erneut evaluiert. Dieses Vorgehen kann damit als eine Variante der *isolation*-Strategie von Experten bezeichnet werden. Aufbauend auf diesem Modell untersuchten die Autoren mit Hilfe von Debuggingaufgaben, welche der respektiven Fähigkeiten gemäß dem Modell Schülerinnen und Schüler während eines 24-stündigen LOGO-Kurses „von selbst“, also ohne entsprechende Instruktion, erlernten. Dazu erhielten sie jeweils ein Bild des gewünschten Ergebnisses des LOGO-Programms (eine geometrische Figur) und den entsprechenden Code. Die Autoren stellten dabei fest, dass die Lernenden zwar aufgrund der graphischen Repräsentation in LOGO kaum Probleme hatten, festzustellen, dass ein Fehler vorlag, aber die Beschreibung des Unterschieds zwischen tatsächlichem und erwartetem Verhalten oftmals sehr ungenau und wenig zielführend war. Nach dieser Betrachtung der Unterschiede, zu der sie explizit

¹¹*Klahr und Carver (1988)* teilen den Schritt der *bug location* in *program representation* und *bug location* auf.

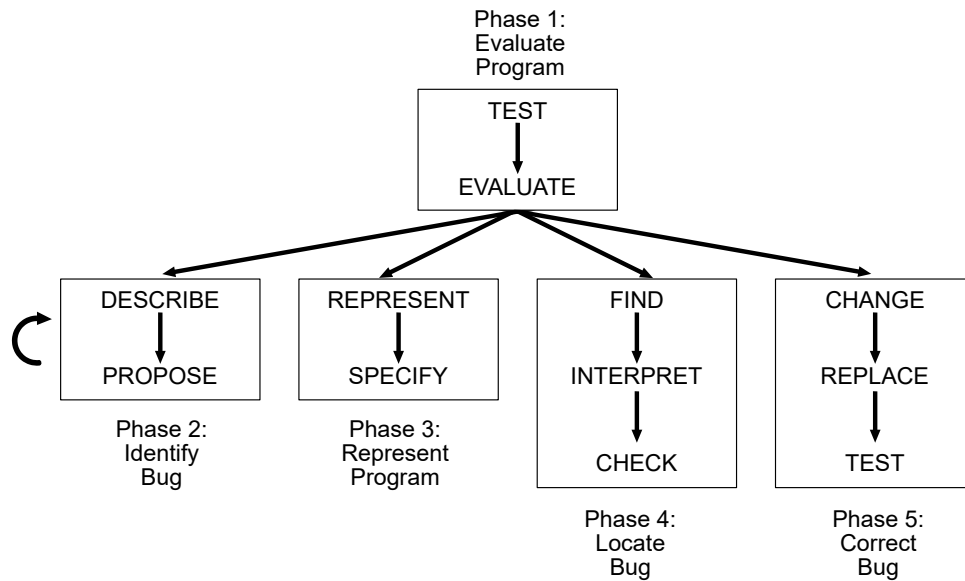


Abbildung 2.7: Modell des Debuggingprozesses in LOGO nach Klahr und Carver (1988)

aufgefordert wurden, stimmte das Verhalten die Schülerinnen und Schüler kaum mit dem entwickelten Modell überein: So nutzten sie vorhandene Indizien – die auch aufgrund der ungenauen Beschreibung des Fehlverhaltens nur eingeschränkt zur Verfügung standen – nur selten, um Vermutungen über den Fehler und seine Position aufzustellen und damit den Suchraum einzugrenzen. Daher konnten die Schülerinnen und Schüler zumeist nur seriell das Programm nachvollziehen und hatten in diesem Schritt große Probleme mit dem Verstehen des Programms. Generell benötigten sie an vielen Stellen Unterstützung durch die Versuchsleiter. Im Unterschied dazu stellte das eigentliche Beheben eines Fehlers, wenn sie ihn schlussendlich gefunden hatten, für die Schülerinnen und Schüler keine Hürde dar. Insgesamt kann das beobachtete Vorgehen damit als Variante der *comprehension*-Strategien bezeichnet werden, die auch Experten für fremde Programme einsetzen. Eine zentrale Barriere, die den Debuggingerfolg der Novizen einschränkte, bestand dabei in den Schwierigkeiten, das Programm nachvollziehen zu können.

Im Unterschied dazu untersuchten Kessler und Anderson (1986) mit Hilfe von Debuggingaufgaben Studierende, die lediglich zwei Tage Programmiererfahrung aufwiesen, um ein empirisch fundiertes Modell des Debuggingprozesses von Novizen zu entwickeln. Sie stellten dabei fest, dass sich das Vorgehen durch vier Schritte beschreiben ließ. Zunächst versuchten die Studierenden, den Code zu verstehen. Allerdings hatte die Mehrheit dabei große Probleme, die Funktionsweise des Programms korrekt zu erfassen. Als Nächstes

fürten sie das Programm aus und wurden hierdurch auf das Fehlverhalten (zumeist in Form einer falschen Ausgabe) aufmerksam. Daraufhin versuchten sie, den Fehler im Code zu lokalisieren, wozu sie wiederholt Hypothesen formulierten und verwarfen. Dies stellte dabei den schwersten Schritt für die Studierenden dar. Abschließend behoben die Probandinnen und Probanden den Fehler. Im Unterschied zu den Ergebnissen von *Carver und Klahr (1986)* hatten die Studierenden dabei größere Probleme, insbesondere weil sie erfolglose Änderungsversuche zur Behebung des Fehlers oftmals nicht rückgängig machten und so zusätzliche Fehler in das Programm einfügten. Auch in dieser Studie unterstützen die Versuchsleiter die Studierenden, indem sie bei Bedarf Hilfestellungen gaben. Vor allem hatten die Novizen die Angewohnheit, zu einem *Trial-and-Error*-Vorgehen zu wechseln, bis sie von den Versuchsleitern wieder auf den richtigen Weg gebracht wurden. Insgesamt kann das Vorgehen der Novizen – wiederum im Unterschied zu *Carver und Klahr (1986)* – der *isolation*-Strategie von Experten zugeordnet werden.

Katz und Anderson (1987) analysierten den Prozess von Studierenden beim Debuggen ihrer eigenen Programme in LISP, um Debuggingvorgehensweisen von Novizen zu identifizieren. Für die Lokalisierung des Fehlers stellten sie drei Herangehensweisen der Studierenden fest:

- *Simple mapping*: Das Fehlverhalten des Programms (wie beispielsweise eine Fehlermeldung) weist direkt auf den Fehler hin.
- *Causal reasoning*: Ausgehend von Informationen, die beispielsweise durch das Ausführen und Testen des Programms gewonnen werden, werden Hypothesen aufgestellt und auf die zugrunde liegenden Fehler geschlossen.
- *Hand simulation*: Das Programm wird sequentiell „per Hand“ ausgeführt und mit dem mentalen Modell verglichen (Schreibtischlauf).

Die Autoren bezeichnen die Strategien *simple mapping* und *causal reasoning* auch als *backward reasoning*, da der Debuggingprozess hierbei vom Fehler aus durchgeführt wird. Im Gegensatz dazu wird im Falle einer *hand simulation* vom Code ausgehend vorgegangen, weshalb es sich dabei um *forward reasoning* handelt. *Forward reasoning* entspricht damit der *comprehension*-Strategie, der im vorherigen Abschnitt für Experten identifiziert wurde, während *backward reasoning* der *isolation*-Strategie bzw. der *wissenschaftlichen Methode* entspricht. Im Folgenden werden in dieser Arbeit durchgängig die Begriffe *isolation*- bzw. *comprehension*-Strategie verwendet, auch wenn in weiteren angeführten Studien teilweise die Begrifflichkeit des *backward* bzw. *forward reasoning* bevorzugt werden. In ihrer Untersuchung stellten *Katz und Anderson (1987)* dabei fest, dass ein Vorgehen gemäß der *comprehension*-Strategie von den Studierenden nur vergleichsweise selten angewandt wurde.

Kim et al. (2018) analysierten in einer Fallstudie das Debuggingverhalten von Studentinnen für das Grundschullehramt in einer blockbasierten Programmiersprache für eigene Pro-

gramme, um Erkenntnisse für die Integration von Informatik – insbesondere bezüglich des Selbstvertrauens in der Programmierung – in die Ausbildung von Grundschullehrkräften zu gewinnen. Sie stellten dabei fest, dass die Probandinnen zunächst gemäß der *isolation*-Strategie den Output des Programms (in Form des Verhaltens eines Roboters) analysierten. Allerdings wurden anschließend kaum Hypothesen aufgestellt, sondern meist seriell (gemäß der *comprehension*-Strategie) das Programm nachvollzogen, was die Autorinnen und Autoren auf die geringe Programmiererfahrung zurückführten. Sie berichteten darüber hinaus von Probandinnen, die eher zufällig das Programm veränderten oder Teile des Codes neu schrieben.

Jayathirtha, Fields und Kafai (2020) untersuchten mit Hilfe von Think-Aloud-Protokollen die Vorgehensweise von Schülerinnen und Schülern beim Debuggen im Kontext von E-Textile-Projekten für Debuggingaufgaben. Dabei konnten sich die Fehler sowohl im Programmcode als auch der physischen Verschaltung der Komponenten befinden. Sie stellten dabei fest, dass die Schülerinnen und Schüler zunächst versuchten, das Programm (bzw. das physische System) zu verstehen (*comprehension*-Strategie). Anschließend wendeten sie, ähnlich wie Experten, ein kombiniertes Vorgehen an, bei sie häufig ausgehend vom Fehlerverhalten gemäß der *isolation*-Strategie Hypothesen aufstellten, diese überprüften, und gegebenenfalls entsprechende Codeabschnitte mit Hilfe der *comprehension*-Strategie genauer untersuchten. Beim Generieren von Hypothesen und Lösungen unterschieden die Autorinnen dabei zwischen zwei Vorgehensweisen: *depth-first* und *breadth-first*. Bei einem *depth-first*-Vorgehen stellten die Schülerinnen und Schüler schnell lediglich wenige Hypothesen auf, beispielsweise ohne das Programm ausgiebig zu testen. Entsprechend der Hypothesen versuchten sie direkt eine Lösung umzusetzen, was häufig ineffizient war. Im Gegensatz dazu formulierten andere Schülerinnen und Schüler gemäß eines *breadth-first*-Vorgehen mehrere Hypothesen, die sie überprüften, bis sie sich für eine der Hypothesen entschieden und entsprechend eine Lösung entwickelten. Neben einem *breadth-first*-Vorgehen stellte auch das häufige Verifizieren bzw. experimentelle Überprüfen der Hypothesen eine erfolgreiche Vorgehensweise dar.

Damit kann zunächst festgehalten werden, dass Novizen prinzipiell ähnliche Vorgehensweisen wie Experten zeigen. Kontrastiert man diese Ergebnisse der einzelnen Studien, sind allerdings Widersprüche im Vorgehen der Novizen festzustellen: Bei Katz und Anderson (1987) dominierte ein Vorgehen gemäß der *isolation*-Strategie, genauso wie bei Kessler und Anderson (1986). Im Unterschied dazu beobachteten Carver und Klahr (1986) und Kim et al. (2018) vorwiegend ein Vorgehen gemäß der *comprehension*-Strategie. Eine mögliche Erklärung könnte sein, ob fremde oder eigene Programme debuggt wurden, wovon auch das Vorgehen von Experten beeinflusst wird. Allerdings wurden sowohl für die beiden Studien mit der dominierenden *comprehension*-Strategie als auch für die *isolation*-Strategie in je einer Studie eigene und in der anderen fremde Programme für die Analyse verwendet. Dabei haben die Probandinnen und Probanden in den Untersuchungen, in denen entgegen der

Studie	Niveau	Methodik	Ergebnis	Programmiererfahrung
<i>Carver und Klahr (1986)</i>	Schule	Fremdes Programm	vorwiegend <i>comprehension</i> -Strategie	24 Stunden
<i>Kessler und Anderson (1986)</i>	Hochschule	Fremdes Programm	vorwiegend <i>isolation</i> -Strategie	2 Tage
<i>Katz und Anderson (1987)</i>	Hochschule	Eigenes Programm	vorwiegend <i>isolation</i> -Strategie	2 Programmierkurse
<i>Kim et al. (2018)</i>	Hochschule	Eigenes Programm	vorwiegend <i>comprehension</i> -Strategie	6 Stunden
<i>Jayathirtha, Fields und Kafai (2020)</i>	Schule	Fremdes Programm	ausgeglichen	> 20 Stunden

Tabelle 2.1: Übersicht der Ergebnisse zum Vorgehen von Novizen

Erwartung (fremdes Programm: eher *comprehension*-Strategie, eigenes Programm: eher *isolation*-Strategie) vorgegangen wurde, besonders wenig Programmiererfahrung (vgl. Tabelle 2.1)

Murphy et al. (2008), *Fitzgerald et al. (2009)* und *Fitzgerald et al. (2008)* untersuchten den Debuggingprozess von Studierenden, die bereits einen Java-Kurs erfolgreich belegt hatten, um die Strategien von Novizen zu identifizieren. Dazu mussten diese eine Debuggingaufgabe für ein Problem bearbeiten, das sie zuvor selbst implementiert hatten, und es wurden sowohl Beobachtungsprotokolle des Debuggingprozesses als auch Interviews mit den Studierenden ausgewertet. Dabei identifizierten die Autorinnen und Autoren 14 Strategien, die unterschiedlich hilfreich für den Debuggingprozess waren (vgl. Abbildung 2.8). Allerdings unterscheiden sich Vorgehen und Zielsetzung von den bisherigen Untersuchungen: Anstatt den allgemeinen Prozess zu beschreiben, werden hier einzelne Handlungen der Studierenden dargestellt, ohne diese in ihrer zeitliche Abfolge zu betrachten. Die resultierenden als Strategien bezeichneten Vorgehensweisen entsprechend damit den bisher als *Taktiken* oder auch *Methoden* bezeichneten Vorgehensweisen, die im Unterschied zu *isolation*- bzw. *comprehension*-Strategie auf einer geringeren Abstraktionsebene angesiedelt sind und ein solches allgemeines Vorgehen unterstützen können. Dies erhärtet die Feststellung, dass bisher keine einheitlichen bzw. eindeutigen Termini und Begrifflichkeiten etabliert sind. Als erfolgreiche Strategien wurden dabei beispielsweise das Lesen der Spezifikation (*gain domain knowledge*) oder Hinzuziehen weiterer Ressourcen wie der Dokumentation

(*using resources*), ein *pattern matching* basierend auf der bisherigen Programmiererfahrung, das Auskommentieren (*isolating the problem*) und Testen von Code sowie das schrittweise *Tracen* des Programmflusses (auch mit Hilfe von Werkzeugen) angewandt. Als weniger zielführende Strategien wurden unter anderem ein *Trial-and-Error-Ansatz* (*Tinkering*) und Versuche, das Problem zu umgehen, identifiziert, indem die Studierenden beispielsweise Code, den sie nicht verstanden hatten, selbst neu schrieben. Damit können die verschiedenen identifizierten Strategien teilweise einem allgemeinen Vorgehen gemäß *isolation-* oder *comprehension-*Strategie zugeordnet werden. So beschreibt etwa die Strategie des *Thinking* das Generieren von Hypothesen als einen Schritt der *isolation-*Strategie, während das *Tracen* zielführend im Rahmen der *comprehension-*Strategie angewendet werden kann. Weitere Verhaltensweisen finden sich in vergleichbarer Form bei Experten, wie beispielsweise die Verwendung des Debuggers, Testen oder aber Auskommentieren als einer Form basalen *Slicings*.

Insgesamt stellten die Autorinnen und Autoren fest, dass weniger die Auswahl der Debuggingstrategien als die Effizienz der Anwendung ausschlaggebend für den Debuggingerfolg war (*Fitzgerald et al., 2008*). So verwendeten viele der Studierenden für die Strategie des *Tracing* nur wenig aussagekräftige *Print-Statements*, hatten Probleme mit dem Formulieren alternativer Hypothesen oder testeten das Programm nur mit den Beispielwerten aus der Aufgabenstellung. Dabei beobachteten die Autorinnen und Autoren auch Fehlvorstellungen, wie etwa die Annahme, dass ein Programm fehlerfrei sei, wenn es erfolgreich kompiliert. Die Studierenden bewerteten in den Interviews dabei das Lokalisieren des Fehlers als schwersten Schritt des Debuggingprozesses. Diese Selbstwahrnehmung stimmt damit mit den Beobachtungen der bisher betrachteten Studien überein.

Munson und Schilling (2016) untersuchten speziell den Umgang mit Fehlermeldungen durch Studierende. Sie stellten dabei fest, dass nur etwa die Hälfte von ihnen gemäß der Heuristik *fix the first compiler error first* vorgingen und Studierende mit besserer Leistung eher diese Heuristik befolgten. Ein solcher ineffizienter Umgang mit Fehlermeldungen und den Informationen, die sie zur Fehlerbehebung beitragen, stellt also potentiell ein typisches Problem von Novizen dar.

Zwischenfazit

Betrachtet man also den Debuggingprozess von Novizen, verwenden diese prinzipiell ein ähnliches Vorgehen (*comprehension-* und *isolation-*Strategie) wie Experten. Allerdings gehen sie dabei oftmals ineffizient vor. So weichen sie zwischenzeitlich häufig von einem zielführenden Verhalten ab und wechseln zu einem *Trial-and-Error-Vorgehen*. Weitere typische Hürden sind eine ungenaue Beschreibung des Fehlverhaltens als Unterschied zwischen tatsächlichem und erwartetem Verhalten, der ineffiziente Umgang mit Fehlermeldungen

Category	Definition/description
Consider alternatives	Notices that one error may have multiple causes
Environmental	Takes advantage of the functionality provided by the programming/debugging environment such as the undo command; uses comments to delineate what has been done and what should happen
Gain domain knowledge	Rereads the specification or reexamines the sample output to gain insight into the problem
Isolating the problem	Comments out or alters code to isolate a problem, replaces variables with constant values, forces a specific flow of control with a constant
Just in Case	Does unnecessary things such as adding unnecessary parentheses or extra braces or fixing spelling errors in a comment
Pattern matching	'Fixes' things that do not 'look right' such as finding a missing brace by inspection
Testing	Verifies arithmetic, tests boundary conditions, predicts output and compares results with predicted values
Thinking	Pondering, reflecting on a possible cause or solution
Tinkering	Makes random and usually unproductive changes such as copying in large chunks of irrelevant code from other programs or replacing an assignment with '= '; compiles again without making any changes
Tracing	Mentally traces: Traces mentally or on paper, comparing the output with the code; Inserts print statements: Follows the flow of control or prints the value of variables; Uses debugger tool: Uses a debugger to step through the code and check the value of variables
Understanding code	Reads the code, tries to understand what the variables were used for or what the program is supposed to do
Using resources	Uses help/JavaDocs, the Java Tutorial, old programs, textbook, the Web
Using tools	Uses the debugger
Work around problem	Replaces 'obscure' code with completely new code, changes the type of loop rather than understanding why the current loop did not work, adds special cases rather than correcting the original problem.

Abbildung 2.8: Debuggingstrategien der Studierenden bei *Fitzgerald et al. (2009)*.

sowie Probleme damit, Hypothesen zu generieren und (fremden) Code zu verstehen. Die hauptsächliche Schwierigkeit scheint dabei in der Lokalisierung des Fehlers und weniger in dessen Behebung zu liegen. Dabei stellt gerade das fehlende Rückgängigmachen von erfolglosen Änderungen ein typisches Problem von Novizen dar. Auch beim Testen oder *Tracen* gehen Novizen oftmals ineffizient vor, indem sie lediglich die Beispielwerte ausprobieren oder nur wenig aussagekräftige Ausgaben verwenden.

2.2.3 Experten vs. Novizen

Novizen scheinen also prinzipiell vergleichbar zu Experten vorzugehen, haben aber Probleme mit verschiedenen Schritten in der Fehlersuche und -behebung. Der direkte Vergleich des Debuggingprozesses von Experten und Novizen für dasselbe Problem gemäß klassischer Expertiseforschung ermöglicht es nun, das Vorgehen beider Gruppen direkt gegenüberzustellen und diese Annahmen zu prüfen. Dabei sind die folgenden fünf Studien anzuführen:

Jeffries (1982) verwendete Debuggingaufgaben, um Studierende, die Programmieranfängerinnen und -anfänger sind (Novizen), mit fortgeschrittenen Studierenden (Experten) zu vergleichen. Allerdings wurde den Probandinnen und Probanden dazu lediglich das ausgedruckte Programm samt der Ergebnisse einiger Testläufe bereitgestellt. Der Autor stellte dabei fest, dass die Experten insgesamt mehr Fehler fanden. Insbesondere drei Unterschiede im Vorgehen wurden identifiziert: Zunächst versuchten alle Probandinnen und Probanden das Programm zu verstehen. Novizen hatten dabei im Unterschied zu Experten große Probleme mit der Abstraktion von einzelnen Codezeilen hin zu Sinneinheiten, während Experten viele typische Muster schnell erfassten. Weiterhin lasen die Experten das Programm in Ausführungsreihenfolge, um es zu verstehen, während die Novizen es linear von oben nach unten durchgingen. Darüber hinaus waren die Experten deutlich effizienter und genauer in der *hand simulation* des Programms. Eine weitere Beobachtung war, dass Novizen bei erfolglosen Korrekturversuchen häufig neue Fehler hinzufügten. Hier bestätigen sich also bereits identifizierte Muster, wie etwa die Probleme von Novizen mit dem Verstehen des Programms (*Carver und Klahr, 1986*), dem ineffizienten *Tracing* (*Murphy et al., 2008*) sowie der Angewohnheit, neue Fehler einzufügen (*Kessler und Anderson, 1986*).

Vessey (1985) untersuchte professionelle Programmierer beim Debuggen eines Cobol-Programms, das mit logischen Fehler versehen wurde und erneut nur in ausgedruckter Form zur Verfügung stand. Auf Basis der Beobachtungsprotokolle wurden Strategiediagramme für jeden Probanden erstellt. Die für die ex-post stattfindende Einteilung in Novizen und Experten erklärende Variable war, die Programme in aussagekräftige *Chunks* einteilen zu können. Bezüglich der Strategien stellte Vessey fest, dass sowohl Novizen als auch Experten zunächst einen *Breadth-first* Ansatz verfolgten und sich einen Überblick über das Programm

verschafften, Novizen allerdings nicht in der Lage waren, einen Gesamtüberblick über das System zu erlangen, und sich daher auf spezifische Stellen *in depth* konzentrierten. Darüber hinaus verfolgten Experten zielstrebig einen Ansatz ohne ständiges Wechseln der Vorgehensweise, zeigten sich ruhig und entspannt und eine ergebnisoffene und Alternativen in Betracht ziehende Einstellung. Auch diese Studie bestätigt damit, dass Novizen zunächst ähnlich wie Experten vorgehen, aber insbesondere Probleme mit dem Verstehen des Programms haben.

Gugerty und Olson (1986) untersuchten den Unterschied zwischen Experten und Novizen im Debuggen von Pascal und LOGO-Programmen. Ergebnis der Beobachtungen war erneut, dass Experten und Novizen zwar ähnlich vorgingen, erstere aber schneller bessere Hypothesen bildeten, da sie weniger Probleme mit dem Verständnis des Codes hatten. Sie führten das Programm auch häufiger aus. Insgesamt verwendeten Novizen und Experten zunächst dieselbe Strategie (Programm verstehen), aber Novizen brauchten dafür fast doppelt so lange. Novizen hatten außerdem die Angewohnheit, neue Fehler hinzuzufügen, da sie beim Experimentieren durchgeführte Änderungen nicht wieder rückgängig machten, wenn diese nicht zum Erfolg führten. Auch hier besteht ein Hauptunterschied also in der Fähigkeit, das Programm zu verstehen. Darüber hinaus wurden bereits beschriebene typische Probleme und Hürden wie das fehlende Rückgängigmachen erfolgloser Änderungen oder das ineffiziente Generieren von Hypothesen identifiziert.

Auch *Nanja und Cook (1987)* untersuchten Studierende beim Debuggen von Pascal-Programmen mit semantischen und logischen Fehlern. Insgesamt zeigte sich in den Beobachtungsprotokollen, dass die Experten im Unterschied zu den Novizen fast alle Fehler sehr schnell behoben und dabei weniger Änderungen vornahmen und kaum neue Fehler hinzufügten. Auch die Autorinnen und Autoren dieser Studie stellten fest, dass die Novizen das Programm nicht in Reihenfolge der Ausführung, sondern von oben nach unten lasen. Hierdurch hatten sie große Probleme, das Programm zu verstehen, und wechselten daher schnell zu einer *isolation*-Strategie mit direkter Suche nach den Fehlern – vergleichbar zu den Ergebnissen von *Vessey (1985)*. Dazu formulierten sie Hypothesen, veränderten den Code entsprechend und überprüften, ob die Änderung erfolgreich war. Die Experten gingen erst zu einem hypothesengeleiteten Vorgehen über, nachdem sie Aufbau und Struktur des Programms verstanden hatten. Insgesamt waren die Novizen bei semantischen Fehlern, für die eine Fehlermeldung – und damit entsprechende Informationen die bei der Generierung von Hypothesen helfen – bereitstand, mit der *isolation*-Strategie erfolgreicher, als bei logischen Fehlern. Novizen verwendeten auch deutlich mehr Ausgaben, um den Wert von Variablen zu überwachen, da sie ineffiziente Positionen im Code wählten. Ebenso stellten die Autoren fest, dass die Novizen die Angewohnheit hatten, neue Fehler in das Programm einzufügen, weil sie erfolglose Änderungen nicht rückgängig machten oder größere Teile des Programms neu schrieben, obwohl lediglich eine Änderung ausreichend gewesen wäre, um den Fehler zu beheben. Auch diese Untersuchung bestätigt damit die bisher identi-

zierten Unterschiede, wie etwa Probleme mit dem Verstehen des Programms, ineffiziente Platzierung von Ausgaben oder das Einfügen neuer Fehler. Darüber hinaus weisen die Ergebnisse darauf hin, dass für verschiedene Fehlertypen unterschiedliche Herangehensweisen zielführend sind, und, dass die Möglichkeit, das Programm auch ausführen zu können, das Vorgehen beeinflusst.

Yen, Wu und Lin (2012) stellten bei ihrer vergleichenden Studie zwischen professionellen Entwicklerinnen und Entwicklern sowie Studierenden fest, dass Experten die Rückmeldungen des Compilers besser verarbeiteten, während Novizen zu einer *Trial-and-Error*-Vorgehensweise neigten. Darüber hinaus wählten die Experten eher eine *comprehension*-Strategie, während die Novizen eher gemäß der *isolation*-Strategie vorgingen. Basierend auf den bisherigen Ergebnissen liegt die Vermutung nahe, dass die Novizen aufgrund der Probleme mit dem Nachvollziehen des Programms gemäß einer *isolation*-Strategie wechselten, die ohne Verständnis des Codes nicht zielführend angewandt werden kann. Auch mit dem Umgang mit Fehlermeldungen sowie *Trial-and-Error* zeigen sich erneut typische Probleme und Hürden von Novizen.

Zwischenfazit

Auch diese Studien bestätigen, dass Novizen und Experten zunächst vergleichbar vorgehen. Der zentrale Unterschied zwischen Experten und Novizen beim Debuggen zeigte sich beim *Verstehen des Programms*. Da Novizen hierbei große Probleme hatten, wechselten sie schnell von einer *comprehension*- zu einer *isolation*-Strategie, die aber ein entsprechendes Programmverständnis für zielführende Hypothesen und damit ein effizientes Debugging voraussetzt, und daher wenig erfolgreich war. Darüber hinaus erhärteten sich viele der für Novizen bereits identifizierten Probleme, wie etwa Schwierigkeiten mit dem Formulieren von Hypothesen, die Angewohnheit, neue Fehler einzufügen, mit Fehlermeldungen ineffizient umzugehen oder ein *Trial-and-Error*-Vorgehen. Auch scheinen unterschiedliche Fehlertypen einen Einfluss auf den Debuggingprozess zu haben.

2.2.4 Eigene vs. fremde Programme

Die Mehrzahl dieser (und der weiteren in diesem Kapitel beschriebenen) Untersuchungen verwendeten sogenannte *Debugging-Aufgaben*, um das Debuggingverhalten der Probandinnen und Probanden zu analysieren bzw. die Debuggingleistung zu messen. Dabei wurden Aufgaben von den Forscherinnen und Forschern vorbereitet und mit verschiedenen Fehlern versehen. Die Probandinnen und Probanden debuggten damit also nicht ihr eigenes Programm, nachdem sie auf einen Fehler aufmerksam geworden waren, sondern ein *fremdes*.

Bereits *Gould und Drongowski (1974)* betonen in einer der ersten wissenschaftlichen Studien zum Thema Debugging, dass dies einen großen Einfluss auf den Debuggingprozess haben kann. Um dieses Problem zu adressieren, können Probandinnen und Probanden beim Debuggen ihrer eigenen Programme untersucht werden (vgl. z.B. *Katz und Anderson (1987)* und *Allwood und Björhag (1990)*). Dabei wird aber die Vergleichbarkeit und Reliabilität der Ergebnisse eingeschränkt. Einen anderen möglichen Ansatz wählten z.B. *Murphy et al. (2008)*. Sie ließen die Probandinnen und Probanden zunächst ein Programm für eine Problemstellung selber implementieren und anschließend eine Debugging-Aufgabe für dasselbe Problem bearbeiten.

Tatsächlich zeigten *Katz und Anderson (1987)* in einem Experiment, dass sich die Vorgehensweisen von Studierenden unterscheiden, je nachdem ob sie ihr eigenes oder ein fremdes Programm debuggen. Die Probandinnen und Probanden benötigten signifikant weniger Zeit, um das eigene Programm initial zu lesen, und behoben auch den ersten Fehler in signifikant weniger Zeit. Tatsächlich unterschied sich aber auch das Vorgehen: Für die eigenen Programme wurde eher eine *isolation-*, für fremde eher eine *comprehension-*Strategie verwendet.

Zwischenfazit

Zusammenfassend deutet dies darauf hin, dass ob der Untersuchungsmethodik der *Debugging-Aufgaben* mit fremdem Code viele der bisher beschriebenen Ergebnisse nur eingeschränkt auf das Debuggen eigener Programme übertragbar sind: Für fremde Programme wird, wie sich auch in den bisherigen Kapiteln zeigte, eher eine *comprehension-*Strategie angewandt, während für eigene Programme, für die Struktur und Aufbau bekannt sind, eher gemäß der *isolation-*Strategie vorgegangen wird. Offensichtlich spielt aber insbesondere Programmverständnis als dem zentralen Unterschied zwischen Experten und Novizen eine deutlich größere Rolle beim Debuggen fremder Programme als beim Debuggen eigener, für die bereits ein mentales Modell des Aufbaus und der Funktionsweise existiert.

2.2.5 Implikationen

Insgesamt ist damit festzuhalten, dass in den letzten 30 Jahren der Debuggingprozess **von Schülerinnen und Schülern** – im Gegensatz zu Studierenden – nur selten systematisch untersucht wurde. Nichtsdestotrotz können die existierenden hochschuldidaktischen Erkenntnisse bezüglich typischer Probleme und Hürden von Novizen im Unterschied zum Vorgehen von Experten eine wertvolle Basis für die Ausgestaltung von Unterrichtskonzepten und -materialien liefern. Aus diesen Ergebnissen lassen sich daher für diese Arbeit drei Implikationen ziehen:

Erstens fehlt es weiterhin an einer Systematisierung der **Fähigkeiten**, die Lehrkräfte im Unterricht den Schülerinnen und Schülern vermitteln müssen, damit diese in die Lage versetzt werden, eigene Programme erfolgreich debuggen zu können. So beschränkt sich der bisherige Forschungsstand auf die Darstellung des Debuggingprozesses von Experten und Novizen oder deren Vergleich sowie einigen Empfehlungen, wie konkreten Problemen, wie etwa mit dem *Tracing*, beigegeben werden könnte. Der Fokus der informatikdidaktischen Forschung liegt damit stärker auf der Charakterisierung des Prozesses von Novizen und Experten und weniger auf der Identifikation notwendiger Fähigkeiten, die im Unterricht vermittelt werden müssen. Die hier erfolgte Betrachtung, Zusammenführung und Interpretation verschiedener Studien liefert zwar bereits einige Indizien für notwendige Fähigkeiten. So wurden zwei allgemeine Vorgehensweisen von Experten identifiziert und es konnte festgestellt werden, dass Novizen prinzipiell einen ähnlichen Ansatz wählen, diesen aber ineffizient anwenden. Darüber hinaus gibt es weitere Taktiken, Strategien, Werkzeuge, Methoden und Wissen, wie etwa über typische Fehler, die Experten einsetzen. Nichtsdestotrotz bedarf es, auch vor dem Hintergrund dieser Vielzahl an unterschiedlichen – uneinheitlichen – Begrifflichkeiten, einer systematischeren Untersuchung des Forschungsstandes und der Einbeziehung weiterer Quellen, wie etwa Programmierlehr- und Schulbüchern, um für Novizen relevante Fähigkeiten zu identifizieren.

Zweitens ist hervorzuheben, dass sich der Debuggingprozess beim Debuggen *eigener* Programme vom Debuggen *fremder* Programme unterscheidet. Ziel dieser Arbeit ist es, die Selbstständigkeit von Schülerinnen und Schülern beim Finden und Beheben von Fehlern in ihren *eigenen* Programmen zu steigern. Daher spielen die Prozesse des Programmverständnisses, die in vielen der angeführten Studien aufgrund der verwendeten *Debugging-Aufgaben* zentral sind, keine Rolle für den Debuggingprozess – obgleich das Verstehen von fremdem Code natürlich eine wichtige Kompetenz für Programmieren im Allgemeinen darstellt. Daher erscheint ein Fokus auf die Vermittlung der *isolation*-Strategie (bzw. *wissenschaftliche Methode* oder *backward reasoning*) im Unterricht zielführend. Deshalb sollte Debugging weiterhin vorwiegend in Situationen eingeübt werden, die dem Debuggen eigener Programme möglichst nahekommen.

Abschließend ist festzustellen, dass Novizen beim Debuggen zwar prinzipiell vergleichbare Vorgehensweisen wie Experten zeigen, diese aber oftmals ineffizient anwenden. Zentrale Probleme, die sich in den Untersuchungen herausstellten, waren etwa das Wechseln zu einem *Trial-and-Error*-Vorgehen, der ineffiziente Umgang mit Fehlermeldungen, die exakte Beschreibung des Fehlverhaltens oder das Generieren von Hypothesen. Diese typischen Hürden und Probleme von Novizen müssen in entsprechenden Konzepten und Materialien für den Unterricht gezielt adressiert werden.

2.3 Fehler

Im vorigen Kapitel wurde festgestellt, dass sowohl das Wissen über typische Fehler für den Debuggingprozess wichtig ist (*Ducasse und Emde, 1988*) als auch, dass für verschiedene Fehlerarten unterschiedliche Vorgehensweisen bei der Fehlersuche hilfreich sind (*Nanja und Cook, 1987*). Damit müssen für unterschiedliche Fehlerarten unterschiedliche Vorgehensweisen an Schülerinnen und Schüler vermittelt werden. Daher soll im Folgenden dieser Einfluss des zugrunde liegenden Fehlers auf den Debuggingprozess genauso wie die Bedeutung verschiedener Fehlertypen für Novizen genauer untersucht und so Auswirkungen für die Vermittlung von Debugging für den Unterricht identifiziert werden.

Wie bereits eingangs angesprochen, sind Programmierfehler auch für professionelle Entwicklerinnen und Entwickler unvermeidlich. Ein solcher *Fehler* im Programmtext, der sich im *Fehlverhalten* des Programms bemerkbar macht, kann dabei aus mehreren Gründen entstehen: Einerseits kann eine *Fehlvorstellung* der Programmiererin bzw. des Programmierers vorliegen – aber beispielsweise auch eine fehlerhafte Spezifikation, mangelndes Wissen über die Syntax der Programmiersprache, eine falsche Auffassung des Problems oder aber ein Flüchtigkeitsfehler. Tatsächlich zeigte *Eisenstadt (1997)* für professionelle Entwicklerinnen und Entwickler, dass Fehlvorstellungen bezüglich der Programmiersprache und ihrer Konzepte eher selten ursächlich für den Fehler sind. Auch bei Novizen belegten *Spohrer und Soloway (1986)*, dass Fehlvorstellungen bezüglich der Semantik der Programmierkonzepte nur eine untergeordnete Rolle spielen. Stattdessen handelt es sich zumeist um andere Probleme („*breakdowns*“), die im Entwicklungsprozess auftreten und eben die verschiedensten Ursachen haben können (vgl. *McCauley et al. (2008)* und *Ko und Myers (2005)*). Daher erscheint es im Rahmen dieser Arbeit nicht sinnvoll, sich auf die Identifikation und Behebung solcher Fehlvorstellungen zu fokussieren – auch wenn die Erforschung solcher Fehlvorstellungen eine klassische Herangehensweise (informatik-)didaktischer Forschung darstellt, und auch für das Debugging durchgeführt wurde (vgl. z.B. *Ettles, Luxton-Reilly und Denny (2018)*, *Soloway und Johnson (1984)* oder *Zehetmeier et al. (2015)*). Stattdessen sollen Konzepte und Materialien für den Unterricht entwickelt werden, die Schülerinnen und Schüler zum selbstständigen Debuggen der tatsächlichen Fehler befähigen – unabhängig davon, ob die Ursache des Fehlers eine Fehlvorstellung, ein Flüchtigkeitsfehler oder aber eine falsche Auffassung der Problemstellung ist. Daher werden im Folgenden verschiedene Kategorisierungen für Fehler bezüglich der Auswirkungen auf den Debuggingprozess und nicht im Hinblick auf deren Ursachen diskutiert.

2.3.1 Fehlerarten

Zur Unterscheidung von verschiedenartigen Fehlern existiert eine Reihe von Kategorisierungen. Diese werden im Folgenden zunächst vorgestellt, bevor der Einfluss entsprechender unterschiedlich gearteter Fehler auf den Debuggingprozess untersucht wird. Eine typische Klassifikation unterscheidet dabei syntaktische, semantische und logische Fehler (vgl. für Java z.B. *Hristova et al. (2003)* oder *Altadmri und Brown (2015)*).

- **Syntaktische Fehler:** Fehler bezüglich Rechtschreibung, Interpunktion oder Reihenfolge von Schlüsselwörtern, die die Syntax der Programmiersprache verletzen (z.B. fehlende oder falsche Klammern oder Verwendung des Zuweisungsoperators (=) statt des Vergleichsoperators (==)).
- **Semantische Fehler:** Fehler bezüglich der Bedeutung der Konstrukte der Programmiersprache (z.B. Aufruf einer nicht *static*-Methode, als wäre sie *static*, oder der Vergleich von *Strings* mit == anstatt von *.equals()*).
- **Logische Fehler:** Verhalten des Programms entspricht nicht der Spezifikation.

Johnson et al. (1983) differenzieren Fehler hingegen in *construct-* und *non-construct-related*. Erstere beziehen sich dabei auf Fehler bezüglich der Verwendung konkreter programmiersprachenspezifischer Konzepte und Konstrukte. Letztere beziehen sich auf allgemeinere Fehler, die nicht abhängig von der konkreten Programmiersprache sind und sich genauer anhand von vier Dimensionen kategorisieren lassen: Konkrete Anweisungen wie Eingaben, Ausgaben, Initialisierungen, Deklarationen, Bedingungen usw. können entweder ganz fehlen, überflüssig sein, sich an der falschen Stelle befinden oder aber falsch formuliert sein.

Eine weitere populäre Möglichkeit, Fehler zu kategorisieren, geht vom Zeitpunkt aus, zu dem das Fehlverhalten des Programms bemerkt wird (vgl. z.B. *Claus und Schwill (2003)*):

- **Kompilierzeitfehler** treten zum Zeitpunkt der Übersetzung des Programms auf und führen zu einer Fehlermeldung inklusive Fehlerbeschreibung und typischerweise Informationen zur genaueren Lokalisierung. (*Syntaxfehler und statische semantische Fehler*)
- **Laufzeitfehler** treten zur Laufzeit auf und haben den Abbruch der Programmausführung und typischerweise eine entsprechende Fehlermeldung zur Folge. (*dynamische semantische Fehler*)

¹²Teilweise auch als Unterkategorie von Laufzeitfehlern angesehen

- Bei **logischen Fehlern**¹² wird das Programm erfolgreich übersetzt und vollständig ausgeführt, allerdings entspricht das Ergebnis der Ausführung nicht der Spezifikation.

Darüber hinaus existieren weitere Kategorisierungen, wie etwa eine Unterscheidung in *konzeptionelle* und *teleologische* Fehler (Wertz, 1982) oder anhand der Aktivitäten des Programmierers bzw. der Programmiererin, etwa in der Spezifikation oder Implementierung (Ko und Myers, 2005).

Unabhängig davon, wie genau einzelne Fehlerarten nun kategorisiert werden, ist festzuhalten, dass der Debuggingprozess sich nach Fehlern unterschiedlicher Art unterscheiden kann und teils erheblich variiert. Eine zentrale Ursache dafür sind die unterschiedlichen zur Verfügung stehenden Informationen. So stehen beispielsweise für einen Kompilierzeitfehler oder Syntaxfehler eine Beschreibung des Fehlers sowie eine Zeilennummer bereit. Daneben sind in modernen IDEs direkte Hinweise auf solche Fehler sowie automatische Korrekturvorschläge enthalten. Für Laufzeitfehler oder manche semantische Fehler steht typischerweise ebenfalls eine Fehlermeldung als Information bereit, während für logische oder *non-construct-related* Fehler lediglich die fehlerhafte Ausgabe Anhaltspunkte liefern kann. Verschiedene empirische Studien belegen diese Unterschiede im Debuggingprozess für unterschiedliche Fehlertypen, vgl. z.B. Gilmore (1991), Yen, Wu und Lin (2012) und Allwood und Björhag (1990). In Konsequenz unterscheiden sich auch die Debuggingvorgehensweisen, die für die jeweiligen Fehlertypen Anwendung finden können, alleine schon abhängig davon, ob das zu debuggende Programm überhaupt lauffähig ist und daher dynamisch oder lediglich statisch analysiert werden kann. So ist offensichtlich das Werkzeug Debugger für Syntaxfehler nicht geeignet. Daher werden in manchen Definitionen von Debugging und gerade auf einem universitären Niveau solche Kompilierzeitfehler sogar ausgeschlossen (vgl. z.B. Ko und Myers (2005)) und Debugging erst beginnend mit Laufzeitfehlern aufgefasst.

Zwischenfazit

Zusammenfassend ist damit festzustellen, dass eine große Anzahl von Klassifikationen für Fehler existiert. Zentral ist dabei weniger die konkrete Klassifikation als vielmehr die Erkenntnis, dass das Debuggingvorgehen für verschiedene Fehlerarten variiert, weil unterschiedliche Informationen (wie etwa eine Beschreibung des Fehlers oder gar eine Zeilennummer) zur Verfügung stehen und unterschiedliche Herangehensweisen zur Fehlersuche angewandt werden können (statische vs. dynamische Analyse). Schülerinnen und Schülern müssen also für unterschiedliche Arten von Fehlern jeweils angepasste Vorgehensweisen vermittelt werden. Grundlage dafür ist es, dass die Lernenden auch eine entsprechend

geeignete Kategorisierung für Fehler kennen, um damit abhängig vom Fehlertyp eine adäquate Vorgehensweise auswählen zu können.

2.3.2 Bedeutung für Novizen

Eine Konsequenz der Unterschiede bezüglich der zur Verfügung stehenden Informationen und anwendbaren Herangehensweisen für verschiedenartige Fehler ist es, dass gewisse Fehlerarten schwerer zu beheben sind als andere (vgl. z.B. *Gilmore (1991)*). Wie gerade dargelegt, wird dabei insbesondere das Finden und Beheben von Kompilierzeit- oder Syntaxfehlern teilweise nicht als Debugging aufgefasst. Für die Entwicklung von Ansätzen für den Informatikunterricht muss damit geklärt werden, welche Fehlerarten Schülerinnen und Schüler vor Probleme stellen, um damit entsprechende Herangehensweisen zu vermitteln, die für diese Fehlerarten hilfreich sind. Gerade die Rolle der Syntax- oder Kompilierzeitfehler im Informatikunterricht muss damit geklärt werden.

Dabei ist es aus mehreren Gründen schwierig, quantitative Aussagen bezüglich der Häufigkeit von unterschiedlichen Fehlern auf Basis des Forschungsstandes zu treffen. Zunächst konzentrieren sich Untersuchungen oftmals nur auf bestimmte Fehlerarten. Zumeist sind das die Fehler, die durch eine statische Analyse des Programms automatisiert erkannt und ausgewertet werden können (vgl. z.B. *McCall und Kölling (2014)*), oder es werden direkt Compilermeldungen analysiert (vgl. z.B. *Jadud (2005)*). Allerdings können solche Compilermeldungen oftmals nicht eindeutig einem konkreten Fehler zugeordnet werden: Eine Fehlermeldung kann in gewissen Programmiersprachen von mehr als einem Fehlertyp verursacht werden, genauso wie ein konkreter Fehlertyp zu mehreren Fehlermeldungen führen kann, je nach Kontext (*McCall und Kölling, 2014*). Weiterhin weisen viele der Studien keine angemessene Stichprobengröße auf, die eine Verallgemeinerung entsprechender Ergebnisse ermöglicht (vgl. z.B. *Spohrer und Soloway (1986)*), oder sie analysieren lediglich den finalen Zustand der Programme und eben nicht die Fehler, die während des Entwicklungsprozesses gemacht werden (vgl. z.B. *Hall et al. (2012)*). Außerdem untersucht erneut die Mehrheit dieser Studien Studierende und nicht etwa Schülerinnen und Schüler.

Im Folgenden werden dennoch existierende Erkenntnisse dargestellt, um Indizien bezüglich der Häufigkeiten verschiedener Fehlertypen herauszuarbeiten. *Hall et al. (2012)* kommen in ihrer Untersuchung der Hausaufgaben von Studierenden zu dem Ergebnis, dass logische Fehler am häufigsten auftreten. Auch *Spohrer und Soloway (1986)* stellen fest, dass entgegen der landläufigen Meinung Fehler, die *non-construct-related* sind, von Studierenden viel häufiger gemacht werden als solche, die *construct-related* sind. *Altadmri und Brown (2015)* analysieren die Häufigkeit verschiedener Fehlertypen basierend auf dem BlueJ-Blackbox-Datensatz, der sowohl die Daten von Schülerinnen und Schülern als auch Studierenden enthält. Die Autoren kommen zu dem Schluss, dass in Summe semantische Fehler häufiger

auftreten als Syntaxfehler – zumindest ab einem gewissen Grad an Programmierkenntnissen. Nichtsdestotrotz ist in ihrer Untersuchung ein Syntaxfehler („Klammerfehler“) der am häufigsten identifizierte Fehler.

Betrachtet man nicht nur die Häufigkeit, sondern die „Schwere“ eines Fehler, kann diese anhand der Zeit gemessen werden, die benötigt wird, um ihn zu beheben (*Altadmri und Brown, 2015*). Dabei ist festzustellen, dass Syntaxfehler im Allgemeinen schnell behoben werden. Für die Lernenden erscheint es schwieriger, vom Compiler nicht erkannte Fehler zu finden. Für solche Fehler benötigen Novizen deutlich länger, um sie aufzuspüren und zu beheben (*Chen und Lim, 2013; Altadmri und Brown, 2015; Murphy et al., 2008*).

Zwischenfazit

Zusammenfassend ist damit festzustellen, dass sich auf Basis des Forschungsstandes keine abschließende Antworten bezüglich der Häufigkeit und Schwere von Fehlern für Novizen geben lassen. Ursache dafür sind insbesondere die vielen verschiedenen Erhebungsmethoden und fehlende Untersuchung logischer Fehler. Die existierenden Ergebnisse deuten darauf hin, dass Syntax- und Kompilierzeitfehler tatsächlich ein eher geringes Problem darstellen, insbesondere weil sie schnell behoben werden. Allerdings stammen die Ergebnisse vorwiegend aus der hochschuldidaktischen Forschung und lassen sich möglicherweise nur eingeschränkt auf Schülerinnen und Schüler übertragen. Um Ansätze für den Unterricht zu entwickeln, sollte daher insbesondere die Bedeutung von Kompilierzeitfehlern für Schülerinnen und Schüler untersucht werden, um festzustellen, ob auch für solche Fehler Hilfestellungen benötigt werden, oder Kompilierzeitfehler für diese Arbeit tatsächlich nicht als Debugging aufgefasst werden sollten.

2.3.3 Implikationen

Insgesamt zeigt sich damit, dass das Debuggingverhalten abhängig vom zugrunde liegenden Fehler teils erheblich variiert, insbesondere aufgrund der unterschiedlichen zur Verfügung stehenden Informationen und anwendbaren Vorgehensweisen. Für diese Arbeit bedeutet das, dass Schülerinnen und Schülern für verschiedene Fehlertypen jeweils angepasste Vorgehensweisen vermittelt werden müssen. Dazu muss ebenfalls eine geeignete Fehlerkategorisierung eingeführt werden, damit die Lernenden, abhängig von der vorliegenden Art des Fehlers, entsprechend adäquate Vorgehensweisen auswählen können.

Darüber hinaus unterscheiden sich verschiedenen Fehlerarten hinsichtlich ihrer Bedeutung für Novizen. Für die Entwicklung von Ansätzen für den Informatikunterricht muss daher geklärt werden, welche Fehlerarten Schülerinnen und Schüler vor Probleme stellen, um

gezielt entsprechende Herangehensweisen bereitzustellen – und ob beispielsweise Kompilierzeitfehler überhaupt adressiert werden müssen. Der vorwiegend hochschuldidaktische Forschungsstand kann dazu keine abschließende Antworten geben, weswegen gerade die Frage nach der Bedeutung von Kompilierzeitfehlern im Rahmen dieser Arbeit geklärt werden sollte.

2.4 Lernende

Für die Entwicklung von Konzepten und Materialien für Debugging im Unterricht ist es zentral, auch die Perspektive der Schülerinnen und Schüler zu betrachten. Existierende Untersuchungen – wiederum fast ausschließlich in der hochschuldidaktischen Forschung – konzentrieren sich dabei insbesondere auf die affektiven Einstellungen sowie konkrete kognitive Lernvoraussetzungen. Im Folgenden sollen diese beiden Dimensionen nun untersucht werden.

2.4.1 Affektive Einstellungen der Lernenden

Selbstwirksamkeitserwartungen (*Bandura, 1982*) determinieren Anstrengung, Ausdauer und Belastbarkeit für eine Aufgabe (*Pajares, 1996*) und sind damit zentral für die Selbstständigkeit im Debugging. Dabei existiert ein Zusammenhang zwischen Selbstwirksamkeitserwartungen und den affektiven Einstellungen gegenüber einer Aufgabe (*Kundu und Ghose, 2016*), die hauptsächlich durch Erfolgserlebnisse beeinflusst werden. Daher soll im Folgenden der Forschungsstand zu den affektiven Einstellungen von Lernenden gegenüber dem Debugging untersucht werden, um Schlussfolgerungen für die Steigerung der Selbstständigkeit in der Fehlerbehebung ziehen zu können.

Die affektive Komponente der Einstellung von Schülerinnen und Schülern im Kontext des Debuggens von Programmen ist – ebenso wie deren Vorgehen – nur selten erhoben oder untersucht worden. Erweitert man die Betrachtung auf Erkenntnisse aus der Hochschulforschung, stellen *Lahtinen, Ala-Mutka und Järvinen (2005)* in ihrer Online-Befragung von Studierenden verschiedener europäischer Hochschulen fest, dass die Studierenden das Finden und Beheben von Bugs in ihren Programmen als schwersten Teil des Programmierenlernens bewerten.

Kinnunen und Simon (2010) führten wiederholt semi-strukturierte Interviews mit Programmieranfängerinnen und -anfängern einer Universität durch. Ergebnis der qualitativen Auswertung war, dass die Studierenden sowohl mit der Wahrnehmung eines Fehlers als auch mit dessen Behebung viele negative emotionale Reaktionen wie insbesondere Frustrati-

on verbanden. Vornehmlich waren die Studierenden häufig von den Fehlern überrascht, da sie davon ausgingen, gerade alles richtig zu machen. In Konsequenz hatten sie meist keine Idee, wie sie bei der Beseitigung dieses Fehlers vorgehen könnten. Außerdem hatten die Fehler oftmals eine negative Auswirkung auf ihre Selbstwirksamkeitserwartungen, und Fehler bzw. die Probleme in der Fehlerbehebung wurden als persönliches Versagen aufgefasst.

Die Einstellungen von Schülerinnen und Schülern untersuchten *Perkins et al. (1986)*. Auf Basis von strukturierten Beobachtungen teilten sie Lernende in *stoppers* und *movers* ein. *Stoppers* sind nicht in der Lage weiterzuarbeiten, wenn sie mit einem Problem – also beispielsweise einem Fehler – konfrontiert sind, und verlieren alle Hoffnung, das Problem selbst lösen zu können. Im Gegensatz dazu experimentieren *movers* und versuchen weiter, ihren Code anzupassen. Weiterhin stellen *Perkins et al. (1986)* fest, dass Fehler für viele Schülerinnen und Schüler frustrierend sind, insbesondere weil sie unmittelbar durch eine Fehlermeldung, ein abstürzendes Programm oder eine falsche Ausgabe angezeigt werden. Auch hier werden sie nicht als Chance wahrgenommen, sondern vielmehr als persönliches Versagen gewertet, wie folgende Beobachtung verdeutlicht.

As a researcher walked by, one student hurriedly cleared his computer screen in order to conceal his program's erroneous output.

Darüber hinaus berichten die Autoren davon, dass auffällig viele Schülerinnen und Schüler schnell bei der Behebung eines Fehlers aufgaben. Nachdem sie allerdings von der Forscherin oder dem Forscher zu erneuten Versuchen ermutigt worden waren, konnten sie das Problem weitestgehend durchaus selbstständig lösen.

Auch *Papert (1980)* berichtet anekdotisch davon, dass Schülerinnen und Schülern oftmals, anstatt zu debuggen, lieber erneut mit der Implementierung beginnen. Er führt dies auf die schulische Fehlerkultur zurück:

What we see as a good program with a small bug, the child sees as "wrong," "bad," "a mistake." School teaches that errors are bad; the last thing one wants to do is pore over them, dwell on them, or think about them. The child is glad to take advantage of the computer's ability to erase it all without any trace for anyone to see. The debugging philosophy suggests an opposite attitude. Errors benefit us because they lead us to study what happened, to understand what went wrong, and, through understanding, to fix it.

Zwischenfazit

Zusammenfassend zeigt sich damit, dass Programmierfehler und das Finden und Beheben dieser für Lernende oftmals mit großer Frustration einhergehen, sodass sie vielfach

vorschnell aufgeben, auch wenn sie womöglich erfolgreich sein könnten. Fehler sind häufig negativ konnotiert und werden als persönliches Versagen mit einem entsprechendem Effekt auf die Selbstwirksamkeitserwartungen wahrgenommen. Um also die Selbstständigkeit von Schülerinnen und Schülern bei der Fehlerbehebung zu fördern, erscheint es zentral, Fehler als „Normalität“ der Programmierung zu etablieren und eine entsprechende positive Fehlerkultur im Unterricht zu schaffen.

2.4.2 Lernvoraussetzungen

Gemäß der Lerntheorie des Konstruktivismus ist Lernen ein ständiger und aktiver Prozess der Anpassung bereits existierender mentaler Modelle, indem neue Erfahrungen gemacht und reflektiert werden (*Phillips, 1995*). Solche – oftmals lebensweltlichen und vorunterrichtlichen – Erfahrungen werden auch als *individuelle Lernvoraussetzungen* bezeichnet (*Prenzel und Doll, 2002*) und müssen berücksichtigt werden, um im Sinne des Konstruktivismus geeignete Konzepte und Materialien für den Unterricht zu entwickeln.

Simon et al. (2008) untersuchten entsprechende Lernvoraussetzungen für das Debugging. Dazu identifizierten sie bereits aus der Lebenswelt ausgeprägte Troubleshooting-Erfahrungen von Studierenden, die sie als *debugging preconceptions* bezeichneten. Ziel der Untersuchung war es, für die gefundenen Ansätze und Strategien zu überprüfen, ob sich diese auf die Programmierung übertragen und sich damit Unterschiede im Debuggingverhalten zwischen Novizen und Experten erklären lassen. Dazu wurden den teilnehmenden Studierenden die vier folgenden Szenarien beschrieben:

- light bulb: Erkläre einem Gast in deiner Wohnung via E-Mail, was er tun kann, wenn das Licht im Schlafzimmer nicht mehr funktioniert.
- telephone: Wie könnte man beim Spiel Flüsterpost herausfinden, wann der erste Fehler passiert? Wie könnte man das Spiel modifizieren, damit keine Fehler mehr auftreten?
- coffee: Wie finde ich in einer fremden Stadt, ohne dass ich die Sprache spreche, den nächsten Starbucks?
- real life: Beschreibe mindestens zwei Beispiele aus deinem Alltag, bei denen du *troubleshooten* musstest. Beschreibe auch, wie du dabei vorgegangen bist.

Die Studierenden mussten ihre Reaktion und ihr Vorgehen für diese Szenarien schriftlich festhalten. Anschließend analysierten die Autorinnen und Autoren die Antworten auf gemeinsame Merkmale hin. Sie stellten fest, dass einige der angewandten Vorgehensweisen denen von Debugging-Experten ähnelten, wie beispielsweise zunächst zu testen

(auch wenn die Studierenden zumeist einen Test direkt mit einer Reparatur/Korrektur kombinierten), Hypothesen aufzustellen oder das Verfolgen eines strukturierten Plans. Einige der identifizierten Strategien hatten im Gegensatz dazu keine Bedeutung für das Debugging, wie beispielsweise gewisse Handlungen erneut durchzuführen (im Sinne von „*have you tried turning it on and off again*“). Darüber hinaus zeigten sich viele Strategien von Debugging-Experten nicht oder nur selten in den Antworten, wie beispielsweise das System zunächst zu analysieren und verstehen oder das Rückgängigmachen erfolgloser Reparaturversuche (*undo*).

Die Autorinnen und Autoren ziehen daraus einige Hinweise für die Unterrichtspraxis. So sollten Strategien wie *undo* explizit eingeübt, der Unterschied zwischen dem Finden und dem Beheben von Fehlern verdeutlicht, der Wert des Testens ohne gleichzeitige Korrekturen (*test-only*) betont und die Formulierung mehrerer Alternativhypothesen für einen Fehler geübt werden. Sie stellen abschließend fest, dass Debugging weniger *gesunder Menschenverstand* ist als andere informatische Konzepte, wie das Sortieren oder Parallelität, und viele Strategien von Debugging-Experten in den identifizierten *preconceptions* fehlen.

Zwischenfazit

Zusammenfassend zeigt die Untersuchung von Lernvoraussetzungen von Studierenden, dass Lernende beim Debuggen teilweise auf Erfahrungen aus dem *Troubleshooting* zurückgreifen. Dies kann helfen, gewisse Verhaltensweisen von Novizen beim Debuggen (vgl. Kapitel 2.2.2) zu erklären, wie beispielsweise die Angewohnheit, erfolglose Änderungen nicht rückgängig zu machen. Allerdings fehlt es an Erkenntnissen bezüglich der Lernvoraussetzungen von Schülerinnen und Schülern und es werden in der existierenden Untersuchung vergleichsweise allgemeine Szenarien genutzt und nicht für das Debugging relevante *Troubleshooting*-Vorgehensweisen und deren entsprechende Charakteristika gezielt untersucht.

2.4.3 Implikationen

Insgesamt kann für die Untersuchung der Lernenden damit festgehalten werden, dass Debugging als schwere und frustrierende Aktivität wahrgenommen wird und sie vielfach vorschnell aufgeben. Um die Selbstwirksamkeitserwartungen von Schülerinnen und Schülern zu steigern, sollte eine Kultur der Wahrnehmung von Programmierfehlern als „Normalität“ der Programmierung und zeitgleich positiv konnotierte Lerngelegenheiten geschaffen werden. So können Beharrlichkeit und schlussendlich Selbstständigkeit in der Fehlerbehebung gesteigert werden.

Darüber hinaus wurden für Studierende Debugging-Lernvoraussetzungen auf Basis ihrer Troubleshooting-Erfahrung identifiziert. Solche Lernvoraussetzungen tragen einerseits dazu bei, das Verhalten von Novizen beim Debugging zu erklären, und sind andererseits gemäß der Lerntheorie des Konstruktivismus zentral für die Gestaltung von Konzepten und Materialien für den Informatikunterricht. Allerdings beschränkt sich der bisherige Forschungsstand auf Studierende, für die aufgrund des Altersunterschiedes ein unterschiedliches Maß an Troubleshooting-Erfahrungen aus ihrem täglichen Leben zu erwarten ist. Gleichzeitig werden in der angeführten Untersuchung eher allgemeine Troubleshooting-Kontexte gewählt und nicht gezielt für das Debugging tatsächlich relevante Vorgehensweisen analysiert. Um entsprechende Lernvoraussetzungen zu identifizieren, die für die Vermittlung von Debugging an Schülerinnen und Schüler wesentlich sind, müssen daher weitere Forschungsaktivitäten unternommen werden.

2.5 Unterricht

Wie bereits festgestellt, unterscheiden sich Debuggingfähigkeiten von Programmierfähigkeiten und entwickeln sich nur eingeschränkt „von selbst“ (Carver und Klahr, 1986). Beispielweise folgern Murphy et al. (2008) oder Kessler und Anderson (1986) daraus, dass Debuggingfähigkeiten explizit unterrichtet werden sollten. Nichtsdestotrotz gibt es überraschend wenige Studien sowohl in Bezug auf universitäre Lehre als auch auf Unterricht, die die explizite Vermittlung von Debugging untersuchen. Diese werden im Folgenden beschrieben, um daraus einerseits existierende *methodische Ansätze*, und andererseits die konkret vermittelten *Inhalte* zu identifizieren, die damit eine Basis für die Entwicklung von Konzepten und Materialien für den Unterricht darstellen können. Dabei ist die Darstellung in Ergebnisse aus der Hochschuldidaktik, für den Unterricht sowie werkzeuggestützte Ansätze gegliedert.

2.5.1 Debuggen in der Hochschullehre

Chmiel und Loui (2004) verwendeten freiwillige Debuggingaufgaben und *debugging logs*, um die Debuggingfähigkeiten von Studierenden zu fördern. Es stellte sich heraus, dass Studierende, die die freiwilligen Debuggingaufgaben bearbeitet hatten, mit Fortschreiten des Semesters signifikant weniger Zeit für das Debuggen ihrer eigenen Programme benötigten als jene, die die Aufgaben nicht bearbeitet hatten. Dieser Zusammenhang spiegelte sich jedoch nicht in den Klausurergebnissen wider, die entgegen den Erwartungen nur geringfügig besser waren. Auf ähnliche Weise setzten Ahmadzadeh, Elliman und Higgins (2007) verpflichtende Debuggingaufgaben ein. Im Unterschied zu vorher erfolgten Durchläufen

des Kurses machten die Studierenden auch hier weniger Fehler, die Klausurergebnisse unterschieden sich aber ebenfalls nicht signifikant. Hier wurden also für die Lernenden Gelegenheiten geschaffen, Debugging zu üben, ohne Vorgehensweisen oder Ähnliches explizit zu vermitteln.

Im Unterschied dazu untersuchten *Katz und Anderson (1987)* den Effekt der Vermittlung verschiedener Vorgehensweisen. Dazu setzten sie auf eine selbstentwickelte Benutzeroberfläche, die unterschiedliche Vorgehensweisen erzwingen konnte: Eine Gruppe Studierender konnte die zu debuggenden Lisp-Programme lediglich Zeile für Zeile gemäß dem Programmfluss untersuchen (*comprehension-Strategie*). Eine andere Gruppe konnte sich gemäß der *isolation-Strategie*, ausgehend von der falschen Ausgabe, dafür entscheiden, verschiedene vorgegebene Teile des Codes zu überprüfen, etwa ob eine Variable am Ende der Funktion mit der korrekten Antwort übereinstimmt. Eine dritte Gruppe (*neutral strategy*) konnte frei durch das Programm navigieren. Im zweiten Teil des Experiments konnten alle Gruppen gemäß den Bedingungen der *neutral strategy* ohne Einschränkungen arbeiten. Dabei zeigte sich, dass Studierende weiterhin das ihnen vermittelte Vorgehen anwendeten. Ansonsten wurden kaum Unterschiede zwischen den Strategien festgestellt. Entsprechende Vorgehensweisen können also explizit vermittelt werden.

Allwood und Björhag (1991) untersuchten in einem Experiment, inwieweit schriftliche Debugging-Hinweise Studierende unterstützen können. Dazu stellten sie einer Experimentalgruppe schriftliche Anhaltspunkte zur Verfügung, die ein systematisches Vorgehen für die Behebung von Fehlern gemäß der *isolation-Strategie* beschrieben. Die Experimentalgruppe hatte Zeit, die Hinweise zu lesen und im Rahmen von Debuggingaufgaben einzüben. Die Kontrollgruppe implementierte zeitgleich die Debuggingaufgaben. In einer anschließenden Testphase mussten beide Gruppen ein weiteres Programm implementieren. In den Screenrecordings und der Aufzeichnung der „think-aloud“-Kommentare zeigte sich, dass sich die Anzahl der gemachten Fehler zwischen Versuchs- und Kontrollgruppe nicht signifikant unterschied, während die Anzahl der beseitigten Fehler (insbesondere semantischer und logischer Art) bei Verfügbarkeit schriftlicher Hinweise signifikant höher war. Da gleichzeitig keine Unterschiede in den verwendeten Vorgehensweisen zwischen den Gruppen erkennbar waren, folgerten die Autoren, dass die Unterschiede auf einer höheren Ebene liegen müssen und vor allem die Anwendung eines systematischen Vorgehens beim Debuggen entscheidend sei. Analog zu *Katz und Anderson (1987)* zeigt sich damit also, dass ein solches gezieltes Vorgehen für erfolgreiches Debugging vermittelt werden kann.

Auch *Böttcher et al. (2016)* vermittelten ein systematisches Debuggingvorgehen angelehnt an die *isolation-Strategie* sowie die Verwendung des Debuggers in einer expliziten Einheit. Dazu mussten die Studierenden zunächst einen Text zum Debuggen lesen und Fragen beantworten. Anschließend folgte eine Live-Coding-Demonstration des im Text vorgestellten Vorgehens zum Debuggen (vgl. Abbildung 2.9), ehe die Studierenden Debuggingaufgaben

bearbeiteten und ihr Vorgehen dokumentierten. Die Auswertung ergab, dass nur wenige Studierende das vermittelte Vorgehen wie gefordert umsetzten und dass sie stattdessen schnell wieder zu einem unsystematischen „Herumstochern“ übergingen. Im Gegensatz zu den bisherigen Studien sind hier also ernüchternde Ergebnisse für die Vermittlung eines systematischen Vorgehens festzuhalten. Dabei wurde in dieser Studie die Anwendung dieses Vorgehen nicht solcherart erzwungen wie bei *Katz und Anderson (1987)*, allerdings war der Ansatz von *Allwood und Björhag (1991)* ähnlich fakultativ.

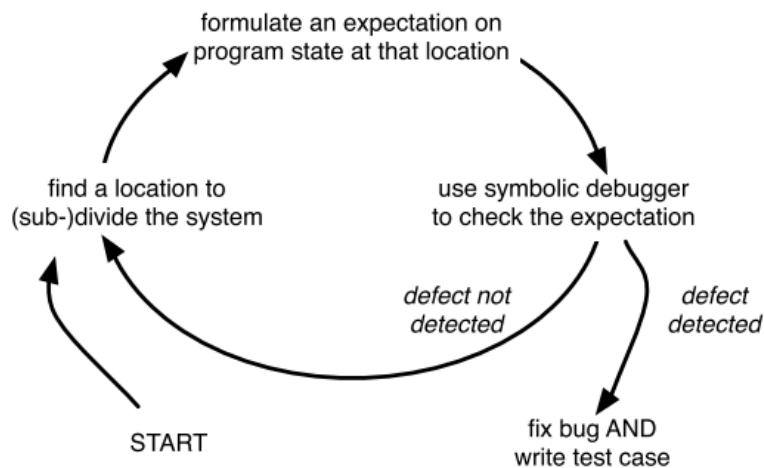


Abbildung 2.9: Debuggingvorgehen nach *Böttcher et al. (2016)*

Eranki und Moudgalya (2016) vermittelten Nicht-Informatik-Studierenden das *Slicing* mit dem Ziel, die Programmierfähigkeiten und das Programmverständnis von Novizen zu verbessern. Zum Einüben verwendeten sie Aufgaben, bei denen die Studierenden Code in die richtige Reihenfolge bringen und teilweise fehlende Zeilen ergänzen mussten. Im Vergleich von Experimental- und Kontrollgruppe stellten die Autoren bei Vermittlung der Strategie einen positiven Effekt auf die Debuggingleistung fest. Auch hier wurde also eine Debuggingvorgehensweise erfolgreich explizit vermittelt. Insgesamt ist festzustellen, dass alle bisherigen Ansätze für die Einübung der jeweils vermittelten Vorgehensweisen Debugging-Aufgaben einsetzen.

¹³Eine zusammenfassende Darstellung beider Teilstudien findet sich bei *Klahr und Carver (1988)*.

2.5.2 Debuggen im Informatikunterricht

Neben dem Hochschulbereich gibt es auch einige Untersuchungen, die sich auf Kinder und Jugendliche im Informatikunterricht bzw. in außerschulischen Projekten fokussieren. Im Folgenden sollen die relevanten Erkenntnisse dieser Studien vorgestellt werden.

Carver (1986) bzw. Carver und Risinger (1987)¹³ vermittelten einen Debuggingprozess angelehnt an der *isolation*-Strategie für LOGO mit vielversprechenden Ergebnissen: Sie entwickelten ein schrittweises Debuggingvorgehen für Schülerinnen und Schüler ohne Programmiererfahrung in LOGO (vgl. Abbildung 2.10 bzw. Kapitel 2.2.2). In zwei Experimenten gaben sie den Schülerinnen und Schülern eine Stunde Debugging-Training als Teil eines größeren LOGO-Curriculums, in der das Vorgehen explizit vermittelt wurde. Darüber hinaus setzten sie *debugging logs* ein, die während der gesamten Zeit im Klassenzimmer vorhanden waren. Mit Hilfe von Debuggingaufgaben wurde an mehreren Zeitpunkten im Schuljahr Debuggingleistung und -verhalten erhoben. Die Ergebnisse (ohne Kontrollgruppe) zeigten in beiden Experimenten einen Wechsel von unsystematischem Ausprobieren hin zu einem systematischen Vorgehen bei der Suche nach Fehlern. Obendrein wurde für die Fehlersuche deutlich weniger Zeit benötigt. Die Schülerinnen und Schüler formulierten vor dem Ausprobieren des Codes mehr Hypothesen, achteten stärker auf den Kontrollfluss, nahmen weniger Code-Änderungen vor (insbesondere an fehlerfreien Stellen) und machten weniger neue Fehler. Erneut wurde damit ein systematisches Vorgehen mit vielversprechenden Ergebnissen eingeführt.

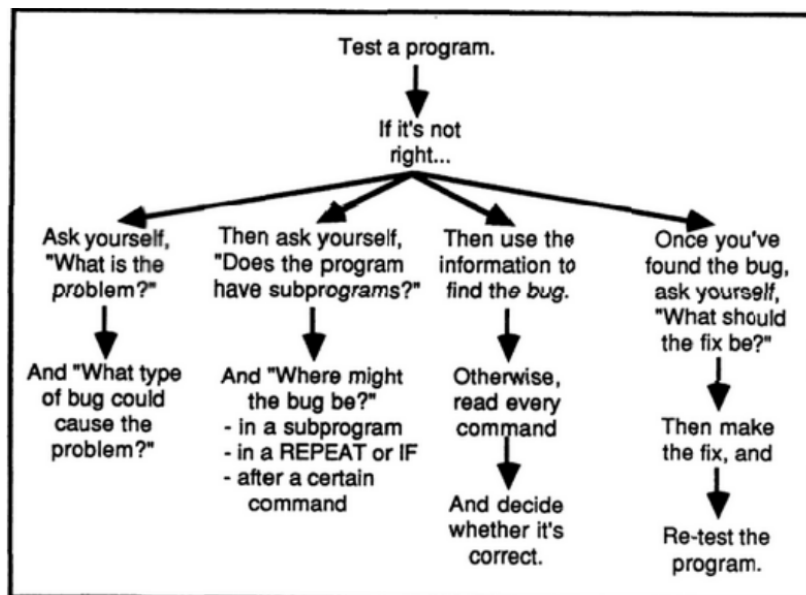


Abbildung 2.10: Schrittweises Debuggingvorgehen nach Carver und Risinger (1987)

Lui et al. (2017) entwickelten im Kontext von E-Textiles *reconstruction kits*, die die einfache und schnelle Verwendung von verschiedenen Sensoren und Aktoren mit dem LilyPad Arduino ermöglichen (vgl. auch Fields, Searle und Kafai (2016)). Diese setzten sie im Rahmen von Workshops als als *Debug-its* bezeichnete Debugging-Aufgaben ein. Im Unterschied zu regulären Debugging-Aufgaben können sich hier Fehler entweder in der physischen Verschaltung oder im Programmcode befinden. Zum einen gab es durch die Autorinnen vorgegebene *Debug-its*, zum anderen erstellten die Schülerinnen und Schüler diese selbst. Die Wissenschaftlerinnen stellten dabei fest, dass insbesondere kollaboratives Arbeiten der Schülerinnen und Schüler und das Erstellen von eigenen *Debug-its* wertvolle Lerngelegenheiten boten. Die Schülerinnen und Schüler zeigten dabei ein hohes Interesse, und betonten – trotz anfänglicher Frustration – mehr Selbstvertrauen, ein besseres Wissen über typische Fehler, als auch Strategien zur Fehlerbehebung erworben zu haben (Fields und Kafai, 2020). Damit erweitern die Autorinnen das Üben von Debugging anhand von Debugging-Aufgaben um eine kreative Komponente, in der die Schülerinnen und Schüler aktiv über Fehler reflektieren, indem sie selbst eigene solche Aufgaben gestalten.

DeLiema et al. (2019) verwendeten im Rahmen von mehrtägigen Programmierworkshops eine Tagebuchmethode, sodass die Schülerinnen und Schüler zu Beginn jedes Tages über ihre bisher eingesetzten Strategien¹⁴ zum Debugging reflektierten und sich für den Tag den Einsatz einer weiteren Strategie vornahmen. Darüber hinaus setzten sie Aktivitäten ein, bei denen die Schülerinnen und Schüler ihre Emotionen beim Debuggen mit Kunst verarbeiteten, indem diese etwa Gedichte, Gemälde oder Comics gestalteten. In ihrer Pre-Post-Auswertung ohne Kontrollgruppe stellten die Autorinnen und Autoren fest, dass sich die Selbsteinschätzung der Debuggingfähigkeiten der Lernenden im Verlauf des Workshops gesteigert hatte. Auch hier wird also insbesondere das eigene Vorgehen beim Debugging reflektiert.

2.5.3 Werkzeugbasierte Ansätze

Einen traditionellen Ansatz zur Vermittlung und Aufbereitung von Inhalten für den Informatikunterricht stellen Werkzeuge dar. Insbesondere zum Programmierenlernen ist der Einsatz geeigneter Werkzeuge, oftmals auch mit Gamification-Elementen, weit verbreitet (vgl. z.B. Malmi, Utting und Ko (2019)). Tatsächlich existiert eine Vielzahl solcher Werkzeuge auch für das Debugging. Auch deren Analyse kann Aufschluss geben, *welche* Debuggingfähigkeiten *wie* vermittelt werden. Dabei lassen sich zwei unterschiedliche Zielsetzungen feststellen:

¹⁴Hier weiter gefasst als nur *Debuggingstrategien* im engeren Sinne, sondern als *critical thinking strategies*, wie beispielsweise „fokussiert bleiben“.

Einen Teil dieser Werkzeuge eint, dass das zentrale Ziel nicht die Vermittlung von Debuggingfähigkeiten ist, sondern die Vermittlung von Programmierfähigkeiten (wie Lee *et al.* (2014), Lee und Ko (2011) und Long (2007)). Debugging bzw. eine *Debugging-first*-Herangehensweise wird lediglich als *Zugang* zum Programmierenlernen gewählt. Dies begründen die Autorinnen und Autoren damit, dass das selbstständige Erstellen eigener Programme sehr komplex sei (vgl. z.B. auch Edwards (2004) oder Lowe (2019)). Statt also mit dem Schreiben eigener Programme zu beginnen, werden zunächst kleine, vorgegebene Programme – zumeist im Kontext einer Miniwelt – nachvollzogen und debuggt. Dabei werden beispielsweise verschiedene Kontrollstrukturen eingeführt und eingeübt. Neben der Visualisierung des Programmablaufs wird weiteres Feedback zur Verfügung gestellt, beispielsweise durch die Erklärung von Fehlermeldungen (Lee und Ko, 2011). Außerdem werden häufig Funktionalitäten zum schrittweisen Nachvollziehen des Programms, vergleichbar mit einem Debugger, zur Verfügung gestellt. Damit stellen diese Werkzeuge aufgrund ihrer Ausrichtung vorwiegend eine Möglichkeit zum Üben von Debugging dar, analog zu Debugging-Aufgaben.

Im Gegensatz dazu haben andere Werkzeuge explizit die Verbesserung von Debuggingfähigkeiten als Ziel. Carter (2014), Lee und Wu (1999) und Liu *et al.* (2017) verwenden dazu ebenfalls automatisiert generierte Debuggingaufgaben. Lediglich Miljanovic und Bradbury (2017) vermitteln explizit Debuggingtechniken wie das Platzieren von Haltepunkten, die Verwendung von *print*-Debugging oder das Auskommentieren von Code. Die jeweiligen Techniken werden dabei mithilfe eines Tutorials eingeführt und müssen danach in einer Debuggingaufgabe angewandt werden.

Darüber hinaus existieren didaktisch reduzierte Werkzeuge wie der Debugger in BlueJ (Kölling, 2000) oder das tangible-Programmiersystem von Sipitakiat und Nusen (2012), das neben dem schrittweisen Nachverfolgen der Ausführung auch *haptische Breakpoints* ermöglicht, in dem die Schülerinnen und Schüler Flaggen neben Anweisungen platzieren können, um diese für die spätere Untersuchung zu markieren. Außerdem existieren tutorielle Systeme, die die Lernenden während ihres Debuggingprozesses durch unterschiedliche Arten von Feedback unterstützen sollen, beispielsweise indem relevante Codeausschnitte auf Wunsch erklärt werden (Kumar, 2002). Auch hier steht aber nicht das Vermitteln von Debuggingfähigkeiten im Vordergrund, sondern lediglich die Unterstützung des Debuggingprozesses.

Zusammenfassend stellen Li *et al.* (2019) in ihrem Review existierender Werkzeuge fest, dass große Diskrepanzen zwischen für Debugging benötigtem Wissen und dem in diesen Werkzeugen vermittelten existieren. Dabei wird insbesondere darauf hingewiesen, dass innerhalb der Werkzeuge zumeist nur unsystematisch einzelne Aspekte des Debugging isoliert aufgegriffen werden und es auch hier an einem „Gesamtkonzept“ fehlt.

2.5.4 Implikationen

Zusammenfassend ist damit festzustellen, dass der am weitesten verbreitete **methodische Ansatz** zur Verbesserung der Debuggingfähigkeiten – sowohl in der universitären Lehre, im Informatikunterricht als auch bei Werkzeugen – darin besteht, Lernenden durch Debugging-Aufgaben die Möglichkeit zum (unsystematischen) Üben zu geben. Darüber hinaus werden *debugging logs* und vergleichbare Methoden zur Reflexion eingesetzt. Zwar konstatieren die verschiedenen Studien positive Effekte auf die Debuggingleistung, allerdings werden Debuggingfähigkeiten auf diese Art und Weise nicht explizit eingeführt und vermittelt, sondern sollen sich weiterhin „von selbst“ durch Übung entwickeln.

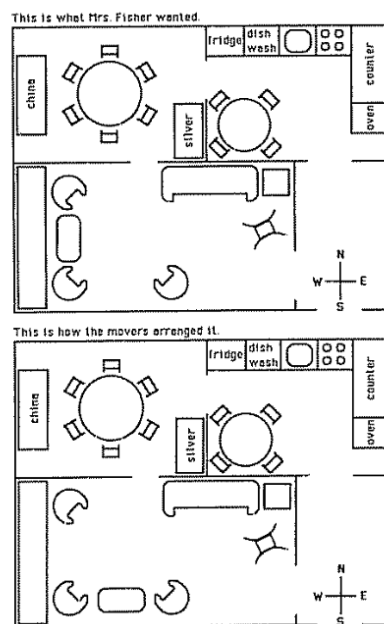
Wenn konkrete **Inhalte** explizit unterrichtet werden, ist das einerseits ein systematisches Debuggingvorgehen, welches sich an der bereits dargestellten *isolation*-Strategie orientiert, indem Hypothesen über die Ursache des Fehlverhaltens aufgestellt und überprüft werden. Insbesondere für den schulischen Kontext zeigen sich vielversprechende Ergebnisse, allerdings fehlt es auch hier an adäquaten empirischen Untersuchungen, die einen Effekt belegen können. Andererseits werden vereinzelt gewisse Debuggingstrategien (wie das *Slicing* oder *Tracing* mit Hilfe des Debuggers oder *print*-Debugging) erfolgreich vermittelt. Auch diese Inhalte werden mit Hilfe von Debuggingaufgaben eingeübt. Dies erfolgt dabei aber zumeist isoliert: Es existiert kein übergreifendes *Konzept*, sondern es werden lediglich einzelne Strategien unterrichtet.

Damit ist festzuhalten, dass die Frage, wie Debugging unterrichtet werden kann, in Anbetracht des Forschungsstandes als bisher unbeantwortet angesehen werden muss – gerade vor dem Hintergrund, dass Debuggingfähigkeiten sich eben nicht „von selbst“ (*Carver und Klahr, 1986*) entwickeln. Wie dargelegt, existieren lediglich vereinzelte Studien, zumeist ohne adäquate empirische Methodik, die sich der expliziten Vermittlung von Debugging widmen. Weiterhin zeigen die wenigen existierenden Ergebnisse eben auch, dass Debuggen explizit unterrichtet werden *kann*. Bezüglich der Fragen, *welche* Debuggingfähigkeiten denn nun *auf welche Art und Weise* unterrichtet werden können, gibt es aber lediglich erste Indizien, insbesondere für ein schulisches Niveau und mit den entsprechenden speziellen Voraussetzungen und Rahmenbedingungen von (Informatik-)Unterricht.

2.6 Transfer

Zentrales Ziel informatischer Bildung ist es, einen Beitrag zur Allgemeinbildung zu leisten, der auch für das Debugging untersucht werden muss. Einer der relevanten Beiträge der Informatik zur Allgemeinbildung sind spezifisch informatische Denk- und Arbeitsweisen, die über die Informatik hinaus Anwendung finden können. Zentrale Annahme solcher als

Computational Thinking bezeichneten Denk- und Arbeitsweisen ist, dass jede bzw. jeder davon profitieren kann (Wing, 2006) und Computational Thinking sowohl gewinnbringend für Probleme des Alltag angewandt werden kann als auch den Transfer allgemeiner kognitiver Fähigkeiten in andere Domänen unterstützt (Wing, 2011). Wie bereits dargelegt, stellt Debugging dabei eine bedeutende Herangehensweise des Computational Thinking dar. Allerdings gibt es für einen Transfer informatischer Vorgehensweisen und Konzepte gemäß dem Computational Thinking viele ernüchternde Ergebnisse (Denning, 2017; Guzdial, 2015). Im Gegensatz dazu existiert für Debugging eine Untersuchung, die auf einen Transfer über die Programmierdomäne hinaus hindeutet: In den bereits beschriebenen (vgl. Abschnitt 2.5.2) Experimenten von Klahr und Carver (1988) wurde neben der Veränderung des Debuggingverhaltens von Schülerinnen und Schülern durch die explizite Vermittlung eines systematischen Vorgehens in der Programmierdomäne auch der Transfer auf Nicht-Programmieraufgaben in zwei Studien untersucht. Zur Erhebung des Transfers wurde den Schülerinnen und Schülern eine Geschichte (z.B. Anweisungen an eine Umzugsfirma, wie Möbel aufgestellt werden sollen) sowie Informationen über die intendierte und die tatsächliche Anordnung der Möbel gegeben (vergleiche Abbildung 2.11). Die Anweisungen sind dabei ähnlich strukturiert wie in LOGO (z.B. durch Überschriften) und sorgen für eine gewisse Nähe zum Programmierkontext.



Here are the directions Mrs. Fisher gave to the movers.

To arrange the dining room,

- Center the china cabinet on the west wall.
- Place the silver cabinet in the south-east corner.
- Put the table in the center of the room.
- Arrange the 6 chairs around the table evenly.

To arrange the living room,

- Place the cabinets against the west wall.
- Place one chair in front of each end of the cabinets.
- Place the square table in the north-east corner.
- Put the sofa on the north wall, next to the square table.
- Place another chair on the south wall, across from the sofa.
- Put the coffee table between the two chairs.
- Put the rocker on the east wall, next to the square table.

To arrange the kitchen,

- Put the refrigerator in the north-west corner.
- Put the dishwasher to the right of the refrigerator.
- Put the sink to the right of the dishwasher.
- Put the stove to the right of the sink.
- Place the counter next to the stove and along the east wall.
- Put the oven along the east wall, next to the counter.
- Place the table in the south-west corner of the room.
- Arrange the 4 chairs around the table evenly.

Change or add one thing to fix Mrs. Fisher's directions

Abbildung 2.11: Exemplarische Aufgabe zum Messen der Transferleistung (Klahr und Carver, 1988)

In beiden Studien wurde der Transfer vor und nach der Intervention in Form der Vermittlung eines systematischen Debuggingvorgehens im Rahmen eines größeren LOGO-

Curriculums erhoben. Während die erste Studie dabei ohne Kontrollgruppe durchgeführt wurde, gab es eine solche für die zweite Studie, die allerdings überhaupt keinen Unterricht in LOGO erhielt. Erwartet wurde für beide Experimente ein Unterschied im Vorgehen der Schülerinnen und Schüler bei der Fehlersuche in den Nicht-Programmier-Kontexten nach der Intervention. Dazu klassifizierten die Autorinnen und Autoren das Vorgehen bei der Bearbeitung der Transferaufgaben durch die Schülerinnen und Schüler wie folgt:

- *Focused Search*: Gezielte und selektive Suche, in der zunächst der grobe Bereich identifiziert wird, in dem sich der Fehler befinden kann (im Beispiel aus Abbildung 2.11 die Anweisungen für den Raum, der falsch aufgebaut wurde). Dieser Bereich wird gelesen und jede Anweisung mental simuliert, bis der Fehler gefunden ist.
- *Self-terminating Brute Force*: Lesen und mentales Simulieren jeder Anweisung, bis der Fehler gefunden ist, danach wird der Rest nicht weiter beachtet.
- *Brute Force*: Lesen und mentales Simulieren jeder Anweisung.

Erwartet wurde als Konsequenz des expliziten Debuggingtrainings mehr *focused search* und eine gesteigerte Genauigkeit der Fehlerkorrekturen. Tatsächlich stellten die Autorinnen und Autoren fest, dass sich das Suchverhalten der Schülerinnen und Schüler nach der Intervention in Richtung einer *focused search* veränderte – im Gegensatz zur Kontrollgruppe. Auch die Genauigkeit in der Fehlersuche hatte sich gesteigert und es gab weniger Stellen, die fälschlicherweise als Fehler eingestuft wurden. Bezieht man die Veränderung der Debuggingfähigkeiten in der Programmierdomäne mit in die Untersuchung ein, ist hervorzuheben, dass die Schülerinnen und Schüler stets entweder besser im Debuggen in LOGO und im Transfer oder bei keinem von beiden wurden. Über die eigentliche Untersuchung hinaus bearbeiteten 200 Schülerinnen und Schüler die Transferaufgaben, die entweder über mindestens 200 Stunden LOGO-Erfahrung ohne explizites Debuggingtraining oder aber über keinerlei Programmiererfahrung verfügten. Dabei konnte kein signifikanter Unterschied zwischen der Leistung der beiden Gruppen festgestellt werden. Dies deuten die Autoren als weiteren Hinweis auf die Bedeutung der expliziten Vermittlung von Debuggingfähigkeiten für einen Transfer über die Programmierdomäne hinaus (*Carver und Risinger, 1987*).

Zwischenfazit und Implikationen

Zusammenfassend lässt sich damit konstatieren, dass für das Debugging im Gegensatz zu anderen Teilaspekten des Computational Thinking zumindest Indizien für einen Transfer über die Programmierdomäne hinaus existieren. Die gewählten Kontexte, für die ein Transfer festgestellt werden konnte, haben dabei aber nur eingeschränkte Bedeutung für

den Alltag. Nichtsdestotrotz bildet diese Untersuchung das empirische Fundament für die Analyse der gesellschaftlichen Ansprüche und damit den Beitrag von Debugging zur Allgemeinbildung. Allerdings fehlt es insbesondere an einer genauen Analyse, welche Debuggingfähigkeiten sich nun gemäß dem Computational Thinking auf den Alltag übertragen lassen. Dazu muss vor allem geklärt werden, welche Debuggingfähigkeiten sich für den Alltag und insbesondere das Troubleshooting gewinnbringend einsetzen lassen.

2.7 Zusammenfassung und Fazit

Abschließend soll nun der informatikdidaktische Forschungsstand zum Debugging zusammengefasst und damit die Ausgangslage für diese Arbeit dargestellt werden.

1. Was unterscheidet Debugging von Testen und Programmieren?

Debugging ist der Prozess des Findens und Behebens von Fehlern in Computerprogrammen. Ausgehend vom Fehlerverhalten des Programms muss dazu der zugrunde liegende Fehler lokalisiert und beseitigt werden. Im Gegensatz dazu ist es das Ziel des *Testens* eines Programms, festzustellen, ob ein solches Fehlverhalten vorliegt. In der professionellen Softwareentwicklung werden diese Tätigkeiten daher oftmals von unterschiedlichen Personen ausgeführt. Dennoch besteht eine enge Verbindung zwischen dem Debuggen und Testen eines Programms: Einerseits stoßen fehlgeschlagene Testfälle den Debuggingprozess an. Andererseits unterstützen Testfälle den Debuggingprozess der Programmiererin bzw. des Programmierers, weil Fehler reproduziert und genauer eingegrenzt werden können.

Debugging findet also während der Programmierung statt. Nichtsdestotrotz handelt es sich beim Programmieren und Debuggen um unterschiedliche Tätigkeiten, die unterschiedliche Fähigkeiten benötigen. So sind gute Programmierer nicht automatisch auch gute Debugger. Debuggingfähigkeiten müssen daher gezielt gefördert und entwickelt werden.

2. Wie läuft der Debuggingprozess von Experten und Novizen ab?

Experten verwenden zwei Vorgehensmodelle zum Debuggen von Programmen, unter anderem abhängig davon, ob es sich um ein bekanntes oder ein unbekanntes Programm handelt. Bei der *comprehension*-Strategie (oder *forward reasoning*) wird dazu der Code sequentiell händisch nachvollzogen und dabei ein mentales Modell des tatsächlichen Programms gebildet und mit dem spezifizierten, gewünschten Programm verglichen. Bei der *isolation*-Strategie (oder auch *backward reasoning* bzw. *wissenschaftliches Debugging*) werden anhand zur Verfü-

gung stehender Informationen wiederholt Hypothesen aufgestellt, experimentell überprüft und gegebenenfalls weiterentwickelt, bis der Fehler gefunden ist. Für die experimentelle Überprüfung und zur Gewinnung weiterer Informationen kommen zusätzliche Strategien, Taktiken, Methoden und Werkzeuge zum Einsatz.

Novizen verwenden ein ähnliches Vorgehen und ähnliche Strategien, wenden diese aber oftmals ineffizient an. Daher wechseln sie häufig zu einem *Trial-and-Error-Ansatz*, gehen ineffizient mit Fehlermeldungen um oder haben Probleme in der Beschreibung des Fehlerverhaltens und dem Generieren von Hypothesen. Ein zentraler Unterschied zwischen Experten und Novizen ist insbesondere das *Verstehen* des Programms. Allerdings spielt dieser Schritt nur beim Debuggen *fremder* Programme eine große Rolle.

3. Welche Rolle spielen unterschiedliche Fehlerarten für den Debuggingprozess?

Es gibt eine Vielzahl an Klassifikationen von Fehlern in unterschiedliche Kategorien. Unabhängig von der konkreten Klassifikation ist festzustellen, dass das Vorgehen beim Debuggen abhängig von den Informationen und Vorgehensweisen variiert, die für verschiedene Fehlertypen in unterschiedlichem Ausmaß zur Verfügung stehen. Bezüglich der Häufigkeit und Schwere der verschiedenen Fehlertypen für Novizen lassen sich keine abschließenden Antworten geben, und insbesondere die Frage nach der Bedeutung von Kompilierzeitfehlern für den Informatikunterricht ist bisher unbeantwortet.

4. Welche Einstellungen und Lernvoraussetzungen haben Lernende bezüglich Debugging?

Bezüglich der Einstellungen von Novizen zum Debugging ist festzuhalten, dass Programmierfehler und der Debuggingprozess oftmals mit großer Frustration verbunden sind. Programmierfehler sind für Lernende negativ konnotiert und werden als persönliches Versagen wahrgenommen. Bezüglich der Lernvoraussetzungen greifen Lernende auf Erfahrungen von *Troubleshooting*-Prozessen in ihrem Alltag zurück, womit sich manche Verhaltensweisen von Novizen beim Debuggen erklären lassen.

5. Welche Ansätze für das Unterrichten von Debugging gibt es?

Bezüglich des expliziten Unterrichts von Debugging existieren nur wenige empirische Erkenntnisse. Vorwiegend wird Lernenden durch Debuggingaufgaben die Möglichkeit gegeben, unsystematisch zu üben. Wenn konkrete Inhalte vermittelt werden, handelt es sich dabei zumeist um ein systematisches Vorgehen gemäß der *isolation*-Strategie oder aber

um einzelne isolierte Strategien ohne zugrunde liegendes Gesamtkonzept. Nichtsdestotrotz deuten diese Ergebnisse darauf hin, dass Debugging explizit vermittelt werden *kann*.

6. Findet ein Transfer von Debuggingfähigkeiten über die Programmierdomäne hinaus statt?

Im Gegensatz zu vielen anderen Herangehensweisen und Konzepten im Rahmen von Computational Thinking existiert für das Debugging tatsächlich eine Studie, die auf einen Transfer von entsprechenden Debuggingfähigkeiten aus der Programmierdomäne hinaus hindeutet.

Fazit

Insgesamt ist festzustellen, dass die informatische und informatikdidaktische Forschung zu Debugging bis auf die Siebzigerjahre zurückgeht. Nichtsdestotrotz mangelt es an Erkenntnissen, die zentral für die Gestaltung von Konzepten und Materialien für den Informatikunterricht sind. Besonders auffällig ist dabei, dass lediglich eine äußerst geringe Anzahl an Untersuchungen für den schulischen Kontext durchgeführt wurde. Dabei existieren offensichtlich große Unterschiede zu den Bedingungen hochschuldidaktischer Lehre. Beispielsweise sind Lehrkräfte oftmals eben selbst keine Experten der Programmierung, weisen Schülerinnen und Schüler deutlich weniger Selbstständigkeit auf als Studierende oder es sind grundlegend unterschiedliche Rahmenbedingungen bezüglich der Lehr-Lern-Arrangements und Lernziele gegeben.

Damit ergibt sich eine Reihe von Forschungsdesideraten:

- Zunächst ist die Frage, wie Debugging explizit unterrichtet werden kann, bisher unbeantwortet. Hier fehlt es insbesondere an geeigneten Konzepten für die Unterrichtspraxis, deren Wirksamkeit adäquat empirisch untersucht wurde.
- Grundlage der Vermittlung von Debugging im Informatikunterricht ist eine Systematisierung von Debuggingfähigkeiten, die Novizen für das Debuggen benötigen und die entsprechend unterrichtet werden müssen. Auch hier ist festzustellen, dass die Frage, welche Fähigkeiten Novizen für erfolgreiches Debugging nun benötigen, bislang unbeantwortet ist.
- Gänzlich unbeachtet ist bisher die Perspektive der Lehrkräfte geblieben. Bezüglich deren Wahrnehmung der Probleme der Schülerinnen und Schüler (wie beispielsweise der Rolle von Kompilierzeitfehlern im Informatikunterricht), aber auch der eingesetzten Ansätze zum Umgang mit Programmierfehlern, existiert keine Untersuchung.

- Zwar deutet der Forschungsstand darauf hin, dass Debuggingfähigkeiten auch auf den Alltag übertragen werden können. Allerdings fehlt es an Erkenntnissen, wie genau dieser Transfer nun ausgestaltet sein kann und welche Debuggingfähigkeiten einen Beitrag zur Allgemeinbildung leisten können.
- Außerdem fehlt es in diesem Zusammenhang an einer Untersuchung der Lernvoraussetzungen – zielgerichtet für benötigte Debuggingfähigkeiten –, die Schülerinnen und Schüler aus der *Troubleshooting*-Erfahrung ihres Alltags besitzen, und welchen Einfluss diese auf ihr Debuggingverhalten haben können.

Nichtsdestotrotz lassen sich aus den existierenden Ergebnissen aber auch einige Gestaltungshinweise für die Entwicklung von Konzepten und Materialien für den Unterricht ableiten:

- Da es das Ziel dieser Arbeit ist, Schülerinnen und Schüler zum Debuggen eigener Programme zu befähigen, erscheint ein Fokus auf die Vermittlung eines Vorgehens gemäß der *isolation*-Strategie (bzw. *wissenschaftliche Methode* oder *backward reasoning*) zielführend – auch in Anbetracht der existierenden vielversprechenden Untersuchung zur Vermittlung eines solchen Vorgehens. Weiterhin sollte Debugging in Situationen eingeübt werden, die dem Debuggen eigener Programme so nahe wie möglich kommen.
- Für unterschiedliche Fehlerarten benötigen die Schülerinnen und Schüler jeweils angepasste Vorgehen bzw. Strategien, die die unterschiedlichen Informationen adressieren, die zur Verfügung stehen.
- Zentral erscheint es, Fehler als „Normalität“ der Programmierung und zeitgleich positiv konnotierte Lernchance für die Schülerinnen und Schüler zu etablieren.
- Hochschuldidaktische Erkenntnisse bezüglich typischer Probleme von Lernenden während des Debuggingprozesses sollten in Konzepten und Materialien angemessen aufgegriffen werden, wie beispielsweise die Angewohnheit, neue Fehler in das Programm einzufügen, weil erfolglose Korrekturversuche nicht rückgängig gemacht werden, Probleme mit dem Formulieren von Hypothesen oder aber der ineffiziente Umgang mit Fehlermeldungen.

Wie in Kapitel 1.3 beschrieben, wurden basierend auf diesem Forschungsstand und der entsprechenden Forschungslücke mit Hilfe des Forschungsformates der didaktischen Rekonstruktion entsprechende Forschungsfragen abgeleitet. Damit sollen in dieser Arbeit die betreffenden Forschungsdesiderate aufgegriffen werden, um schlussendlich geeignete Konzepte und Materialien für den Informatikunterricht zu entwickeln, die das Schlüsselproblem Debugging angemessen adressieren.

Teil II:
Zugrunde liegende Perspektiven

The process of debugging, going an correcting the program and then looking at the behavior, and then correcting it again, and finally iteratively getting it to a working program, is in fact, very close to learning about learning.

NICHOLAS NEGROPONTE

3 Fachliche Klärung

Im Zuge der Untersuchung der *zugrunde liegenden Perspektiven* auf das Thema Debugging soll im Folgenden zunächst die *fachliche Klärung* vorgenommen werden. Ziel der fachlichen Klärung ist es, die grundlegende Sachstruktur des Unterrichtsgegenstandes herauszuarbeiten, allerdings aus einer fachdidaktischen Perspektive, d.h. ausgehend von einer *Vermittlungsabsicht* (Kattmann et al., 1997). Diese Sachstruktur stellt die Grundlage für den weiteren Forschungsprozess dar. Im Gegensatz zur fachlichen Klärung eines *Themengebietes* werden dabei für den *Prozess* Debugging nicht zentrale Konzepte herausgearbeitet, sondern konkrete Fähigkeiten (vgl. die Argumentation in Kapitel 1.3). Im Folgenden sollen daher relevanten Debuggingfähigkeiten identifiziert werden. Dies ermöglicht es zu analysieren, welche dieser Fähigkeiten möglicherweise bereits im Informatikunterricht aufgegriffen werden, welchen Beitrag diese Fähigkeiten zur Allgemeinbildung leisten oder aber welche Fähigkeiten die Schülerinnen und Schüler bereits aus ihrem Alltag als Lernvoraussetzung mitbringen.

3.1 Ziele der Untersuchung

Die Notwendigkeit einer fachlichen Klärung für das Debugging ergibt sich aus dem Forschungsstand: Wie bereits dargelegt, ist Debugging eine praktische *Notwendigkeit* der Softwareentwicklung und damit eine Tätigkeit, die ohne eine zugrunde liegende Theorie seit den Anfängen der Programmierung ausgeübt wird. Dabei beschränkt sich die bisherige Forschung auf die Charakterisierung des Debuggingprozesses von Experten und Novizen oder deren Vergleich. Dazu existiert eine Vielzahl von uneinheitlich verwendeten Begrifflichkeiten wie etwa Ansätze, Taktiken, Strategien, Werkzeuge, Methoden und Wissen, die im Kontext des Debugging verwendet werden (siehe Kapitel 2.2). Welche konkreten Fähigkeiten Novizen für erfolgreiches Debugging aber nun benötigen und ihnen damit vermittelt werden müssen, ist bisher ungeklärt. Es fehlt somit an einer Systematisierung relevanter Debuggingfähigkeiten aus einer Vermittlungsabsicht.

Neben dem Stand der Forschung stellen, wie auch von Kattmann et al. (1997) im Rahmen des Modells der didaktischen Rekonstruktion betont, Lehrbücher einen möglichen Zugang zur fachlichen Klärung eines Unterrichtsgegenstandes dar. Allerdings offenbart sich hier eine ähnliche Lücke wie im Stand der Forschung: Zunächst gibt es kaum (Lehr-)Bücher, die sich dediziert dem Thema Debugging widmen. Betrachtet man darüber hinaus Programmierlehrbücher, finden sich auch hier lediglich vereinzelt und isoliert gewisse Techniken (McCauley et al., 2008).

Die Frage, welche Debuggingfähigkeiten Programmieranfängerinnen bzw. Programmieranfängern vermittelt werden sollten, ist damit bisher unbeantwortet. Dabei stellt gerade die Auswahl der im Unterricht zu vermittelnden Kompetenzen die Grundlage jedes fachdidaktischen und unterrichtlichen Handelns dar, ohne deren Klärung sich Fragen nach dem methodischen Vorgehen oder der adäquaten Aufbereitung im Unterricht gar nicht erst sinnvoll stellen. Daher soll im Folgenden eine Systematisierung relevanter Fähigkeiten vorgenommen und somit die erste Forschungsfrage dieser Arbeit beantwortet werden:

(RQ1) Was sind relevante Debuggingfähigkeiten für Programmieranfängerinnen und -anfänger?

3.2 Methodik

Kattmann et al. (1997) schlagen zur fachlichen Klärung die hermeneutisch-analytische Untersuchung geeigneter Quellentexte wie Lehrbücher vor. *Grillenberger (2019)* verwendet dazu ein teilautomatisiertes Verfahren, um die zentralen Begriffe des Themengebietes „Daten“ herauszuarbeiten. *Schmidt (2011)* nutzt aufgrund der Breite des Themas „Stoffe, Teilchen und Atome“ das Verfahren der Globalanalyse (vgl. *Bortz und Döring (2006)*). Nichtsdestotrotz wird in einem Großteil der am Modell der didaktischen Reduktion ausgerichteten Arbeiten die qualitative Inhaltsanalyse nach Mayring eingesetzt (vgl. z.B. *Sander (2018)* oder *Frerichs (1999)*). Auch für die fachliche Klärung des Themas Debugging erscheint eine *inhaltlich-strukturierende* qualitative Inhaltsanalyse geeignet: Ziel der Untersuchung ist es, relevante Fähigkeiten zu sammeln, zu strukturieren und systematisieren. Dabei fehlt es bisher an entsprechenden Theorien oder etablierten Fachtermini. Die qualitative Inhaltsanalyse hilft in solchen Fällen, die unterschiedlichen lokalen Theorien und verwendeten Begriffe vergleichend zu analysieren und zu einer Sachstruktur zu verdichten. Nach *Mayring (2014)* wird die *strukturierende* qualitative Inhaltsanalyse jedoch theoriegeleitet auf Basis eines theoretisch fundierten deduktiven Kategoriensystems durchgeführt. Vor dem Hintergrund des begrenzten Forschungsstandes erscheint ein solches deduktives Verfahren ungeeignet. Daher orientiert sich die Methodik dieser Untersuchung an der *konventionellen* qualitativen Inhaltsanalyse nach *Hsieh und Shannon (2005)*, die Zielsetzung und Vorgehen mit Mayrings *strukturierter* qualitativer Inhaltsanalyse teilt. Im Unterschied zu Mayrings Variante werden die Kategorien sowie deren Bezeichnung dabei induktiv auf Basis des Datenmaterials gebildet. Sowohl die strukturierte Inhaltsanalyse nach *Mayring (2014)* als auch die konventionelle qualitative Inhaltsanalyse nach *Hsieh und Shannon (2005)* können damit als Varianten einer inhaltlich-strukturierten qualitativen Inhaltsanalyse eingeordnet werden (*Schreier, 2014*).

Zentrale Phasen dieses Vorgehens sind einerseits die Sammlung und Auswahl des Materials, in diesem Fall des **Dokumentenkorpus**, und andererseits dessen induktive **Auswertung**. Im Folgenden soll auf diese beiden Schritte näher eingegangen werden.

Dokumentenkorpus

Für den Prozess des Debugging existieren im Vergleich zu „traditionellen“ Themengebieten wie *Programmierung*, *Softwaretests* oder *Datenbanken* nur wenige dedizierte Lehrbücher. Als Tätigkeit im Kontext der Programmierung widmen allerdings manche Programmierlehrbücher dem Thema Debugging ein Kapitel. Daher wurden entsprechende Lehrbücher nach Inhalten zum Debugging durchsucht und gegebenenfalls in das Korpus aufgenommen.

Dennoch bedarf es einer Erweiterung des Korpus für eine aussagekräftige Analyse. Daher wurden zum einen bayerische Schulbücher untersucht und, falls sie Inhalte für das Debugging enthielten, herangezogen. Darüber hinaus wurden auch wissenschaftliche Beiträge zum Thema Debugging analysiert: Zu diesem Zweck wurden relevante Bibliotheken (ACM Digital Library, IEEE Digital Library, Google Scholar) systematisch durchsucht und Dokumente mit einer Mindestlänge von vier Seiten identifiziert, die entsprechende Schlüsselwörter enthielten („debugging“, „debugging strategies“, „debugging education“, „debugging competences“, „debugging skills“).

Damit ergab sich folgendes Korpus:

- 14 wissenschaftliche Beiträge
- 3 bayerische Schulbücher
- 18 Programmierlehrbücher
- 3 Monographien mit einem Fokus auf Debugging

Auswertung

Die Dokumente des Textkorpus wurden anschließend ausgewertet. Dazu wurden sie auf Aussagen und Ergebnisse zu folgendem Aspekt analysiert:

- Welche Debuggingfähigkeiten werden Novizen vermittelt oder als relevant angesehen?

Wie in Kapitel 2.1 bereits dargelegt, unterscheiden sich Debuggingfähigkeiten konzeptionell von allgemeinen Programmierfähigkeiten. Daher konzentriert sich diese Analyse explizit auf Debuggingfähigkeiten – im Gegensatz zu Fähigkeiten, die als allgemeine Pro-

grammierfähigkeiten betrachtet werden. Aus diesem Grund wurden auch Fähigkeiten wie Programmverständnis ausgelassen, obwohl sie offensichtlich notwendige Voraussetzungen für den Debuggingprozess sind (vgl. Kapitel 2.2).

Ausgangspunkt der Analyse war ein Korpus von zehn zufällig ausgewählten Dokumenten, das anschließend jeweils schrittweise erweitert wurde. Als Analyseeinheiten dienten dabei die entsprechenden Textpassagen der Dokumente. Diese wurden induktiv kodiert, wobei die Bezeichnungen der Kodierung zunächst jeweils direkt aus dem Text übernommen wurden. Auf Basis der Kodierungen in den zehn initialen Dokumenten wurden erste Kategorien gebildet, die damit das anfängliche Kategoriensystem darstellten, welches für jedes folgende Dokument als Grundlage zur Verfügung stand und gegebenenfalls erweitert wurde. Für jede neue Kodierung wurde überprüft, ob sie in eine bereits bestehende Kategorie eingeordnet werden konnte oder ob eine neue Kategorie angelegt werden musste. Jeweils nach der Analyse von fünf Dokumenten wurde versucht, einzelne Kategorien zusammenzufassen oder hierarchisch anzuordnen. Gegebenenfalls unterschiedliche Termini wurden vereinheitlicht, wie auch in den Ergebnissen jeweils ausgeführt wird. Das finale Kategoriensystem und die Definitionen der daraus resultierenden Fähigkeiten (Oberkategorien) und deren Ausprägungen (Subkategorien) erlauben damit die Strukturierung des Untersuchungsgegenstandes.

3.3 Ergebnisse

Im Folgendem wird das Ergebnis der induktiven *inhaltlich-strukturierenden* qualitativen Inhaltsanalyse dargelegt. Dazu werden die resultierenden Fähigkeiten ausführlich beschrieben. Das finale Kategoriensystem mitsamt den Verweisen auf das Korpus zur Gewährleistung des qualitativen Gütekriteriums der Transparenz ist in Tabelle 3.1 dargestellt.

3.3.1 Anwenden eines systematischen Debuggingvorgehens

Im Korpus finden sich, wie bereits im Forschungsstand festgestellt, eine Reihe an unterschiedlichen Begrifflichkeiten für die Bezeichnung von Vorgehensweisen im Kontext des Debuggings. Hierzu gehören die Begriffe Strategie, Methode, Ansatz, Taktik oder auch Technik. Dabei kann – wie sich bereits im Forschungsstand (siehe Kapitel 2.2) angedeutet hat – zwischen zwei Ebenen unterschieden werden, auf die mit diesen Termini Bezug genommen wird. Zum einen existieren Vorgehensweisen, die den kompletten Prozess des Debuggens auf einer abstrakteren Ebene beschreiben (z.B. *isolation*-Strategie, wissenschaftliche Methode, *forward-reasoning*-Ansatz). Daneben werden konkrete Vorgehensweisen genannt, die für einzelne Schritte eines solchen allgemeinen Vorgehens angewendet werden (z.B. *Slicing*

Oberkategorie	Unterkategorie	Quellen
Anwenden von Debuggingstrategien	Auskommentieren und Slicing	<i>Eranki und Moudgalya (2016), Miljanovic und Bradbury (2017), Fitzgerald et al. (2009) und Metzger (2004)</i>
	Logging und Tracing	<i>Miljanovic und Bradbury (2017), Erbs und Stolz (1984), Sande und Sande (2014), Barnes, Kölling und Gosling (2016), Meyer (2018), Passig und Jander (2013), Fitzgerald et al. (2009), Becker (2015) und Metzger (2004)</i>
	Assertions Testen	<i>Briggs und Haxsen (2016), Ratz et al. (2018) und Metzger (2004) Klein (2013), Barnes, Kölling und Gosling (2016), Padmanabhan (2016), Fitzgerald et al. (2009), Metzger (2004) und Agans (2002)</i>
	Informationen sammeln Hilfe suchen	<i>Ulllenboom (2020), Fitzgerald et al. (2009), Metzger (2004) und Agans (2002) Passig und Jander (2013), Fitzgerald et al. (2009), Metzger (2004), Agans (2002) und Dörn (2020)</i>
Anwenden eines systematischen Vorgehens	<i>isolation-Strategie</i>	<i>Katz und Anderson (1987), Carver und Risinger (1987), Gugerty und Olson (1986), Allwood und Björhag (1991), Böttcher et al. (2016), Vessey (1985), Nanja und Cook (1987), Yen, Wu und Lin (2012), Passig und Jander (2013), Metzger (2004), Agans (2002) und Zeller (2009)</i>
	<i>comprehension-Strategie</i>	<i>Katz und Anderson (1987), Vessey (1985), Nanja und Cook (1987), Yen, Wu und Lin (2012) und Gugerty und Olson (1986)</i>
Verwendung von Werkzeugen	Debugger	<i>Böttcher et al. (2016), Klima und Selberherr (2010), Krienke (2002), Boles (1999), Niemeyer (2020), Ulllenboom (2020), Barnes, Kölling und Gosling (2016), Brauer (2015), Becker (2015), Dörn (2020) und Dörn (2019)</i>
	IDE-Feedback	<i>Krienke (2002), Niemeyer (2020), Barnes, Kölling und Gosling (2016), Brauer (2015) und Staatsinstitut für Bildungsforschung München (2008)</i>
Anwenden von Mustern und Heuristiken für typische Fehler		<i>Ulllenboom (2020), Carver und Risinger (1987), Chmiel und Loui (2004), Mueller (2018), Briggs und Haxsen (2016), Niemeyer (2020), Brauer (2015), Padmanabhan (2016) und Passig und Jander (2013)</i>

Tabelle 3.1: Resultierendes Kategoriensystem der inhaltlich-strukturierenden qualitativen Inhaltsanalyse.

oder *print*-Debugging). Daher werden die auf einer allgemeinen Ebene angesiedelten Vorgehensweisen in der Analyse unter der Kategorie *Systematisches Vorgehen* angeordnet, um diese im Korpus inhärente Trennung der zwei Ebenen abzubilden.

Dementsprechend stellt die erste identifizierte relevante Debuggingfähigkeit die Anwendung eines *systematischen Vorgehens* dar. Ein solches Vorgehen (manchmal auch als „Strategie“ oder *Methode* bezeichnet) bedeutet, dass systematisch auf einer Makroebene ein zielgerichteter Plan verfolgt wird, um Fehler zu finden und zu beheben. Wie bereits ausführlich im Forschungsstand dargestellt (vgl. Kapitel 2.2), existieren dabei insbesondere zwei solcher Vorgehensmodelle, die von Experten wie Novizen gleichermaßen angewendet werden. Einerseits handelt es sich dabei um ein Vorgehen gemäß der *isolation*-Strategie (auch *backward reasoning* oder *wissenschaftliche Methode*). Dabei wird das Programm zunächst getestet und das Fehlverhalten beobachtet. Aufbauend auf Informationen, die dabei gewonnen werden, werden Hypothesen über Art und Ort des Fehlers aufgestellt. Diese Hypothesen werden nun experimentell verifiziert und, falls erforderlich, wiederholt verfeinert oder ganz verworfen, bis der Fehler gefunden ist. Um die korrekte Behebung sicherzustellen, wird das Programm zum Abschluss erneut getestet.

Andererseits wird bei der *comprehension*-Strategie (bzw. *forward reasoning*) der Code sequentiell nachvollzogen und dabei ein mentales Modell von Aufbau und Struktur des Programmes gebildet und mit dem intendierten verglichen. Während die *isolation*-Strategie damit vorwiegend beim Debuggen eigener Programme eingesetzt wird, findet die *comprehension*-Strategie insbesondere bei fremden Programmen Anwendung.

3.3.2 Anwenden von Debuggingstrategien

Die oben angesprochenen auf einer konkreteren Ebene angesiedelten Vorgehensweisen, die das allgemeine systematische Vorgehen durch spezifische Methoden, Taktiken, Techniken oder Strategien unterstützen, werden im Folgenden unter der Bezeichnung *Debuggingstrategien* subsumiert. Dabei konnten verschiedene Arten solcher Strategien (Unterkategorien) identifiziert werden:

- **Auskommentieren und Slicing:** Bei diesen Strategien wird der Problemraum reduziert, indem einzelne Codeteile auskommentiert bzw. temporär entfernt werden. Hierdurch verringert sich die Komplexität der Fehlersuche.
- **Logging und Tracing:** Mit Hilfe solcher Strategien kann der Wert von Variablen sowie der Programmablauf während der Ausführung nachvollzogen werden. Dazu können beispielsweise mit Hilfe von textuellen Ausgaben (*print*-Debugging) Statusmeldungen oder die aktuelle Belegung von Variablen ausgegeben werden, um so Hypothesen zu überprüfen, weitere Informationen zu gewinnen oder den Program-

mablauf nachzuvollziehen. Dies kann auch durch das Setzen von Breakpoints, die die Programmausführung bei Erreichen der entsprechenden Anweisung pausieren, oder durch das „händische“ Nachvollziehen in Form eines Schreibtischlaufs oder auch durch Tracetabellen (vgl. *Erbs und Stolz (1984)*) realisiert werden.

- **Assertions:** Assertions erlauben das Formulieren von Zusicherungen bezüglich des Programmzustands zu gewissen Zeitpunkten. Sie fungieren dabei als explizit im Code repräsentierte Hypothesen, die automatisiert zur Laufzeit überprüft werden. Entsprechende Abweichungen helfen dabei, den Fehler zu lokalisieren.
- **Testen:** Beim Testen wird mit Hilfe von gezielt gewählten Eingaben das Verhalten des Programmes geprüft. Dies erlaubt es, das (gegebenenfalls fehlerhafte) Verhalten genauer zu analysieren, sowie – ausgehend von den Eingaben, für die das Programm sich fehlerhaft verhält – Rückschlüsse über Art und Position des Fehlers zu ziehen, und damit Hypothesen aufzustellen oder zu validieren.
- **Informationen sammeln:** Indem weitere Ressourcen hinzugezogen werden, wie etwa die Spezifikation (oder auch Aufgabenstellung) oder auch die Beschreibung einer API-Schnittstelle, beispielsweise einer verwendeten externen Bibliothek, können zusätzliche Erkenntnisse über das Problem gewonnen werden, die bei der Formulierung von Hypothesen bzw. dem Vergleich von tatsächlichem und intendiertem Programm behilflich sind.
- **Hilfe suchen:** Die letzte Kategorie subsumiert verschiedene Strategien, in denen gezielt externe Hilfe gesucht wird. Beispiele umfassen etwa eine systematische Web-suche unter der Verwendung entsprechender Stichwörter, die genaue Beschreibung eines Problems, entweder zur Selbstreflexion (*rubberduck debugging*¹⁵) oder zur Einbindung von Experten bzw. einer (Online-)Community.

Dabei war festzustellen, dass sich insbesondere Schul- und Programmierlehrbücher vor allem auf die Vermittlung von konkreten Strategien wie *print*-Debugging konzentrierten, während das allgemeine systematische Vorgehen insbesondere in der Forschungsliteratur kodiert wurde. Weitere im Kontext von Fehlern und Fehlerbehandlung angeführte Ansätze wie etwa die Kommentierung des Codes oder das Abfangen von Fehlern (beispielsweise durch Ausnahmebehandlung) stellen hingegen keine Strategien zum Debugging dar, sondern setzen im Sinne *robuster Programmierung* darauf, die Qualität der Software von vornherein zu steigern. Daher wurden sie in dieser Analyse nicht berücksichtigt.

¹⁵Der Debuggingprozess bzw. das Problem wird verbalisiert und z.B. einer „Gummiente“ erklärt. Dies adressiert das typische Phänomen, das allein das Erklären des eigenen Problems für eine andere Person ausreicht, um selbst den Fehler zu finden.

3.3.3 Anwendung von Heuristiken und Mustern für typische Fehler

Entwicklerinnen und Entwickler wenden oft eine verkürzte Version des systematischen Debuggingprozesses an: Aus ihrer Erfahrung kennen sie typische Fehler und deren mögliche Ursachen und generieren und überprüfen entsprechende Hypothesen direkt. Um dieses „Lernen aus Fehlern“ zu unterstützen, führen viele professionelle Entwicklerinnen und Entwickler *debugging logs* oder „Tagebücher“, in denen sie ihre Debugging-Erfahrungen dokumentieren. Ein solcher Katalog von Fehlern kann dabei bei der Beseitigung zukünftiger ähnlicher Fehler hilfreich sein. Die Bedeutung von Heuristiken und Mustern für typische Fehler wird auch für Novizen betont. Heuristiken wie *fix the first compiler error first* helfen diesen im Umgang mit Fehlermeldungen. Aufgrund der geringen Programmiererfahrung von Novizen unterstützen darüber hinaus gerade Muster für typische Fehler, welche sich sonst üblicherweise erst mit fortschreitender Programmiererfahrung ausbilden. Indem entsprechende Ursachen typischer Fehler vermittelt werden, werden Novizen direkt mögliche Ansatzpunkte zur Fehlerbehebung bereitgestellt und sie bei der Generierung von Hypothesen unterstützt.

3.3.4 Verwenden von Werkzeugen

Auch die Verwendung von Werkzeugen unterstützt das allgemeine Debuggingvorgehen und wird Novizen für die Fehlersuche und -behebung – vorwiegend in Programmierlehr- bzw. Schulbüchern – vermittelt. Dazu gehört einerseits der Umgang mit dem Debugger. Der Debugger ist ein zumeist direkt in Entwicklungsumgebungen enthaltendes Werkzeug, das das schrittweise Nachvollziehen des Programms Anweisung für Anweisung oder das Setzen und Springen von Breakpoint zu Breakpoint erlaubt. Darüber hinaus finden sich im Korpus explizite Hilfestellungen, um das Feedback entsprechender Entwicklungsumgebungen zu interpretieren und zur Fehlerlokalisierung und -beseitigung zu nutzen. Professionelle Werkzeuge wie der Einsatz von automatisierten Fehlerlokalisierungsansätzen wie *delta debugging*, *Back-in-Time-Debuggern*, die es auch erlauben, die Programmausführung rückwärts nachzuvollziehen und vergleichbare professionelle werkzeuggestützte Ansätze finden sich hingegen nicht im Korpus.

3.4 Interpretation

Für die Entwicklung von Konzepten und Materialien für das Debugging im Unterricht ist es zentral, zunächst zu klären, welche Fähigkeiten Novizen dazu vermittelt werden müssen. Dabei konzentriert sich der bisherige Forschungsstand vorwiegend auf eine Beschreibung

des Debuggingprozesses, wobei eine Vielzahl verschiedener Begriffe mit unterschiedlicher Verwendung bzw. unterschiedlichem Fokus festzustellen ist. Daneben befassen sich viele Ansätze der informatischen Forschung zum Debugging mit den Anforderungen von Experten. In diesem Abschnitt wurden daher eine auf die Bedürfnisse von Novizen abzielende inhaltlich-strukturelle qualitative Inhaltsanalyse auf Basis eines Korpus aus wissenschaftlichen Beiträgen, Schulbüchern, Programmierlehrbüchern sowie Monographien mit einem Schwerpunkt auf Debugging durchgeführt. Im Sinne des Modellierungscharakters dieses Vorgehens wurden dabei die für eine unterrichtliche Vermittlung wesentlichen Aspekte identifiziert.

Diese wesentlichen Aspekte lassen sich in einem Modell von Debuggingfähigkeiten für Novizen darstellen (vgl. Abbildung 3.1): Voraussetzung für erfolgreiches Debugging sind dabei, wie in Kapitel 2.1 herausgearbeitet, zunächst gewisse allgemeine Programmierfähigkeiten. Darüber hinaus wird, wie auch in Kapitel 2.2 betont, ein Verständnis für das zu debuggende Programm und dessen Struktur und Aufbau benötigt. Den Kern des Modells bilden die in dieser Analyse identifizierten vier dedizierten Debuggingfähigkeiten. Dabei stellt das Anwenden eines systematischen Debuggingvorgehens die Grundlage für einen erfolgreichen Debuggingprozess dar. Dieses kann dabei durch die Anwendung konkreter Debuggingstrategien, der Verwendung von Werkzeugen sowie Anwendung von Heuristiken und Mustern unterstützt werden.

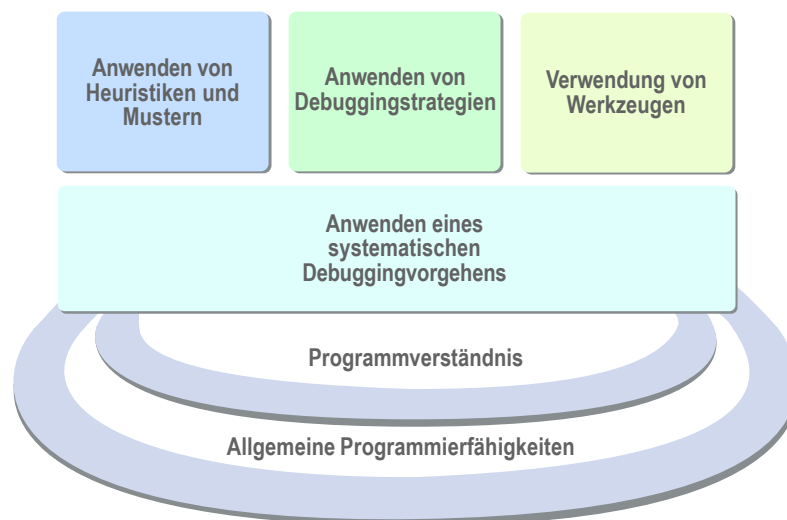


Abbildung 3.1: Modell der Debuggingfähigkeiten für Novizen

Ein Vergleich mit teilweise gleichzeitig zu dieser Analyse entstandenen Ansätzen zur Systematisierung von Debugging hilft dabei, das Modell und die Validität der Ergebnisse

im Sinne korrelativer Gültigkeit (vgl. *Mayring (1985)*) abzusichern. Dazu werden im Folgenden zwei weitere Ansätze zur Systematisierung relevanter Debuggingfähigkeiten für den Unterricht herangezogen, die zeitgleich zu dieser Arbeit und ebenfalls mit expliziter Vermittlungsabsicht für Novizen entstanden sind.

K-8 Debugging-Learning Trajectory

Rich et al. (2019) entwickelten auf Basis einer Analyse von Lernzielen einen *K-8 Learning Trajectory*. Dazu untersuchten sie analog zu dieser Untersuchung systematisch und induktiv maßgebliche Forschungsliteratur und identifizierten Lernziele mit Bezug auf Debugging. Diese wurden anschließend kategorisiert, nach Komplexität angeordnet und ein Lernpfad vorgeschlagen (vgl. Abbildung 3.2). Dabei wurden die Lernziele verschiedenen Dimensionen zugeteilt:

- D1 Strategien, um Fehler zu finden und zu beheben
- D2 Fehlerarten
- D3 Die Bedeutung von Fehlern für den Problemlöseprozess

Die Dimension D2 mit Lernzielen wie *„errors can be caused by missing, as opposed to incorrect, information within instructions“* entspricht dabei der Fähigkeit des Modells der Debuggingfähigkeiten für Novizen, Heuristiken und Muster für typische Fehler anzuwenden. Betrachtet man die fünf angeführten Lernziele der Dimension 1, also Strategien, um Fehler zu finden und zu beheben, ist festzustellen, dass *Rich et al. (2019)* keine Unterteilung zwischen einem allgemeinen systematischen Vorgehen und Debuggingstrategien, die diesen Prozess unterstützen, vornehmen. Im Learning Trajectory stellt die Debuggingstrategie *„Iterative refinement can help fix errors: This is the first step toward understanding that Observe → Hypothesize → Modify → Test [...] can be used to debug code“*, die im Modell der Debuggingfähigkeiten für Novizen als eine Variante eines systematischen Vorgehens bezeichnet wird, zwar auch die Grundlage aller weiteren Strategien dar. Allerdings wird diese „nur“ als weitere Debuggingstrategie (wie beispielsweise das schrittweise Ausführen) bezeichnet.

Trotz der Einteilung der Lernziele in „unplugged“ (graue Box) und „computer-based“ (weiße Box) findet sich keines mit explizitem Werkzeugbezug, auch wenn beispielsweise die Verwendung des Werkzeugs *Debugger* unter dem Lernziel *„Step-by-step execution“* oder der Umgang mit Feedback der IDE unter *„Compile errors should be fixed in the order the compiler reports them“* subsumiert werden könnten. Gerade werkzeuggestützte Aspekte wie etwa der Umgang mit dem Debugger waren in unserer Analyse im Gegensatz dazu oftmals Inhalt der Programmierlehrbücher und finden sich daher auch als zentraler Bestandteil des Modells der Debuggingfähigkeiten für Novizen. Lernziele der Dimension 3 hingegen zeigen

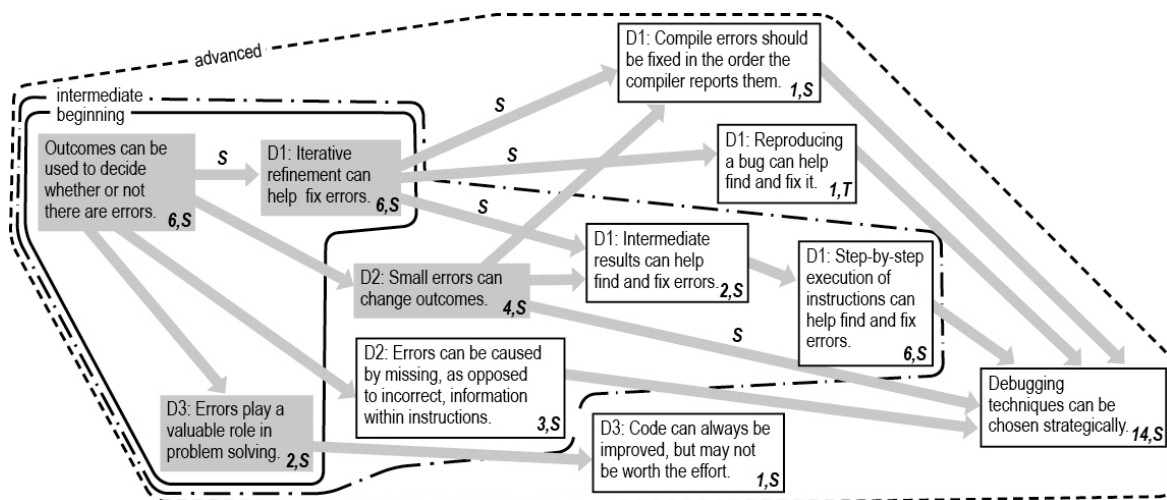


Abbildung 3.2: Learning Trajectory (Rich et al., 2019)

die Bedeutung von Fehlern bzw. von Fehlerbehebung über den Prozess des Debuggens hinaus auf. Dementsprechend sind sie nicht mehr zentral für den Prozess an sich, sondern betonen vielmehr dessen gesellschaftliche Dimension.

Zusammenfassend zeigen sich also inhaltliche Ähnlichkeiten der Ergebnisse, insbesondere bezüglich der Aspekte *typische Fehler* und *Debuggingstrategien* – obwohl die Studie von Rich et al. (2019) sowohl eine andere Zielgruppe (K8) hat, sich auf ein engeres Korpus fokussiert als auch mit *Lernzielen* statt *Fähigkeiten* eine andere Zielsetzung verfolgt. Hauptunterschied des Modell der Debuggingfähigkeiten für Novizen ist insbesondere die Herausstellung eines *systematischen Debuggingvorgehens* sowie die Betonung des Werkzeugaspektes.

Framework for Teaching Debugging

Einen anderen Ansatz wählten Li et al. (2019). Anstatt induktiv vorzugehen, orientierten die Autorinnen und Autoren sich an einem Framework für Troubleshooting nach Jonassen und Hung (2006) (vgl. auch Kapitel 5) und übertrugen es auf das Debugging. Für die fünf Arten von Wissen des Frameworks wurden deduktiv entsprechende Belege und Bezüge in der Literatur herausgearbeitet. Damit ergab sich folgendes Modell:

- *Domain knowledge*: Verständnis der jeweiligen Programmiersprache
- *System knowledge*: Verständnis des Programms, das debuggt werden soll
- *Procedural knowledge*: Wissen, wie beispielsweise ein Testfall aufgerufen werden kann oder Funktionalitäten der IDE wie *breakpoints* genutzt werden können

- *Strategic Knowledge*: Wissen darüber, wie effektiv debuggt werden kann, unterteilt in:
 - *Global strategies*: Strategien, die nicht kontextabhängig sind, wie beispielsweise *isolation-Strategie*, *comprehension-Strategie*
 - *Local strategies*: Kontextabhängige Strategien, spezifisch für ein konkretes Programm, wie das Vergleichen von Ein- und Ausgabe oder das Verwenden von *print-Anweisungen*. Dabei sind lokale Strategien oftmals konkrete Ausprägungen globaler Strategien, wie etwa die Verwendung des Debuggers als lokale Ausprägung der globalen Strategie *forward tracing*
- *Previous experience*: Wissen über typische Fehler

Vergleicht man das Framework mit dem Modell der Debuggingfähigkeiten für Novizen, finden sich große Übereinstimmungen. So werden – wie auch in den bisher diskutierten Modellen – sowohl die *Anwendung von Heuristiken und Mustern* als *previous experience* als auch Debuggingstrategien im Sinne des Modells als *local strategies* betont. Darüber hinaus weist das Framework von *Li et al. (2019)* aber auch den *Umgang mit Werkzeugen* als *procedural knowledge* explizit auf. Außerdem werden *global strategies* als abstraktere Ebene konkreter Debuggingstrategien angeführt, die der *Anwendung eines systematischen Vorgehens* ähneln. Dabei bilden ebenfalls Wissen bezüglich der Programmiersprache und des konkreten Programms das Fundament.

Damit stimmen das deduktiv entwickelte Framework von *Li et al. (2019)* und das induktiv entwickelte Modell der Debuggingfähigkeiten für Novizen weitestgehend überein. Lediglich der explizite Zusammenhang zwischen globalen und lokalen Debuggingstrategien findet sich nicht im Modell der Debuggingfähigkeiten für Novizen.

Zusammenfassend stellt insbesondere das heterogene Korpus, welches sich eben nicht nur aus Forschungsliteratur, sondern auch aus Programmierlehr- und Schulbüchern zusammensetzt, damit eine methodische Besonderheit im Vergleich zu den anderen diskutierten Modellen dar. Gerade aufgrund des praktischen Charakters von Debugging als sich durch Erfahrung entwickelnde Notwendigkeit der Programmierung erscheint es sinnvoll, Dokumente wie Lehr- und Schulbücher mit aufzunehmen, die auf entsprechenden praktischen Erfahrungen basieren. Darüber hinaus trägt diese Heterogenität zur Varianzmaximierung (vgl. z.B. (*Patton, 2002*)) der Stichprobe bei, sodass die Wahrscheinlichkeit, wesentliche Informationen nicht erheben zu können, sinkt (*Reinders, 2016*). Dabei zeigten sich Unterschiede der Schwerpunkte der verschiedenen Dokumentarten. Während in der Forschungsliteratur insbesondere die Rolle eines systematischen Vorgehens und an Novizen vermittelt wird, konzentrieren sich Schul- und Programmierlehrbücher vorwiegend auf die Vermittlung konkreter Strategien wie etwa das *print-Debugging* sowie Werkzeuge wie den Debugger.

3.5 Fazit

In dieser induktiven Untersuchung von Schulbüchern, Programmierlehrbüchern und wissenschaftlicher Literatur wurden vier dedizierte Debuggingfähigkeiten für Novizen identifiziert und in einem Modell der Debuggingfähigkeiten für Novizen dargestellt: Aufbauend auf allgemeinen Programmierfähigkeiten und Programmverständnis ist die *Anwendung eines systematischen Debuggingvorgehens*, also das Verfolgen eines zielgerichteten Plans, die Grundlage für einen erfolgreichen Debuggingprozess. Dieses Vorgehen wird dabei einerseits durch die Anwendung konkreter *Debuggingstrategien* unterstützt, wie beispielsweise das *print-Debugging* zum *Logging/Tracing* und damit dem Überprüfen von Hypothesen oder Nachverfolgen des Programmfluss oder aber dem *Slicing*, um den Suchraum zu reduzieren. Auch die *Anwendung von Mustern und Heuristiken* kann das systematische Vorgehen unterstützen, beispielsweise indem Muster für typische Fehler die direkte Formulierung und Überprüfung von Hypothesen ermöglichen. Zudem müssen entsprechende *Werkzeuge* wie etwa der *Debugger* oder auch Feedback der IDE adäquat eingesetzt werden.

Weitere, zeitgleich mit dieser Arbeit entstandene Kategorisierungen kommen dabei zu vergleichbaren Ergebnissen, daher ist eine theoretische Absicherung der Validität des resultierenden Modells gegeben. Trotz der Ähnlichkeiten verfügt das vorgestellte Modell der Debuggingfähigkeiten für Novizen über ein Herausstellungsmerkmal: Die Anwendung eines systematischen Debuggingvorgehens wird als explizite Fähigkeit angeführt und als Grundlage für einen erfolgreichen Debuggingprozess betrachtet.

Das Modell der Debuggingfähigkeiten für Novizen stellt damit einerseits im Sinne der didaktischen Strukturierung die Basis für die Vermittlung von Debugging dar: Für die jeweiligen Fähigkeiten müssen nun geeignete Ansätze und Materialien für den Informatikunterricht entwickelt und auf Wirksamkeit evaluiert werden. Andererseits stellt die fachliche Klärung aber auch die Basis für die weitere Untersuchung der zugrunde liegenden Perspektiven dar: So kann darauf aufbauend analysiert werden, welche Fähigkeiten möglicherweise bereits im Informatikunterricht aufgegriffen werden, welchen Beitrag diese Fähigkeiten zur Allgemeinbildung leisten oder aber welche Fähigkeiten die Schülerinnen und Schüler bereits aus ihrem Alltag als Lernvoraussetzung mitbringen.

4 Perspektive der Lehrkräfte

Debugging im Unterricht stellt für die Lehrkräfte eine große Herausforderung dar: Zunächst haben sie – genauso wie professionelle Softwareentwicklerinnen und -entwickler (Perscheid et al., 2017) – Debugging oftmals selbst nicht systematisch gelernt. Darüber hinaus fehlt es für die Vermittlung von Debugging an entsprechenden fachdidaktischen Ansätzen sowie konkreten Konzepten und Materialien für den Unterricht. In Konsequenz berichten Lehrkräfte häufig davon, von PC zu PC zu eilen, um allen Schülerinnen und Schülern möglichst gerecht zu werden – ein Phänomen, das umgangssprachlich auch als *Turnschuhdidaktik* bezeichnet wird. Um geeignete Konzepte und Materialien für den Unterricht zu entwickeln, die die Lehrkräfte unterstützen können, müssen die vorhandenen Erfahrungen der Lehrerinnen und Lehrer sowie ihre persönliche Sichtweise auf Debugging im Unterricht miteinbezogen werden. Aufbauend auf der *fachlichen Klärung* soll daher im Folgenden die Perspektive der Lehrkräfte untersucht werden. Dazu wird mittels einer Interviewstudie analysiert, mit welchen Herausforderungen die Lehrkräfte beim Debuggen im Unterricht konfrontiert sind und wie sie mit diesen Herausforderungen umgehen. Die Berücksichtigung dieser Perspektive ist für die Akzeptanz und damit Praxiswirksamkeit zu entwickelnder Konzepte und Materialien essentiell (Diethelm et al., 2011) und ermöglicht es, entsprechende Gestaltungshinweise abzuleiten.

4.1 Ziele der Untersuchung

Lehrkräfte sind tagtäglich mit den Programmierfehlern der Schülerinnen und Schüler konfrontiert. Allerdings fehlt es bisher an Untersuchungen, die die unterrichtspraktische Realität, wie beispielsweise die zumeist anekdotisch wiedergegebene Problematik der *Turnschuhdidaktik* in den Blick nehmen. Ziel dieser Untersuchung ist es daher, Herausforderungen der Unterrichtspraxis für Lehrkräfte im Kontext des Debuggings sowie deren bisherigen Umgang mit diesen zu analysieren, um Gestaltungshinweise für die Entwicklung von Konzepten und Materialien abzuleiten. Dazu wird **RQ2: Wie gehen Lehrkräfte mit Programmierfehlern im Unterricht um und wie vermitteln sie Debugging?** in mehrere Teilfragen unterteilt.

Zunächst soll untersucht werden, wie Lehrkräfte den Umgang sowie die Probleme der Schülerinnen und Schüler mit Programmierfehlern bewerten und wie sie selbst im Unterricht darauf reagieren. So können insbesondere Herausforderungen der Unterrichtspraxis, aber auch bestehende Best Practices erfasst werden.

(RQ2.1) Wie gehen Schülerinnen und Schüler sowie Lehrkräfte mit Programmierfehlern im Klassenzimmer um?

Darüber hinaus soll untersucht werden, welche Debuggingfähigkeiten auf welche Art und Weise und zu welchem Zeitpunkt im Unterricht vermittelt werden, um basierend auf den Erfahrungen der Lehrkräfte mit ihren Ansätzen Hinweise für die Gestaltung von Konzepten und Materialien ableiten zu können:

(RQ2.2) Wie wird Debugging im Unterricht vermittelt?

Weiterhin sollen die Gründe für die (Nicht-)Vermittlung von Debugging erhoben werden, um wesentliche Bedingungen für die Akzeptanz entsprechender Konzepte und Materialien in der Unterrichtspraxis zu erhalten:

(RQ2.3) Aus welchen Gründen wird Debugging (nicht) zum Unterrichtsthema?

4.2 Methodik

Für die Untersuchung dieser Forschungsfragen wurden ein Querschnittsdesign gewählt und Interviews mit 12 Gymnasiallehrkräften aus verschiedenen Bundesländern Deutschlands durchgeführt. Die qualitative Untersuchungsmethodik mit einem teilstrukturierten Leitfaden erlaubt dabei eine Kontrastierung der Fälle. Somit wird ein tiefer Einblick in die Unterrichtspraxis ermöglicht und es können beispielsweise nicht nur vermittelte Debuggingfähigkeiten, sondern insbesondere auch dabei gesammelte Erfahrungen der Lehrkräfte erfasst und ausgewertet werden. Der Leitfaden (siehe Anhang A) wurde dabei nach dem Verfahren von *Helfferrich (2019)* entwickelt. Die Interviews wurden anschließend wörtlich transkribiert und in Schriftdeutsch überführt. Zur Auswertung wurde die strukturierende qualitative Inhaltsanalyse nach *Mayring (2014)* angewandt. Durch die Kategorisierung der Textpassagen mit Hilfe eines Kodierleitfadens können so systematisch bestimmte Aspekte gemäß dem Forschungsinteresse aus dem Material herausgefiltert werden. Im Folgenden werden die Stichprobenziehung und die Auswertung im Detail beschrieben.

Sampling und Stichprobe

Da ein übergreifendes Ziel die Identifikation von Best Practices und existierenden Ansätzen für Debugging im Unterricht ist, sollten herausragende Lehrkräfte befragt werden. Auf diese Weise wird die potentiell „obere Grenze“ dessen erfasst, was im Informatikunterricht zu Debugging unterrichtet wird. Dementsprechend wurden im Sinne einer deduktiven Stichprobenziehung kriteriengeleitet gezielt geeignete Lehrkräfte ausgewählt (*Reinders, 2016*). Die Lehrkräfte der Stichprobe mussten dabei

- über mindestens fünf Jahre Berufserfahrung verfügen und
- entweder in der Aus- und Weiterbildung von Lehrkräften tätig sein
- oder Kooperationen mit Universitäten unterhalten.

Um gemäß dem Ziel der Varianzmaximierung qualitativer Stichprobenziehung eine möglichst hohe Heterogenität in der Stichprobe herzustellen (vgl. z.B. (Patton, 2002)), wurden nach Möglichkeit Lehrkräfte aus unterschiedlichen Bundesländern mit entsprechend unterschiedlichen curricularen Rahmenbedingungen und Anforderungen oder zumindest aus unterschiedlichen Regionen innerhalb der Bundesländer ausgewählt. Heterogenität in der Stichprobe senkt dabei die Wahrscheinlichkeit, wesentliche Informationen nicht erheben zu können (Reinders, 2016). Nichtsdestotrotz unterrichten alle befragten Lehrkräfte ob der curricularen Vorgaben objektorientierte Programmierung und verwenden als Programmiersprache fast ausschließlich Java.

Auswertung

Die Interviews wurden zunächst wörtlich transkribiert und anschließend mit Hilfe der strukturierenden qualitativen Inhaltsanalyse nach Mayring (2014) ausgewertet. Das deduktive Kategoriensystem (vgl. Tabelle 4.1) orientiert sich dabei an den drei Forschungsfragen und baut auf der zuvor vorgenommen fachlichen Klärung auf. Anschließend wurden Ankerbeispiele für die jeweiligen Kategorien identifiziert. Um zu vermeiden, dass wichtige Aspekte aufgrund zuvor definierter Kategorien vernachlässigt werden, wurden induktive Ergänzungen zugelassen. Die transkribierten Interviews bilden die Grundlage der Auswertung und die entsprechenden Textpassagen dienen als Analyseeinheiten. Für die eigentliche Codierung wurde die Software MAXQDA verwendet. Bezüglich der Interrater-Übereinstimmung wurden zwei der transkribierten Interviews ebenfalls von einem zweiten Forscher kodiert.

4.3 Ergebnisse

4.3.1 Wie gehen Schülerinnen und Schüler sowie Lehrkräfte mit Programmierfehlern im Klassenzimmer um? (RQ2.1)

Einschätzung der Probleme der Schülerinnen und Schüler

Erkenntnis 1: Die Reaktion der Schülerinnen und Schüler ist abhängig von der Lehrkraft.

Kategorie	Subkategorie	Ankerbeispiel
Umgang mit Fehlern im Informatikunterricht	Informationen über die Reaktionen der Schülerinnen und Schüler auf Fehler	Die motivierten und begeisterten Schülerinnen und Schüler versuchen, Fehler selbst zu beseitigen.
	Informationen über die Reaktionen der Lehrkräfte auf Programmierfehler der Schülerinnen und Schüler	Ich bemerke, wenn ein Schüler seine Hand hebt. Dann gehe ich hin und versuche, ihm zu helfen.
	Merkmale der Interaktion zwischen Lehrkraft und Schülerin bzw. Schüler	Die Schüler sagen einfach: „Es gab eine Fehlermeldung, ich weiß nicht, was sie bedeutet“.
	Informationen über häufige Fehler von Schülerinnen und Schülern	Typische Fehler sind z.B., dass sie versuchen, eine Methode aufzurufen, ohne vorher das entsprechende Objekt zu erzeugen.
Im Unterricht vermittelte Debuggingfähigkeiten	Aussagen zum Unterrichten eines systematischen Debuggingprozesses	Nein, so etwas stelle ich nicht vor.
	Aussagen zum Unterrichten von Debuggingstrategien	Dann weise ich sie darauf hin: „Füge einfach eine Zeile ein, die genau das ausgibt, was du wissen willst“.
	Aussagen über Heuristiken und Muster für häufige Fehler	Dann versuche ich zu erklären, woher der Fehler typischerweise kommt.
	Aussagen über die Vermittlung des Gebrauchs von Werkzeugen	Ich stelle den Debugger als Werkzeug vor.
	Art und Weise, Debugging zu unterrichten, z.B. auf individueller Basis oder in expliziten Debugging-Lektionen	Wenn es sich um einen häufigen Fehler handelt, dann spreche ich ihn immer vor der ganzen Klasse an, wenn es ein individuelles Problem ist, dann nur mit dem jeweiligen Schüler.
	Aussagen über den Zeitpunkt, zu dem Debugging vermittelt wird, wie z.B. zu Beginn des Kurses oder bei Bedarf	Normalerweise, wenn wir über Arrays sprechen, stelle ich den Debugger vor.
	Aktivitäten, um Debugging zu vermitteln	Ich verteile fehlerhaften Quellcode zum Üben.
Motivation der Lehrkräfte, Debugging (nicht) zu unterrichten	Aussagen darüber, warum Lehrkräfte Debugginginhalte in ihren Unterricht aufnehmen	Insbesondere die Hilflosigkeit und Frustration der Schülerinnen und Schüler.
	Aussagen darüber, warum Lehrkräfte Debugginginhalte in ihrem Unterricht vernachlässigen	Ich muss mich kurz fassen, denn die Lehrplanthemen haben Priorität.
	Aussagen von Lehrkräften, ob sie im Laufe ihrer Berufserfahrung ihren Unterricht in Bezug auf das Debugging verfeinert haben	Was sich im Laufe der Zeit entwickelt hat, ist meine Erfahrung: Ist das wirklich ein individuelles Problem oder werden viele Schüler damit zu kämpfen haben?

Tabelle 4.1: Kategoriensystem für die qualitative Inhaltsanalyse

Die Beobachtungen der Lehrkräfte bezüglich der Reaktionen der Schülerinnen und Schüler auf Fehler sind unterschiedlich. Einige Lehrkräfte berichten, dass die Mehrheit bei Problemen sofort die Hilfe der Lehrkraft sucht. Andere berichten, dass sie zunächst versuchen, Fehler durchaus alleine oder mit Mitschülerinnen bzw. Mitschülern zu lösen. Manchmal kann dieses Verhalten jedoch auch nur die Folge der Nichtverfügbarkeit der Lehrkraft sein, weil sie damit beschäftigt ist, anderen Schülerinnen oder Schülern zu helfen:

Erstmal müssen sie es selbst versuchen, weil meistens bin ich irgendwo unterwegs [und helfe woanders]. [...] [Manchmal] kommt man zeitlich gar nicht durch, dass man allen helfen kann, sondern dann sagt man irgendwann: „Ok, und jetzt ist Cut“, dann hab ich vielleicht der Hälfte weitergeholfen und der Rest muss es sich dann eben anschauen, wenn wir die Verbesserung machen. [I2, #00:04:06]

Erkenntnis 2: Unterstützung ist insbesondere für „schwächere“ Schülerinnen und Schüler notwendig.

Wenn Schülerinnen und Schüler versuchen, Fehler selbstständig zu beheben, unterscheiden die Lehrkräfte zwei Gruppen: „gute“ Schülerinnen und Schüler, die wenig Hilfe benötigen, und „schwächere“ Schülerinnen und Schüler. Letztere sind häufig überfordert und wenden einen unsystematischen Trial-and-Error-Ansatz an. Eine Lehrkraft fasst ihre Erfahrungen wie folgt zusammen:

Die, die programmieren können, erkennen den Fehler und beheben ihn, und die, die nicht programmieren können, probieren einfach etwas aus, bis keine Fehlermeldung mehr da ist [...] sie machen halt solange Strichpunkte überallhin oder „ints“ irgendwohin, bis diese Fehlermeldung nicht [mehr] auftritt. [I9, #00:08:29]

Erkenntnis 3: Auch Kompilierzeitfehler und der Umgang mit entsprechenden Fehlermeldungen stellen selbst nach einiger Programmiererfahrung eine große Hürde für Schülerinnen und Schüler dar.

Die Lehrkräfte berichten, dass einfache Syntaxfehler (wie fehlende Semikolons oder Klammern) nach einigen Wochen problemlos behoben werden. Viele Lehrkräfte sind sich jedoch einig, dass andere Arten von Kompilierzeitfehlern, wie etwa im Umgang mit Datentypen, weiterhin große Schwierigkeiten bereiten. So antwortet eine Lehrkraft auf die Frage nach logischen Fehlern:

In den 10. Klassen ist es weniger das große Problem, weil die meisten Schüler in der Zeit gar nicht so weit kommen, dass dieses Programm komplett irgendwie läuft. [I7, #00:04:37]

Eine Lehrkraft beschreibt Probleme, die sich aus mehrdeutigen (Java-) Fehlermeldungen ergeben, die an der falschen Stelle erscheinen – zumindest aus der Sicht der Schülerinnen und Schüler:

*„Reached end of file while parsing“ oder so etwas [ist] für die Schüler jetzt nicht wirklich klar, weil der Fehler an einer anderen Stelle steht, als sie ihn eigentlich gemacht haben.
[I8, #00:10:16]*

Erkenntnis 4: Die Unterstützung durch die Lehrkräfte wird erschwert, da die Schülerinnen und Schüler nicht in der Lage sind, konkrete Fragen zu stellen.

Insgesamt stellen die Schülerinnen und Schüler überwiegend unpräzise Fragen, wenn sie schließlich die Lehrkraft um Hilfe bitten.

*Der Großteil der Schüler schaut dann aber entsetzt, meldet sich und sagt: „Das ist rot, da ist ein Fehler“, und will dann im Endeffekt von dir die Lösung präsentiert haben.
[I11, #00:03:22]*

Reaktion der Lehrkräfte

Erkenntnis 5: Die Lehrkräfte eilen meist von einem Schüler-PC zum anderen und versuchen zu helfen.

Im Allgemeinen stellten die Lehrerinnen und Lehrer eine große Hilflosigkeit und Frustration im Umgang mit Fehlern fest, sodass sie am Ende tatsächlich von einem Schüler-PC zum anderen hasten, den Fehler erklären und Hinweise zur Fehlerbehebung geben.

Die meisten Fehlermeldungen, wenn man bei Google eingibt, kriegt man ja raus, woran es liegt, aber das trauen sich oder wollen relativ wenige, die haben immer da gerne den Lehrer nebendran. [I7, #00:11:00]

Erkenntnis 6: Wenn Lehrkräfte Schülerinnen und Schülern individuell helfen, bemühen sie sich, vorwiegend Verständnisschwierigkeiten und Fehlvorstellungen zu adressieren, was aufgrund der zur Verfügung stehenden Zeit oftmals nicht möglich ist.

In dem kleinen Zeitraum, der für die individuelle Unterstützung zur Verfügung steht, besteht das primäre Ziel gewöhnlich darin, zugrunde liegende Verständnisschwierigkeiten und Fehlvorstellungen zu beseitigen.

Je nachdem wie viele Schüler man dann hat und wie viele gerade Hilfe brauchen, wie viel Zeit man also für den Einzelnen hat, versucht man dann wirklich sich hinzusetzen und zu sagen: „Lies dir das doch mal durch, was bedeutet das? Was könnte dann das

Problem sein? Wie hast du es denn sonst gemacht wo kein Fehler war?“ Wenn man jetzt aber gerade viel hat, [...] dann sagt man halt schnell: „Da fehlt ein Strichpunkt“, „da ist das 's' klein“, „da hast du das 'new' vergessen“, also je nachdem wie die allgemeine Situation gerade ist. [I7, #00:03:22]

Erkenntnis 7: Eine Best Practice stellt das Einfordern von Selbstständigkeit im Umgang mit Fehlern dar.

Drei der befragten Lehrkräfte verfolgen einen Ansatz gemäß der „Turnschuhdidaktik“ und ausgiebiger individueller Unterstützung nur in den ersten Wochen und verlangen von den Schülerinnen und Schülern danach eigenständiges Denken und Experimentieren. So erwarten sie von den Lernenden, dass sie, bevor die Lehrkraft um Hilfe gebeten werden kann, selbstständig Versuche zur Fehlerbehebung unternehmen und zuerst die Mitschülerinnen und Mitschüler konsultieren. Eine Lehrkraft berichtet zum Beispiel, dass eines ihrer Hauptprinzipien darin besteht, dass jede Fehlermeldung der Klasse nur einmal erklärt wird:

Also solche Fehlermeldungen werden einmal dann in der 10. für alle angesprochen, aber einmal, das ist auch bekannt, jede Fehlermeldung darf einmal gefragt werden, danach: „Nein, hatten wir schon, entweder ihr habt es euch notiert oder ihr habt es euch gemerkt oder jemand von der Gruppe weiß es, nicht mehr meine Sache“, da bin ich stur. [I11, #00:06:49]

Eine andere Lehrkraft berichtet:

Weil sonst bin ich tatsächlich der Ersatz für den Compiler, der dann immer sagt: „Denk dran, hier noch ein Strichpunkt und da.“ Dann trauen sie sich nicht mal auf Compile zu drücken. Nein, das müssen sie einfach lernen, sich mit den Fehlern auseinanderzusetzen, sonst bin ich da im Unterricht auch überfordert. [I3, #00:22:35]

Interessanterweise tendieren Lehrkräfte, die einen solchen Ansatz des Einforderns von Selbstständigkeit verfolgen, auch dazu, Kompilierzeitfehler als eine geringere Hürde zu sehen.

Bei Syntaxfehlern, [...] da sagt ja der Compiler was Sache ist, das verstehen die [Schülerinnen und Schüler] normalerweise schon. [I12, #00:08:15]

Eine Lehrkraft berichtet von mehr Eigenständigkeit, spezifischeren Fragen und einem insgesamt besseren Umgang mit Fehlern im Vergleich zu ihrem Unterricht, bevor sie verstärkt Selbstständigkeit eingefordert hat.

Diese Situation „ja da war eine Fehlermeldung“, - ja was stand denn da?“, - weiß ich nicht, die habe ich weggeklickt“, also das Wichtigste an der Fehlermeldung war das rote Kreuz oben rechts, zumachen, weg, und dann den Lehrer rufen, nicht umgekehrt, [...], können sie mir mal übersetzen?“ [habe ich vorher ganz oft erlebt] [I11, #00:24:34]

4.3.2 Wie wird Debugging im Unterricht vermittelt? (RQ2.2)

Vermittelte Fähigkeiten

Erkenntnis 8: Die Lehrkräfte vermitteln keinen systematischen Debuggingprozess.

Keine der Lehrkräfte vermittelt irgendeine Art von modellhaftem Debuggingprozess, der über „Lies zuerst die Fehlermeldungen“ hinausgeht. So konnte dieser Kategorie keine Kodierung zugeordnet werden, obwohl explizit danach gefragt wurde.

Erkenntnis 9: Es werden verschiedene Debuggingstrategien unsystematisch vermittelt.

Die Mehrheit der Lehrkräfte gibt den Schülerinnen und Schülern mindestens eine Debuggingstrategie an die Hand, aber die entsprechenden Strategien werden – mit wenigen Ausnahmen – nicht systematisch unterrichtet oder geübt. Ein Schwerpunkt liegt dabei auf Strategien, die das schrittweise Nachvollziehen von Code ermöglichen. Daher gehören die Verwendung des Debuggers, *print-Debugging* oder die Verwendung anderer werkzeugspezifischer Funktionen, die beim *Tracen* helfen – siehe Erkenntnis 11 – zu den am häufigsten genannten Strategien. Darüber hinaus wurden manchmal Internet-Recherchen zur Behandlung von Fehlermeldungen, Auskommentieren, basales Slicing oder Testen (insbesondere das Testen durch visuelle Beobachtung) erwähnt.

Erkenntnis 10: Die Lehrkräfte konzentrieren sich auf Heuristiken und Muster für häufige Fehler.

Dennoch liegt das Hauptaugenmerk der Lehrerinnen und Lehrer auf Mustern und Heuristiken für den Umgang mit typischen Fehlern, insbesondere für Laufzeit- und Kompilierzeitfehler. Dazu gehören beispielsweise die Heuristik, die oberste Fehlermeldung zuerst zu beheben, die Zuordnung von konkreten Fehlermeldungen zu Fehlern oder aber Muster für typische Fehler zu geben:

Ich versuche zu erklären, wo der Fehler typischerweise auftritt. Oft dauert die Schleife zu lange, sie haben ihr Array zu klein gemacht oder vergessen, dass es bei 0 beginnt. Solche Dinge. [I7, #00:09:38]

Erkenntnis 11: Als Werkzeuge werden insbesondere der Debugger und weitere werkzeugspezifische Features eingeführt, die beim schrittweisen Nachvollziehen des Codes helfen – allerdings mit gemischten Ergebnissen.

Hinsichtlich der Werkzeuge, die für das Debugging im Unterricht eingeführt werden, berichten Lehrerinnen und Lehrer teilweise vom Debugger – allerdings fast ausschließlich in einer didaktisch reduzierten Version (wie z.B. in BlueJ) – sowie weiterer werkzeugspezifischer Features, wie z.B. dem Objektinspektor in Greenfoot und BlueJ¹⁶. Beim Debugger sind sich die Lehrkräfte einig, dass er nur den „guten“ Schülern und Schülerinnen zu helfen scheint. Dies spiegelt sich auch in seinem Einsatz wider:

Allerdings ist das eine Strategie, die jetzt mit den „schwächeren“ Schülern nicht so hilfreich ist. Also da finde ich den Debugger schwierig. Also da merke ich, die trauen sich da nicht ran. Also ich stelle das denen vor. Die „guten“ Schüler kommen damit gut zurecht [...], aber einige „schwächere“ Schüler tun sich da schwer. [I3, #00:24:11]

Eine Lehrkraft berichtet, dass die Einführung des Debuggers nicht funktioniert hat. Folglich blieb es für sie ein einmaliges Experiment.

Das hat eher [...] noch mehr Aufwand und mehr Verwirrung gekostet. [I8, #00:08:32]

Wann Debugging unterrichtet wird

Aus den Daten sind zwei Zeitpunkte zu unterscheiden, zu denen Debugging unterrichtet wird: Zu Beginn des Kurses oder Schuljahres oder bei Bedarf – wenn entsprechende Fähigkeiten erforderlich sind.

Erkenntnis 12: Der Versuch, Kenntnisse vorwiegend zu Beginn des Programmierunterrichts „auf Vorrat“ aufzubauen, hilft den Schülerinnen und Schülern kaum.

Einige Lehrkräfte berichten, dass sie Maßnahmen ergriffen haben, um Probleme bereits in einem frühen Stadium des Programmierkurses bzw. Schuljahres zu verhindern. Diese Maßnahmen bestanden beispielsweise aus der Unterscheidung verschiedener Fehlertypen, grundlegender Heuristiken für die Behandlung bestimmter Fehler (oder Fehlermeldungen) oder der Einführung eines werkzeugspezifischen Debuggers oder Objektinspektors. Die Lehrkräfte berichten jedoch, dass unterstützende Materialien und „Wissen auf Vorrat“ (vgl. *Dohmen (2000)*) kaum verwendet oder angewendet wurden.

¹⁶Der Objektinspektor ermöglicht es, die Werte der statischen und Instanzfelder eines Objekts anzuzeigen, vgl. *Kölling et al. (2003)*.

Die Schüler glauben, sie haben es verstanden, und dann schauen sie es nie wieder an. [I9, #00:14:42]

Erkenntnis 13: Die meisten Debuggingfähigkeiten werden bei Bedarf vermittelt.

Hauptsächlich wird Debugging bedarfsorientiert unterrichtet. Ein gängiges Beispiel hierfür ist die Einführung von Arrays. Für dieses Thema werden die entsprechenden Fehlermeldungen (und ihre typischen Ursachen) oder Strategien, wie z.B. die Verwendung des Debuggers, in der Regel in zwei möglichen Phasen behandelt: entweder während der allgemeinen Einführung in das Thema oder als Reaktion auf den ersten fehlerhaften Speicherzugriff und die entsprechende Fehlermeldung.

Und wenn wir dann sowas wie NullPointerExceptions haben, wenn wir mit Objekten anfangen oder jetzt bei den Feldern, dann diese ArrayIndexOutOfBoundsException, wenn das das erste Mal kommt, lege ich meistens den Bildschirm über den Beamer hin und sage: „Hört zu, Leute, die Fehlermeldung kann jedem von euch in den nächsten Wochen öfters passieren, das könnte dahinterstecken.“ Dann thematisieren wir das mal, und dann versuche ich das zu klären und zu klären, wo da häufig der Fehler ist. [I7, #00:09:38]

Wie Debugging unterrichtet wird

Erkenntnis 14: Lehrkräfte tendieren dazu, keine expliziten Unterrichtseinheiten oder Stunden zum Thema Debugging durchzuführen, sondern stattdessen relevante Fähigkeiten auf individueller Basis zu vermitteln.

Keine der Lehrkräfte verwendet eine explizite Unterrichtsstunde oder -phase für das Debuggen, außer falls sie den Debugger einführt. Wenn Inhalte alle Schülerinnen und Schüler betreffen (wie z.B. das erste Auftreten eines bestimmten Fehlers, die Einführung in den Debugger, ...), werden diese vor der gesamten Klasse angesprochen. Ein großer Teil der Unterstützung der Schülerinnen und Schüler ist jedoch individueller Natur. Die Lehrerinnen und Lehrer berichten auch, dass in diesen einzelnen Unterstützungsphasen durch Beobachtung im Sinne von *Lernen am Modell* (vgl. Bandura (1962)) gelernt wurde, z.B. wie die Lehrkraft bei der Fehlersuche im Code der Schülerin bzw. des Schülers die Strategie des Auskommentierens anwendet.

Oftmals ist es so, dass ich es einmal mehr oder minder vorführe, also mehr ein „Lernen durch Beobachtung“ als ein „Lernen durch Anleitung“. [I11, #00:20:48]

In Bezug auf bestimmte debuggingbezogene Aktivitäten erwähnen einige der Lehrkräfte die Verwendung von Debuggingaufgaben. Auch *debugging logs* oder *reviews* wurde vereinzelt angeführt.

4.3.3 Aus welchen Gründen wird Debugging (nicht) zum Unterrichtsthema? (RQ2.3)

Erkenntnis 15: Lehrkräfte vermitteln Debuggingfähigkeiten hauptsächlich aufgrund ihrer Erfahrung mit der Hilflosigkeit von Schülerinnen und Schülern.

Der Hauptgrund, den Lehrkräfte für die Integration von Debugginginhalten in ihren Unterricht angeben, ist die erlebte Hilflosigkeit und Frustration der Schülerinnen und Schüler im Umgang mit Programmierfehlern. Einige der Lehrkräfte berichten von einer Weiterentwicklung ihrer Debugginginhalte, die sie im Laufe der Jahre an die typischen Fehler der Schülerinnen und Schüler angepasst haben.

Vor allem die Hilflosigkeit der Schüler oder die Frustration der Schüler [...], es gibt halt in der 10ten immer ein paar, die aufgegeben haben. Das würde ich gerne vermeiden. [...] Wenn der Lehrer sagt, es ist gut, Fehler zu machen, zählt es einfach nicht so viel, wie wenn der Computer sagt: „Funktioniert nicht“. Dann ist ein Fehler keine Herausforderung, sondern Frustration. [I9, #00:21:47]

Erkenntnis 16: Die wichtigsten Gründe dafür, Debugging nicht zu unterrichten, sind Zeitmangel, dass Debugging kein explizites Thema im Lehrplan ist sowie fehlende Konzepte und Materialien.

Alle Lehrkräfte würden gerne mehr debuggingbezogene Themen in ihren Unterricht integrieren. Die Lehrkräfte begründen ihre Entscheidung, Debugging im Unterricht auszusparen, mit dem herrschenden Zeitmangel. Dazu gehören sowohl Unterrichtszeit als auch Zeit für die Vorbereitung und Erstellung geeigneter Konzepte. Als weiterer Grund wird angegeben, dass das Debugging kein explizit genannter Inhalt des Lehrplans ist – obwohl es einen zentralen Schritt in der Programmierung darstellt:

In den Lehrplänen ist es eigentlich nicht so als eigenes Thema enthalten, [...] und okay, ich programmiere, was brauche ich zum Programmieren, ich brauche die Programmierkonzepte, ich brauche die Datenstrukturen, also das sind sozusagen Themen, die dann in der Unterrichtsplanung landen, und das Debugging ist mehr auf den Prozess gemünzt und wird daher selten Unterrichtsgegenstand. [I1, #00:17:15]

Daher „vernachlässigen“ Lehrkräfte Debugging oftmals zugunsten von Inhalten, die ausdrücklich in den Lehrplänen gefordert werden. Darüber hinaus argumentieren sie, dass

sie keine geeigneten Konzepte kennen und einen Ansatz benötigen, um mehr Inhalte in geeigneter Weise zu vermitteln. Sie berichten, dass es kein Material gibt, auch nicht in (Schul-)Büchern:

Auch in der Literatur [...], man findet etwas zu Datenbanken, zum Programmieren, zu diesen ganzen Feldern der Informatik, aber [Debugging] ist selten so ein Thema für sich. [I1, #00:16:47]

Erkenntnis 17: Eine wichtige Quelle für die Vermittlung von Debugging ist das eigene Debuggingverhalten der Lehrkräfte.

Vier der Lehrkräfte gaben ausdrücklich an, dass die Strategien und die Unterstützung, die sie den Schülerinnen und Schülern bieten, auf ihrem eigenen persönlichen Debuggingverhalten basieren:

[Ich habe] überlegt, wie habe ich das denn selbst damals an der Universität gemacht und wie hat das funktioniert? [I2, #00:28:04]

4.4 Interpretation

Im Folgenden sollen die Ergebnisse der Interviewstudie interpretiert und mit existierenden Erkenntnissen des Forschungsstandes kontrastiert werden. Zunächst erlaubt diese Untersuchung wertvolle Einblicke in die Unterrichtspraxis und deren Herausforderungen, wobei aufgrund der bewussten Auswahl von Lehrkräften, die entweder mit Universitäten und aktueller Forschung kooperieren oder in der Lehrkräfteausbildung tätig sind, tendenziell die „obere Grenze“ dessen erfasst wurde, was im Unterricht zum Debugging vermittelt wird.

Auch für diese Lehrkräfte scheint jedoch die anekdotisch wahrgenommene *Turnschuhdidaktik* tatsächlich Realität der Schulpraxis zu sein: Die meisten Lehrkräfte berichten davon, von Arbeitsplatz zu Arbeitsplatz zu eilen, um die Schülerinnen und Schüler bei der Fehlerbehebung zu unterstützen (Erkenntnis 5), während die Lernenden ein unsystematisches Trial-and-Error-Verhalten zeigen, oftmals frustriert sind und unsystematische Fragen stellen und es den Lehrkräften damit erschweren, zielgerichtet zu unterstützen (Erkenntnis 2, 4).

Betrachtet man die Aussagen der Lehrkräfte bezüglich der von ihnen vermittelten Debugginginhalte, ist durchaus eine Übereinstimmung mit dem Modell der Debuggingfähigkeiten für Novizen festzustellen: Die befragten Lehrkräfte unterrichten verschiedene Debuggingstrategien – wenn auch vorwiegend unsystematisch – (Erkenntnis 9), Heuristiken für häufige Fehler (Erkenntnis 10) sowie die Verwendung von Werkzeugen (Erkenntnis

11). Dies gilt jedoch nicht für die Anwendung eines systematischen Debuggingvorgehens (Erkenntnis 8): Dieser Kategorie konnte keine Kodierung zugeordnet werden, obwohl explizit danach gefragt wurde. Vermutlich wenden Lehrkräfte beim Debugging selbst ein systematisches Vorgehen an, allerdings handelt es sich dabei möglicherweise um einen unbewussten Prozess. Die meisten dieser Fähigkeiten werden dabei nicht explizit oder für die ganze Klasse in dedizierten Unterrichtseinheiten oder -phasen zum Debugging vermittelt (Erkenntnis 13), sondern in der individuellen Beratung von Schülerinnen und Schülern vorgeschlagen oder vorgeführt (Erkenntnis 14). Und auch in der individuellen Unterstützung werden vorwiegend Verständnisschwierigkeiten oder Fehlvorstellungen adressiert und nicht Vorgehensweisen unterrichtet, die dabei helfen, Fehler selbstständig zu finden und zu beheben (Erkenntnis 6).

Insgesamt zeigt sich damit, dass Debugging im Informatikunterricht mit gemischtem Erfolg (Erkenntnis 11) und nur selten explizit (Erkenntnis 13, 14) vermittelt oder thematisiert wird, auch weil es Lehrkräften an einer entsprechenden Systematik relevanter Fähigkeiten sowie entsprechenden Konzepten für den Informatikunterricht fehlt (Erkenntnis 16).

Darüber hinaus tragen die Erfahrungen und Berichte der Lehrkräfte aber auch zur Identifikation von Hinweisen für die Entwicklung von Materialien und Konzepten bei – sowohl hinsichtlich der Akzeptanz und Verbreitung in der Unterrichtspraxis als auch bezüglich der inhaltlichen Ausgestaltung –, die im Folgenden diskutiert werden.

Vorlieben der Lehrkräfte. Die Daten weisen darauf hin, dass die Lehrkräfte Debugging auf der Grundlage ihres eigenen Debuggingverhaltens unterrichten (Erkenntnis 17): Eine Lehrkraft, die selbst typischerweise den Debugger verwendet, um Fehler zu finden, unterrichtet den Umgang mit diesem auch. Im Gegensatz dazu konzentrieren sich Lehrkräfte, die selbst nie mit dem Debugger gearbeitet haben und stattdessen *print*-Debugging verwenden, eher auf solche Strategien. Lehrkräfte haben zumeist – genauso wie professionelle Entwicklerinnen und Entwickler (Perscheid et al., 2017) – nie systematisch Debuggen „gelernt“. Gerade vor dem Hintergrund, dass Lehrkräfte oftmals selbst nur geringe Programmiererfahrung haben, sollten solche persönlichen Präferenzen und eine entsprechende Varianz in Materialien und Konzepten berücksichtigt werden, um die Akzeptanz in der Schulpraxis sicherzustellen.

Einpassung in den Unterricht. Die Lehrkräfte berichten davon, dass die Vermittlung entsprechender Debuggingfähigkeit bereits zu Beginn des Programmierunterrichts im Sinne von „Wissen auf Vorrat“ (Dohmen, 2000) nicht geeignet erscheint (Erkenntnis 12). So nützt die Einführung des Debuggers wenig, bevor die diesbezüglichen Programmierprojekte der Schülerinnen und Schüler den Einsatz dieses Werkzeuges rechtfertigen, und findet kaum

Anwendung. Stattdessen sollten entsprechende Fähigkeiten *bei Bedarf* vermittelt werden. Darüber hinaus müssen die erforderlichen Konzepte und Materialien sich zeitlich und thematisch passend in die bisherige Unterrichtsplanung integrieren lassen (Erkenntnis 16), um vor dem Hintergrund der knappen Unterrichtszeit und curricularer Zwänge die Akzeptanz in der Praxis sicherzustellen.

Rolle von Kompilierzeitfehlern. Wie in Kapitel 2.3 diskutiert, wird Debugging gerade auch in der Hochschullehre teilweise als beginnend mit Laufzeitfehlern aufgefasst und das Finden und Beheben von Kompilierzeitfehlern nicht als Debugging im eigentlichen Sinne betrachtet. Die Daten dieser Untersuchung weisen aber darauf hin, dass insbesondere auch Kompilierzeitfehler für viele Schülerinnen und Schüler eine große Hürde darstellen (Erkenntnis 3). Auch für die Behebung von Kompilierzeitfehlern werden also geeignete Vorgehensweisen benötigt – zumindest dann, wenn im Unterricht traditionelle textbasierte Programmierung und nicht block- oder *framebased*-Ansätze verwendet werden, bei denen viele dieser Fehler nicht mehr möglich sind (Altadmri, Kölling und Brown, 2016). Eine mögliche Erklärung für die Bedeutung von Kompilierzeitfehlern kann die begrenzte Unterrichtszeit in der Schule sein: Spätestens am Ende der Unterrichtsstunde erhalten die Schülerinnen und Schüler die Lösungen für die Aufgaben, unabhängig davon, wie viele Fehler in ihrem Programm noch vorhanden sind. Daher bilden sie keine Heuristiken und Erfahrungen im Umgang mit bestimmten Fehlern aus. Dementsprechend müssen Schülerinnen und Schülern in entsprechenden Konzepten und Materialien Vorgehensweisen und Strategien für alle Fehlertypen vermittelt werden. Möglicherweise wird dieses Ergebnis allerdings dadurch eingeschränkt, dass Lehrkräfte die von Lernenden häufig gemachten Fehler nur schlecht einschätzen können (Brown und Altadmri, 2014).

Selbstständigkeit. Die Berichte der Lehrkräfte lassen darauf schließen, dass die Reaktionen der Schülerinnen und Schüler unterschiedlich und von der Lehrkraft und ihrem Handeln abhängig sind (Erkenntnis 1). Anhand der Daten kann folgende These aufgestellt werden: Lehrkräfte, die ein hohes Maß an Selbstständigkeit einfordern, berichten von mehr Selbstständigkeit bei der Fehlerbehebung (Erkenntnis 7) und stufen beispielsweise auch Kompilierzeitfehler als eine geringere Hürde ein. Das Konzept der *erlernten Hilfflosigkeit* (Peterson, Maier und Seligman, 1993) könnte hier eine Rolle spielen, da es keine nennenswerten Unterschiede in der Art der Debuggingstrategien gibt, die den Schülerinnen und Schülern jeweils beigebracht werden. Allerdings verwenden zwei dieser drei Lehrkräfte bereits mit Beginn des Programmierunterrichts einen (agilen) projektbasierten Ansatz, der einen großen Einfluss auf ihre epistemologischen Unterrichtsansätze haben kann. Nichtsdestotrotz erscheint ein starker Fokus auf die Förderung von Selbstständigkeit in Materialien und Konzepten daher vielversprechend. Obwohl die Rolle erlernter Hilfflosigkeit mit den

gegebenen Daten nicht weiter untersucht werden kann, bietet sie doch eine interessante Perspektive für zukünftige Forschung.

Schülerinnen und Schüler. Gemäß den Ausführungen der Lehrkräfte bilden „durchschnittliche“ bis „schwache“ Schülerinnen und Schüler die primäre Zielgruppe für Konzepte und Materialien bezüglich Debugging (Erkenntnis 2, 11). Dabei ist der hilflose und unsystematische Trial-and-Error-Ansatz, den die Lehrkräfte für diese „schwächeren“ Schülerinnen und Schüler beschreiben, im Einklang mit der Literatur (Murphy et al., 2008). Auch Beobachtungen, wie das schnelle „Wegklicken“ von Fehlermeldungen oder die Frustration und Hilflosigkeit decken sich mit dem Forschungsstand (Perkins et al., 1986) und müssen in entsprechenden Konzepten und Materialien adressiert werden, indem die Schülerinnen und Schüler angemessen unterstützt werden.

Limitationen und Güte

Ziel dieser Untersuchung war es, die Perspektive der Lehrkräfte, also deren Erfahrungen und auch existierende Ansätze aus der Unterrichtspraxis zu erheben. Daher ist die Studie und ihr methodisches Design darauf ausgelegt, übergreifende Muster zu identifizieren und Kernprobleme aufzudecken – und damit zur Theoriegenese und nicht zu deren Quantifizierung beizutragen. Damit stellen die Ergebnisse natürlich nur einen Ausschnitt der Unterrichtspraxis dar: Wie in qualitativen Studien üblich, ist die Reichweite der Untersuchung begrenzt und es besteht kein Anspruch auf eine (exemplarische) Verallgemeinerung (Bortz und Döring, 2006). Darüber hinaus unterrichteten fast alle der befragten Lehrkräfte in der Programmiersprache Java – auch aufgrund der Verbreitung von Java für den Programmierunterricht (Hubwieser et al., 2015). Damit besteht die Gefahr, dass insbesondere sprachspezifische Probleme und Vorgehensweisen festgestellt wurden (*Selektionsbias*). Dennoch lassen hauptsächlich drei Aspekte vermuten, dass die Ergebnisse nicht nur für die untersuchten Lehrkräfte zutreffen und sprachunabhängig sind, also die **Validität** dieser Untersuchung gewährleistet ist. Einerseits sind große Ähnlichkeiten in den Berichten der Lehrkräfte festzuhalten. So ergab sich im Zuge der iterativen Durchführung und Auswertung der Interviews, dass die Erhebung zusätzlicher Interviews keine Veränderungen in den Ausprägungen der Kategorien mehr nach sich zog (*Empirische Sättigung*, vgl. Strübing et al. (2018)). Weiterhin decken sich die Ergebnisse der Untersuchung mit dem Forschungsstand (der insbesondere wenig Bezug zu Java hat), wie etwa bezüglich der im Unterricht vermittelten Inhalte oder aber der Frustration der Schülerinnen und Schüler (*korrelative Gültigkeit*, vgl. Mayring (1985)). Drittens sollte durch die bewusste Auswahl von Lehrkräften, die entweder mit Universitäten und aktueller Forschung kooperieren oder in der Lehrkräfteausbildung tätig sind, die „obere Grenze“ dessen erhoben werden, was zum The-

ma Debugging im Informatikunterricht vermittelt wird (*Stichprobengültigkeit*, vgl. Mayring (1985)). Dies lässt nicht darauf schließen, dass es *keine* Unterschiede gibt zwischen Lehrkräften, deren Art und Weise, mit Programmierfehlern umzugehen, Debugging im Unterricht zu adressieren und – auch in Konsequenz – den Herausforderungen der Unterrichtspraxis –, wie sich auch im Zuge der Darstellung der Ergebnisse gezeigt hat. Eine differenzierte Betrachtung der Ursachen solcher Unterschiede ist aber eben nicht Ziel dieser Studie, deren Interpretation daher nicht über Thesen zu plausibel erscheinenden Zusammenhängen hinausgehen kann.

Zur Absicherung der Wissenschaftlichkeit und Qualität der gewonnenen Ergebnisse wurden darüber hinaus weitere Maßnahmen unternommen: Das deduktive, auf Basis unter anderem der fachlichen Klärung entwickelte Kategoriensystem, trägt überdies zur Validität der Untersuchung bei (*Semantische Gültigkeit, Konstruktvalidität*, vgl. Mayring (1985)). Durch die Interrater-Übereinstimmung soll hingegen die **Intersubjektivität** der Untersuchungsergebnisse sichergestellt werden (vgl. Rädiker und Kuckartz (2019)). Außerdem soll durch die Verfahrensdokumentation im Sinne der Darstellung des Erhebungsinstrumentes und der Analyseschritte sowie die Regelgeleitetheit des systematisches Vorgehens (vgl. Mayring (2002)) die **Transparenz** des Vorgehens gewährleistet werden¹⁷.

4.5 Fazit

Zusammenfassend untersucht diese Studie damit die bisher vernachlässigte Perspektive der Lehrkräfte. Lehrkräfte sind tagtäglich mit den Programmierfehlern der Schülerinnen und Schüler konfrontiert und die Berücksichtigung ihrer Perspektive ist daher für die Akzeptanz und Verbreitung von Konzepten und Materialien essentiell.

Bezüglich der Art und Weise, wie die Schülerinnen und Schüler im Programmierunterricht mit Fehlern umgehen, und der Herausforderungen, die sich daraus für die Lehrkräfte ergeben, berichten sie, dass besonders „schwächere“ Schülerinnen und Schüler oft überfordert und hilflos sind. Sie verwenden häufig einen Trial-and-Error-Ansatz, sind unselbstständig und fordern die Hilfe der Lehrkraft ein, wobei sie zumeist nur unpräzise Fragen stellen. Gerade Kompilierzeitfehler stellen auch nach einer gewissen Programmiererfahrung ein durchaus nennenswertes Problem für viele der Schülerinnen und Schüler dar. Die Lehrkräfte wiederum eilen tatsächlich oftmals von einem Arbeitsplatz zum anderen und versuchen, die Schülerinnen und Schülern bei der Fehlerbehebung zu unterstützen (*Turnschuhdidaktik*). Dementsprechend müssen Schülerinnen und Schüler entsprechende Vorgehensweisen

¹⁷Zur Diskussion etablierter Gütekriterien qualitativer Forschung und von Maßnahmen zur Qualitätssicherung, insbesondere auch der unterschiedlichen Zuordnung, in den (naturwissenschaftlichen) Fachdidaktiken siehe Göhner und Krell (2020).

vermittelt werden, damit diese selbständig ihre Fehler finden und beheben können, sodass auch die Lehrkräfte entlastet sind.

Im Hinblick auf den Unterricht von Debugging und Umgang der Lehrkräfte mit den Problemen der Schülerinnen und Schüler zeigen die Ergebnisse, dass einige Strategien und Heuristiken, aber kein systematisches Vorgehen zum Finden und Beheben von Fehlern vermittelt werden – und damit eben kaum Vorgehensweisen, die die Selbstständigkeit der Schülerinnen und Schüler fördern. Abgesehen von der Einführung des Debuggers setzt keine der Lehrkräfte eine explizite Unterrichtsphase oder -einheit zur Vermittlung von Debuggingfähigkeiten ein. Dabei fehlt den Lehrkräften ein Konzept bzw. eine Systematisierung dessen, was vermittelt werden könnte respektive müsste: Die Daten deuten darauf hin, dass sie Debugging auf der Grundlage ihres eigenen Debuggingprozesses unterrichten – wobei sie Debugging typischerweise selbst auf unstrukturierte Weise „gelernt“ haben. Die Erfahrung der Lehrkräfte weist dabei darauf hin, dass die Vermittlung von Debugging „auf Vorrat“ kaum erfolgreich ist, aber insbesondere das aktive Einfordern von Selbstständigkeit eine erfolgreiche Best Practice darstellt.

Der Hauptgrund für Lehrkräfte, Debugging nicht zu unterrichten, ist dabei der Mangel an Zeit – sowohl im Unterricht als auch für die Vorbereitung geeigneter Konzepte und Materialien. Außerdem berichten sie, dass Debugging kein expliziter Inhalt des Lehrplans ist und es an Konzepten und Materialien für den Unterricht fehlt.

Diese Ergebnisse erlauben es damit, folgende Hinweise für die Gestaltung von Konzepten und Materialien für den Unterricht abzuleiten: Konzepte und Materialien sollten. . .

- die Vielfalt der persönlichen Debuggingvorlieben der Lehrkräfte berücksichtigen,
- sich zeitlich und thematisch passend in bisherige Unterrichtsplanung bedarfsorientiert integrieren lassen,
- auch Ansätze zum Umgang mit Kompilierzeitfehlern enthalten,
- sich vor allem auf das Fördern aber auch Fordern von Selbstständigkeit fokussieren,
- und insbesondere auch auf die Bedürfnisse „schwacher“ bis „durchschnittlicher“ Schülerinnen und Schüler ausgerichtet sein.

5 Klärung gesellschaftlicher Ansprüche

Informatikunterricht hat den Anspruch, zur Allgemeinbildung der Schülerinnen und Schüler beizutragen. Gemäß dem Forschungsformat der didaktischen Rekonstruktion für die Informatikdidaktik ist es dabei auch für die didaktische Strukturierung von Unterrichtsgegenständen zentral, den allgemeinbildenden Gehalt des Untersuchungsgegenstandes herauszuarbeiten (Diethelm *et al.*, 2011). Dies ermöglicht es, einerseits Hinweise zur Ausgestaltung von Konzepten und Materialien abzuleiten, um die allgemeinbildenden Ansprüche an das Fach Informatik zu erfüllen. Andererseits ergeben sich aus der Betrachtung der lebensweltlichen Bedeutung Hinweise auf mögliche Vorerfahrungen der Schülerinnen und Schüler bezüglich Debugging aus ihrem Alltag, die wesentlich für die im nächsten Kapitel vorgenommene Untersuchung der Perspektive der Lernenden sind. Aufbauend auf der *fachlichen Klärung* sollen daher im Folgenden gesellschaftliche Ansprüche, insbesondere im Kontext von Computational Thinking, für das Debugging untersucht werden.

5.1 Ziel

Ziel dieses Kapitels ist es, gesellschaftliche Ansprüche von Debugging zu analysieren. Diese beziehen sich dabei vor allem auf den Beitrag zur Allgemeinbildung. Traditioneller Ausgangspunkt, um den allgemeinbildenden Gehalt eines Faches oder konkreten Unterrichtsgegenstand herauszustellen, sind dabei die Ansätze nach Heymann (1996) oder Klafki (1993). Spezifisch für die Informatik und ihren Allgemeinbildungsanspruch strukturieren beispielsweise Döbeli Honegger (2016), Passey (2017) oder Vogel, Santo und Ching (2017) Argumente für informatische Bildung, etwa bezüglich der Berufs- und Arbeitswelt, informatischer Problemlösekompetenzen, der Welterklärung oder überfachlicher Kompetenzen, die im Unterschied zu Klafki und Heymann weitere Dimensionen in Betracht ziehen. Dabei konzentriert sich die Argumentation bezüglich des allgemeinbildenden Anspruchs des Schulfach Informatik im deutschsprachigen Raum insbesondere auf die Argumente der Welterklärung und spezifischer informatischer Denkweisen im Sinne des Computational Thinking (Seegerer, Michaeli und Romeike, 2019). Aufbauend auf der *fachlichen Klärung* des Debugging soll daher im Folgenden analysiert werden, welchen Beitrag Debugging zu diesen Dimensionen von Allgemeinbildung leisten kann:

(RQ3) Welchen Beitrag können Debuggingfähigkeiten zur Allgemeinbildung leisten?

5.2 Debugging und Allgemeinbildung

Zur Klärung der gesellschaftlichen Ansprüche eines Unterrichtsgegenstandes schlagen *Die-thelm et al. (2011)* die Untersuchung von Curricula und Bildungsstandards vor. Allerdings findet Debugging nur selten überhaupt explizit in deutschen wie internationalen Curricula und Standards Berücksichtigung. Stattdessen wird in vielen der entsprechenden Dokumente (vgl. z.B. *Röhner et al. (2016)*, *Gesellschaft für Informatik (2008)* und *Seehorn et al. (2011)*) und Curricula (vgl. z.B. *Schulqualität und Bildungsforschung (2009)*) vom Finden und Beheben von Fehlern oder dem Umgang mit Fehlermeldungen gesprochen. Dementsprechend kann ein solches Vorgehen in diesem Fall nicht angewendet werden.

Dabei ermöglicht die Betrachtung der Lebenswelt der Schülerinnen und Schüler erste Ansatzpunkte bezüglich des allgemeinbildenden Gehalts des Debugging: Tatsächlich kommen sie dort in Kontakt mit Phänomenen, die die Konsequenz von Programmierfehlern darstellen – beispielsweise in Form der persönlichen Nutzung von fehlerbehafteter Anwendungssoftware oder aber medialer Berichterstattung prominenter Sicherheitslücken oder Softwarefehler. Damit kann Debugging zur *Welterklärung* beitragen: Dort machen die Schülerinnen und Schüler die Erfahrung, dass Fehler unvermeidbarer Bestandteil der Programmierung sind, was zum Verständnis dieser Phänomene der Lebenswelt beiträgt. Warum diese Fehler nun aber teilweise nicht gefunden werden, betrifft in deutlich stärkerem Ausmaß den Bereich der Softwarequalität, insbesondere bezüglich des Testens und der Verifikation von Programmen, als das Debugging (vgl. dazu *Michaeli und Romeike (2017)*).

Darüber hinaus ermöglichen das „Lernen aus Fehlern“ und eine entsprechende Auffassung von Fehlern als positiv konnotierter Lernanlass, gemäß einem *erfahrungsbasierten Lernen* (*Dewey, 1986*) der Wiederholung von Fehlern vorzubeugen. *Papert (1980)* betont, dass Debugging, gerade aufgrund der besonderen Bedeutung und Unvermeidbarkeit von Fehlern in der Programmierung, zu einem veränderten Umgang mit Fehlern als einer *überfachlichen Kompetenz* auch außerhalb der Informatik führen könnte:

Many children are held back in their learning because they have a model of learning in which you have either „got it“ or „got it wrong“. But when you program a computer you almost never get it right the first time. Learning to be a master programmer is learning to become highly skilled at isolating and correcting bugs. [...] The question to ask about the program is not whether it is right or wrong, but if it is fixable. If this way of looking at intellectual products were generalized to how the larger culture thinks about knowledge and its acquisition we might all be less intimidated by our fears of being wrong.

[...] In the LOGO environment, children learn that the teacher too is a learner, and that everyone learns from mistakes.

Gerade da das Schaffen einer entsprechenden Fehlerkultur eine große Herausforderung darstellt, erscheinen Programmierfehler und deren Behebung hier vielversprechend. Allerdings ist das Debugging gemäß dem Forschungsstand für Lernende zumeist mit großer Frustration verbunden, und die Fehler werden üblicherweise eben nicht als positiv konnotierte Lernchance wahrgenommen (vgl. Kapitel 2.4). Der Ansatz des *productive failure* (Kapur, 2008) baut dabei explizit darauf, Lernende zunächst Fehler machen zu lassen, um damit ein nachhaltigeres Lernen zu ermöglichen. Debugging kann dabei als eine Form von *productive failure* angesehen werden (Lui et al., 2017; Searle, Litts und Kafai, 2018). Ein gesellschaftlicher Anspruch des Debuggings als Unterrichtsgegenstand ist damit, zu einem veränderten Umgang mit Fehlern als positiv konnotiertem Lernanlass im Sinne eines Lernens aus Fehlern beizutragen. Dabei ist die Bedeutung von Fehlern für Lernprozesse, insbesondere im Kontext des Problemlösens, bereits ausführlich aufgearbeitet worden (vgl. z.B. VanLehn et al. (2003) und Oser, Hascher und Spychiger (1999)).

Weiterhin ist Debugging eine bedeutende Herangehensweise des Computational Thinking (Yadav et al., 2011; Kazimoglu et al., 2012; Brennan und Resnick, 2012) und soll daher gewinnbringend für Probleme des Alltags eingesetzt werden können (Wing, 2011). Wie bereits angesprochen finden und beheben Schülerinnen und Schüler auch in ihrem Alltag regelmäßig Fehler: Wenn beispielsweise „das Internet nicht mehr geht“ oder das Fahrrad nicht mehr funktionieren will, *troubleshooten* sie. Der Forschungsstand (vgl. Kapitel 2.6) deutet darauf hin, dass Debuggingfähigkeiten – im Unterschied zu ernüchternden Ergebnissen für andere Bereiche von Computational Thinking (Guzdial, 2015) – tatsächlich auch auf den Alltag übertragen werden können. Allerdings fehlt es an Erkenntnissen, wie genau dieser Transfer nun wirkt und welche Debuggingfähigkeiten daher einen Beitrag zur Allgemeinbildung im Sinne von Computational Thinking leisten können. Dementsprechend soll im Folgenden der Zusammenhang zwischen *Troubleshooting* im Alltag und dem Debugging in der Programmierung aus der Perspektive des Computational Thinking ausführlich untersucht werden. Zu diesem Zweck wird systematisch die vorhandene *Troubleshooting*-Literatur untersucht, um den *Troubleshootingprozess* zu charakterisieren und Bezüge, Gemeinsamkeiten und Unterschiede zum Debugging herauszuarbeiten.

5.3 Computational Thinking und Debugging

5.3.1 Problemlösen und Troubleshooten

Problemlösen wird von Anderson und Crawford (1980) als „any goal-directed sequence of cognitive operations“ definiert. Den Problemlöseprozess unterteilt Gick (1986) in drei Schritte (vgl. Abbildung 5.1): Zunächst wird eine entsprechende Repräsentation des Problems gebildet. Wenn es nötig ist und nicht etwa auf bisherige Erfahrungen für ähnlich gelagerte Probleme

zurückgegriffen werden kann (*Schema*), wird nun eine Lösung gesucht. Diese wird abschließend umgesetzt und deren Erfolg evaluiert. Welche Fähigkeiten nun benötigt werden, um ein „guter“ Problemlöser zu sein, hängt stark vom Problem ab, welches durch verschiedene Faktoren wie Abstraktheit, Strukturiertheit oder Erfolgskriterien charakterisiert wird (*Jonassen, 2000*).

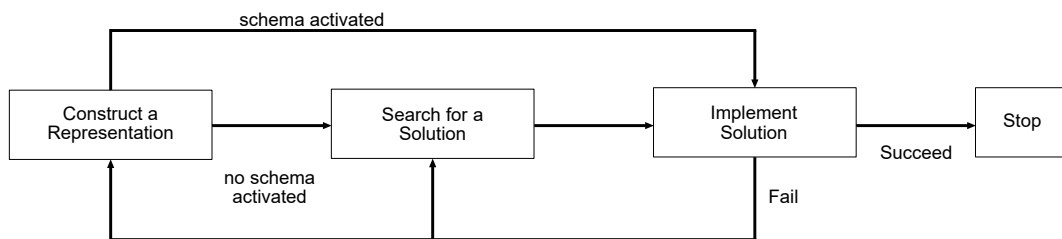


Abbildung 5.1: Problemlöseprozess nach Gick (1986)

Jonassen (2000) identifiziert elf unterschiedliche Klassen von Problemen gemäß dieser Kriterien, wie etwa algorithmische, logische, oder Entscheidungsprobleme. Eine der Problemklassen stellt dabei die Klasse der Troubleshooting-Probleme dar:

Troubleshooting is among the most common forms of everyday problem solving and in many domains is synonymous with problem solving, perhaps because the inoperative entities that involve troubleshooting are most easily perceived as problems. Mechanics who troubleshoot your inoperative car or computer programmers who debug your inoperative computer are always recognized as problem solvers. The primary purpose of troubleshooting is fault state diagnosis. That is, some part or parts of a system are not functioning properly, resulting in a set of symptoms that have to be diagnosed and matched with the user's knowledge of various fault states.

Troubleshooting bezeichnet also den Prozess der Lokalisierung der Ursache für ein Fehlverhalten eines Systems und die anschließende Reparatur oder den Austausch der fehlerhaften Komponente (vgl. auch *Morris und Rouse (1985)*). Debugging ist dabei Troubleshooting in der Domäne der Programmierung, ein Spezialfall allgemeinen Troubleshootings (*Katz und Anderson, 1987*). Als weitere Beispiele (und damit andere Domänen) für Troubleshooting-Probleme führt *Jonassen (2000)* unter anderem die Bestimmung von Chemikalien in der qualitativen Analyse, die Identifizierung von Kommunikationspannen in einem Ausschuss oder die Bestimmung, warum Zeitungsartikel schlecht geschrieben sind, an.

5.3.2 Troubleshootingprozess

Wie läuft nun aber der allgemeine Troubleshootingprozess ab? *Jonassen und Hung (2006)* beschreiben das Vorgehen als iterativen Prozess:

1. *Construct problem space*: Analog zum allgemeinen Problemlöseprozess muss zunächst ein mentales Modell des Systems und seiner Komponenten erstellt werden, inklusive Zielen und erwünschtem bzw. normalem Verhalten des Problems.
2. *Identify fault symptoms*: Durch den Vergleich von mentalem Modell, insbesondere gewünschtem bzw. normalem Verhalten, mit dem tatsächlichen Verhalten, werden Fehlverhalten und erste Indizien für dessen Ursache identifiziert.
3. *Diagnose fault(s)*: Zunächst werden Symptome und System mit den eigenen Erfahrungen abgeglichen und vergleichbare Fälle herangezogen, um den Problemraum zu verkleinern. Wenn bisher kein vergleichbares Problem gelöst wurde, müssen wiederholt Hypothesen formuliert und überprüft werden. Hierbei wird versucht, den Problemraum iterativ und rekursiv einzuschränken. Dabei werden auch verschiedene Strategien, wie beispielsweise *serial elimination* oder *space split* angewandt.
4. *Generate and verify solutions*: Schließlich werden Lösungen in der Reihenfolge ihrer Erfolgsaussicht angewandt und überprüft.
5. *Remember experience*: Zum Abschluss werden die Fehlerursache und die entsprechende Lösung implizit für zukünftige Troubleshootingprozesse abgespeichert.

Analog dazu teilen *Schaafstal, Schraagen und Van Berl (2000)* im Rahmen ihrer *task analysis* auf Basis von Beobachtungsstudien und Forschungsstand den Troubleshooting-Prozess in vier Schritte ein (vgl. Abbildung 5.2): Zunächst muss das eigentliche Problem konkretisiert, also sowohl das Fehl- als auch das korrekte Verhalten des Systems identifiziert werden. Aufbauend darauf werden Hypothesen über mögliche Ursachen aufgestellt, wobei bisherige Erfahrungen einfließen (*recognition-primed decision making* nach *Klein (1989)*), aber auch Strategien diesen Prozess unterstützen. Diese Hypothesen müssen anschließend überprüft werden, wobei der Auswahl adäquater Testmethoden und -werkzeuge große Bedeutung zukommt. Für diesen Test werden erwartetes und tatsächliches Verhalten verglichen, damit Informationen gewonnen und die Hypothese akzeptiert, verworfen oder verfeinert. Abschließend wird der Fehler behoben und zur Überprüfung erneut getestet, ob die Korrektur tatsächlich erfolgreich war und/oder weitere Fehler vorliegen. Beispielsweise bestätigen *Dounas-Frazer et al. (2016)* diesen Ablauf in einer Think-Aloud-Studie für die Domäne des Schaltungsentwurfes.

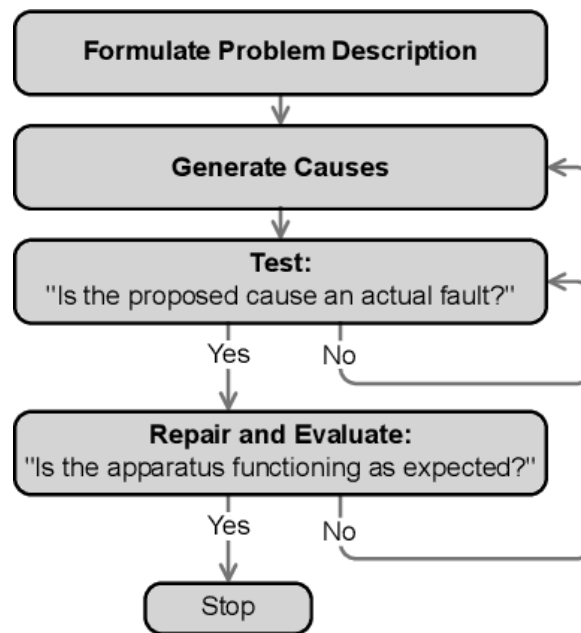


Abbildung 5.2: Visualisierung des Troubleshooting-Prozesses von *Dounas-Frazer et al. (2016)*, basierend auf der *cognitive task analysis* nach *Schaafstal, Schraaggen und Van Berl (2000)*.

5.3.3 Troubleshootingfähigkeiten

Analog zum Debugging stellt sich nun die Frage, welche Fähigkeiten als „guter“ Troubleshooter benötigt werden. *Morris und Rouse (1985)* identifizieren in ihrem Review drei wesentliche Fähigkeiten sowie Probleme „schwacher“ Troubleshooter:

- Die Fähigkeit, Tests durchzuführen, um Informationen über das System und etwaige Fehler zu erhalten. Im Kontext von elektronischen Problemen könnte dies etwa die Bedienung eines Oszilloskops sein. „Schwache“ Troubleshooter wenden hierbei oftmals nur eingeschränkt zielführende Tests an.
- Die Fähigkeit, defekte Komponenten zu ersetzen. „Schwachen“ Troubleshootern fehlen hierbei oftmals elementare Kenntnisse oder sie haben Probleme in der Ausführung.
- Die Fähigkeit, ein systematisches Vorgehen¹⁸ bei der Suche nach der Ursache anzuwenden, wie etwa wiederholt Hypothesen zu generieren, vom Fehler aus das System rückwärts nachzuverfolgen, o.ä. „Schwache“ Troubleshooter nutzen hierbei die zu Verfügung stehenden Informationen oftmals nur ineffizient, haben Probleme mit dem

¹⁸ „employ some kind of strategy“, *Morris und Rouse (1985)*.

Generieren von Hypothesen und Überprüfen dieser und sind in ihrem Vorgehen kaum flexibel. Dieser Schritt stellt die größte Herausforderung im Troubleshootingprozess dar, ist aber auch am ehesten unabhängig von der konkreten Domäne.

Jonassen und Hung (2006) kategorisieren für Troubleshooting benötigtes Wissen in fünf Kategorien:

- *Domain knowledge*: Allgemeines Wissen über die Domäne, wie etwa Theorien und Prinzipien, auf deren Grundlage das System oder Gerät entworfen wurde.
- *System/Device knowledge*: Konzeptionelles Wissen über die Funktionsweise eines Systems, wie etwa Aufbau/Struktur sowie Funktion und Zusammenspiel der einzelnen Komponenten. Dieses Wissen stellt dabei den Hauptunterschied zwischen Novizen und Experten beim Troubleshooten dar und unterteilt sich in:
 - *topological knowledge*: Wo befindet sich welche Komponente?
 - *functional knowledge*: Welche Funktion hat welche Komponente?
- *Performance/Procedural knowledge*: Wissen über das Durchführen von Wartungsaktivitäten oder das Testen einzelner Komponenten für das spezifische System, wie etwa das Messen von Spannung.
- *Strategic knowledge*: Wissen, das hilft Hypothesen zu bestätigen oder neue Alternativen zu suchen, wenn die bestehenden Hypothesen oder Lösungen als falsch oder nicht durchführbar bewertet werden müssen. Dabei lassen sich zwei Arten von Strategien unterscheiden:
 - *Global strategies*: Strategien, die unabhängig von der konkreten Domäne oder dem konkreten System sind und helfen den Problemraum einzuschränken. Beispiele sind etwa *Trial-and-Error*, *topographic search*¹⁹, binäre Suche oder *functional/discrepancy detection*²⁰.
 - *Local strategies*: Strategien, die spezifisch für eine konkrete Domäne oder ein konkretes System sind.
- *Experiential knowledge*: Bisherige Troubleshootingerfahrung.

Darüber hinaus existiert eine Vielzahl an Untersuchungen, die sich mit dem Troubleshooting in konkreten Domänen, wie etwa elektronischen Schaltungen (*Rasmussen und Jensen*,

¹⁹*Forward topographic search*: Ausgehend von dem Punkt, an dem das Gerät/System normal funktioniert, das System nachverfolgen. *Backward topographic search*: analog rückwärts vom Fehler aus vorgehen.

²⁰Fehler isolieren, indem nach Diskrepanzen gesucht wird zwischen dem, was bei einem normalen Systembetrieb erwartet wird, und dem tatsächlichen Verhalten.

1974; Reed und Johnson, 1993), Produktionssystemen (Bereiter und Miller, 1989; Konradt, 1995) oder Radarsystemen (Schaafstal, Schraagen und Van Berl, 2000) auseinandersetzen. Dort werden jeweils angewandte lokale wie globale Troubleshooting-Strategien identifiziert und zeigen, dass auch globale Strategien in unterschiedlichen Kontexten unterschiedliche Bedeutung haben.

5.3.4 Troubleshooting und Debugging

Im Folgenden sollen die Erkenntnisse für den allgemeinen Troubleshootingprozess nun in Bezug zum Debugging gestellt und interpretiert werden. Zunächst ist festzustellen, dass Debugging einen Spezialfall von Troubleshooting darstellt: Das Finden und Beheben von Fehlern in der Domäne der Programmierung. Dies zeigt sich auch am Ablauf des Troubleshootingprozesses: Dabei werden wiederholt Hypothesen formuliert, in Experimenten überprüft und gegebenenfalls verfeinert, bis die Ursache des Fehlers gefunden ist – analog zur *isolation*-Strategie, der für Experten beim Debugging eigener Programme induktiv festgestellt wurde. Auch beim Troubleshooting muss dabei zunächst das entsprechende System (Debuggen: der Programmcode) verstanden werden, ausgehend vom Fehlverhalten des Systems (Debuggen: des Programms) Informationen und Indizien gesammelt werden, die die Generierung von Hypothesen anleiten, und abschließend der Fehler (Debuggen: Bug) behoben werden. Auch unterstützen gewisse Strategien die Generierung und Überprüfung von Hypothesen.

Diese Ähnlichkeiten zeigen sich auch im Vergleich der für das Troubleshooting benötigten Fähigkeiten mit dem Modell der Debuggingfähigkeiten für Novizen (vgl. Kapitel 3): Betrachtet man die Klassifizierung von Morris und Rouse (1985), so ähnelt die allgemeine „Fähigkeit, Tests durchzuführen“, der domänenspezifischen *Anwendung von Werkzeugen* für das Debugging. Auch hier müssen entsprechende Fähigkeiten ausgeprägt sein, um beispielsweise den Debugger oder das Feedback der IDE zielführend für die Überprüfung und Verfeinerung von Hypothesen einzusetzen. Die „Fähigkeit, defekte Komponenten zu ersetzen“, bedeutet, übertragen auf das Debugging, den gefundenen Fehler nun tatsächlich zu korrigieren. Hierfür werden entsprechende Programmierfähigkeiten benötigt, wobei dieser Schritt im Debugging generell die geringste Hürde darstellt, während die größten Schwierigkeiten bei der Lokalisierung des Fehlers liegen (Gould, 1975; McCauley et al., 2008). Die dritte Fähigkeit nach Morris und Rouse (1985) bezieht sich auf ein systematisches Vorgehen. Entsprechend Kapitel 2.2 findet beim Debuggen eigener Programme vorwiegend ein hypothesengeleiteter Prozess Anwendung. Auch die typischen Schwierigkeiten von „schwachen“ Troubleshootern, die Morris und Rouse (1985) beschreibt, weisen große Ähnlichkeiten zum Debuggen auf (Aufstellen von Hypothesen, ineffiziente Nutzung von Informationen, vgl. Kapitel 2.2.2).

Betrachtet man das Modell des Troubleshootingwissens nach *Jonassen und Hung (2006)*, ist festzuhalten, dass dieses, wie in Kapitel 3.4 bereits diskutiert, von *Li et al. (2019)* als „Framework for teaching debugging“ in die Domäne der Programmierung übertragen wurde. *Domain* und *system/device knowledge* beziehen sich dabei auf grundlegende Programmierfähigkeiten und ein Verständnis des zu debuggenden Programms, für das genauso die Funktion (*functional knowledge*) und der Ort (*topographic knowledge*) der einzelnen Bestandteile (bzw. Module) bekannt sein müssen. Spezifisch für den eigentlichen Troubleshootingprozess wird nun einerseits *performance/procedural knowledge*, also – übertragen auf das Debugging – Wissen über die *Anwendung von Werkzeugen* benötigt. Darüber hinaus wird *experiential knowledge*, also für das Debuggen die *Anwendung von Mustern und Heuristiken für typische Fehler* verlangt. Außerdem ist Wissen über Strategien erforderlich. Lokale Strategien für das Debugging sind dabei (vgl. Kapitel 2.2) beispielsweise *Slicing*, *print-Debugging* oder aber Testfälle. Daneben existieren globale Strategien – die zwar jeweils unabhängig von der konkreten Domäne sind, aber nichtsdestotrotz in manchen Domänen größere Bedeutungen als in anderen haben. Für viele der lokalen Debuggingstrategien gibt es entsprechende globale Troubleshootingstrategien. So kann beispielsweise für die lokale Strategie, die Ausführung eines bestimmten Falles zu erzwingen und den tatsächlichen Programm-Output mit dem erwarteten Output zu vergleichen, *functional/discrepancy detection* als die entsprechende allgemeine globale Troubleshootingstrategie angesehen werden (*Li et al., 2019*). Analog dazu kann die Strategie des *print-Debuggings* zum schrittweisen Nachvollziehen eines Programmablaufs als lokales Gegenstück einer globalen *forward topographic search* angesehen werden.

Sowohl das Beispiel des Oszilloskops von *Morris und Rouse (1985)* als auch das des „Messens von Spannung“ bei *Jonassen und Hung (2006)* deuten aber an, dass werkzeugbezogene Fähigkeiten sehr domänenspezifisch und nur eingeschränkt generalisierbar sind. Das gilt analog für die Rolle von Erfahrung (*experiential knowledge*), die sich auf eine konkrete Domäne bezieht. Im Gegensatz dazu findet sich das allgemeine systematische Troubleshooting-Vorgehen analog im Debugging – und es existieren Indizien, dass nach dem Unterrichten eines systematischen Vorgehens zum Debuggen dieses aus der Domäne der Programmierung hinaus übertragen wird (vgl. Kapitel 2.6). Dasselbe gilt für Strategien: Lokale Troubleshootingstrategien für die Domäne der Programmierung, wie etwa das gezielte Nachverfolgen des Wertes einer Variable mit Hilfe von *print-Anweisungen*, haben entsprechende globale, und damit kontextunabhängige Strategien, wie etwa die *topographische Suche*. Damit scheint vor allem die Förderung dieser beiden Debuggingfähigkeiten vielversprechend, um im Sinne des Computational Thinking einen Beitrag zur Allgemeinbildung zu leisten.

5.4 Fazit

Zusammenfassend zeigt sich, dass Debugging insbesondere auf drei Arten einen Beitrag zu Allgemeinbildung leisten kann: Zur (Welt-)Erklärung des Phänomens „fehlerhafte Software“ aus der Lebenswelt, bezüglich des Lernens aus Fehlern als überfachliche Kompetenz sowie als Herangehensweise des Computational Thinking für das Troubleshooting. Im Rahmen dieses Kapitels wurde dabei die letzte Dimension eingehend untersucht.

Dabei zeigte sich, dass das Debugging ein Spezialfall von allgemeinem Troubleshooting und damit ein Aspekt von Problemlösen ist. Dabei sind sowohl das systematische Debuggingvorgehen als auch Debuggingstrategien wie Tracing oder Testen (*lokale Strategien*) Manifestationen des allgemeinen Troubleshooting-Vorgehens oder *globaler Troubleshootingstrategien* wie *topographischer Suche* oder *functional/discrepancy detection*. Bisherige Forschungsergebnisse deuten zumindest darauf hin, dass ein Transfer aus der Domäne der Programmierung heraus tatsächlich stattfinden kann. Somit kann Debugging mit der Vermittlung eines systematischen Vorgehens und entsprechender Strategien einen Beitrag zum Troubleshooting und damit zur Allgemeinbildung im Sinne des Computational Thinking leisten.

Diese Ergebnisse erlauben nun einerseits Hinweise zur Ausgestaltung von Konzepten und Materialien, wie etwa die Betonung des allgemeinen Debuggingvorgehens oder einer positiven Fehlerkultur, um die allgemeinbildenden Ansprüche an das Fach Informatik zu erfüllen. Andererseits wurden auf diese Weise Vorgehensweisen des Troubleshooting identifiziert, die die Schülerinnen und Schüler in ihrem Alltag möglicherweise bereits anwenden und die damit aufgrund der Bezüge zur Fehlersuche und -behebung in der Programmierung konkrete Lernvoraussetzungen für das Debugging im Unterricht darstellen, die im folgenden Kapitel untersucht werden sollen.

6 Perspektive der Lernenden

Gemäß der Lerntheorie des Konstruktivismus ist Lernen ein ständiger und aktiver Prozess der Anpassung bereits existierender mentaler Modelle, indem neue Erfahrungen gemacht und reflektiert werden (*Phillips, 1995*). Daher müssen die bereits vorhandenen Erfahrungen der Lernenden für die Entwicklung geeigneter Konzepte und Materialien für den Unterricht berücksichtigt werden (*Bransford, Brown und Cocking, 2000; Bonar und Soloway, 1985*). Solche – oftmals lebensweltlichen und vorunterrichtlichen – Erfahrungen werden als *individuelle Lernvoraussetzungen* bezeichnet und können im Allgemeinen sowohl kognitive, affektive oder auch motivationale Faktoren umfassen (*Prenzel und Doll, 2002*). Für die Ausgestaltung spezifischer Lehr-Lern-Settings sind dabei insbesondere konkrete domänenspezifische kognitive Lernvoraussetzungen (wie Präkonzepte, Vorwissen oder Vorstellungen) relevant. In der Informatikdidaktik existiert eine Vielzahl von Untersuchungen solcher domänenspezifischer kognitiver Lernvoraussetzungen zu verschiedenen Themen. Für den Bereich der Programmierung ermittelten *Onorato und Schwaneveldt (1987)* sowie *Miller (1981)* Lernvoraussetzungen, indem sie Lernende beim „Programmieren“ in natürlicher Sprache beobachteten. *Gibson und O’Kelly (2005)* analysierten den Problemlösungsprozess der Schülerinnen und Schüler für Suchprobleme, *Kolikant (2001)* untersuchte deren Lernvoraussetzungen in Bezug auf Parallelität und Synchronisation, und in der „Commonsense Computing“-Reihe wurden Lernvoraussetzungen für verschiedene Themen wie Sortierung oder Logik erhoben (*Simon et al., 2006; Lewandowski et al., 2007; VanDeGrift et al., 2010*). Im Folgenden sollen nun Debugging-Lernvoraussetzungen von Schülerinnen und Schülern identifiziert werden, indem ihr Vorgehen beim Troubleshooting mit Hilfe eines Escape-Rooms untersucht wird.

6.1 Ziele der Untersuchung

Lernvoraussetzungen stellen die Basis für die Entwicklung von Konzepten und Materialien für den Unterricht dar. Aber welche lebensweltlichen und vorunterrichtlichen Erfahrungen haben Schülerinnen und Schüler für das Debugging? Wie bereits dargelegt, finden und beheben sie Fehler in ihrem Alltag, indem sie *troubleshooten*. Debugging ist ein Spezialfall des Troubleshootings: Troubleshooting in der Domäne der Programmierung. Demnach werden ähnliche Fähigkeiten und Strategien beim Troubleshooten und Debuggen verwendet (*Li et al., 2019*). Daher können entsprechende alltägliche Troubleshooting-Erfahrungen als Vorerfahrungen für das Debuggen angesehen werden (*Simon et al., 2008*).

Im Folgenden werden daher die Vorgehensweisen von Schülerinnen und Schülern beim *Troubleshooten* untersucht, um domänenspezifische kognitive Lernvoraussetzungen für das

Debuggen zu identifizieren, und damit die vierte Forschungsfrage dieser Arbeit zu beantworten:

(RQ4) Welche debuggingspezifischen Lernvoraussetzungen bringen Schülerinnen und Schüler aus ihrem Alltag mit?

Bisherige Ansätze zur Analyse von Lernvoraussetzungen für das Debugging von Studierenden (vgl. Kapitel 2.4.2 bzw. *Simon et al. (2008)*) verwenden dafür Kontexte aus der realen Welt, wie z.B. „Wie würdest du vorgehen, wenn eine Glühbirne nicht mehr funktioniert“. Im Gegensatz dazu geht der hier verfolgte Ansatz einen Schritt weiter: Anstatt die Probandinnen und Probanden nur zu befragen, wie sie reagieren *würden*, wenn sie sich in einer bestimmten Situation befänden, werden sie *tatsächlich* in diese Situation gebracht, indem ein Escape-Room als Methode eingesetzt wird. Darüber hinaus sind die Aufgaben auf Basis der bereits dargelegten Literatur auf die Untersuchung debuggingspezifischer Vorgehensweisen zugeschnitten.

6.2 Spezifischer Forschungsstand: Escape-Rooms als Unterrichtsbzw. wissenschaftliche Untersuchungsmethode

Seit einigen Jahren haben sogenannte (Live-)Escape-Rooms (oder auch Escape-Games, Exit-the-Room-Spiele, Breakout-Games, usw.), bei denen Teilnehmende in einen Raum „eingeschlossen“ werden und aus diesem entkommen müssen, weite Verbreitung durch kommerzielle Anbieter gefunden, die die ursprüngliche Idee aus einem Subgenre digitaler Point-and-Click-Adventures in die Realität übertragen haben (*Nicholson, 2015*). Ein typischer Escape-Room hat dabei eine übergreifende Geschichte und die Teilnehmerinnen und Teilnehmer müssen in einer gegebenen Zeit eine Vielzahl an Rätseln und Aufgaben lösen und damit oftmals entsprechende Schlösser o.Ä. entsperren, die wiederum zu neuen Rätseln und Aufgaben führen. Um zu gewinnen, müssen alle Rätsel in der gegebenen Zeit gelöst werden. Entweder müssen die Spielerinnen und Spieler dann aus dem Raum entkommen und/oder einen bestimmten Gegenstand finden.

In den letzten Jahren ist ein zunehmendes Interesse zum Einsatz solcher Escape-Rooms in verschiedenen Bereichen sowohl informeller als auch formaler Bildung zu beobachten. Die Nutzung eines Escape-Rooms als *Unterrichtsmethode* bietet verschiedene spannende pädagogische Möglichkeiten, wie z.B. einen ansprechenden und motivierenden Kontext durch die Anwendung von Elementen der Gamification (*Borrego et al., 2017; Nicholson, 2018*). Darüber hinaus können Fähigkeiten wie Kollaboration, kritisches Denken und Problemlösen auf natürliche Weise gefördert werden (*Pan, Lo und Neustaedter, 2017; Friedrich et al., 2019; Hacke, 2019*). Die Zielsetzung solcher Räume reicht dabei von der Vermittlung allgemeiner

Problemlösungs- oder Teamkoordinationsfähigkeiten (z.B. *Williams (2018) und Friedrich et al. (2019)*) bis hin zur Vermittlung von Inhalten, die spezifisch für eine bestimmte Domäne oder ein bestimmtes Fachgebiet sind, wie z.B. Informatik (*Hacke, 2019*), Pharmazie (*Eukel, Frenzel und Cernusca, 2017*) oder Physik (*Vörös und Sárközi, 2017*).

Für viele dieser Escape-Rooms, die als Unterrichtsmethode eingesetzt werden, sind beispielsweise die Gestaltungskriterien, die Motivation der Lernenden oder der tatsächliche Lernerfolg evaluiert worden. Das Potential von Escape-Rooms als Methode zur Untersuchung von Problemlöse- und Lernprozessen ist im Gegensatz dazu aber bisher weitgehend ungenutzt.

Järveläinen und Paavilainen-Mäntymäki (2019) führten eine vergleichende Fallstudie durch, in der sie die Lernprozesse von drei studentischen Teams in einem Escape-Room zur Vermittlung einer informationswissenschaftlichen Forschungsmethode analysierten. Sie stellten fest, dass die verschiedenen Teams auf ihrem Weg durch den Escape-Room unterschiedliche Lernprozesse einsetzten.

In der Informatik analysierte *Hacke (2019)* Verhaltensmuster in Problemlösungsprozessen in einem Escape-Room für den Informatikunterricht. Zu diesem Zweck untersuchte der Autor die Videoaufnahmen von 38 Schülerinnen- und Schüler-Gruppen. Für die Analyse verwendete er ein deduktives Kategoriensystem zur Klassifizierung von Verhaltensmustern im Problemlösungsprozess. Anschließend wurde der Einfluss dieser Muster auf den Gesamterfolg im Spiel evaluiert. Der Autor identifizierte dabei vielversprechende Verhaltensmuster und Gruppenverhalten. Dazu gehörten beispielsweise die Benutzung der Tafel für Notizen, ein Koordinator (statt eines Leiters) zur Abstimmung des Teams in der Gruppenzusammensetzung oder strukturierte Aufgabenlöser im Team. Die meisten dieser Muster befinden sich aber auf einer eher abstrakten Ebene der „Teamzusammensetzung“ und Merkmalen des Teams und befassen sich nicht näher mit dem eigentlichen Problemlöseprozess.

Zusammenfassend ist damit festzustellen, dass die Anwendung von Escape-Rooms als Forschungsmethode eine – bisher – kaum genutzte, aber vielversprechende Möglichkeit darstellt, Problemlösungsprozesse zu untersuchen. Sie ermöglicht es, die Prozesse in einer „natürlichen“ Umgebung zu beobachten. Auf diese Weise kann die Analyse bereits vorhandener Debugging-Lernvoraussetzungen einen Schritt weiter geführt werden als bisher: Anstatt Teilnehmende nur beschreiben zu lassen, wie sie in einer gegebenen Situation reagieren und vorgehen würden, können die tatsächlichen Problemlösungsprozesse und das Verhalten in einer realen Problemlösungssituation erfasst werden. Damit erscheint ein solcher Escape-Room-Ansatz vielversprechend und bietet insbesondere folgende Potentiale für die Erhebung von Debugging-Lernvoraussetzungen:

- Der tatsächliche Troubleshootingprozess und dessen Charakteristika und Merkmale können in einer „natürlichen Umgebung“ beobachtet werden.

- Im Unterschied zur Erhebung durch offene Fragebögen oder Interviews werden damit auch die Reaktionen der Schülerinnen und Schülern erfasst, wenn ihr ursprünglicher Plan nicht aufgeht.
- Darüber hinaus sind die Teilnehmerinnen und Teilnehmer nicht in der Lage, ihre abschließende Antwort umfassend zu planen oder zu revidieren, wie dies bei einer schriftlichen Erhebung der Fall wäre.

6.3 Erhebungsinstrument Escape-Room

Bei der Gestaltung des Erhebungsinstruments für diese Untersuchung bieten dabei die beschriebenen existierenden Ansätze sowohl als Unterrichts- als auch wissenschaftliche Erhebungsmethode ein gewisses Maß an Orientierung und Anhaltspunkten, wie beispielsweise das allgemeine Vorgehen (Clarke *et al.*, 2017) in der Entwicklung eines solchen Konzeptes. Dennoch lassen sich etwaige Erfahrungen ob des Forschungsinteresses und der spezifischen wissenschaftlichen Zielsetzung nur eingeschränkt übertragen. Daher wurde zur Entwicklung des Escape-Rooms und seiner Aufgaben ein Design-Based-Research-Ansatz gewählt. Im Folgenden soll beschrieben werden, wie das entsprechende Erhebungsinstrument zur Beobachtung der Schülerinnen und Schüler beim Troubleshooting entwickelt wurde und wie der daraus resultierende Escape-Room mit den entsprechenden Aufgaben aufgebaut ist.

6.3.1 Entwicklung

Nachdem die Zielsetzung des Raums klar definiert worden war, wurde in einem ersten Schritt (vgl. auch das Framework zur Entwicklung von Escape-Rooms für den Unterricht von Clarke *et al.* (2017)) die übergreifende Geschichte festgelegt, in die alle Aufgaben thematisch eingebettet werden sollten. Als Setting wurde aufgrund der organisatorischen Gegebenheiten (Nutzung von Räumen der Universität) das „Büro eines Professors“ gewählt und aufgrund der Vielfalt der damit verbundenen Möglichkeiten mit einer Geschichte über das „alte Ägypten“ verbunden: Ziel der Teilnehmenden ist es, den Fluch des Pharaos zu beenden, indem sie im Büro eines an Ausgrabungen im Tal der Könige beteiligten Professors für Archäoinformatik einen aus der Grabungsstätte geraubten Gegenstand finden müssen. Im nächsten Schritt wurden die einzelnen Aufgaben entwickelt. Wegen des Forschungsvorhabens ergaben sich dabei verschiedenartige Kriterien für die Gestaltung der einzelnen Aufgaben.

Inhaltliche Kriterien

Zunächst sind dabei inhaltliche Kriterien von Bedeutung. Ziel der Untersuchung ist es, den Troubleshootingprozess von Schülerinnen und Schülern zu beobachten, um daraus Rückschlüsse auf ihre Debugging-Lernvoraussetzungen zu ziehen. Wie bereits in Kapitel 5 dargelegt, erfüllen Troubleshootingaufgaben gemäß *Jonassen und Hung (2006)* unter anderem folgende Kriterien:

Sie

- sind nicht vollständig definiert,
- erfordern, dass ein mentales Modell des zu troubleshootenden Systems konstruiert wird,
- haben bekannte Lösungen mit klaren Erfolgskriterien,
- erfordern, dass ein Urteil über die Art des Problems gebildet wird,
- beinhalten üblicherweise nur einen Fehler, auch wenn dieser zu mehreren beobachtbaren Fehlverhalten führen kann.

Darüber hinaus muss, wie bereits herausgearbeitet, ein Bezug zum Debugging hergestellt werden, sodass insbesondere solche Troubleshooting-Fähigkeiten, die konkreten Debuggingfähigkeiten entsprechen, in der Lösung der Aufgabe angewendet werden können. Also vor allem (vgl. Kapitel 5):

- Ein systematisches Vorgehen gemäß dem allgemeinen Troubleshooting-Vorgehen anzuwenden.
- Globale Strategien zu lokalen Debuggingstrategien, wie etwa *functional/discrepancy detection* oder *topographische Suche* einzusetzen.

Methodologische Kriterien

Des Weiteren muss die Anwendung entsprechender Verhaltensweisen und deren Charakteristika durch die Teilnehmenden auch beobachtbar sein, um empirisch erhoben werden zu können. Daher muss das Verhalten entweder

- direkt und extern beobachtbar sein, sodass direkte Rückschlüsse auf Charakteristika und Merkmale des Troubleshooting-Prozess gezogen werden können, oder

- durch die Teamarbeit und offene Kommunikation zwischen den Teammitgliedern beobachtbar gemacht werden (vgl. z.B. *Fields, Searle und Kafai (2016)*), indem diese Kommunikation aktiv gefördert oder notwendig wird.

Kriterien des Escape-Room-Settings und der praktischen Durchführung

Weiterhin ergeben sich durch das gewählte Setting des Escape-Rooms (statt beispielsweise einer Laborsituation) zusätzliche Kriterien:

- **Verständlichkeit:** Aufgrund des freien Problemlösecharakters der Aufgaben in einem Escape-Room ist die Verständlichkeit der Aufgaben zentral. Grundlegend für die Beobachtung eines zielgerichteten Troubleshootingprozesses ist, dass Schülerinnen und Schüler die Zielsetzung der Aufgabe in angemessener Zeit erfassen können.
- **Praktikabilität:** Da für die praktische Durchführung Räume der Universität genutzt werden, die jeweils nur für einen gewissen Zeitraum zur Verfügung stehen, müssen die Aufgaben als solche in angemessener Zeit variabel auf- und abgebaut werden können.
- **Verkettbarkeit:** Die Lösung der einzelnen Aufgaben muss als „Eingabe“ für Schlösser geeignet sein, die im Sinne einer Staffelung dann weitere Aufgaben freigeben. Daher müssen die Lösungen typischerweise aus Zahlenkombinationen, kurzen alphanumerischen Zeichenfolgen und Ähnlichem bestehen.
- **Thematische Passung:** Die Aufgaben sollten thematisch in die übergreifende Geschichte eingebettet sein.
- **Zeitliche Passung:** Die Komplexität der Aufgaben muss an das gewählte zeitliche Limit angepasst sein.

Gemäß dieser Kriterien wurden initial 10 Aufgaben entwickelt und in mehreren Iterationen sowohl mit anderen Forscherinnen und Forschern als auch Schülerinnen und Schülern erprobt und weiterentwickelt. In der Beobachtung und Auswertung zeigten sich dabei verschiedene Herausforderungen und Verbesserungspotentiale in der (Weiter-)Entwicklung der Aufgaben, die wie folgt adressiert wurden:

Was ist die Lösung? Aufgrund des Troubleshooting-Charakters der Aufgaben war es oftmals das Ziel, Fehler in einem System zu finden. Eine typische Hürde in der Bearbeitung solcher Aufgaben war es, zu erkennen, ob nun die *Fehler* (beispielsweise fehlerhafte Zahlen) oder deren *Korrektur* das Ergebnis darstellen, das beispielsweise zum Öffnen des nächsten Schlosses benötigt wird. Um dieses – auf das Spiel als solches – bezogene Problem zu lösen,

wurden einheitlich die *Fehler* als Lösung für alle Aufgaben festgelegt, da der Fokus in der Mehrheit der Aufgaben auf der Lokalisierung und nicht der Behebung lag. Diese Konvention wurde den Teilnehmenden zu Beginn des Spiels kommuniziert und das Problem damit reduziert.

Was muss ich hier tun? Für manche der Aufgaben hatten die Teilnehmenden ohne externe Hinweise des Spielleiters Probleme, die konkrete Zielsetzung der Aufgabe zu erfassen. Entsprechend wurden zusätzliche Hinweise eingebaut oder bestehende konkretisiert. Zudem befanden sich in den ersten Iterationen viele Materialien für spätere Aufgaben bereits zugänglich im Raum, die für Verwirrung sorgten – auch weil es an entsprechenden Möglichkeiten mangelte, diese zunächst wegzusperren und im weiteren Verlauf „freizuspielen“. Um zu verhindern, dass Schülerinnen und Schüler aufwendig und erfolglos versuchen, einen Zusammenhang zwischen Materialien unterschiedlicher Aufgaben herzustellen, wurden Möglichkeiten geschaffen (beispielsweise indem weitere verschließbare Boxen geeigneter Größe angeschafft wurden), diese, soweit möglich, gestaffelt „freizuspielen“.

Komplexität und Zeit. In den ersten Iterationen waren – genretypisch – einige der benötigten Materialien im Raum versteckt. Dabei zeigte sich, dass Schülerinnen und Schüler große Probleme hatten, diese Gegenstände effizient zu suchen und zu finden. Da die Suche keinen Beitrag zum Untersuchungsziel leistet, wurde daraufhin auf versteckte Gegenstände (als weitverbreitete Aufgabe in Escape-Rooms (Nicholson, 2015)) komplett verzichtet und alle nötigen Gegenstände wurden zentral im Raum direkt auffindbar positioniert. Auch war festzustellen, dass die Schülerinnen- und Schülergruppen nur selten parallel und arbeitsteilig arbeiteten, wenn gerade mehrere Aufgaben verfügbar waren. Daher wurde insgesamt die Anzahl und die Komplexität der Aufgaben reduziert.

Bruteforce. Ein weiteres auffälliges Muster war, dass viele Gruppen von Schülerinnen und Schüler versuchten, einzelne Schlösser über das Ausprobieren aller möglichen Kombinationen zu öffnen – gerade wenn bereits eine oder zwei Teile der Zahlenkombination bekannt waren. Diese Verhaltensweise, die nicht in den eigentlichen Aufgaben, sondern in der *Umgehung* dieser angewandt wurde, konnte allerdings nur eingeschränkt unterbunden werden, beispielsweise durch entsprechende Schlösser. Auch wurden weitere „Abkürzungen“, wie das Herausziehen von Materialien aus verschlossenen Truhen oder einem Tagebuch zu unterbinden versucht, indem diese Materialien fest fixiert wurden.

Neben diesen vorwiegend für Kriterien des Escape-Room-Settings relevanten Faktoren wurden aber auch methodologische und inhaltliche Probleme identifiziert:

Beobachtbarkeit des Vorgehens. Für einige der Aufgaben war das Vorgehen der Teilnehmenden zunächst nur eingeschränkt beobachtbar, insbesondere weil Aufgaben „im Kopf“ gelöst wurden – oder es zumindest versucht wurde. Das Troubleshooting-Vorgehen und dessen Merkmale sowie angewandte Strategien konnten so nicht erhoben werden. Indem zusätzlich externe Hilfsmittel wie z.B. bunte Steine zur Markierung von Positionen zur Verfügung gestellt und von den Teilnehmenden genutzt wurden, konnten entsprechende Prozesse sichtbar gemacht werden. Außerdem wurden Maßnahmen ergriffen, die Kommunikation der Teilnehmenden aktiv zu fördern, um so interne Prozesse beobachtbar zu machen: Beispielsweise wurde bei einer Aufgabe eine zeitliche Beschränkung bis zum nächsten Versuch eingeführt, in der die Teilnehmenden das weitere Vorgehen planen konnten.

Bezug zu Debugging. Auch zeigte sich bei manchen Aufgaben, dass in der Praxis kein klarer Bezug zwischen den (Troubleshooting)-Handlungen und Debugging hergestellt werden konnte.

Aufgrund der Kombination aus fehlendem bzw. unklarem Bezug zu Debugging und Problemen in der Beobachtbarkeit wurden einige Aufgaben gestrichen, weil sie keinen Beitrag zum Ziel der Untersuchung leisten konnten.

6.3.2 Resultierende Aufgaben

Im Folgenden werden die für das Forschungsinteresse aussagekräftigsten finalen Aufgaben beschrieben, die in dieser Untersuchung ausgewertet wurden. Jede Aufgabe wird zunächst dargelegt und anschließend das jeweils intendierte Untersuchungsziel angeführt. Eine Übersicht der Aufgaben und der jeweiligen Untersuchungsziele findet sich in Tabelle 6.1. Das erwartete Vorgehen der Teilnehmenden für die jeweiligen Aufgaben ist aus Gründen der Nachvollziehbarkeit im Detail direkt bei der Darstellung der Ergebnisse beschrieben.

Monitor: Die Teilnehmenden finden einen Monitor, der nicht zu funktionieren scheint. Um ihn zu „reparieren“, müssen sie lediglich das neben dem Monitor liegende Stromkabel mit dem Monitor verbinden. Mit dem Monitor ist weiterhin ein Streamingempfänger verbunden (und mit einem „Nicht anfassen“-Aufkleber versehen), sodass, nachdem der Strom verbunden ist, auf dem Monitor ein Timer mit der Restspielzeit und die nächste Aufgabe angezeigt werden. Ziel dieser Aufgabe ist es, das Troubleshooting-Vorgehen und die Anwendung einer *functional/discrepancy detection*-Strategie zu beobachten.

Kabelsalat: Die Teilnehmenden finden acht in Reihe geschaltete USB-Verlängerungskabel, die eine LED-Taschenlampe mit einem Netzteil verbinden (vgl. Abbildung 6.1). Außerdem finden sie eine fest im Raum montierte Box, in die sie ohne die Taschenlampe nicht hineinleuchten können. Zwei der Kabel sind defekt. Die Aufgabe besteht darin, die defekten Kabel zu identifizieren und mit den verbleibenden Kabeln eine ausreichend lange Kette zu bilden, um in die Box leuchten und die Zahlen darin lesen zu können. Ziel dieser Aufgabe ist es, das Troubleshooting-Vorgehen und die Anwendung einer *functional/discrepancy detection*-Strategie zu beobachten.

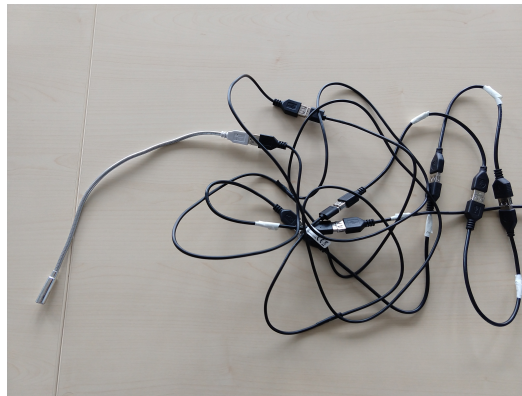


Abbildung 6.1: Kabelsalat

Telefon abhören: Die Teilnehmenden entdecken eine Kiste mit fünf heraushängenden Kabeln (vgl. Abbildung 6.2). Beim Einstecken eines Kabels in einen der fünf Stecker erhalten sie eine akustische Rückmeldung darüber, wie viele Kabel richtig stecken. Sie müssen die richtige Reihenfolge der Kabel finden, ähnlich wie im Spiel *Mastermind*. Ziel dieser Aufgabe ist es, das Troubleshooting-Vorgehen zu beobachten.

Tal der Könige: Die Teilnehmenden finden eine Karte des Tals der Könige mit einer Route darauf. Außerdem finden sie eine Wegbeschreibung, wobei einige der Pfeile, die die Route beschreiben, falsch sind (vgl. Abbildung 6.3). Das Papier mit der Route enthält den Hinweis, dass 6 Fehler zu identifizieren sind. Diese Aktivität wurde entwickelt, um die Anwendung einer topographischen Suchstrategie zu analysieren.

Mr. X: Eine Webanwendung (auf einem zuvor gefundenen Tablet) zeigt eine Karte des U-Bahn-Systems von Kairo (vgl. Abbildung 6.4). Die Teilnehmenden haben die Aufgabe, herauszufinden, wohin Mr. X, dessen Startposition gegeben ist, fährt. Zu diesem Zweck

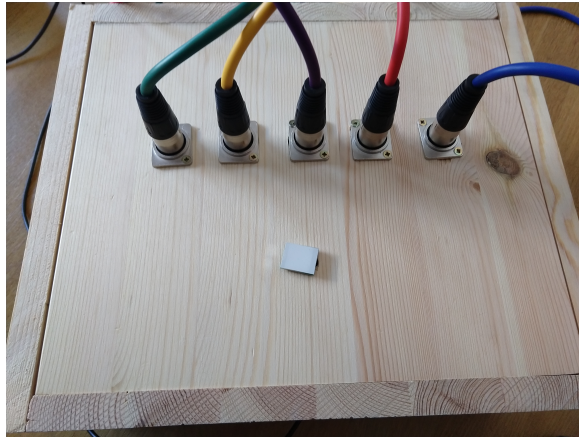


Abbildung 6.2: Telefon abhören

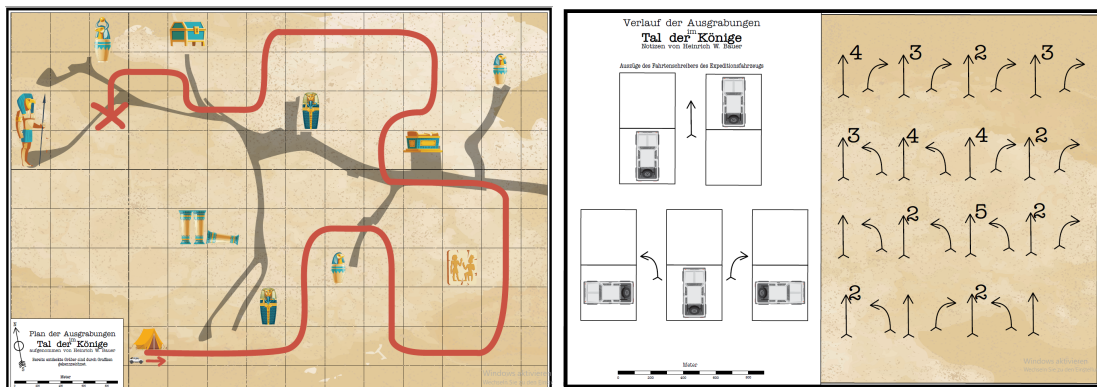


Abbildung 6.3: Tal der Könige

können sie zwei „Wächter“ in den U-Bahn-Tunneln platzieren und erhalten eine Rückmeldung darüber, ob Mr. X an ihnen vorbeigefahren ist. Nach einer Wartezeit von 30 Sekunden erhalten sie eine weitere Chance, die Wächter zu platzieren. Diese Aktivität wurde entwickelt, um die Anwendung einer topographischen Suchstrategie zu analysieren.

6.4 Methodik

Um die Forschungsfrage zu beantworten, wurde ein Querschnittsstudiendesign angewandt. Über eine eigens eingerichtete Website wurde der „informatische Escape-Room“ an Schulen in der näheren Umgebung beworben, um Schulklassen zu gewinnen. Im Folgenden werden der konkrete Ablauf, die Stichprobe sowie die Auswertung im Detail beschrieben.

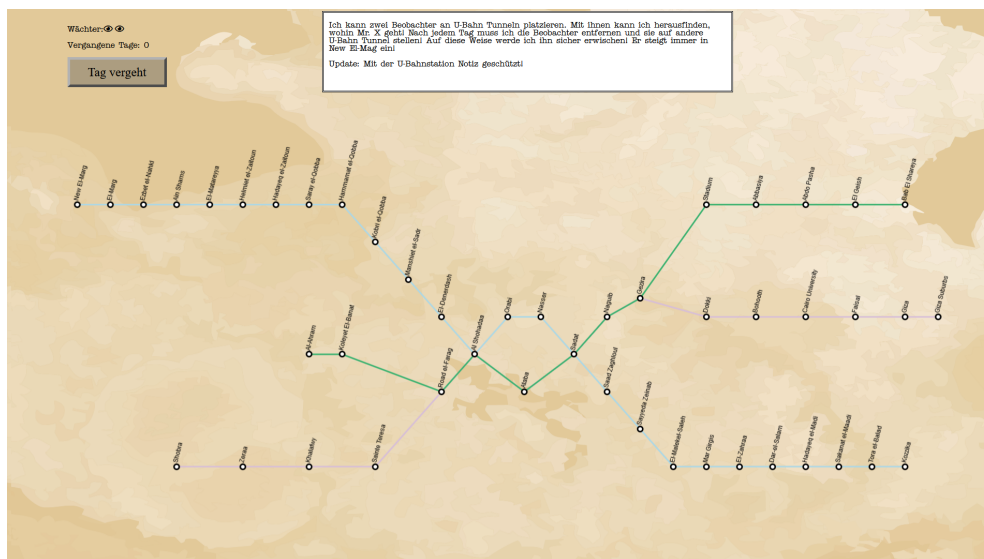


Abbildung 6.4: Mr. X

Aufgabe	Untersuchungsziel
Monitor	Troubleshooting-Vorgehen; „functional/discrepancy detection“-Strategie
Kabelsalat	Troubleshooting-Vorgehen; „functional/discrepancy detection“-Strategie
Telefon abhören	Troubleshooting-Vorgehen
Tal der Könige	Topographische Suchstrategie
Mr. X	Topographische Suchstrategie

Tabelle 6.1: Übersicht der resultierenden Aufgaben

Ablauf

Jede Schulklasse wurde in Gruppen von 4-6 Schülerinnen bzw. Schülern aufgeteilt, teilweise randomisiert, in der Mehrheit aber nicht. Je nach Größe der Schulklasse und Verfügbarkeit entsprechend geeigneter Räume an der Universität spielten eine oder zwei Gruppen (in unterschiedlichen Räumen) gleichzeitig den Escape-Room. Die restlichen Schülerinnen und Schüler erhielten solange ein Parallelprogramm ohne Bezug zu Debugging. Jede der Gruppen bekam vor bzw. im Raum eine kurze Einführung in die Geschichte, die Funktionsweise der verschiedenen Schlösser und Regeln des Raums²¹ und hatte dann 60 Minuten Zeit, alle Rätsel zu lösen und den gestohlenen Gegenstand zu finden. Über eine im Raum angebrachte hochauflösende Überwachungskamera konnte das Verhalten der Schülerinnen und Schüler aufgezeichnet werden. Mit Hilfe der Zweiwege-Audiofunktion der Kamera

²¹Beispielsweise, dass sie keine Kraft einsetzen müssen, etwaige weitere Türen nicht weiter beachten oder dass alle „festgeklebten“ Gegenstände nicht abgenommen werden dürfen

konnte darüber hinaus sowohl die Kommunikation der Teilnehmenden erhoben als auch bei Bedarf mit ihnen interagiert werden, um das Spiel zu steuern und Hilfestellungen zu geben. In den verschiedenen Durchführungen wurden abhängig vom Alter und im Zuge der Erprobung einzelne Aufgaben durchrotiert. Die Aufgaben waren dabei immer so angeordnet, dass die Schülerinnen und Schüler zwei „Pfade“ durch den Raum parallel verfolgen konnten, die jeweils zu zwei Teilen einer PIN für den finalen Safe führten. Gerade bei großen Gruppen konnten so die Fälle verringert werden, in denen sich alle Teilnehmenden im Kreis um einzelne Materialien stellten, sodass das Verhalten nicht mehr oder nur eingeschränkt beobachtbar war.

Stichprobe

Der Escape-Room wurde zu sieben Gelegenheiten eingesetzt. Jedes Mal kam dafür eine Schulklasse zu Besuch. Insgesamt spielten damit etwa 150 Schüler in 28 Gruppen von vier bis sechs Personen den Raum. Alle Schülerinnen und Schüler kamen aus Gymnasien der Umgebung und waren zwischen 14 und 18 Jahre alt. Einige hatten bereits Erfahrung in Informatik, aber höchstens ein Jahr formalen Informatikunterricht. Darüber hinaus hatten einige der Schülerinnen und Schüler bereits Erfahrung mit Escape-Rooms, die bei bestimmten „Meta-Faktoren“ Vorteile bieten könnten, z.B. wie man einen Raum durchsucht, bestimmte Schlösser bestimmten Aufgaben anhand der Anzahl der Lösungseingaben zuordnet usw. Insgesamt wurden 32 Stunden Videomaterial erhoben.

Auswertung

Ziel dieser Untersuchung ist es, relevante Debugging-Lernvoraussetzungen durch die Analyse des Troubleshooting-Verhaltens der Schülerinnen und Schüler zu identifizieren. Damit entsprechende Verhaltensweisen als relevant erachtet werden können, müssen sie dabei zumindest in einer gewissen Häufigkeit auftreten. Daher erscheint eine Kombination aus qualitativer und quantitativer Auswertung nach *Mayring (2001)* geeignet, die aus drei Schritten besteht:

1. Zunächst wurden die Videodaten mit Hilfe einer strukturierenden qualitativen Inhaltsanalyse nach *Mayring, Gläser-Zikuda und Ziegelbauer (2005)* (vergleichbar zu *Hacke (2019)*) analysiert, um alle einzelnen *Handlungen* der Schülerinnen und Schüler für die jeweiligen Aufgaben zu erfassen. Dazu wurde ein deduktives Kategoriensystem entwickelt (vgl. Auszug in Tabelle 6.2), das alle möglichen Handlungen der Schülerinnen und Schüler enthält. Um zu vermeiden, dass wichtige Aspekte aufgrund zuvor definierter Kategorien vernachlässigt werden, waren induktive Ergänzungen möglich.

Die Videodaten stellen dabei die Grundlage der Auswertung dar. Videopassagen mit einer Länge von mindestens 5 Sekunden dienen als Analyseeinheiten. Bezüglich der Interrater-Übereinstimmung wurden die Videos von 6 Gruppen (ca. 20 Prozent der Daten) ebenfalls von einem zweiten Forscher kodiert.

2. Im nächsten Schritt wurden die Kategorien nach der Häufigkeit ihres Auftretens im Material angeordnet. Für die weitere Analyse wurden nur Handlungen berücksichtigt, die in mindestens 20 Prozent der Gruppen beobachtet wurden.
3. Zuletzt wurden die Debugging-Lernvoraussetzungen identifiziert. Dazu wurden die relevanten Handlungen für jede Gruppe abhängig vom jeweiligen Kontext, in dem sie gezeigt wurden, interpretiert und entweder einem Schritt des allgemeinen Troubleshootingprozesses (vgl. Abbildung 5.2) oder einer bestimmten Troubleshootingstrategie zugeordnet (analog zu *Dounas-Frazer et al. (2016)* für das Troubleshooting in elektrischen Schaltungen). Nach der Zuordnung wurden Charakteristika und Merkmale dieser analysiert, beispielsweise wie die Schülerinnen und Schüler einen bestimmten Schritt des Troubleshootingprozesses durchgeführt oder eine bestimmte Strategie angewendet haben. Die Verallgemeinerung dieser Merkmale und Charakteristika über alle Gruppen hinweg ermöglicht damit die Identifikation von Debugging-Lernvoraussetzungen.

Aufgabe	Kategorie
Kabelsalat	Kabelkette in Steckdose {1,2,3} einstecken Kabel einzeln überprüfen Kabelkette schrittweise und systematisch kürzen/verlängern Kabelkette zufällig kürzen/verlängern/ändern Taschenlampe einzeln überprüfen Verbindungen der Kabel überprüfen
Monitor	ausgestecktes Kabel finden Ein/Aus-Knopf drücken Andere Knöpfe drücken Monitor untersuchen Anschlüsse des Monitors untersuchen Anschlüsse an der Steckdose überprüfen Kabel einstecken

Tabelle 6.2: Auszug des Kategoriensystem für die strukturierende qualitative Inhaltsanalyse (Schritt 1)

6.5 Ergebnisse

Im Folgenden werden die beobachteten Troubleshooting-Verhaltensweisen und Strategien für die einzelnen Aufgaben beschrieben. Für jede Aufgabe werden dazu zunächst die erwarteten Vorgehensweisen der Schülerinnen und Schüler entsprechend dem Troubleshootingprozess (siehe Abbildung 5.2) und der Troubleshootingstrategien (vergleiche Kapitel 5.3.3) dargelegt. Anschließend wird das tatsächlich beobachtete Verhalten charakterisiert, ehe die jeweiligen Ergebnisse zusammengefasst und im Hinblick auf Debugging-Lernvoraussetzungen interpretiert werden.

6.5.1 Monitor

Die Monitor-Aufgabe stellte immer das erste Rätsel des Raums dar, da die Schülerinnen und Schüler nach dem Lösen der Aufgabe einen Timer mit der verfügbaren Restzeit sowie notwendige Informationen für die nächste Aufgabe erhielten. Folgende Vorgehensweise wurde erwartet:

1. Bemerkten, dass der Monitor nicht funktioniert (*formulate problem description*).
2. Basierend auf dieser Beobachtung die Hypothese aufstellen, dass kein Strom verbunden ist (*generate causes*).
3. Den Ein/Aus-Knopf bzw. andere Knöpfe drücken und/oder die Anschlüsse des Monitors systematisch prüfen (*test*).
4. Das ausgesteckte Kabel suchen (wenn nicht schon vorher gefunden) und einstecken und damit die Aufgabe lösen (*repair and evaluate*).

Bei der Analyse der Videodaten wurden die folgenden Verhaltensweisen beobachtet: Bevor die Schülerinnen und Schüler den Raum betraten, erhielten sie den Hinweis, dass „der böse Professor etwas Verdächtiges an seinem Monitor gemacht hatte“, sodass alle Gruppen den Monitor innerhalb der ersten Minuten untersuchten und feststellten, dass er nicht funktionierte. Einige Gruppen (30 %) entdeckten sofort das lose Kabel (das direkt neben dem Monitor lag), steckten es ein und lösten damit die Aufgabe. Andere Gruppen (etwa 35 %) kontrollierten systematisch die Anschlüsse am Bildschirm und/oder an den Steckdosen und fanden dann entweder sofort das Kabel oder begannen mit der Suche danach. Nur wenige Gruppen formulierten explizit eine entsprechende Hypothese; dennoch kann vermutet werden, dass sie eine typische Ursache für ein solches Problem als Reaktion auf die *interne* Hypothese, dass der Bildschirm möglicherweise nicht richtig verbunden ist, auf der Grundlage ihrer Alltagserfahrung überprüften.

Ein beträchtlicher Anteil der anderen Gruppen (35 %) benötigte deutlich mehr Zeit oder sogar einen Hinweis zur Lösung dieser Aufgabe. Die meisten dieser Gruppen drückten den Ein/Aus-Schalter und stellten fest, dass keine Reaktion auf dem Bildschirm zu sehen war. Dennoch formulierten sie keine Hypothese über den Grund dafür. Stattdessen inspizierten sie unsystematisch den Bereich um den Monitor herum oder die Steckdosenleiste, ohne jedoch zu verfolgen, welche Kabel wohin führen. Einige Schülerinnen und Schüler hielten das abgesteckte Kabel sogar in der Hand, wussten aber nicht, was sie damit machen wollten. Schließlich begannen sie damit, den Raum nach anderen Aufgaben zu durchsuchen.

Ein weiteres häufiges Verhalten, das viele Gruppen zeigten, war der Versuch, andere Knöpfe des Monitors zu drücken, z.B. um die Eingabe für den Bildschirm zu ändern. Ein wahrscheinlicher Grund dafür ist, dass die Wahl der falschen Quelle eine häufige Fehlerursache ist, auf die die Schülerinnen und Schüler in ihrem täglichen Leben gestoßen sein könnten. Ohne Stromversorgung sahen sie keine Rückmeldung auf dem Bildschirm, sodass ihnen diese Heuristik in diesem speziellen Fall nicht half (außer weitere Hinweise zu erhalten, die darauf hindeuteten, dass kein Strom angeschlossen ist).

Zusammenfassend weisen die Daten für diese Übung auf zwei Debugging-Lernvoraussetzungen hin. Zunächst wendeten einige Schülerinnen und Schüler einen systematischen Fehlersuchprozess an und ließen Erfahrungen aus ihrem Alltag entsprechend den Erwartungen einfließen. Eine beträchtliche Anzahl von ihnen war jedoch nicht in der Lage, eine initiale Hypothese darüber aufzustellen, warum der Monitor nicht funktionierte. Insgesamt waren die Probleme, die viele Schülerinnen und Schüler mit dieser Aufgabe hatten, sowie der Grad der Hilflosigkeit, der auch beim Debugging üblich ist (*Perkins et al., 1986*), überraschend.

6.5.2 Kabelsalat

In der Kabelsalat-Aufgabe wurde das folgende Verhalten erwartet:

1. Die Kabelkette einstecken und feststellen, dass die Taschenlampe nicht leuchtet (*formulate problem description*).
2. Mehrere Hypothesen über die Ursache aufstellen, etwa dass die Steckdose, der Adapter, die Kabel oder die Taschenlampe defekt sind (*generate causes*).
3. Diese Hypothesen systematisch überprüfen (*test*), indem die jeweilige Komponente mit Hilfe einer *functional/discrepancy detection*-Strategien überprüft wird.
4. Zu dem Schluss kommen, dass einige der Kabel nicht funktionieren, diese Kabel identifizieren und die Kabelkette ohne sie zusammenstecken, sodass mit Hilfe der

Taschenlampe in die Box geleuchtet und die Zahlen darin gelesen werden können (*repair and evaluate*).

Wie erwartet versuchten alle Schülerinnen und Schüler zunächst, die Kabelkette mit der Taschenlampe an ihrem Ende in eine Steckdose zu stecken. Nachdem sie festgestellt hatten, dass die Taschenlampe nicht leuchtete, kontrollierten fast alle Gruppen die Verbindung der einzelnen USB-Kabel. Dies kann als *Anwendung von Mustern für typische Fehler* beschrieben werden, da die Schülerinnen und Schüler wahrscheinlich ähnliche Erfahrungen in ihrem Alltag gemacht haben. Zu diesem Zeitpunkt stellten viele Gruppen ausdrücklich die Hypothese auf, dass die Steckdose defekt sein könnte. Um diese Hypothese zu überprüfen, schlossen sie die Kabel an eine andere Steckdose im Raum an. Als Reaktion darauf hatte die Mehrheit der Gruppen aus einem von zwei Gründen Probleme, die Aufgabe zu lösen:

- Ein Teil (etwa 37 %) der Schülerinnen und Schüler war **nicht in der Lage, eine alternative Hypothese aufzustellen**. Einige Gruppen überprüften sogar eine dritte Steckdose. Wenn ihr erster Ansatz nicht funktionierte – selbst nachdem sie den Hinweis erhalten hatten, dass die Steckdosen und die Taschenlampe einwandfrei funktionieren – fehlte ihnen eine Richtung, um weiter an dem Problem zu arbeiten.
- Andere Gruppen (ca. 30 %) stellten die Hypothese auf, dass die Taschenlampe kaputt sein könnte. Allerdings fehlte ihnen ein Ansatz zur **Prüfung dieser Hypothese**. Einige Gruppen überprüften die Taschenlampe schließlich mit einer kürzeren Kabelkette.

Überraschenderweise kontrollierten nur etwa 25 % der Gruppen, ob die Taschenlampe funktioniert, wenn sie direkt und ohne Kabelkette an das Netzteil angeschlossen ist. Viele Gruppen benötigten Hinweise, wie z.B. „die Kabel überprüfen“, oder „dass Steckdose und Taschenlampe in Ordnung sind“. An dem Punkt, an dem sie herausfanden, dass einige der Kabel defekt sein könnten, überprüften die meisten Gruppen systematisch ein Kabel nach dem anderen, um die defekten zu identifizieren. Andere Gruppen erweiterten die Kabelkette systematisch Schritt für Schritt. Keine Gruppe wandte für dieses Problem eine binäre Suche an. Angesichts der geringen Anzahl von Kabeln und der (effizienteren) Methode, jedes Kabel einzeln zu untersuchen, erscheint die Verwendung einer solchen binären Suche auch als ungeeignet.

Zusammenfassend deuten die Daten für diese Aufgabe auf drei Debugging-Lernvoraussetzungen hin. Zunächst ist festzustellen, dass die meisten Schülerinnen und Schüler zunächst einen systematische Troubleshootingprozess entsprechend der Erwartungen anwendeten und sogar Erfahrungen aus ihrem täglichen Leben mit einbezogen – allerdings nur für ihre erste Hypothese. Sie hatten große Probleme damit, alternative Hypothesen aufzustellen, nachdem ihr erster Ansatz nicht erfolgreich war. Überraschenderweise hatten die Schülerinnen und Schüler darüber hinaus große Probleme, wenn sie die richtige Hypothese hatten, die entsprechende Komponente isoliert zu testen.

6.5.3 Telefon abhören

In dieser Aufgabe wurde das folgende Verhalten erwartet:

1. Zufällig Kabel mit Steckern verbinden (*formulate problem description*).
2. Dabei einen Zusammenhang zwischen der Anzahl der Pieptöne und den verbundenen Kabeln herstellen und damit das System verstehen (*generate causes/test*).
3. Systematisch Schritt für Schritt die richtige Position für jedes Kabel finden, indem jeder der verbleibenden Stecker für ein Kabel überprüft wird (*repair and evaluate*).

Die meisten Schülerinnen und Schüler begannen damit, alle fünf Kabel anzuschließen. Abhängig von der Anzahl der passenden Stecker erhielten sie Audio-Feedback in Form von Pieptönen. Alle Gruppen begannen dann, die Positionen der Kabel nach dem Zufallsprinzip zu ändern. Auf diese Weise wurde die Anzahl der Pieptöne entweder erhöht oder verringert. Wenn die Anzahl der Pieptöne und damit die Anzahl der korrekt positionierten Stecker abnahm, zeigten die Schülerinnen und Schüler ein interessantes Muster: Oftmals machten sie die Veränderungen, die zu dem Rückgang geführt hatten, nicht rückgängig, sondern wechselten weitere Kabel. Eine Gruppe hatte zum Beispiel bereits drei Stecker durch zufälliges Vertauschen der Positionen, meist von benachbarten Steckern, richtig positioniert. Dennoch machten sie so lange weiter, bis nur noch ein Piepton zu hören war, entfernten schließlich alle Stecker und wandten sich vorerst einer anderen Aufgabe zu. Obwohl dies nicht mit abschließender Sicherheit gesagt werden kann, schien die Mehrheit der Gruppen, die dieses Verhalten zeigten, das Konzept des „Je-mehr-Pieptöne-desto-besser“ verstanden zu haben. Nichtsdestotrotz machten die Schülerinnen und Schüler, zumindest für diesen eher explorativen Prozess, bestimmte Änderungen nicht rückgängig, wenn sie das Ergebnis verschlechtert oder zumindest nicht verbessert hatten.

Die meisten Gruppen wechselten nach einiger Zeit zu einem systematischeren Ansatz: Etwa die Hälfte entfernte alle Stecker und begann von vorn, indem sie alle 5 Positionen für den ersten Stecker überprüften, dann die restlichen vier für den nächsten und so weiter. Die andere Hälfte identifizierte die bereits korrekt positionierten Kabel, indem sie diese nacheinander entfernte, und testete dann nur noch die verbleibenden Positionen auf die restlichen Stecker.

Zusammengefasst zeigen die Daten für diese Aufgabe zwei Debugging-Lernvoraussetzungen. Die meisten Gruppen lösten diese Aufgabe entsprechend den Erwartungen: Nachdem sie das System in einer explorativen Art und Weise verstanden hatten, wandten sie einen systematischen Prozess an, bei dem sie Schritt für Schritt die richtige Position für jedes Kabel prüften. Allerdings konnte das interessante Muster identifiziert werden, dass die Schülerinnen und Schüler am Ende ihrer Exploration des Systems erfolglose Änderungen

nur selten rückgängig machten, obwohl sie die Bedeutung der Anzahl der Pieptöne bereits verstanden zu haben schienen. Ähnliches Verhalten ist beim Debugging üblich, wo Novizen oft zusätzliche Fehler hinzufügen, indem sie erfolglose Korrekturversuche nicht rückgängig machen (*Gugerty und Olson, 1986*).

6.5.4 Tal der Könige

In der Tal-der-Könige-Aufgabe wurde das folgende Verhalten erwartet:

1. Das System der Pfeilrichtungen und der Route anhand des gegebenen Beispiels verstehen und die ersten Pfeile mit der Route vergleichen (*formulate problem description*).
2. Die Route nachverfolgen und dabei Hilfsmittel verwenden, die helfen, die falschen Pfeile und/oder die aktuelle Position festzuhalten (*apply forward topographic search strategy*).

Überraschenderweise mussten etwa 75 % der Gruppen die Route mehr als zweimal nachverfolgen. Für die meisten stellten dabei die ersten drei falschen Pfeilrichtungen (von insgesamt sechs) kein Problem dar, aber je länger die Route war, desto mehr Fehlidentifikationen oder Verwechslungen wurden beobachtet. Dabei konnten folgende Ursachen für diese Probleme ausgemacht werden:

Zunächst verfolgten die Schülerinnen und Schüler ihren Fortschritt oft mit den Fingern, aber meist nur auf der Karte oder der Route – trotz der Arbeit in der Gruppe. Dies führte nach einiger Zeit zu Verwirrung über die aktuelle Position, insbesondere bei gleichzeitig stattfindenden Gruppendiskussionen. Ein weiteres häufiges Muster war, dass ein oder zwei Schülerinnen und Schüler an der Route und der Karte arbeiteten, während eine andere Schülerin bzw. ein anderer Schüler die von ihnen festgestellten Fehler aufschrieb. Dabei gab es häufig Probleme in ihrer Kommunikation: Einige Gruppen identifizierten alle sechs Fehler einwandfrei, aber sie vergaßen oder versäumten es, einen davon aufzuschreiben, oder schrieben ihn zweimal auf. Im Allgemeinen begannen viele Gruppen erst nach dem ersten Nachverfolgen der gesamten Route damit, zusätzliche Hilfsmittel zu verwenden und sich Notizen zu machen. Obwohl sie beispielsweise in der Lage waren, die falschen Pfeile oder zusätzliche Informationen direkt auf dem Kartenpapier zu markieren, entschieden sich viele Gruppen dazu, dies auf einem separaten Blatt Papier zu tun – möglicherweise, weil sie das Spielmaterial nicht beschädigen wollten. Einige Gruppen transformierten sogar die Route in die Pfeil-Darstellung und verglichen sie dann, machten aber Fehler im Prozess dieser Überführung. Im Allgemeinen bestand ein häufiges Muster darin, dass die Schülerinnen und Schüler beim erneuten Nachverfolgen der Route die zuvor korrekt identifizierten falschen Pfeile als richtig erkannten, was zu noch mehr Verwirrung führte.

Zusammenfassend ist festzustellen, dass die Probleme vieler Schülerinnen und Schüler durchaus überraschend waren. Erwartet wurde, dass die Aufgabe recht einfach sei, da sie sich an den Unplugged-Debugging-Übungen für Grundschul Kinder orientierte, wenn auch mit einer verlängerten Route. Während es kein Problem darstellte, das System zu verstehen, wie z.B. die Bedeutung der Pfeile, hatten Schülerinnen und Schüler Mühe, ihre aktuelle Position zu verfolgen, während sie falsche Pfeile nachverfolgten und/oder als richtig erkannten, insbesondere gegen Ende der Route. Ähnliche Probleme mit dem *Tracing* sind auch in der Programmierung für Novizen üblich (Lister et al., 2004).

6.5.5 Mr. X

In dieser Aufgabe wurde das folgende Verhalten erwartet:

1. Mit der gegebenen Startposition anfangen.
2. Die Ausgangsstation systematisch isolieren, indem Wächter aufgestellt werden, die so viele Informationen wie möglich über die Route von Mr. X liefern, z.B. an Tunneln nach Umsteigestationen (*apply forward topographic search strategy*).

Für diese Aufgabe konnten mehrere verschiedene Verhaltensmuster der Schülerinnen und Schüler beobachtet werden. Ein großer Prozentsatz (etwa 60 %) wandte tatsächlich eine optimale oder nahezu optimale Strategie an: Ihre Annäherung ging immer von dem gegebenen Einstiegspunkt (der Endstation der Linie) von Mr. X aus. Darauf aufbauend *tracten* sie die Route von Mr. X durch das U-Bahn-System, indem sie Wächter an U-Bahn-Tunneln platzierten, die viele Informationen lieferten, wie z.B. Stationen direkt nach Umsteigepunkten. Auf der Grundlage der erhaltenen Rückmeldungen isolierten und identifizierten sie die fragliche U-Bahn-Station Schritt für Schritt auf systematische Weise. Während sie die 30 Sekunden warteten, bis sie die Beobachter wieder platzieren konnten, diskutierte eine Gruppe zum Beispiel die Platzierung der nächsten Beobachter wie folgt: (A) „Lass uns den Wächter hier platzieren, hier muss er vorbei“ (B) „Ja, genau, hier muss er sowieso vorbei [daher hilft uns das nicht]“. Dann wies B auf eine bessere Stelle hin, die mehr Informationen darüber lieferte, wohin Mr. X ging: (B) „Hier wissen wir, ob er von der blauen auf die grüne U-Bahn-Linie wechselt“. Diese Gruppen benötigten nur 3 bis 5 Iterationen zur Lösung der Aufgabe.

Ein übliches Muster für weniger effiziente Gruppen war es, eine andere Art von *topographischer Strategie* anzuwenden: Anstatt die mögliche Route, ausgehend vom gegebenen Einstiegspunkt durch das U-Bahn-System, nachzuvollziehen, stellten sie ihre Wächter auf entfernte Äste des U-Bahn-Systems, um herauszufinden, ob Mr. X an ihnen vorbeigefahren war. Teilweise wählten sie sogar Bahnhöfe in der Nähe der jeweiligen Endstation anstelle der Stationen direkt nach den Umsteigestationen. Auf diese Weise benötigten sie eine wesentlich größere Anzahl von Iterationen, um die richtige Station zu finden.

Ein weiteres häufiges Muster war, dass die Schülerinnen und Schüler zunächst die U-Bahn-Tunnel direkt neben oder ganz in der Nähe der Eingangsstation kontrollierten. Erst nachdem sie bemerkt hatten, dass es aufgrund der 30-sekündigen Wartezeit lange dauern würde, die Aufgabe auf diese Weise zu lösen, veränderten sie ihr Vorgehen.

Zusammenfassend wurden zwei verschiedene topographische Strategien beobachtet, die von den Schülerinnen und Schülern angewandt wurden: Ein Teil verfolgte den Weg von der Eingangsstation auf effiziente Weise. Andere Schülerinnen und Schüler isolierten die U-Bahnstation, indem sie die Zweige nacheinander ausschlossen, was für diese spezielle Aufgabe ineffizient war. Bei der Fehlersuche stellt auch die Auswahl geeigneter Orte für die Platzierung von *print*-Anweisungen oder Haltepunkten eine große Herausforderung für Novizen dar (Murphy et al., 2008).

6.6 Interpretation

Im Folgenden werden die Ergebnisse aus den einzelnen Aufgaben generalisiert und im Hinblick auf Programmierung und Debugging interpretiert. Daneben werden daraus entstehende Implikationen für das Debugging im Unterricht diskutiert.

6.6.1 Generalisierung und Diskussion der Ergebnisse

Debugging-Lernvoraussetzung: Schülerinnen und Schüler haben Probleme mit der Generierung von Hypothesen, insbesondere von Alternativhypothesen.

Die Daten zeigen, dass die Mehrheit der Schülerinnen und Schüler ein Vorgehen vergleichbar mit dem diskutierten allgemeinen Troubleshooting-Prozess anwendete, auch wenn sie nicht jede Hypothese explizit formulierten. Obwohl sie im Allgemeinen einem Plan folgten, hatten sie Probleme in bestimmten Phasen des Troubleshooting-Prozesses. Für das Debugging (wie auch für das Troubleshooting, vgl. Morris und Rouse (1985)) ist die Fähigkeit, effektiv mehrere zielführende Hypothesen zu generieren, ein entscheidender Unterschied zwischen Experten und Novizen (Gugerty und Olson, 1986; Kim et al., 2018). Bei der Monitor-Aufgabe hatte ein beachtlicher Teil der Schülerinnen und Schüler Probleme mit der **Formulierung einer initialen Hypothese**. Im Gegensatz dazu waren alle Schülerinnen und Schüler in der Lage, eine erste Hypothese für die Kabelsalat-Aufgabe zu generieren und zu testen. Ursächlich könnte das unterschiedliche Domänenwissen sein, das für diese Aufgaben benötigt wird: Zwar haben alle Schülerinnen und Schüler Erfahrung im Umgang mit USB-Kabeln, aber wahrscheinlich haben nicht alle Routine beim Anschließen eines Computermonitors. Darüber hinaus deuten die Daten darauf hin, dass

sie Probleme mit der **Formulierung alternativer Hypothesen oder von mehr als einer Hypothese** haben. Viele Gruppen kamen nicht weiter, nachdem sie ihre erste Hypothese in der Kabelsalat-Aufgabe getestet hatten und ablehnen mussten. Einige Gruppen versuchten sogar eine dritte Steckdose. Dies steht im Einklang mit der Literatur, so berichten beispielsweise *Bereiter und Miller (1989)*, dass ein falscher Troubleshooting-Ansatz selbst angesichts widersprüchlicher Hinweise oft nicht verworfen wird. Für das Debugging stellten *Murphy et al. (2008)* fest, dass Schülerinnen und Schüler häufig nicht erkennen, dass sie feststecken und ihren Ansatz ändern müssen. Sie kommen zu dem Schluss, dass insbesondere das Nachdenken über alternative Fehlerursachen im Unterricht betont werden sollte. Auch *Jayathirtha, Fields und Kafai (2020)* betonen die Bedeutung von mehreren Hypothesen für das Debugging. Die Debugging-Lernvoraussetzungen, die in der vorliegenden Studie gefunden wurden, könnten das Debuggingverhalten der Schülerinnen und Schüler erklären und deuten darauf hin, wie wichtig es ist, die Schülerinnen und Schüler im Unterricht zu vermitteln, wie sie Hypothesen und insbesondere Alternativhypothesen generieren können.

Beim Debuggen von Code ist es ein übliches Verhalten von Schülerinnen und Schülern, einen Trial-and-Error-Ansatz anzuwenden, wie z.B. das Hinzufügen von Semikolons oder geschweiften Klammern oder das Erhöhen oder Verringern von Schleifenparametern (vgl. Kapitel 4). Die Daten zeigten ähnliche Muster erst, nachdem die Lernenden nicht mehr weiterkamen. Dies deutet darauf hin, dass Schülerinnen und Schüler, die einen solchen Trial-and-Error-Ansatz beim Debuggen anwenden, möglicherweise nicht wissen, wie sie an dem Problem weiterarbeiten sollen.

Debugging-Lernvoraussetzung: Schülerinnen und Schüler sind nicht in der Lage, ein System effektiv zu testen und einzelne Komponenten isoliert zu überprüfen.

In der Kabelsalat-Aufgabe stellten die Schülerinnen und Schüler die Hypothese auf, dass die Steckdose defekt sein könnte. Um dies zu testen, verwendeten sie eine andere Steckdose. Sie hatten aber große Probleme damit, ihre Hypothese – dass die Taschenlampe möglicherweise nicht funktioniert – zu überprüfen. Insbesondere waren sie nicht in der Lage, die Komponente „Taschenlampe“ isoliert zu testen. Für das Debuggen stellten *Murphy et al. (2008)* und *Fitzgerald et al. (2009)* fest, dass Studierende üblicherweise *Testen* als Strategie zum Debuggen einsetzen, aber meist nur mit bereitgestellten Beispielwerten. Sie verwenden nur selten spezifische Fälle wie Randbedingungen. Gerade da ähnliche Verhaltensweisen auch für das Troubleshooting in anderen Domänen berichtet werden (*Morris und Rouse, 1985*), liegt die Vermutung nahe, dass Schülerinnen und Schüler **keine Vorerfahrungen in Bezug auf effektives Testen** aus ihrem Alltag haben. Daher müssen entsprechende Fähigkeiten bereits auf einem basalen Niveau eingeführt und eingeübt werden – z.B. wie eine einzelne isolierte Komponente eines Programms getestet werden kann.

Debugging-Lernvoraussetzung: Änderungen rückgängig zu machen ist nicht intuitiv.

Bei der Reparatur eines Systems machen **Schülerinnen und Schüler selten ihre Änderungen rückgängig**, wenn sie den Fehler nicht behoben haben. Dazu machten *Murphy et al. (2008)* ähnliche Beobachtungen in ihrer qualitativen Studie über Strategien von Novizen beim Debuggen von Code, obwohl es in der IDE Unterstützung für ein solches schnelles und direktes *Undo* von Änderungen gibt. Aufgrund dieses „Nicht-rückgängig-Machens“ ist es ein weit verbreitetes Muster, dass Novizen neue Fehler in ihren Code einführen, wenn sie debuggen (*Gugerty und Olson, 1986*). Die Daten deuten darauf hin, dass das Rückgängigmachen von Veränderungen keine intuitive Vorgehensweise ist, die Schülerinnen und Schüler aus ihrem täglichen Leben mitbringen, und daher im Unterricht explizit adressiert werden muss.

Debugging-Lernvoraussetzung: Schülerinnen und Schüler nutzen Domänenwissen und beziehen Heuristiken und Muster für typische Fehler in ihren Troubleshootingprozess ein.

Die erhobenen Daten zeigen, dass Schülerinnen und Schüler ihre bisherigen Erfahrungen in die Fehlerbehebung einfließen lassen, wie z.B. die Überprüfung, ob die USB-Kabel richtig verbunden sind, ob die Stromversorgung angeschlossen ist oder ob der richtige Eingang für den Bildschirm gewählt wurde. Die Anwendung solcher Heuristiken und Muster für häufige Fehler ist eine wesentliche Debuggingfähigkeit, und professionelle Entwicklerinnen bzw. Entwickler verwenden diese Muster und Heuristiken auf der Grundlage ihrer Erfahrung, um ihren Debuggingprozess „abzukürzen“. Um das Lernen aus vergangenen Fehlern zu unterstützen, führen viele professionelle Entwicklerinnen und Entwickler ein *debugging log* oder Debugging-Tagebuch, mit dem sie ihre Debugging-Erfahrung dokumentieren (*Perscheid et al., 2017*). Offensichtlich fehlt Programmier-Novizen die entsprechende Erfahrung. Deshalb müssen die Lehrerinnen und Lehrer den Aufbau solcher Muster und Heuristiken unterstützen, indem sie Methoden wie Debugging-Tagebücher (vgl. *Carver und Risinger (1987)*) anwenden oder dafür sorgen, dass Situationen entstehen, in denen Schülerinnen und Schüler diese Muster und Heuristiken aufbauen können.

Debugging-Lernvoraussetzung: Schülerinnen und Schüler haben Probleme mit *cognitive load* und der effektiven Verwendung von Hilfsmitteln bei der topographischen Suche bzw. dem *Tracing*.

In der Mr.-X-Aufgabe wählten viele Schülerinnen und Schüler tatsächlich gut geeignete U-Bahn-Tunnel, um ihre Wächter zu platzieren. Im Gegensatz dazu haben Novizen beim Debuggen oftmals große Probleme bei der Anwendung vergleichbarer Strategien, wie beispielsweise dem *print-Debugging*, da sie häufig ungeeignete Orte oder Ausgaben verwenden

(Murphy et al., 2008) – obwohl gerade *print-Debugging* eine der am häufigsten angewandten Strategien ist (Fitzgerald et al., 2009). Eine These, die aus dem Datenmaterial hervorgeht, ist, dass das Hauptproblem im Programmierkontext darin besteht, kein mentales Modell des Programmablaufs zu haben oder nicht in der Lage zu sein, ein solches zu erstellen (vergleiche auch die Probleme von Novizen mit der *comprehension*-Strategie, Kapitel 2.2) – was in dieser Aufgabe durch das U-Bahn-Netz gegeben war.

Die Schülerinnen und Schüler hatten auch Mühe, die Route bei der Aufgabe „Tal der Könige“ nachzuverfolgen. Dies war überraschend, da eine einfache Pfeilnotation und damit konkretes (statt symbolisches) *Tracing* (vgl. Détienné und Soloway (1990)) ohne zusätzliche syntaktische Hürden verwendet wurde. Für das Nachvollziehen von Programmcode in der Programmierung werden die Fähigkeiten von Novizen als allgemein schlecht eingeschätzt (Lister et al., 2004). Schülerinnen und Schüler verfügen oft nicht über die notwendige Genauigkeit (Perkins et al., 1986), sind nicht in der Lage, das Abstraktionsniveau anzuheben und haben im Allgemeinen Probleme mit *cognitive load* (Cunningham et al., 2017). Es ist zu vermuten, dass in diesem Fall ein ähnliches Problem mit *cognitive load* besteht, da die Schülerinnen und Schüler vor allem mit den letzten drei falschen Pfeilen zu kämpfen hatten. Darüber hinaus verwendeten sie eher ineffiziente externe Darstellungen, wie es auch für das *Tracing* in der Programmierung berichtet wird (Cunningham et al., 2017). Damit deuten die Daten darauf hin, dass dieses Problem außerhalb des Programmierkontextes existiert – auch wenn diese Aufgabe durchaus enge Bezüge zu Programmierung aufweist. Um effizientes *Tracing* im Unterricht zu ermöglichen, müssen Schülerinnen und Schüler Erfahrungen sammeln, indem sie *Tracing* üben und lernen, wie externe Hilfsmittel zielführend genutzt werden.

Debugging-Lernvoraussetzung: Schülerinnen und Schüler sind schnell frustriert.

Eine weitere interessante Beobachtung aus den Daten ist, dass die Mehrheit der Schülerinnen und Schüler vergleichsweise schnell frustriert war. Im Gegensatz zu typischen Aufgaben hatten sie aufgrund des Problemlösecharakters des Escape-Rooms oftmals eben kein „Rezept“ zur Verfügung, wie die verschiedenen Aufgaben zu lösen seien. Anstatt hartnäckig zu bleiben und sich durch eine Aufgabe durchzukämpfen, gaben viele Schülerinnen und Schüler auf und zogen es vor, im Raum nach weiteren Hinweisen zu suchen oder an einem anderen Problem zu arbeiten. Insbesondere wenn sich die Gruppen aufteilten, um gleichzeitig an verschiedenen Aufgaben zu arbeiten und eine Gruppe einen Durchbruch erzielte, kamen (und blieben) alle Teammitglieder zusammen, um an der Problemstellung zu arbeiten, für die sie nun einige Anhaltspunkte hatten. Dieses Verhalten kann als „Suche nach Erfolgserlebnissen“ bezeichnet werden, anstatt sich weiter an einer im Moment frustrierenden Aufgabe zu versuchen. Für Schülerinnen und Schüler, die Code debuggen, gibt es ähnliche Erkenntnisse (Kinnunen und Simon, 2010), gerade da das Debuggen ein

Prozess ist, bei dem ein gewisses Maß an Ausdauer und Beharrlichkeit benötigt wird. Die Daten deuten darauf hin, dass das Durchhalten in einer Aufgabe bis zu einem gewissen Grad unabhängig von der Programmierung ist und im Unterricht behandelt werden muss, z.B. durch das Bereitstellen von Strategien, mit deren Hilfe Schülerinnen und Schüler sich selbst helfen können, wenn sie nicht mehr weiter wissen.

6.6.2 Einordnung in der Forschungsstand

Wie in Kapitel 2.4.2 dargelegt, identifizierten *Simon et al. (2008)* mit einem ähnlichen Ansatz *debugging preconceptions* von Studierenden. Anstatt Videodaten aus einem Escape-Room zu verwenden, analysierten sie die Antworten der Studierenden für bestimmte Szenarien. Die Escape-Room-Methodik erlaubt es, das Verhalten in Aktion und detaillierter zu analysieren. Darüber hinaus sind die Schülerinnen und Schüler deutlich jünger und haben vermutlich weniger Troubleshooting-Erfahrungen aus ihrem täglichen Leben. Außerdem haben die Escape-Room-Aufgaben im Gegensatz zu den eher offenen Szenarien von *Simon et al. (2008)* immer nur eine richtige Lösung – entsprechend der Charakterisierung von Troubleshootingaufgaben nach *Jonassen und Hung (2006)*.

Nichtsdestotrotz liefern die Ergebnisse dieser Untersuchung zusätzliche Belege für einige ihrer Ergebnisse, wie z.B. dass das *Undo* für Schülerinnen und Schüler unnatürlich erscheint und die Lernenden oft keine alternativen Hypothesen aufstellen. Die Escape-Room-Methodik und die Aufgaben, die entsprechend der theoretischen Grundlage speziell entwickelt wurden, ermöglichen es jedoch, weitere Debugging-Lernvoraussetzungen zu identifizieren, wie z.B. dass Schülerinnen und Schüler Domänenwissen, Heuristiken und Muster einbeziehen oder welche Probleme sie mit bestimmten Strategien wie der topographischen Suche oder dem Testen haben.

6.6.3 Limitationen und Güte

Was die eingesetzte Escape-Room-Methodik angeht, waren die Möglichkeit, das Verhalten, den Prozess und die Strategien von Schülerinnen und Schülern zu beobachten, insgesamt zufriedenstellend. In den Daten gab es zwar sowohl Gruppen, in denen eine Schülerin bzw. ein Schüler allein eine bestimmte Aufgabe löste, als auch Gruppen, die eine bestimmte Aufgabe so lösten, dass – aufgrund des Kamerawinkels – das Vorgehen nicht genau beobachtet werden konnte. Folglich wurden diese Fälle in der Analyse ausgelassen. Nichtsdestotrotz konnten für die überwiegende Mehrheit (und angesichts der großen Zahl der Teilnehmenden) der Aufgaben und Gruppen die Aktionen der Schülerinnen und Schüler beobachtet und zusätzliche Erkenntnisse aus Gruppendiskussionen gewonnen werden. Darüber hinaus zeigten alle Gruppen aufgrund des Szenarios ein hohes Maß an Motiva-

tion. Da der Raum nicht linear war, konnten die Teilnehmenden manchmal parallel an bestimmten Aufgaben arbeiten. Sie konnten den Raum so stets nach weiteren Hinweisen für spätere Problemstellungen durchsuchen. Dies könnte die Tendenz beeinflusst haben, eine bestimmte Aufgabe aufzugeben und sich vorerst auf eine andere zu konzentrieren (obwohl nur wenige Gruppen beobachtet wurden, die sich aufteilten, um insgesamt parallel an Aufgaben zu arbeiten). Auch wurden die Gruppen aus den Schulklassen nicht oder nur teilweise randomisiert gezogen. Aufgrund der Zielsetzung der Untersuchung, relevante Lernvoraussetzungen über alle Probandinnen und Probanden zu identifizieren, und nicht vergleichende Aussagen über die Vorgehensweisen und Erfolgsquoten einzelner Schülerinnen bzw. Schüler und Gruppen zu treffen, erscheint eine solche Randomisierung allerdings nicht notwendig.

Für die Analyse der Troubleshootingprozesse zur Identifizierung von Debugging-Lernvoraussetzungen erscheint dieser Ansatz geeignet und damit die **Validität** der Untersuchung gewährleistet. Sicherlich lässt sich nicht das gesamte Troubleshooting-Verhalten auf die Domäne des Debuggens übertragen. Aus diesem Grund konnten einige der initial entwickelten Aufgaben keine signifikanten Erkenntnisse zur Forschungsfrage beitragen. Daher wurden diese Aufgaben in dieser Analyse ausgelassen. Die Aufgaben, deren Auswertung in dieser Untersuchung dargelegt wird, zeigen jedoch eine starke Verbindung zur Literatur, in der der Zusammenhang zwischen Debugging und Troubleshooting hergestellt (*Semantische Gültigkeit, Konstruktvalidität* vgl. Mayring (1985)), und insbesondere im dritten Schritt des kombiniert qualitativ-quantitativen Auswertungsverfahrens augenfällig wird. Infolgedessen waren debuggingbezogene Verhaltensweisen wie ein systematischer Troubleshootingprozess oder bestimmte globale Troubleshootingstrategien notwendig und beobachtbar. Die Ergebnisse unterstützen diese Annahme: Die meisten der Debugging-Lernvoraussetzungen, die in dieser Untersuchung identifiziert werden, spiegeln sich – wie oben diskutiert – im Debuggingverhalten von Programmieranfängerinnen und -anfängern bzw. im Troubleshooting-Verhalten gemäß Literatur wider (*korrelative Gültigkeit*, vgl. Mayring (1985)). Darüber hinaus trägt die Größe der Stichprobe zur (exemplarischen) Verallgemeinerung der Ergebnisse (*Stichprobengültigkeit*, vgl. Mayring (1985)) bei.

Zur Absicherung der Wissenschaftlichkeit und Qualität der gewonnenen Ergebnisse wurden zudem weitere Maßnahmen unternommen: Durch die Interrater-Übereinstimmung soll die **Intersubjektivität** der Untersuchungsergebnisse sichergestellt werden (vgl. Rädiker und Kuckartz (2019)). Außerdem soll durch die Verfahrensdokumentation im Sinne der Darstellung des Erhebungsinstrumentes und der Analyseschritte sowie die Regelgeleitetheit des systematischen Vorgehens (vgl. Mayring (2002)) die **Transparenz** der Vorgehensweise gewährleistet werden.

6.7 Fazit

Zusammenfassend werden in dieser Untersuchung auf Basis von debuggingbezogenem Troubleshooting-Verhalten von Schülerinnen und Schülern Debugging-Lernvoraussetzungen identifiziert. Der innovative methodische Ansatz mittels eines Escape-Rooms als Erhebungsinstrument bietet dabei im Vergleich zu einer schriftlichen Erhebung der Reaktionen der Teilnehmenden den Vorteil, dass der tatsächlichen Fehlerbehebungsprozess und die eingesetzten Strategien in einer natürlichen Umgebung beobachtet werden können, einschließlich der Reaktionen, die auftreten, wenn ein erster Ansatz nicht funktioniert. Die Kommunikation innerhalb der Gruppen, die bei der Gestaltung der Aufgaben aktiv gefördert wurde, erwies sich als geeignet, die Prozesse der Schülerinnen und Schüler beobachtbar zu machen.

Damit konnten Debugging-Lernvoraussetzungen identifiziert werden, die zum Verständnis des Debuggingverhaltens von Novizen beitragen. Insbesondere deuten die Ergebnisse darauf hin, dass viele typische Verhaltensweisen von Novizen beim Debuggen bereits vor der ersten Programmiererfahrung existieren. Damit stellen die Debugging-Lernvoraussetzungen Hinweise für die Entwicklung von Konzepten und Materialien für den Unterricht dar:

- Schülerinnen und Schüler wenden intuitiv einen systematischen Prozess für die Fehlersuche an, haben aber große Probleme mit dem Aufstellen von Hypothesen, insbesondere Alternativhypothesen. Im Unterricht muss großer Wert darauf gelegt werden, solche Verhaltensweisen einzuüben.
- *Undo* ist für Schülerinnen und Schüler keine intuitive Verhaltensweise, die sie aus ihrem Alltag mitbringen. Dieses Rückgängigmachen muss also explizit adressiert werden.
- In den Troubleshooting-Aufgaben haben die Schülerinnen und Schüler Heuristiken und Muster mit in ihren Prozess einbezogen, die sie aus Erfahrung gelernt hatten. Zum Debuggen fehlen den Novizen solche Erfahrungen. Deshalb sollten die Schülerinnen und Schüler aktiv dabei unterstützt werden, sich entsprechende Erfahrungen, Muster und Heuristiken anzueignen.
- Die Daten weisen darauf hin, dass effizientes Testen im täglichen Leben von Schülerinnen und Schülern keine große Rolle spielt, was zu einem Mangel an Erfahrung bei der Anwendung von *Testen* als Strategie zur Fehlerbehebung führt. Darüber hinaus hatten Schülerinnen und Schüler im Zusammenhang mit einer topographischen Suche Probleme mit dem Tracing. Entsprechende Debuggingstrategien wie *print-Debugging*, Auskommentieren oder Testen müssen daher im Debugging-Unterricht explizit behandelt und auf einem basalen Niveau eingeführt werden.

- Analog zum Forschungsstand bezüglich der Frustration und Hilflosigkeit beim Debuggen, zeigten die Schülerinnen und Schüler vergleichbare Muster beim Troubleshooten. Gerade da das Debuggen ein Prozess ist, bei dem ein gewisses Maß an Ausdauer und Beharrlichkeit benötigt wird, muss dieser Umstand adressiert werden, z.B. durch das Bereitstellen von Strategien, mit deren Hilfe Schülerinnen und Schüler sich selbst helfen können, wenn sie beim Debuggen nicht weiter kommen.

Teil III:

Didaktische Strukturierung

Testing proves a programmer's failure. Debugging is the programmer's vindication.

BORIS BEIZER

7 Entwicklung von Konzepten und Materialien für den Informatikunterricht

In diesem Kapitel sollen nun im Sinne der *didaktischen Strukturierung* auf Basis der erfolgten Untersuchung der *zugrunde liegenden Perspektiven* Gestaltungskriterien für Konzepte und Materialien für das Debugging im Informatikunterricht in einer Synthese zusammengeführt werden. Damit wird der folgenden Forschungsfrage nachgegangen:

(RQ5) Wie sollten Konzepte und Materialien für die Vermittlung von Debuggingfähigkeiten im Unterricht gestaltet werden?

Aufbauend auf solcherart identifizierten Kriterien wird eine exemplarische Umsetzung eines Unterrichtskonzepts für die Sekundarstufe I und die textbasierte Programmierung beschrieben.

7.1 Synthese: Gestaltungskriterien

Die *fachliche Klärung* stellt mit dem resultierenden Modell der Debuggingfähigkeiten für Novizen die inhaltliche Basis der Vermittlung von Debugging dar. Darüber hinaus konnten sowohl aus dem Forschungsstand (vgl. Kapitel 2.7) als auch der *Untersuchung der zugrunde liegenden Perspektiven* verschiedene Gestaltungshinweise für entsprechende Konzepte und Materialien identifiziert werden. Diese lassen sich wie folgt als Gestaltungskriterien (GK) zusammenfassen:

Konzepte und Materialien . . .

GK1: . . . sollten Selbstständigkeit fordern und fördern. Um der berichteten Hilflosigkeit von Schülerinnen und Schülern und der daraus resultierenden *Turnschuhdidaktik* zu begegnen, ist es zentral, die Selbstständigkeit der Lernenden zu steigern. Bestätigt wird diese Notwendigkeit durch deren Troubleshooting-Verhalten: Schülerinnen und Schüler hatten große Probleme und waren sehr frustriert, wenn sie nicht weiterwussten. Daher müssen Konzepte und Materialien Strategien und Vorgehensweisen bereitstellen, mit deren Hilfe Schülerinnen und Schüler sich selbst helfen können, wenn sie nicht mehr weiterkommen. Einen vielversprechenden Ansatz dafür stellt gemäß den Erfahrungen der Lehrkräfte vor dem Hintergrund *erlernter Hilflosigkeit* das explizite Einfordern dieser Selbstständigkeit im Rahmen entsprechender Unterrichtskonzepte und -materialien dar.

GK2: ... sollten einen Fokus darauf legen, das Formulieren von Hypothesen einzuüben. Grundlegend für effektives Debugging ist das Verfolgen eines zielgerichteten Plans. Aus dem Forschungsstand geht hervor, dass für das Debuggen *eigener* Programme dazu ein Vorgehen gemäß der *isolation*-Strategie (bzw. *wissenschaftliche Methode* oder *backward reasoning*) angewandt werden sollte – auch in Anbetracht der existierenden vielversprechenden Untersuchungen zur Vermittlung eines solchen Vorgehens (siehe Kapitel 2.5). Wie die Untersuchung in Kapitel 6 zeigt, wenden Schülerinnen und Schüler im Troubleshooting tatsächlich intuitiv einen vergleichbaren systematischen Prozess für die Fehlersuche an. Große Probleme haben sie aber hauptsächlich mit dem Aufstellen von Hypothesen, gerade auch mit Alternativhypothesen – genauso wie Programmiernovizen beim Debugging. In entsprechenden Konzepten und Materialien muss daher großer Wert darauf gelegt werden, vor allem das explizite Formulieren von (mehreren) Hypothesen über Fehlerursachen einzuüben.

GK3: ... sollten insbesondere die Bedürfnisse „schwächerer“ Schülerinnen und Schüler adressieren. Die Lehrkräfte haben davon berichtet, dass vor allem „schwache“ und „durchschnittliche“ Lernende Probleme mit dem Debugging haben. Daher sollten Konzepte und Materialien insbesondere auch an die Bedürfnisse dieser Schülerinnen und Schüler angepasst werden. Zentral erscheint es dabei, Fehler als natürlichen Bestandteil der Programmierung und zugleich positiv konnotierte Lernchance zu etablieren, um – gemäß dem Forschungsstand, den Berichten der Lehrkräfte sowie dem beobachteten Verhalten beim Troubleshooting – Frustration und vorschnelles Aufgeben zu verhindern, sowie im Sinne überfachlicher Kompetenzen einen Beitrag zur Allgemeinbildung zu leisten.

GK4: ... sollten Debuggingstrategien explizit vermitteln. Nach den Berichten der Lehrkräfte werden Debuggingstrategien im Informatikunterricht nicht explizit – etwa in einer eigenen Unterrichtsphase – eingeführt, sondern vorwiegend in der individuellen Beratung vorgeschlagen oder vorgeführt. Der Forschungsstand zeigt allerdings, dass Novizen viele dieser Strategien ineffizient anwenden. Auch in der Untersuchung der Lernvoraussetzungen zeigte sich, dass Schülerinnen und Schüler für Strategien, wie etwa das *Testen* oder *Tracing*, nur wenige Vorerfahrungen aus ihrem Alltag haben und diese nur selten zielführend anwenden können. Entsprechende Debuggingstrategien (vgl. Kapitel 3) müssen daher in Konzepten und Materialien explizit behandelt und auf einem basalen Niveau eingeführt werden.

GK5: ... sollten den aktiven Aufbau von Erfahrung unterstützen. Darüber hinaus müssen für das Debugging Muster und Heuristiken für typische Fehler angewendet werden. Im Troubleshooting zeigte sich, dass Schülerinnen und Schüler ihre bisherigen Erfahrungen

berücksichtigen und in den Fehlersuchprozess miteinbeziehen. Für das Debuggen und die Domäne der Programmierung fehlen Novizen solche Erfahrungen allerdings. Gerade unter den Rahmenbedingungen schulischen Unterrichts, für den die Lehrkräfte berichten, dass in Anbetracht der Zeit oftmals doch einfach die Lösung für das Problem von der Lehrkraft genannt oder am Ende der Stunde (unreflektiert) übernommen wird, bauen Schülerinnen und Schüler entsprechende Erfahrungen nur eingeschränkt von selbst auf. Deshalb sollten in Materialien und Konzepten Gelegenheiten geschaffen werden, um die Schülerinnen und Schüler aktiv dabei zu unterstützen, dass sie sich entsprechende Erfahrungen, Muster und Heuristiken (vgl. Kapitel 3) aneignen und diese abrufen können.

GK6: ... sollten typische Hürden und Probleme von Lernenden beim Debugging adressieren. Aus dem Forschungsstand gehen einige typische Probleme von Novizen beim Debuggen hervor (vgl. Kapitel 2.2.2). Exemplarisch dafür steht die Angewohnheit, neue Fehler in das Programm einzufügen, weil erfolglose Korrekturversuche nicht rückgängig gemacht werden. Aus der Untersuchung der Perspektive der Lernenden zeigte sich, dass *Undo* eben keine intuitive Verhaltensweise von Schülerinnen und Schülern darstellt, die sie aus ihrem Alltag mitbringen. Darüber hinaus haben sie beispielsweise Probleme mit dem Umgang mit Fehlermeldungen oder der Platzierung von zielführenden Ausgaben sowie der genauen Beschreibung des Fehlverhaltens als Diskrepanz zwischen erwartetem und tatsächlichem Verhalten. Daher muss diesen und weiteren typischen Problemen in Konzepten und Materialien vorgebeugt werden bzw. diese adressiert werden.

GK7: ... sollten unterschiedliches Vorgehensweisen für verschiedene Fehlerarten vermitteln. Der Forschungsstand zeigt, dass der Debuggingprozess sich für verschiedene Fehlerarten unterscheidet. So benötigen die Schülerinnen und Schüler abhängig vom konkreten Fehlertyp jeweils angepasste Ansätze, Strategien und Werkzeuge. Diese Unterschiede müssen in entsprechenden Konzepten und Materialien aufgegriffen werden. Dabei stellen insbesondere – im Unterschied zur tertiären Bildung – auch Kompilierzeitfehler eine Hürde für Programmieranfängerinnen und -anfänger dar, die demgemäß in Konzepten und Materialien explizit aufgenommen werden müssen.

GK8: ... sollten der Vielfalt der persönlichen Debuggingvorlieben der Lehrkräfte Rechnung tragen. Die Lehrkräfte berichten, ganz unterschiedliche Debuggingstrategien und -werkzeuge in ihrem Unterricht zu vermitteln. Die Vorgehensweisen, die sie ihren Schülerinnen und Schülern nahebringen (wollen), sind dabei oftmals abhängig von ihren eigenen Gewohnheiten. Daher sollte für die Akzeptanz in der Unterrichtspraxis gerade in der Vermittlung entsprechender Debuggingstrategien und -werkzeuge in geeigneten Konzepten und Materialien die erforderliche Flexibilität und (Wahl-)Freiheit gewährleistet sein und

eben nicht der „eine Königsweg“ propagiert und erzwungen werden. So favorisieren auch professionelle Entwicklerinnen und Entwickler individuell unterschiedliche Ansätze und Strategien zum Debuggen (*Perscheid et al., 2017*).

GK9: ... sollten variabel bei Bedarf im Unterricht einsetzbar sein. Lehrkräfte erläutern, dass nur wenig Zeit für die Vermittlung von Debugging zur Verfügung steht und es eben oftmals kein explizites Thema des Curriculums darstellt. Entsprechende Konzepte und Materialien müssen sich für die Akzeptanz in der Schulpraxis daher in angemessenem Zeitumfang in den Unterricht einpassen lassen. Auch berichten die Lehrpersonen von der Erfahrung, dass die Vermittlung von *Wissen auf Vorrat* kaum erfolgreich war. Daher müssen beispielsweise entsprechende Strategien dann eingeführt werden können, wenn die Schülerinnen und Schüler diese benötigen, sodass sie diese direkt anwenden können.

GK10: ... sollten tatsächliche (relevante) Debuggingfähigkeiten fördern. Bei der Untersuchung des Forschungsstandes hat sich gezeigt, dass ein Unterschied im Vorgehen für das Debuggen eigener und fremder Programme besteht. Ziel des allgemeinbildenden Informatikunterrichts ist es zunächst, die Schülerinnen und Schüler dazu zu befähigen, Fehler in ihren eigenen Programmen finden und beheben zu können. Daher sollte versucht werden, Debugging in Situationen einzuüben, die möglichst nahe am Debuggen eigener Programme sind. Nachdem wie in Kapitel 2.1 herausgearbeitet, *Testen* und *Debuggen* fundamental unterschiedliche Tätigkeiten sind, sollte, wenn „fremde“ Programme zu Übungszwecken eingesetzt werden, der Fokus für die Schülerinnen und Schüler weniger darauf liegen, festzustellen, *ob* ein Fehler für ein gegebenes Programm vorliegt, als vielmehr, diesen zu finden und zu beheben.

Limitationen

Zentrale Limitation dieser Gestaltungskriterien für Konzepte und Materialien stellt der Fokus auf textbasierte Programmierung dar: Zum einen beschränkt sich der existierende Forschungsstand fast ausschließlich auf textbasierte Programmiersprachen, zum anderen bilden die in dieser Arbeit untersuchten Erfahrungen der Lehrkräfte die Schulpraxis an deutschen Gymnasien ab, in der oftmals textbasierte Sprachen eingesetzt werden. Dies zeigt sich exemplarisch an der Rolle der Kompilierzeitfehler, die in vielen blockbasierten Programmiersprachen so nicht möglich sind (*Altadmri, Kölling und Brown, 2016*), oder aber an der Hilflosigkeit und Frustration, die ebenfalls im Vergleich zu textbasierter Programmierung eher untypisch für blockbasierte Programmierung ist. Damit stellt sich zunächst die Frage der (uneingeschränkten) Übertragbarkeit dieser Kriterien. Nichtsdestotrotz finden

sich auch in der blockbasierten Programmierung beispielsweise unterschiedliche Fehlerklassen, für die verschiedenartige Strategien und Vorgehensweisen hilfreich sind, genauso wie üblicherweise die bisherige Erfahrung oder ein systematisches Vorgehen eine große Rolle spielen.

7.2 Exemplarische Umsetzung

Im Folgenden soll nun eine exemplarische Umsetzung eines Unterrichtskonzepts gemäß der Gestaltungskriterien (GK) vorgestellt werden. Als Zielgruppe wurden Programmieranfängerinnen und -anfänger und damit die Sekundarstufe I festgelegt. Aufgrund der entsprechenden möglichen Limitationen der Gestaltungskriterien als auch der Bedeutung in der Schulpraxis wurde dabei die textbasierte Programmierung und als konkrete Programmiersprache in den Materialien ob ihrer Verbreitung Java gewählt (Hubwieser et al., 2015). Das Unterrichtskonzept teilt sich dabei in mehrere Bausteine auf, die variabel und bei Bedarf eingesetzt werden können – gegliedert durch die verschiedenen relevanten Debuggingfähigkeiten (GK 9).

7.2.1 Systematisches Vorgehen

Die Grundlage des Modells der Debuggingfähigkeiten für Novizen und damit den ersten Baustein stellt dabei die Vermittlung eines systematischen Vorgehens dar. Dazu wurde eine entsprechend didaktisch adaptierte Variante des systematischen Vorgehens gemäß der *isolation*-Strategie entwickelt (vgl. Anhang C bzw. Abbildung 7.1). Gemäß GK7 wurde dabei eine Unterscheidung für verschiedene Fehlerklassen vorgenommen. Als Kategorisierung wurde die Einteilung in Kompilierzeit-, Laufzeit- und logische Fehler (vgl. Kapitel 2.3) gewählt. Diese Kategorisierung erscheint zielführend, da sie (im Gegensatz zu einer Einteilung in syntaktische und semantische oder *construct-related* und *non-construct-relatede* Fehler) aus der Perspektive der Schülerinnen und Schüler und der Art und Weise, in der sie auf den Fehler aufmerksam werden, vorgenommen wird²²: Für Kompilierzeitfehler erhalten sie beim Übersetzen eine Fehlermeldung des (Java-)Compilers, und die Entwicklungsumgebung weist direkt im Code auf einen entsprechenden Fehler hin. Bei Laufzeitfehlern wird die Programmausführung abgebrochen und es öffnet sich üblicherweise ein Konsolenfenster mit einer entsprechenden Fehlermeldung des Java-Interpreters. Logische Fehler sind hingegen nur anhand eines Unterschieds zwischen erwartetem und tatsächlichem Verhalten festzustellen.

²²So treten beispielsweise statische semantische Fehler zur Kompilierzeit und dynamische semantische Fehler zur Laufzeit auf.

Das entwickelte Vorgehen unterscheidet sich dabei für die jeweiligen Fehlertypen: Für Kompilierzeitfehler wird auf das Lesen der Fehlermeldung hingewiesen. Im Falle von Laufzeitfehlern wird auf Basis der Heuristik, die *erste* Fehlermeldung zu betrachten, eine Hypothese über die Fehlerursache aufgestellt und entsprechend überprüft. Für logische Fehler wird hingegen auf Grundlage des erwarteten und tatsächlichen Verhaltens des Programms eine Vermutung aufgestellt und überprüft. Damit wird auch die entsprechende Reihenfolge des Auftretens verschiedener Fehlertypen adressiert: Nur weil ein Programm frei von Kompilierzeitfehlern ist, heißt es eben nicht, dass es korrekt ist – eine typische Fehlvorstellung von Lernenden (Stamouli und Huggard, 2006). Außerdem wird innerhalb des Vorgehens gemäß GK6 das *Undo* explizit betont, wenn Änderungen eben nicht erfolgreich waren, sowie eine Heuristik für den Umgang mit Fehlermeldungen bereitgestellt.

Dieses Vorgehen wird in etwa 45 bis 90 Minuten im Unterricht eingeführt. Im dazu entwickelten Material (vgl. Anhang B) bearbeiten die Schülerinnen und Schüler zunächst *prototypenbasierte* Debuggingaufgaben. Anstatt wie in traditionellen Debuggingaufgaben direkt mit einer großen Menge fremden Codes konfrontiert zu sein, werden in mehreren Iterationen aufeinander aufbauende Prototypen eines Programms verwendet. Auf diese Art und Weise sehen sich die Schülerinnen und Schüler in jedem neuen Prototypen nur mit vergleichsweise wenig „fremdem“ Code ausgesetzt und kennen sich im „alten“ Code bereits aus. Dies ermöglicht eine Annäherung an das Debuggen eigener Programme (GK10). Beispielsweise ist im ersten Prototypen eines Pongspiels lediglich die Bewegung des Balles umgesetzt, und im nächsten werden zusätzlich die Schläger und deren Steuerung eingefügt. Da die Debugging- und nicht die Testfähigkeiten der Schülerinnen und Schüler gefördert werden sollen (GK10), ist die Anzahl der vorhandenen Fehler je Debuggingaufgabe gegeben. Aus dem gleichen Grund wurde darauf geachtet, dass das Fehlverhalten des Programms schnell ersichtlich ist, sodass direkt mit dem eigentlichen Debugging begonnen werden kann. Anschließend versuchen die Schülerinnen und Schüler, die gefundenen Fehler in verschiedene Kategorien einzuordnen. Darauf aufbauend wird das systematische Vorgehen anhand des Plakates (siehe Abb. 7.1) eingeführt, demonstriert und anhand weiterer prototypenbasierter Debuggingaufgaben eingeübt. Bei jedem Fehler werden die Schülerinnen und Schüler dabei aufgefordert, ihre Hypothesen explizit zu verschriftlichen (GK2). Das Plakat nimmt dabei eine zentrale Position im Klassenraum ein, um die Lernenden auch über die konkrete Intervention hinaus zu unterstützen und es der Lehrkraft zu ermöglichen, ein entsprechendes Vorgehen einzufordern (GK1).

7.2.2 Strategien und Werkzeuge

Erweitert wird dieses Vorgehen durch die Einführung konkreter Strategien bzw. Werkzeuge. Um dabei den Gestaltungskriterien der Vorlieben der Lehrkräfte (GK8) sowie der

Bedarfsorientierung (GK9) zu genügen, werden diese in kurzen Unterrichtsphasen in Form von *Skill-Cards* (vgl. z.B. *Schmalfeldt (2019)*) eingeführt: Die Schülerinnen und Schüler erkunden dabei zunächst – ganz ohne Debuggingkontext – die Möglichkeiten der jeweiligen Strategie, wie beispielsweise die Funktionalitäten eines Debuggers oder aber die Möglichkeiten von *print*-Anweisungen. Anschließend wenden sie diese Strategien bzw. Werkzeuge in Debuggingaufgaben an. Zunächst werden sie dabei eng geführt, um die Möglichkeiten zum Debugging aufzuzeigen, etwa indem sie dazu aufgefordert werden, einen Breakpoint an einer bestimmten Stelle zu setzen und den Zustand einer Variable zu beschreiben, um so auf einen Fehler zu stoßen. Schritt für Schritt sollen die Schülerinnen und Schüler Debuggingstrategien und Werkzeuge danach in offeneren Kontexten anwenden (GK4). Die Debuggingaufgaben werden dabei so gewählt, dass die Strategie einen tatsächlichen Mehrwert in Fehlerbehebung bietet (vgl. Anhang B).

Abschließend ergänzen die Schülerinnen und Schüler die Skill-Card für die jeweilige Strategie bzw. das jeweilige Werkzeug (vgl. Abbildung 7.2). Dazu beschreiben sie einerseits, welche Fragen mit Hilfe der Strategie beantwortet werden können, und andererseits, für welche Art von Fehler diese Strategie bzw. das Werkzeug hilfreich ist. Darüber hinaus halten die Schülerinnen und Schüler dort weitere Hinweise zum Einsatz der Strategie fest, die später durch weitere Erfahrung ergänzt werden können. Eine gegebenenfalls von der Lehrkraft vorbereitete Version der Skill-Card wird zusätzlich an der dafür vorgesehenen Position auf dem Plakat im Klassenzimmer ergänzt (vgl. Abbildung 7.1). Dieses Vorgehen erscheint sowohl für Werkzeuge, wie beispielsweise konkrete Funktionalitäten der Entwicklungsumgebung, als auch Debuggingstrategien geeignet.

7.2.3 Heuristiken und Muster für typische Fehler

Professionelle Entwicklerinnen und Entwickler pflegen oftmals ein *debugging log* – eine persönliche Sammlung von Fehlern und deren Lösung –, um ihre bisherigen Erfahrungen, gerade für komplexe oder aber häufig auftretende Fehler, festzuhalten und für zukünftige Probleme darauf zurückgreifen zu können. Vergleichbare Ansätze wurden bereits in der universitären Lehre bzw. Schule eingesetzt (vgl. Kapitel 2.5). Analog dazu sammeln die Lernenden über den Verlauf des Schuljahres/Programmierunterricht durchgängig die Fehler, mit denen sie zu kämpfen hatten, entweder individuell oder für alle sichtbar im Klassenzimmer. Gerade für den Umgang mit aus der Perspektive der Schülerinnen und Schüler oftmals kryptischen Fehlermeldungen von Kompilier- und Laufzeitfehlern können so entsprechende Heuristiken und Muster durch die aktive Reflexion der Lösung des Problems aufgebaut werden sowie als Nachschlagehilfe dienen (GK5).

7.3 Fazit

In diesem Kapitel wurden aufbauend auf der Untersuchung der *zugrunde liegenden Perspektiven* Kriterien identifiziert, die die Gestaltung von Konzepten und Materialien anleiten. Diese wurden weiterhin genutzt, um ein exemplarisches Konzept für die Sekundarstufe I zu entwickeln. In diesem werden die 10 Gestaltungskriterien wie folgt umgesetzt:

Das entwickelte Konzept ...

- GK1 ... sollte Selbstständigkeit fordern und fördern:** Das vermittelte systematische Vorgehen mitsamt der eingeführten Debuggingstrategien ist in Form des entwickelten Plakates jederzeit präsent im Klassenraum. Gleiches gilt für die *debugging logs*. Dies ermöglicht es der Lehrkraft, auf diese entsprechende Hilfestellung zu verweisen und eben erst dann weiterführende individuelle Hilfe anzubieten, wenn die Schülerin bzw. der Schüler die entsprechenden Ansätze erfolglos ausprobiert hat. Neben dem Einfordern von Selbstständigkeit wird diese aber auch gefördert: Ein konkretes Vorgehen, das explizite Formulieren von Hypothesen oder aber die Skill-Cards als Hilfestellung und Wiederholung für einzelne Strategien stellen entsprechende Unterstützung für eine eigenständige Fehlersuche und -behebung dar.
- GK2 ... sollte einen Fokus darauf legen, das Formulieren von Hypothesen einzuüben:** Für das Unterrichtskonzept wurde eine didaktisch-adaptierte Variante eines systematischen Vorgehens gemäß der *isolation*-Strategie entwickelt. Diese betont dabei ein hypothesengeleitetes Vorgehen, insbesondere durch die explizite Verschriftlichung der Hypothesen in der Übungsphase.
- GK3 ... sollte insbesondere die Bedürfnisse „schwächerer“ Schülerinnen und Schüler adressieren:** Durch das tendenziell kleinschrittige Vorgehen sowie die explizite Verschriftlichung von Hypothesen werden insbesondere die Bedürfnisse „schwächerer“ Schülerinnen und Schüler adressiert und diese zu mehr Selbstständigkeit – im Gegensatz zum vorherrschenden Trial-and-Error im Debugging, angeleitet.
- GK4 ... sollte Debuggingstrategien explizit vermitteln:** Die einzelnen Debuggingstrategien werden auf einem basalen Niveau und zunächst kleinschrittig in kurzen Unterrichtsphasen explizit eingeführt und eingeübt, um nicht vorhandene Vorerfahrungen entsprechender Vorgehensweisen aus dem Alltag angemessen zu adressieren.
- GK5 ... sollte den aktiven Aufbau von Erfahrung unterstützen:** Durch den Einsatz von *debugging logs* werden die Schülerinnen und Schüler einerseits zur Reflexion über ihre bisherigen Fehler angeleitet und können diese andererseits als Nachschlagewerk im Fehlerfall verwenden. Außerdem trägt die Verwendung von Debuggingaufgaben

mit „typischen“ Fehlern von Programmieranfängerinnen und -anfängern dazu bei, entsprechende Erfahrungen aufzubauen.

- GK6 ... sollte typische Hürden und Probleme von Lernenden beim Debugging adressieren:** Innerhalb des Unterrichtskonzepts werden verschiedene typische Probleme von Novizen aufgegriffen. So ist das *Undo* expliziter Bestandteil des entwickelten systematischen Vorgehens. Generell soll durch das systematische Vorgehen einem *Trial-and-Error* als üblichem Verhalten von Novizen vorgebeugt werden. Auch werden beispielsweise die zielführende Platzierung von *print*-Anweisungen sowie die Aussagekraft der entsprechenden Ausgaben und das Vergleichen des tatsächlichen und erwarteten Verhaltens betont sowie Heuristiken für den Umgang mit Fehlermeldungen bereitgestellt.
- GK7 ... sollte unterschiedliche Vorgehensweisen für verschiedene Fehlerarten vermitteln:** Innerhalb des didaktisch-adaptierten systematischen Vorgehens wird explizit nach verschiedenen Fehlerklassen unterschieden und jeweils ein angepasstes Vorgehen präsentiert. Darüber hinaus wird auch für die Debuggingstrategien und Werkzeuge klar benannt, für welche Art von Fehler sie hilfreich sein können.
- GK8 ... sollte der Vielfalt der persönlichen Debuggingvorlieben der Lehrkräfte Rechnung tragen:** Im Rahmen des Konzepts wird es den Lehrkräften ermöglicht, eigenständig die in ihren Augen bedeutsamen Strategien und Werkzeuge auszuwählen (wobei gerade letztere zusätzlich abhängig von den eingesetzten Programmierwerkzeugen der Lehrkraft sind) und angepasst an ihre Bedürfnisse zu vermitteln – beispielsweise basierend auf ihrem persönlichen Debuggingprozess.
- GK9 ... sollte variabel bei Bedarf im Unterricht einsetzbar sein:** Das systematische Vorgehen kann variabel ab dem Zeitpunkt eingeführt werden, an dem die Schülerinnen und Schüler regelmäßig mit Fehlern zu kämpfen haben und Kontakt zu verschiedenen Fehlerklassen relevant werden. Die einzelnen Werkzeuge und Strategien können darauf aufbauend bei Bedarf in einer kurzen Unterrichtsphase nahegebracht und direkt im Kontext der aktuellen Unterrichtsinhalte vermittelt werden. Beispielsweise könnte bei der Einführung von Arrays – wie von den Lehrkräften berichtet – der Debugger vorgestellt und anhand typischer Fehler, die die Schülerinnen und Schüler bereits gemacht haben, eingeübt werden. Dies erlaubt die direkte Anwendung des Gelernten und stellt gleichzeitig eine Integration in das bestehende Unterrichtskonzept mit einem angemessenem zeitlichem Umfang sicher.
- GK10 ... sollte tatsächliche (relevante) Debuggingfähigkeiten fördern:** Einerseits wird durch den Ansatz der prototypenbasierten Debuggingaufgaben der Einfluss fremden Codes reduziert. Andererseits stellt die gegebene Anzahl der Fehler, auf die die

Schülerinnen und Schüler direkt aufmerksam werden, sicher, dass nicht das Testen sondern tatsächlich Debuggingfähigkeiten gefördert werden.

Damit wurde ein integratives Konzept entwickelt, das den Informatik- bzw. Programmierunterricht durchgängig und bedarfsorientiert begleitet. Dieses entwickelte Konzept ist dabei weitgehend unabhängig von der Programmiersprache Java, die lediglich für die Ausgestaltung der konkreten Materialien gewählt wurde, und lässt sich analog auf andere textbasierte Programmiersprachen übertragen. Im Folgenden soll nun die Wirksamkeit der expliziten Vermittlung von Debuggingfähigkeiten anhand des Konzepts sowie dessen Umsetzung in der Schulpraxis untersucht werden.

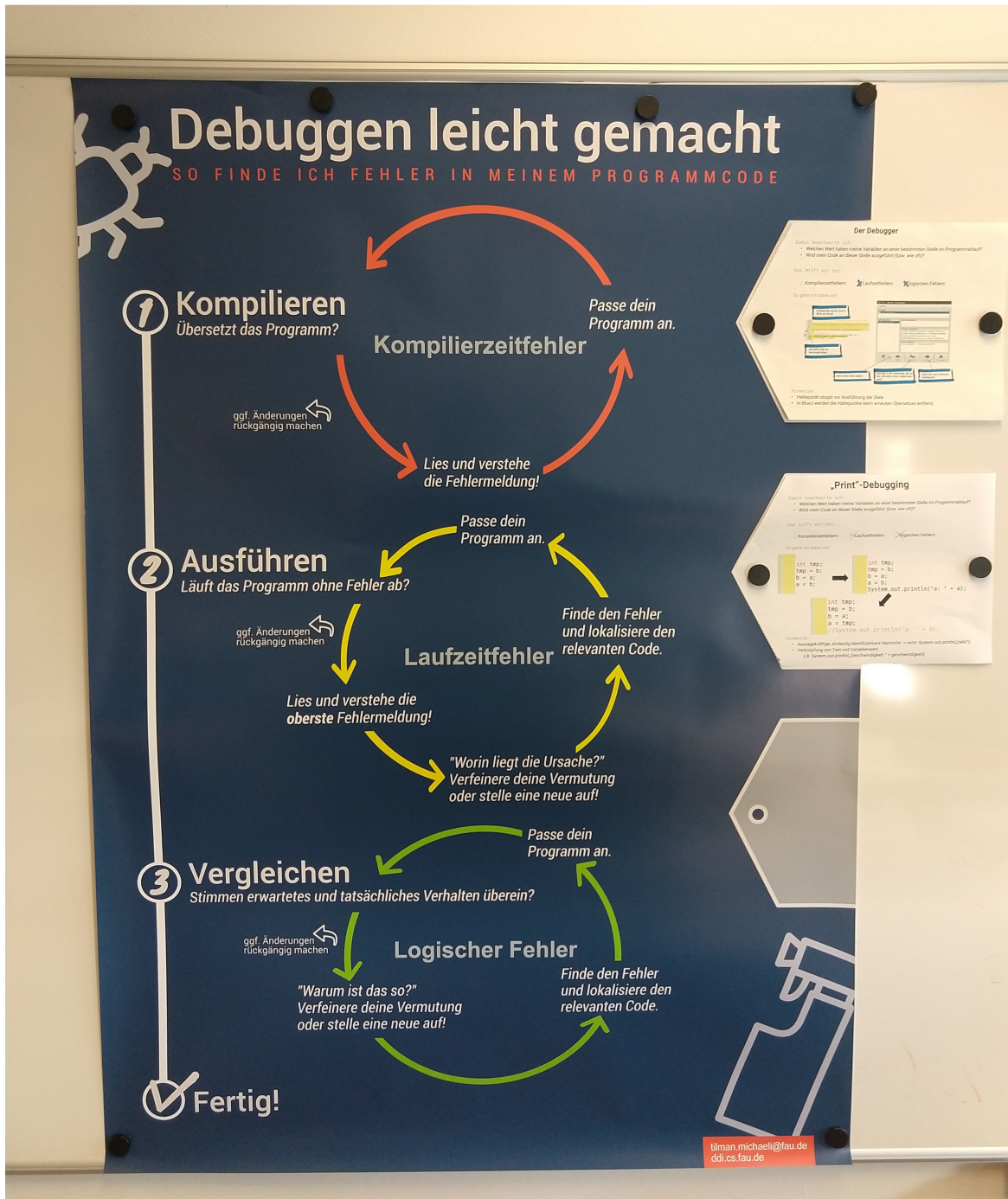


Abbildung 7.1: Plakat zum vermittelten Vorgehen und Skill-Cards

Der Debugger

Damit beantworte ich:

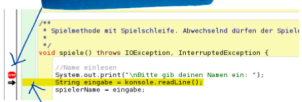
- Welchen Wert haben meine Variablen an einer bestimmten Stelle im Programmablauf?
- Wird mein Code an dieser Stelle ausgeführt (bzw. wie oft)?

Das hilft mir bei:

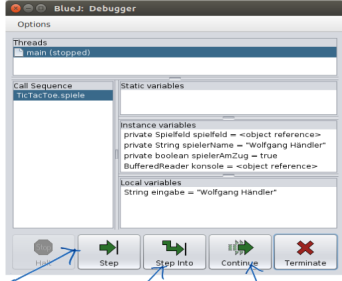
Kompilierzeitfehlern Laufzeitfehlern logischen Fehlern

So gehe ich dabei vor:

Haltepunkt setzen durch Klick am Rand



Aktuelle Zeile ist hervorgehoben



Gehe eine Zeile weiter

Springe in die Methode, die in der aktuellen Zeile aufgerufen wird

Gehe bis zum nächsten Haltepunkt

Hinweise:

- Haltepunkt stoppt vor Ausführung der Zeile, in der er gesetzt ist.
- In BlueJ werden die Haltepunkte beim erneuten Übersetzen entfernt.

Abbildung 7.2: Exemplarische ausgefüllte Skill-Card „Debugger“

Teil IV:

**Implementierung im
Informatikunterricht**

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

BRIAN KERNIGHAN

8 Der Effekt der expliziten Vermittlung von Debugging

Im Zuge der didaktischen Strukturierung wurden Gestaltungskriterien für Konzepte und Materialien für die Vermittlung von Debugging im Unterricht erarbeitet und darauf aufbauend ein konkretes Unterrichtskonzept entwickelt. Betrachtet man bisherige Ansätze zur Vermittlung von Debugging aus dem Forschungsstand (vgl. Kapitel 2.5), ist festzustellen, dass Lernenden vorwiegend die Möglichkeit zum (unsystematischen) Üben von Debugging gegeben wird, indem Debuggingaufgaben eingesetzt werden. Darüber hinaus existieren Ansätze, die explizit etwa ein systematisches Vorgehen (*Carver und Risinger, 1987*) oder auch konkrete Strategien (*Eranki und Moudgalya, 2016*) vermitteln, die aber lediglich eingeschränkt empirisch auf ihre Wirksamkeit überprüft wurden. Daher soll im Folgenden die Wirksamkeit der expliziten Vermittlung von Debugging anhand des entwickelten Unterrichtskonzepts empirisch untersucht werden.

8.1 Ziele der Untersuchung

Ziel dieser Untersuchung ist es, die Wirksamkeit der expliziten Vermittlung von Debugging im Unterricht unter den realen Bedingungen der Schulpraxis zu beforschen. Wie bereits diskutiert, stellt die Konfrontation mit Programmierfehlern eine große Hürde und Quelle für Frustration für Lernende dar (vgl. Kapitel 2.4). Daher stellt die Förderung der Selbstständigkeit der Schülerinnen und Schüler einen wichtigen Aspekt (vergleiche Gestaltungskriterium 1) dar. Nach *Pajares (1996)* determinieren Selbstwirksamkeitserwartungen (vgl. *Bandura (1982)*) die Anstrengung, Ausdauer und Belastbarkeit der Lernenden für eine Aufgabe und sind damit zentral für die Selbstständigkeit. Daher soll einerseits untersucht werden, welchen Einfluss die explizite Vermittlung auf die Selbstwirksamkeitserwartungen der Schülerinnen und Schüler hat. Darüber hinaus soll evaluiert werden, inwieweit sich ihre tatsächliche Debuggingleistung verändert:

(RQ6) Welchen Effekt hat die explizite Vermittlung von Debuggingfähigkeiten auf Selbstwirksamkeit und Debuggingleistung?

Für die empirische Untersuchung wurde dazu die *Vermittlung eines systematischen Vorgehens* gemäß dem in Kapitel 7 beschriebenen Unterrichtskonzept ausgewählt: Einerseits bildet ein systematisches Debuggingvorgehen die Grundlage für einen erfolgreichen Debuggingprozess (vgl. Kapitel 3). Andererseits eignet sich gerade der allgemeine Debuggingprozess, um (besser als beispielsweise für eine konkrete Debuggingstrategie oder ein Werkzeug wie

den Debugger) die Wirksamkeit der explizite Vermittlung mit dem unsystematischen Üben von Debugging zu vergleichen. Damit ergeben sich die folgenden zwei Teilfragen für diese Untersuchung:

- **(RQ6.1)** Hat die Vermittlung eines systematischen Vorgehens einen positiven Effekt auf die Selbstwirksamkeitserwartungen der Schülerinnen und Schüler?
- **(RQ6.2)** Hat die Vermittlung eines systematischen Vorgehens einen positiven Effekt auf die Debuggingleistung der Schülerinnen und Schüler?

8.2 Methodik

Um diese Forschungsfragen zu beantworten, wurde ein Pre-Post-Kontrollgruppen-Test-Design gewählt. Zunächst wurde die Intervention in einer 10. Klasse für besonders leistungsstarke Schülerinnen und Schüler ($n = 14$, Werkzeug Greenfoot und Programmiersprache Stride) pilotiert, um ausgehend von den gewonnenen Erkenntnissen der Durchführung Anpassungen vorzunehmen. Ergebnisse aus einer solchen Untersuchung ohne Kontrollgruppe helfen bei der Beantwortung der Forschungsfragen allerdings nur eingeschränkt, da mögliche Zuwächse in Selbstwirksamkeitserwartungen und Leistung der Schülerinnen und Schüler auch lediglich auf die zusätzliche Übung im Debuggen zurückzuführen sein könnten. Um den Einfluss der Intervention im Gegensatz zum reinen (unsystematischen) Üben von Debuggen, z.B. durch Debuggingaufgaben, zu untersuchen, wurden zwei 10. Klassen als Versuchs- ($n = 13$) und Kontrollgruppe ($n = 15$) herangezogen. Dabei wurden explizit zwei Klassen ausgewählt, die von derselben Lehrkraft mit dem identischen Unterrichtskonzept (unter Verwendung von BlueJ und Java) unterrichtet wurden und die im Curriculum zum Untersuchungszeitpunkt gleich weit fortgeschritten waren.

Ablauf

Die jeweils 90-minütige Durchführung – geleitet vom Autor – bestand aus einem Pretest, einer etwa 10-minütigen Intervention (in der Experimentalgruppe) und einem Posttest. Wie in Abbildung 8.1 dargestellt, bestanden Pre- und Posttests aus einem Fragebogen zur Erhebung der Selbstwirksamkeitserwartungen (siehe Anhang D) für das Debugging (vier Items mit einer fünfstufigen Likert-Skala) sowie Debugging-Aufgaben zur Beurteilung der Leistung der Schülerinnen und Schüler. Zusätzlich wurde nur im Posttest die Einschätzung der Lösbarkeit der Aufgaben erhoben. Zur Messung der Debuggingleistung wurde die Anzahl der korrigierten Fehler (analog zu *Fitzgerald et al. (2008)*) herangezogen. Dazu wurden

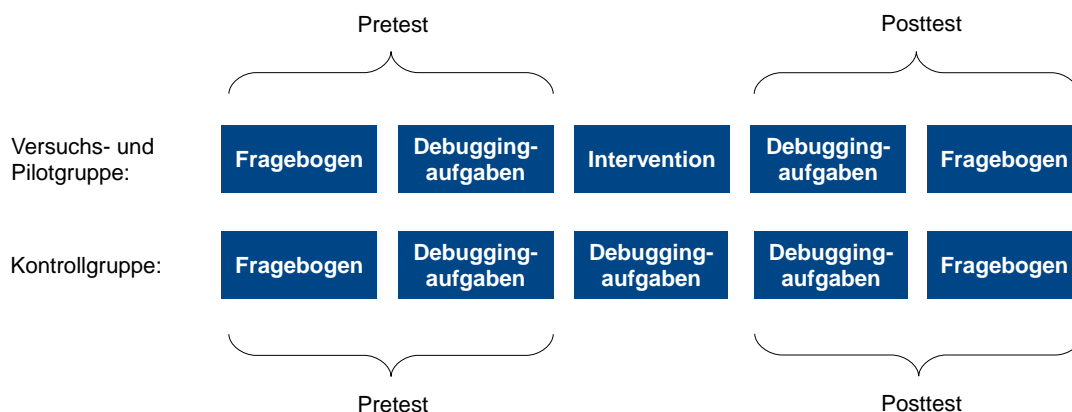


Abbildung 8.1: Untersuchungsdesign

sowohl die Arbeitsblätter, auf denen Fehler und deren Korrekturen von allen Gruppen notiert werden mussten, als auch der Code ausgewertet.

Intervention

Innerhalb der Intervention wurde das systematische Vorgehen zum Debuggen von Programmen, das im vorherigen Kapitel entwickelt und erläutert wurde, anhand eines Plakats (vgl. Anhang C bzw. Kapitel 7) vorgestellt, mit den Schülerinnen und Schülern diskutiert und exemplarisch durchgespielt. Das Plakat war für die restliche Unterrichtsstunde im Klassenzimmer präsent. Im Unterschied zum in Kapitel 7 beschriebenen Konzept kategorisierten die Schülerinnen und Schüler unterschiedliche Fehlertypen nicht selber und das Vorgehen wurde nicht in einer expliziten Phase eingeübt. Im Material für den Posttest wurden die Schülerinnen und Schüler der Experimentalgruppe stattdessen dazu aufgefordert, das zuvor eingeführte Vorgehen anzuwenden.

Messen von Debuggingleistung

Wie bereits festgestellt, unterscheidet sich das Vorgehen für das Debuggen eigener und fremder Programme und Programmverständnis nimmt für das Debuggen fremder Programme eine zentrale Rolle ein. Vor diesem Hintergrund stellt sich die methodische Frage, wie sich Debuggingleistung dediziert messen lässt und der Einfluss von Störvariablen wie

Codeverständnis minimiert werden kann. In Forschungsstand sind dafür verschiedene Ansätze üblich:

- **Traditionelle Debuggingaufgaben:** Von den Forscherinnen und Forschern wird fremder Code bereitgestellt und mit Fehlern versehen (vgl. z.B. *Nanja und Cook (1987)*).
- **Implementieren und dann Debuggen:** Die Probandinnen und Probanden implementieren zunächst selbst die Lösung für ein gegebenes Problem, um anschließend eine von den Forscherinnen und Forschern vorbereitete Version für dasselbe Problem zu debuggen, die mit Fehlern versehen wurde (vgl. z.B. *Fitzgerald et al. (2008)*).

Dabei wird bei der Verwendung traditioneller Debuggingaufgaben dem Unterschied zwischen dem Debuggen eigener und fremder Programme nicht Rechnung getragen. Wenn die Probandinnen und Probanden das Programm zunächst selbst implementieren, wird sich dem Debuggen eigener Programme angenähert, da sie mit Problemstellungen und zumindest einer möglichen Modellierung bzw. Struktur des Programms bereits vertraut sind. Allerdings erschien dieses methodische Vorgehen für den Kontext dieser Untersuchung ungeeignet, insbesondere da mit Pre- und Posttest ein entsprechend hoher zeitlicher Aufwand notwendig gewesen wäre. Stattdessen wurde das Konzept der prototypenbasierten Debuggingaufgaben verwendet, das bereits im vorherigen Kapitel für die Vermittlung von Debuggingfähigkeiten für eigene Programme beschrieben wurde.

Da Debugging- und nicht die Testfähigkeiten der Schülerinnen und Schüler untersucht werden sollten, wurde die Anzahl der Fehler pro Prototyp angegeben. Aus dem gleichen Grund wurde darauf geachtet, dass das Fehlverhalten des Programms schnell erkennbar war, sodass direkt mit der Fehlerlokalisierung begonnen werden konnte. Daher war es nicht notwendig, Randfälle zu überprüfen, um fehlerhaftes Programmverhalten beobachten zu können. Bei den verwendeten Fehlern handelte es sich sowohl um typische syntaktische (z.B. fehlende Klammern oder Datentypen), Laufzeit- (z.B. fehlende Initialisierung, die zu Laufzeitfehlern führt) als auch um typische logische Fehler (z.B. fehlende Methodenaufrufe oder vertauschte Bewegungsrichtungen).

8.3 Ergebnisse

8.3.1 Hat die Vermittlung eines systematischen Vorgehens einen positiven Effekt auf die Selbstwirksamkeitserwartungen der Schülerinnen und Schüler? (RQ6.1)

Zunächst wird der Zuwachs der Selbstwirksamkeitserwartungen, der sich als Mittelwert der vier Items ergibt, für Pilot-, Versuchs- und Kontrollgruppe pre und post untersucht.

Die Antworten der fünfstufigen Likert-Skala wurden auf die Skala 0 (stimme nicht zu) bis 4 (stimme zu) abgebildet. Die Mittelwerte bewegen sich folglich zwischen 0 und 4. Dazu wird ermittelt, ob ein signifikanter Anstieg der Selbstwirksamkeitserwartungen zwischen Pre- und Post-Test innerhalb der einzelnen Gruppen feststellbar ist. Aufgrund der Stichprobengrößen werden dafür stets nicht-parametrisierte Verfahren zur Prüfung auf Signifikanz verwendet (Rasch et al., 2010). Dementsprechend werden die Rangfolgen im Pre- und Posttest mithilfe des Wilcoxon-Vorzeichen-Rang-Tests – einem nicht-parametrischen Test für abhängige Stichproben – analysiert.

In Tabelle 8.1 sind die jeweiligen Mediane und der p-Wert des Wilcoxon-Vorzeichen-Rang-Tests (H_0 : Kein oder negativer Versuchseffekt) dargestellt ²³.

	Median pre	Median post	Wilcoxon-Test
Pilotgruppe	2,75	3,25	$p = 0,044^*$
Kontrollgruppe	2,25	2,50	$p = 0,083$
Versuchsgruppe	2,25	2,75	$p = 0,001^*$

Tabelle 8.1: Einfluss auf Selbstwirksamkeitserwartungen

In allen drei Gruppen ist also ein Anstieg der Selbstwirksamkeitserwartungen festzustellen. Dieser ist allerdings nur für die Pilot- und die Versuchsgruppe signifikant auf einem Signifikanzniveau von $\alpha = 0,05$. Die Effektstärken nach Cohen liegen bei $d = 0,56$ (Pilot) bzw. $d = 0,54$ (Versuch), dies entspricht einem mittleren Effekt (Cohen, 1988). Obgleich das aktive Üben von Debugging die Selbstwirksamkeitserwartungen verbessert, scheint ein systematisches Vorgehen die Selbstwirksamkeitserwartungen stärker positiv zu beeinflussen.

8.3.2 Hat die Vermittlung eines systematischen Vorgehens einen positiven Effekt auf die Debuggingleistung der Schülerinnen und Schüler? (RQ6.2)

Für Unterschiede in der Debuggingleistung werden die Versuchs- und die Kontrollgruppe in Pre- und Posttest verglichen. Die Debuggingleistung wird anhand der Anzahl der behobenen Fehler gemessen. Ein pre-post-Vergleich der Debuggingleistung innerhalb der einzelnen Gruppen analog zur Untersuchung der Selbstwirksamkeitserwartungen ist nicht zielführend, da in Pre- und Posttest unterschiedliche Fehler zu beheben waren.

Um festzustellen, ob sich die Leistung der Versuchsgruppe signifikant von der Leistung der Kontrollgruppe unterscheidet, wird geprüft, ob die beiden Stichproben derselben Grundgesamtheit entstammen. Nur wenn dies nicht der Fall ist, kann von einem signifikan-

²³Signifikante Testergebnisse zu einem Signifikanzniveau von $\alpha = 0,05$ sind durch ein * gekennzeichnet.

ten Unterschied ausgegangen werden. Auch findet ob der Stichprobengröße wieder ein nicht-parametrisierten Test, der Mann-Whitney-U-Test, Anwendung. Im Gegensatz zum Wilcoxon-Vorzeichen-Rang-Test ist dieser Test für unabhängige Stichproben ausgelegt. Die p-Werte des Mann-Whitney-U-Tests (H_0 : Stichproben kommen aus derselben Grundgesamtheit) sind in Tabelle 8.2 dargestellt.

	Mann-Whitney-U-Test
Versuch- vs. Kontrollgruppe Pre	$p = 0,191$
Versuch- vs. Kontrollgruppe Post	$p = 0,049^*$

Tabelle 8.2: Einfluss auf Debuggingleistung

Aufgrund der Ergebnisse des Mann-Whitney-U-Tests kann die Nullhypothese für den Vergleich der Pre-Tests auf einem Signifikanzniveau von $\alpha = 0,05$ nicht abgelehnt werden: Die Debuggingleistung der Schülerinnen und Schüler unterscheidet sich vor Durchführung der Intervention nicht signifikant. Im Gegensatz dazu zeigt sich ein signifikanter Unterschied im Posttest: Die Schülerinnen und Schüler der Versuchsgruppe weisen eine höhere Debuggingleistung (Median = 4, bei insgesamt 9 zu behebbenden Fehlern) auf als die der Kontrollgruppe (Median = 2). Im Posttest wurden für das Ermitteln der Debuggingleistung Aufgaben mit höherem Schwierigkeitsgrad herangezogen, da in beiden Gruppen ein Lerneffekt zwischen Pre- und Posttest anzunehmen ist. Die Effektstärke nach Cohen liegt bei $d = 0,69$ und entspricht einem mittleren Effekt (Cohen, 1988).

Die höhere Debuggingleistung spiegelt sich auch in der wahrgenommenen Schwierigkeit der Aufgaben durch die Schülerinnen und Schüler wider. Diese wurde ex post im Fragebogen mit Hilfe einer fünfstufigen Likert-Skala erhoben. Wiederum auf die Skala 0 (stimme nicht zu) bis 4 (stimme zu) abgebildet, ergeben sich folgende Mittelwerte:

	Aufgaben Pre	Aufgaben Post
Kontrollgruppe	3,07	1,47
Versuchsgruppe	3,23	2,92

Tabelle 8.3: Mittelwerte für *Aufgaben gut lösbar*

Die Ergebnisse lassen darauf schließen, dass ein systematisches Vorgehen den Unterschied machen kann: Wird Schülerinnen und Schülern ein solches systematisches Vorgehen an die Hand gegeben, so können diese ihren Erfolg beim Lokalisieren und Beheben von Fehlern signifikant verbessern.

8.4 Interpretation

Damit zeigt sich, dass die explizite Vermittlung von Debugging in Form eines systematischen Vorgehens einen positiven Effekt auf Selbstwirksamkeitserwartungen und Debuggingleistung hat. In Kapitel 2.5 wurden verschiedene Studien diskutiert (*Böttcher et al., 2016; Carver und Risinger, 1987; Allwood und Björhag, 1991*), die ebenfalls ein entsprechendes systematisches Vorgehen unterrichteten, teils mit vielversprechenden Ergebnissen. Im Gegensatz zu bisherigen Ansätzen wurde diese Intervention nun auch auf Wirksamkeit untersucht, und es stellt sich heraus, dass das theoretisch abgeleitete Potenzial sich auch empirisch bestätigt. Dabei bezieht das hier entwickelte systematische Vorgehen auch weitere Erkenntnisse über Novizen und deren Debuggingprozess mit ein. So werden, wie im letzten Kapitel anhand der Gestaltungskriterien dargelegt, etwa die Probleme beim Formulieren von Hypothesen oder aber das Einfügen von neuen Fehlern adressiert. Außerdem stellt die explizite Unterscheidung in verschiedene Fehlertypen ein hervorstechendes Merkmal dar, das sich aus dem Forschungsprozess als notwendig ergab.

Darüber hinaus sind damit auch Rückschlüsse über einige der Gestaltungskriterien möglich: Ein typisches Phänomen des Informatikunterrichts ist es, dass die Lehrkraft von Schülerin bzw. Schüler zu Schülerin bzw. Schüler eilt, Fehler erklärt und Hinweise zur Fehlerbehebung gibt. Zumal das Konzept der „erlernten Hilflosigkeit“ dabei eine Rolle spielen könnte, erscheint es zentral, die Selbstständigkeit von Schülerinnen und Schülern im Debugging zu steigern (Gestaltungskriterium 1). Der untersuchte Ansatz ist darauf ausgerichtet, das Selbstvertrauen der Lernenden im Debugging zu fördern, indem ihnen eine explizite Vorgehensweise zur Verfügung gestellt wird, wenn sie auf Fehler stoßen. Damit soll ebenfalls dem *Trial-and-Error-Ansatz* entgegengewirkt werden, der vor allem für schwächere Schülerinnen und Schüler üblich ist, indem ein geplanter und zielgerichteter Prozess gefördert wird. Die Ergebnisse deuten darauf hin, dass dieser Ansatz tatsächlich zu diesem Ziel beiträgt.

Bezüglich der Unterscheidung in verschiedene Fehlertypen (Gestaltungskriterium 7) war in der Durchführung festzustellen, dass sich die Schülerinnen und Schüler der Existenz verschiedener Klassen von Fehlern nicht bewusst waren, obwohl sie bereits regelmäßig mit ihnen konfrontiert worden waren. Erst die explizite Reflexion über die Fehler, auf die sie innerhalb der Intervention stießen, brachte die Lernenden dazu, die Unterschiede zu erkennen und zu erfassen, wie sie mit ihnen in Kontakt kamen, welche Informationen ihnen jeweils zur Verfügung standen und wie sie mit den verschiedenen Fehlern umgingen.

Das systematische Vorgehen wurde dabei sowohl mit Java und BlueJ (Experimental- und Kontrollgruppe) als auch mit Stride und Greenfoot (Pilotprojekt) getestet. Die positiven Ergebnisse in beiden Fällen deuten darauf hin, dass dieser Ansatz unabhängig von Werkzeugen und (textbasierten) Programmiersprachen ist (im Gegensatz zu beispielsweise *Carver*

und Risinger (1987)). Gerade vor dem Hintergrund der im Unterricht üblichen Heterogenität bei den verwendeten Werkzeugen und Sprachen sowie der Verallgemeinerung der Ergebnisse erscheint dies zentral.

Güte und Limitationen

Nach Bortz und Döring (2006) müssen für die Güte quantitativer Studien insbesondere die externe und interne Validität des Forschungsdesigns, die statistische Validität in der Auswertung sowie die Konstruktvalidität des Erhebungsinstrument gewährleistet sein. Eine mögliche Limitation der **externen Validität** und damit der Generalisierbarkeit der Untersuchung ist die geringe Stichprobengröße und die fehlende Randomisierung der Schülerinnen und Schüler. Diese sind allerdings zumindest teilweise bedingt durch das gewählte Setting unter den realen Bedingungen der Schulpraxis, welches wiederum im Gegensatz zu einem Laborsetting zu einer höheren externen Validität beiträgt (Bortz und Döring, 2006).

Für die **interne Validität** dieser Untersuchung bedeutet dies allerdings auch, dass daher keine Daten über den tatsächlichen Debuggingprozess der Schülerinnen und Schüler (z.B. in Form von Beobachtungen, Eye-Tracking oder aber Screen-Recordings) und mögliche Veränderungen nach der Intervention erhoben werden konnten. Allerdings zeigen die unstrukturierten Beobachtungen aus den Klassenzimmern, dass die Lernenden tatsächlich erfolgreich einen systematischen Ansatz angewandt haben, der letztendlich zu einer Steigerung der Selbstwirksamkeit und einer höheren Debuggingleistung führte. Darüber hinaus wurden zur Sicherstellung der internen Validität weitere Variablen kontrolliert, indem zwei Schulklassen gewählt wurden, die von der gleichen Lehrkraft nach dem gleichen Konzept unterrichtet wurden und die zum Erhebungszeitpunkt gleich weit fortgeschritten waren. Weiterhin wurde im Fragebogen ebenfalls die Motivation der Schülerinnen und Schüler erfasst, hier zeigten sich aber keine signifikanten Einflüsse auf die unabhängigen Variablen. Auch wurde die *time on task* sowie Merkmale der Lehrerpersönlichkeit kontrolliert, indem der Unterricht durch den Autor durchgeführt wurde. So konnte auch die Hilfestellung für die Schülerinnen und Schüler in der Bearbeitung der Debuggingaufgaben, die essentiell für die Vergleichbarkeit der Gruppen ist, für Pre- und Posttest konstant gehalten werden. Eine Vergrößerung der Stichprobe zugunsten der externen Validität würde damit, insbesondere ob der schulpraktischen Gegebenheiten, zu einer geringeren internen Validität führen – ein typisches Phänomen für entsprechende Forschungsdesigns (Ko und Fincher, 2019).

Darüber hinaus ist die **statistische Validität** durch das a-priori festgelegte Signifikanzniveau sowie die Berechnung adäquater Effektgrößen (Cohens d) sichergestellt.

Bezüglich der **Konstruktvalidität** des Erhebungsinstrumentes muss zwischen dem Fragebogen und den Debuggingaufgaben unterschieden werden. Nachdem kein existierendes Instrument für die Messung von Selbstwirksamkeitserwartungen für das Debugging genutzt werden konnte, wurden die Items in Anlehnung an *Danielsiek, Toma und Vahrenhold (2018)* entwickelt. Mit Hilfe einer konfirmatorischen Faktorenanalyse wurde die Eindimensionalität überprüft und damit die *faktorielle Validität* der Items sichergestellt (*Krebs und Menold, 2014*). Die zusätzlich erhobene Motivation, für die sich keine Signifikanz für die Untersuchung zeigte, wurde mit Hilfe einer verkürzten Fassung des „Intrinsic Motivation Inventory“²⁴ (basierend auf *Deci und Ryan (2012)*) erhoben. Auch für die Messung von Debuggingleistung fehlt es bisher an einem etablierten Instrument. Wie bereits beschrieben, ist für traditionelle Debuggingaufgaben dabei lediglich eine eingeschränkte Konstruktvalidität zu erwarten. Daher wurden prototypenbasierte Debuggingaufgaben verwendet, um den Unterschieden im Vorgehen und dem Einfluss von Codeverständnis Rechnung zu tragen. Außerdem wurden gezielt typische Fehler von Programmieranfängerinnen und -anfängern herangezogen und darauf geachtet, dass tatsächlich Debugging- und nicht etwa Testfähigkeiten gemessen werden. Eine explizite – üblicherweise in mehreren Iterationen durchgeführte – Überprüfung der Konstruktvalidität der prototypenbasierten Aufgaben wurde darüber hinaus nicht vorgenommen und stellt eine Aufgabe für zukünftige informatikdidaktische Forschung dar.

8.5 Fazit

Zusammenfassend sind damit, trotz der kleinen Stichprobe und fehlenden Randomisierung, Indizien für die Wirksamkeit des entwickelten Unterrichtskonzepts erbracht worden. Zentrale Ergebnisse der Untersuchung sind:

- Die Vermittlung eines systematischen Vorgehens zum Finden und Beheben von Programmierfehlern hat einen positiven Einfluss auf die Debugging-Selbstwirksamkeitserwartungen.
- Schülerinnen und Schüler, die ein systematisches Vorgehen vermittelt bekommen haben, zeigen zudem höhere Leistungen im Debuggen als Schülerinnen und Schüler, die Debuggen ausschließlich (unsystematisch) geübt haben.

Diese Ergebnisse bestätigen damit einerseits die Bedeutung eines solchen systematischen Vorgehens als Grundlage für erfolgreiches Debugging (und damit ein zentrales Herausstellungsmerkmal des Modells der Debuggingfähigkeiten für Novizen) und andererseits die entwickelten Gestaltungskriterien, nach denen das Konzept ausgerichtet wurde. Zen-

²⁴<https://selfdeterminationtheory.org/>

trale Konsequenz dieser Ergebnisse ist damit eine empirische Verdichtung der These, dass Debugging explizit vermittelt werden sollte, anstatt – wie vorwiegend üblich – lediglich Gelegenheiten zum unsystematischen Üben zu geben.

9 Gestaltung und Evaluation einer Fortbildung zum Transfer in die Unterrichtspraxis

Zentrales Ziel dieser Arbeit ist es, Konzepte und Materialien zu entwickeln, um das Schlüsselproblem, das Debugging für den Informatikunterricht darstellt, angemessen zu adressieren. Dazu wurden auf Basis der Untersuchung der *zugrunde liegenden Perspektiven* Gestaltungskriterien abgeleitet und gemäß dieser Kriterien ein konkretes Unterrichtskonzept entwickelt und ausschnittsweise auf Wirksamkeit untersucht. Um nun zu einer tatsächlichen Weiterentwicklung und Professionalisierung des Informatikunterrichts und damit zur Praxiswirksamkeit beizutragen, ist es notwendig, die Implementierung und den Transfer der gewonnenen Erkenntnisse und Ergebnisse in die Unterrichtspraxis in den Blick zu nehmen (Gräsel und Parchmann, 2004). Eine traditionelle Möglichkeit für den Transfer fachdidaktischer Innovationen stellt die Fortbildung von Lehrkräften dar. Daher wird im Folgenden in einem ersten Schritt ein Fortbildungsformat für den Transfer der Forschungsergebnisse dieser Arbeit entwickelt und durchgeführt.

Ein solcher Transfer von Forschungsergebnissen geht dabei über eine reine Dissemination hinaus (Prenzel et al., 2010): Eine Veränderung der Schulpraxis ist keine geradlinige Übernahme zur Verfügung gestellter Materialien (Fincher, Kolikant und Falkner, 2019). Stattdessen werden fachdidaktische Innovationen abhängig von Kontext, Erfahrungen und Vorstellungen der Lehrkräfte individuell für den eigenen Unterricht angepasst (House, 1974). Daher soll zur Evaluation der intendierten Transferwirkung untersucht werden, welchen Einfluss die Fortbildung auf die Unterrichtspraxis und den Unterricht bezüglich des Debuggens tatsächlich hat. Dies erlaubt – neben der Untersuchung des Transfererfolgs – die Beforschung der konkreten Umsetzung durch die Lehrkräfte. So ermöglichen die Adaptionen, die sie an den Konzepten und Materialien vornehmen, aber auch die Erfahrungen, die sie dabei machen, die Weiterentwicklung des Unterrichtskonzepts. Eine derartige enge Theorie-Praxis-Verschränkung kann damit zu einer nachhaltigen Innovation und konkreten Unterrichtsweiterentwicklung beitragen.

9.1 Fortbildung, professionelle Kompetenz und Professionswissen

Die Fort- und auch Weiterbildung von Lehrkräften ist ein zentraler Gegenstand (informatik-) didaktischer Forschung. Im Folgenden sollen Faktoren dargelegt werden, die zentral für

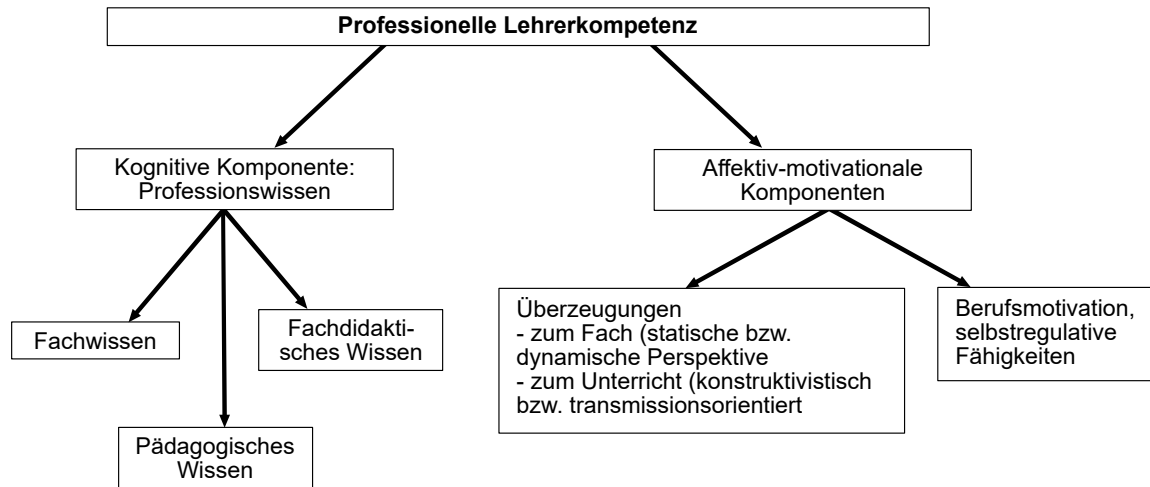


Abbildung 9.1: Modell professioneller Kompetenz von Lehrkräften nach Blömeke, Suhl und Döhrmann (2012)

die Ausgestaltung solcher Angebote sind, um darauf aufbauend anschließend die Entwicklung und Durchführung eines Fortbildungsangebotes für das Debuggen im Unterricht zu beschreiben.

Ziel der Fortbildung von Lehrkräften ist es, die Qualität des Unterrichts und damit der Lernprozesse der Lernenden zu steigern (Richardson und Placier, 2001). Dafür wird im Rahmen von Fortbildungen (bzw. allgemein der Lehrkräftebildung) angestrebt, die *professionelle Kompetenz* der Lehrkräfte weiterzuentwickeln, die bestimmt, „wie gut eine Lehrkraft die Anforderungen ihres Berufs bewältigen kann“ (Kunter et al., 2011). Professionelle Kompetenz wird dabei als erwerbbarer Disposition angesehen, die sich als Zusammenspiel aus Professionswissen und affektiv-motivationalen Komponenten, wie etwa subjektiven Überzeugungen, ergibt (Baumert und Kunter, 2006). Blömeke, Suhl und Döhrmann (2012) konzeptualisieren dies, wie in Abbildung 9.1 dargestellt.

Professionswissen beschreibt dabei das Wissen, „über das die Angehörigen einer bestimmten Profession verfügen (hier des Lehrerberufs) und das für diese Profession charakteristisch ist“ (Harms und Riese, 2018). Es existieren verschiedenste Ansätze, um das Professionswissen von Lehrkräften zu konzeptualisieren (z.B. Anderson und Page (1995), Magnusson, Krajcik und Borko (1999), oder Tamir (1988))²⁵. Zusammenfassend werden in der Forschung dabei drei Dimensionen des Professionswissens – basierend auf der Kategorisierung nach Shulman (1987) – als zentral erachtet (Borowski et al. (2010), vgl. auch Abbildung 9.1):

²⁵Für eine ausführliche Diskussion unterschiedlicher Ansätze siehe Niermann (2016).

- **Content knowledge (CK):** Fachwissen der Lehrkräfte, also vertieftes Hintergrundwissen bezüglich der curricularen Inhalte.
- **General pedagogic knowledge (PK):** Allgemein-didaktisches und damit fachunabhängiges pädagogisches Wissen, beispielsweise über effektive Klassenführung, Lernprozesse oder Diagnostik.
- **Pedagogical content knowledge (PCK):** Fachdidaktisches Wissen, wie etwa über Lernvoraussetzungen der Lernenden oder fachspezifische Unterrichtsmethoden.

Im Rahmen einer fachbezogenen Fortbildung stellt nun insbesondere die Veränderung des Professionswissens in Form von Fachwissen und fachdidaktischem Wissen die Zielgröße dar. Um eine solche Veränderung des Professionswissens als Teil der professionellen Kompetenz zu erzielen, muss – gemäß der Lerntheorie des Konstruktivismus – zunächst ermittelt werden, welche Basis die Lehrkräfte aus ihrer bisherigen Ausbildung und Unterrichtspraxis mitbringen. Diese Aufgabe für die Ausgestaltung der Lehrkräftebildung fasst das Modell der didaktischen Rekonstruktion für die Lehrkräftebildung nach *van Dijk und Kattmann (2007)* explizit (vgl. Abbildung 9.2). Dabei handelt es sich um eine Adaption des Modells der originalen didaktischen Rekonstruktion (vgl. Kapitel 1.3.1 bzw. Abbildung 1.2). Die Stelle der *fachlichen Klärung* bzw. des „Stoffs“ im originalen Modell nehmen nun die entwickelten Unterrichtskonzepte und Materialien ein. Weiterhin wird anstatt der *Perspektiven der Lernenden* nun das Professionswissen von Lehrkräften als Lernvoraussetzung aufgefasst und untersucht. Aus dem Zusammenspiel dieser beiden Perspektiven ergibt sich die Grundlage für die Ausgestaltung von entsprechenden Angeboten zur Lehrkräftebildung. Auch spezifische Modelle zur Fortbildung von Lehrkräften wie beispielsweise das Angebots-Nutzungsmodell der Lehrkräftefortbildung (*Lipowsky und Rzejak, 2017*) oder der DZLM-Kompetenzrahmen (*Barzel und Selter, 2015*) betonen diese konstruktivistische Perspektive – aufbauend auf der *professionellen Kompetenz der Lehrkräfte* –, die daher leitend für die Entwicklung der Fortbildung war.

9.2 Entwicklung einer Fortbildung für das Debugging im Unterricht

Ziel der Fortbildung ist es, die Unterrichtspraxis der Lehrkräfte und damit auch die Debuggingfähigkeiten der Schülerinnen und Schüler zu beeinflussen, indem die im Rahmen dieser Arbeit entwickelte fachdidaktische Innovation in Form der Gestaltungskriterien sowie des konkreten Unterrichtskonzepts und der entsprechenden Materialien für das Debugging im Unterricht den Lehrkräften vermittelt werden. Gemäß dem Modell der didaktischen Rekonstruktion für die Lehrkräftebildung nach *van Dijk und Kattmann (2007)* ergibt sich die

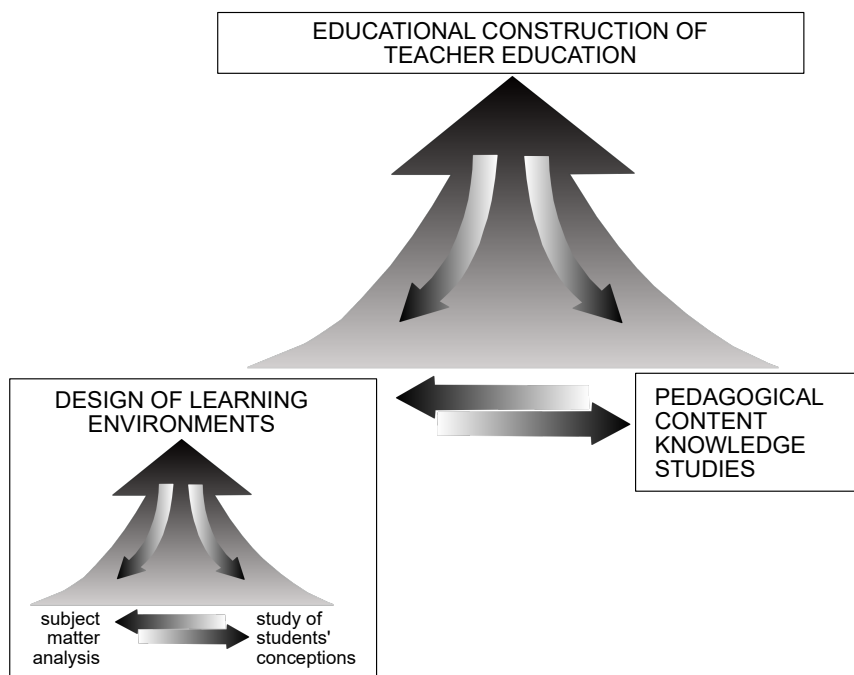


Abbildung 9.2: Didaktische Rekonstruktion für die Lehrkräftebildung nach *van Dijk und Kattmann (2007)*

Grundlage der Ausgestaltung von Angeboten zur Lehrkräftebildung aus dem Zusammenspiel von fachdidaktischer Innovation sowie dem Professionswissen der Lehrkräfte. Das im Rahmen der Fortbildung vermittelte Konzept und die entsprechenden Materialien sind dabei bereits in Kapitel 7 beschrieben worden. Bezüglich der professionellen Kompetenz der Lehrkräfte ermöglicht die vorgenommene Untersuchung der *Perspektive der Lehrkräfte* (vgl. Kapitel 4) Einsichten, die im Folgenden zusammengefasst werden, um darauf aufbauend Hinweise für die Gestaltung der Fortbildung abzuleiten und anschließend die Konzeption und Durchführung zu beschreiben.

9.2.1 Professionelle Kompetenz und Professionswissen der Lehrkräfte für das Debugging

Aus der in Kapitel 4 dargestellten Untersuchung der Perspektive der Lehrkräfte lassen sich Rückschlüsse auf das Professionswissen der Lehrkräfte in Form von *content* und *pedagogical content knowledge*, aber auch auf die Einstellungen zu Debugging als Unterrichtsgegenstand als affektiv-motivationaler Bestandteil der professionellen Kompetenz ziehen.

Content knowledge: Bezüglich des Fachwissens der Lehrkräfte zeigte sich in der Untersuchung, dass die Lehrkräfte Debugging selbst nur selten explizit gelernt hatten – analog zu professionellen Entwicklerinnen und Entwicklern (Perscheid et al., 2017) – und ihnen daher auch entsprechende Konzepte und Inhalte für die Vermittlung im Unterricht fehlten. In Konsequenz müssen – um das Ziel der Fortbildung zu erreichen – auch fachliche Kompetenzen vermittelt werden, obwohl die Innovation des Unterrichtskonzepts eigentlich nicht in der Fachlichkeit, sondern in der fachdidaktischen Vermittlung von Debuggingfähigkeiten liegt. Es offenbarten sich ebenso Unterschiede in den persönlichen Debuggingvorgehensweisen der Lehrkräfte: Während einige der Lehrkräfte selbst vorwiegend mit Hilfe von (basalen) Strategien wie dem *print*-Debugging arbeiteten und kaum Erfahrungen mit Werkzeugen wie etwa dem *Debugger* hatten, vermittelten andere solche „fortgeschrittenen“ Methoden auch ihren Schülerinnen und Schülern. Damit ergibt sich eine gewisse Heterogenität innerhalb der Lehrkräfte, die im Rahmen der Fortbildung adressiert werden muss.

Pedagogical content knowledge: Bezüglich des fachdidaktischen Wissens zeigte sich, dass die Lehrkräfte kaum explizite Einheiten zur Vermittlung von Debuggingfähigkeiten in ihrem Unterricht einsetzten. Vorwiegend wurde versucht, den Schülerinnen und Schülern in der individuellen Betreuung Hilfestellungen zu geben. Als einzige Methoden fanden traditionelle Debuggingaufgaben Verwendung, die den Lernenden lediglich die Möglichkeiten zum unsystematischen Üben geben. Auch berichteten die meisten Lehrkräfte davon, gemäß der „Turnschuhdidaktik“ im Klassenzimmer von einem PC zum anderen zu eilen, um den Schülerinnen und Schülern zu helfen. Damit ist zu erwarten, dass Debuggen zu unterrichten für die Mehrheit der Teilnehmenden neu ist. Andere Lehrkräfte schilderten wiederum Best Practices, die sie erfolgreich im Unterricht einsetzten. Neben der Einführung des entwickelten Unterrichtskonzepts erscheint es daher zentral, den Austausch zwischen den teilnehmenden Lehrkräften zu bestehenden Best Practices und Vorgehensweisen zu fördern und zu einer Vernetzung der Lehrkräfte beizutragen.

Einstellungen: Im Rahmen der Untersuchung wurde außerdem analysiert, aus welchen Gründen die Lehrkräfte Debugging (nicht) unterrichten. Dabei zeigte sich, dass sie insbesondere aufgrund der wahrgenommenen Hilflosigkeit der Schülerinnen und Schüler ein Interesse daran haben, entsprechende Inhalte in ihren Unterricht aufzunehmen. Allerdings sorgen der Mangel an Unterrichtszeit und geeigneten Konzepten und Materialien dafür, dass sie Debugging in ihrem Unterricht aussparen. Darüber hinaus betonten die Lehrkräfte, dass Debugging oftmals keinen expliziten Inhalt des Lehrplans darstellt, und sie es daher häufig zugunsten von Inhalten „vernachlässigen“, die ausdrücklich gefordert werden. Daher sollte für eine tatsächliche Praxiswirksamkeit im Rahmen der Fortbildung die

Bedeutung des Themas und die Vorteile für die eigene Unterrichtspraxis durch eine gestiegene Selbstständigkeit der Schülerinnen und Schüler herausgearbeitet werden – auch wenn bei der Teilnahme an einem entsprechendem Fortbildungsformat von einem prinzipiellen Interesse am Thema ausgegangen werden kann.

9.2.2 Konzeption

Aus der Analyse der zu erwartenden professionellen Kompetenz bezüglich des Debuggings, existierenden Erfahrungen aus bisher durchgeführten Fortbildungen sowie Forschungsergebnissen zur didaktischen Ausgestaltung von Fortbildungen wurden folgende Prinzipien für die Fortbildung abgeleitet.

Gemeinsames Problembewusstsein schaffen. Für eine tatsächliche Veränderung der Schulpraxis ist es zentral, gemeinsam mit den Lehrkräften und ausgehend von ihrer Unterrichtspraxis zu arbeiten, anstatt lediglich „Top-Down“ Materialien zur Verfügung zu stellen (*Henderson, Beach und Finkelstein, 2011*). Wesentlich dafür ist es zunächst, an die Erfahrungen der Lehrkräfte aus ihrem eigenen Unterricht anzuschließen. Durch die Reflexion des eigenen Unterrichts soll ein gemeinsames Problembewusstsein geschaffen werden, sodass die Bedeutung und das Potential des Themas Debugging verdeutlicht wird.

Konkrete Konzepte und Materialien für den eigenen Unterricht. Für die Umsetzung und Erprobung der Inhalte der Fortbildung in der Unterrichtspraxis ist es hilfreich, möglichst konkrete Materialien bereitzustellen, die mehr oder weniger direkt eingesetzt werden können. Dies ermöglicht den Lehrkräften in einer ersten Phase ohne größeren Aufwand den direkten Einsatz und reduziert damit Hürden des Transfers (*Farmer, Gerretson und Lassak, 2003*). Allerdings passen Lehrkräfte diese Konzepte und Materialien – wie eingangs diskutiert – typischerweise spätestens in einer zweiten Iteration an ihre konkreten Bedürfnisse an. Daher sollen die Lehrkräfte aktiv zum Experimentieren und Adaptieren der Materialien als Ausgangspunkt unterrichtlichen Handelns ermuntert werden.

Fachwissen und fachdidaktisches Wissen vermitteln. In der Betrachtung des Professionswissens von Lehrkräften zum Thema Debugging zeigt sich, dass, um das Ziel der Weiterentwicklung der Unterrichtspraxis zu erreichen, neben fachdidaktischen auch entsprechende fachliche Kompetenzen den Lehrkräften vermittelt werden müssen. Obwohl die in der Fortbildung eingeführte Innovation nicht auf einer fachlichen Ebene liegt, ergibt sich hier ein entsprechender Handlungsbedarf. Als besonders geeignet erscheint das Konzept des *didaktischen Doppeldeckers* (*Wahl, 2013*), das es ermöglicht, dass die Lehrkräfte durch die

Bearbeitung der Materialien für die Schülerinnen und Schüler gleichzeitig entsprechendes Fachwissen erwerben und Möglichkeiten zur Vermittlung erfahren.

Selbstbestimmtes und aktives Lernen. Erfolgsfaktoren für Fortbildungen sind insbesondere ein Wechselspiel zwischen theoretischer Vermittlung und praktischen Erprobungen sowie selbstbestimmtes und eigenverantwortliches Lernen (*Darling-Hammond, Hyler und Gardner, 2017; Lipowsky, 2004*). Gerade aufgrund der Heterogenität hinsichtlich des Fachwissens der Lehrkräfte sollte es ihnen daher ermöglicht werden, passend zu ihren individuellen Lernvoraussetzungen beispielsweise neue Debuggingstrategien selbstbestimmt auszuprobieren.

Vernetzung und Austausch. Kollaboration, der Austausch von Ideen und die Vernetzung mit und zwischen den teilnehmenden Lehrkräften, ist essentiell für einen nachhaltigen Transfer der Fortbildungsinhalte in die Unterrichtspraxis (*Darling-Hammond, Hyler und Gardner, 2017*). Daher sollen entsprechende Möglichkeiten aktiv unterstützt werden.

9.2.3 Durchführung

Basierend auf diesen Prinzipien wurde ein Wochenendworkshop (Freitagabend bis Sonntagmittag) entworfen und im November 2019 mit 16 Teilnehmerinnen und Teilnehmern aus verschiedenen Bundesländern durchgeführt. Die Veranstaltung fand dabei in einem Konferenzhotel statt, sodass die Teilnehmenden dort übernachteten und entsprechend verpflegt wurden. Die drei Tage hatten jeweils unterschiedliche Zielsetzungen:

- **Tag 1:** Problemaufriss und Reflexion des eigenen Unterrichts
- **Tag 2:** Vermittlung und Exploration von fachlichen und fachdidaktischen Inhalten
- **Tag 3:** Umsetzung für den eigenen Unterricht

Vor Beginn des Workshops wurden die Lehrkräfte zudem aufgefordert, ihren persönlichen Debuggingprozess und wie sie selbst Debuggen gelernt haben in Form von „Debuggingbiographien“ (vgl. z.B. *Knobelsdorf und Schulte (2005)*) zu reflektieren. Die Auswertung dieser Biographen offenbarte, dass tatsächlich die Mehrheit der Lehrkräfte nie systematisch Debuggen gelernt hat, sondern sich die entsprechenden Fähigkeiten zumeist selbst angeeignet hatte. Auch bestätigte sich die erwartete Heterogenität bezüglich des Umfangs der verwendeten Debuggingstrategien und -werkzeuge und auch der allgemeinen Programmiererfahrung außerhalb des Unterrichts.

Tag 1

Zu Beginn des Workshops wurde am ersten Abend Debuggen als Schlüsselproblem der Unterrichtspraxis aus der Perspektive der Lehrkräfte charakterisiert: Dazu reflektierten Lehrkräfte in Gruppen zunächst mit Hilfe der **Persona-Methode**²⁶ aus dem Design Thinking (*Uebersnickel und Brenner, 2016*), wie „typische“ Schülerinnen bzw. Schüler in ihrem Unterricht beim Debuggen vorgehen und welche Probleme sie haben (vgl. Abbildung 9.3 bzw. Anhang E). Der Austausch mit anderen Lehrkräften und das gemeinsame explizite Festhalten der Probleme der Schülerinnen und Schüler diente dazu, sich in die Perspektive der Lernenden hineinzusetzen und deren typische Probleme zu reflektieren (ein zentraler Bestandteil des PCK). Mithilfe der Persona-Methode wurde ein gemeinsames Problembewusstsein geschaffen und für den restlichen Workshop sowie den Transfer in die eigene Unterrichtspraxis die Zielgruppe festgehalten und charakterisiert.

In einem zweiten Schritt tauschten die Lehrkräftegruppen die Personas untereinander aus und beschrieben jeweils, wie sie die jeweilige Schülerin bzw. den Schüler in ihrem derzeitigen Unterricht unterstützen würden. Auf diese Art und Weise sollten die Lehrkräfte einerseits bestehende Best Practices austauschen und andererseits reflektieren, wo sie Bedarf zur Weiterentwicklung ihres Unterrichts sehen. So konnte ein gemeinsames Problembewusstsein geschaffen werden.

Tag 2

Nach diesem gemeinsamen Problemaufriss begann der zweite Tag der Fortbildung mit einem Vortrag mit anschließender Diskussion, der einen **Blick in die professionelle Welt der Softwareentwicklung** ermöglichte. Dazu berichtete ein professioneller Softwareentwickler aus seiner Berufspraxis, wie Debugging dort abläuft, welchen Stellenwert es einnimmt und wie Entwicklerinnen und Entwickler debuggen „lernen“. Damit sollte den Lehrkräften entsprechendes Hintergrundwissen über Debugging vermittelt werden (CK).

In der nächsten Phase wurden gemeinsam mit den Teilnehmenden die fachdidaktischen Konzepte erarbeitet. Dazu wurden einerseits der Prozess des Debugging aufbereitet und geklärt (wie beispielsweise vom Testen abgegrenzt) und entsprechende Debuggingfähigkeiten systematisiert (vgl. Kapitel 3, CK) und andererseits Möglichkeiten zur Vermittlung der jeweiligen benötigten Fähigkeiten aufgezeigt (vgl. Kapitel 7, PCK).

²⁶Personas sind im Design Thinking Stereotype bestimmter Kunden- oder Nutzergruppen, die es ermöglichen, sich in die Perspektive der jeweiligen Gruppe hineinzusetzen und für deren spezifische Anforderungen Angebote zu entwickeln oder aber zu überprüfen, ob die Bedürfnisse der verschiedenen Gruppen adäquat abgedeckt werden, vgl. *Uebersnickel und Brenner (2016)*.

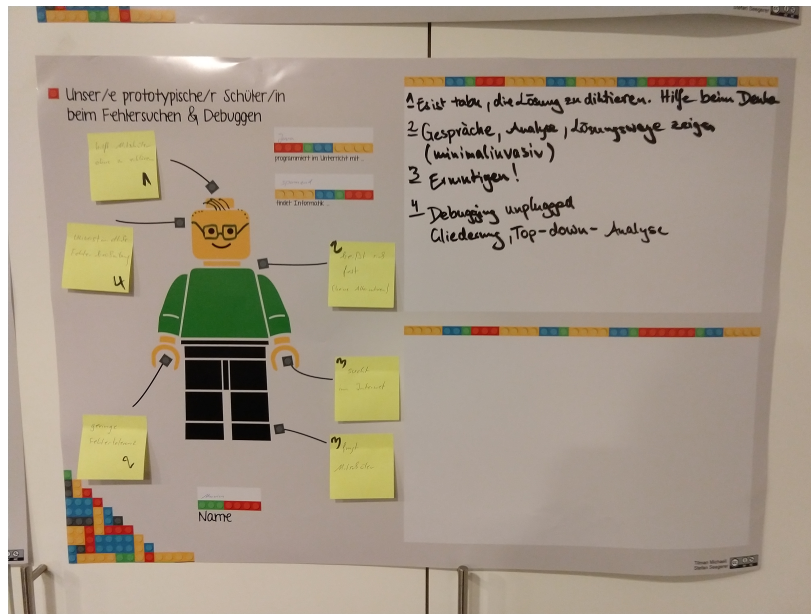


Abbildung 9.3: Persona aus dem Workshop

Anschließend **explorierten** die Lehrkräfte die entwickelten Unterrichtsmaterialien **selbstgesteuert**. Gemäß dem Prinzip des *didaktischen Doppeldeckers* erarbeiteten sich die Lehrkräfte dabei zum einen entsprechende fachliche Kompetenzen (CK), indem sie beispielsweise die Materialien zur Einführung von Debuggingstrategien bearbeiteten, mit denen sie bisher kaum oder keine Erfahrung hatten (wie etwa dem Debugger). Zum anderen probierten sie damit direkt Methoden zur Vermittlung diesbezüglicher Fähigkeiten aus und erwarben damit geeignete fachdidaktische Kompetenzen (PCK). Alle in dieser Phase entstandenen Ideen, Anregungen, Fragen und Anmerkungen wurden gesammelt und anschließend im Plenum diskutiert.

Tag 3

Am letzten Tag der Fortbildung wurde den Lehrkräften die Möglichkeit gegeben, die Konzepte auf ihre eigenen Unterrichtsmaterialien zu übertragen. Auf Basis der Personas erstellte jede Lehrkraft im Rahmen eines *Think-Pair-Share* (Kaddoura, 2013) dabei einen konkreten Plan, welche Ideen sie aus der Fortbildung für ihren Unterricht mitnehmen und dort anwenden wollte, der anschließend mit einer weiteren Lehrkraft ausgetauscht wurde. Abschließend diskutierten die Lehrkräfte in Gruppen über ihre Ideen und Ziele für ihre eigenen Unterrichtspraxis und stellten die Ergebnisse dieser Diskussion im Plenum vor.

9.3 Transfer in die Praxis

Die Gestaltung der Fortbildung wurde von den Lehrkräften allgemein positiv bewertet. Unter anderem wurde der hohe Anteil aktiver Arbeitszeit, der Fachvortrag sowie die intensive Vernetzung untereinander gelobt. Auch die Möglichkeit, die konkreten Materialien auszuprobieren, wurde positiv bewertet. In der abschließenden Reflexion zeigte sich ein klarer Konsens für die verstärkte Integration des Themas Debugging in den eigenen Unterricht. Der tatsächliche Transfer der Fortbildungsinhalte in die Praxis sowie die Erfahrungen der Lehrkräfte im Rahmen dieses Prozesses sollen im Folgenden untersucht werden.

9.3.1 Ziele der Untersuchung

Um den Erfolg einer Fortbildung zu evaluieren, existieren vielfältige Möglichkeiten. Da in dieser Teilstudie explizit der Transfer in die Unterrichtspraxis in den Blick genommen werden sollte, ist es das erste Ziel dieser Untersuchung, auch die tatsächliche Veränderung des Unterrichts der Lehrkräfte zu analysieren – im Gegensatz zu einer Untersuchung der Veränderung der Selbstwirksamkeitserwartungen oder des Professionswissens direkt am Ende der Fortbildung (vgl. z.B. *Geldreich, Talbot und Hubwieser (2018)*). Darüber hinaus sollte zweitens untersucht werden, welche Erfahrungen die Lehrkräfte mit dem Einsatz des Unterrichtskonzepts bzw. ihrer individuellen Adaptionen gemacht hatten. Dies ermöglicht es, verschiedenste Aspekte des Unterrichtskonzepts in unterschiedlichen Kontexten zu beforschen. Damit können beispielsweise langfristige multikausale (Neben-)Effekte der Vermittlung von Debuggingfähigkeiten für die Lehrkräfte, aber auch die Schülerinnen und Schüler (z.B. bezüglich der Selbstständigkeit) festgestellt werden, die etwa durch Beobachtungen oder formative Evaluation der Lehrkräfte empirisch gestützt sind – und damit über die vorgenommene empirische Überprüfung der Wirksamkeit (vgl. Kapitel 8) hinausgehen (*Gräsel und Parchmann, 2004*). So können Rückschlüsse auf die Gestaltungskriterien und das konkrete Unterrichtskonzept gezogen werden, um in einer engen Theorie-Praxis-Verschränkung zu einer nachhaltigen Innovation und konkreten Unterrichtsweiterentwicklung beizutragen.

Allerdings wurde bereits kurz nach der Fortbildung im November 2019 aufgrund der Covid-19-Pandemie die Schulpraxis für den Erhebungszeitraum des Schuljahres 2019/20 weitgehend eingeschränkt. Aufgrund der besonderen Anforderungen des ab März 2020 vorwiegend „digitalen“ Fernunterrichts konnten die Lehrkräfte daher kaum Erfahrungen bezüglich der Wirkung entsprechender Maßnahmen sammeln.

Daher muss die Zielsetzung der Untersuchung eingeschränkt werden: Es kann lediglich das erste Ziel – die Veränderung des Unterrichts – untersucht werden, da über die Wirkung

auf Schülerinnen und Schüler sowie die möglicherweise veränderte Situation für die Lehrkräfte im Klassenzimmer keine Aussagen getroffen werden können. Jedoch ist auch für dieses Ziel von einem enormen Einfluss der pandemiebedingten Unterrichtssituation auszugehen. So ist zu erwarten, dass die zur Verfügung stehende Zeit sowie die Bereitschaft für die Umsetzung neuer Konzepte vor dem Hintergrund der zusätzlichen Belastung und Anforderungen an die Lehrkräfte stark eingeschränkt waren. Daher kann unter diesen besonderen Bedingungen das Ziel der Evaluation nicht eine quantitative Untersuchung und statistische Generalisierung der Veränderungen im Unterricht sein, sondern eine genaue Analyse der Fälle, in denen die Lehrkräfte trotz der oder bereits vor den Schulschließungen entsprechend ihren Unterricht verändert haben. Daher wird die folgende Forschungsfrage untersucht:

(RQ7) Wie ändert sich der Unterricht der an der Fortbildung beteiligten Lehrkräfte im Bezug auf Debugging?

9.3.2 Methodik

Zur Untersuchung der Forschungsfrage wurde eine qualitative Fallstudienmethodik gewählt, um die Veränderungen der Unterrichtspraxis der Lehrkräfte umfassend abbilden zu können (Borchardt und Göthlich, 2007). Die Forschungsmethode der Fallstudie erlaubt dabei die Merkmale der einzelnen Fälle ausführlich zu beleuchten (Bortz und Döring, 2006). Im Folgenden werden die Schritte der Datenerhebung und Auswahl der Fälle sowie der Datenauswertung im Detail beschrieben.

Datenerhebung und Auswahl der Fälle

Gegen Ende des Schuljahres wurden telefonische teilstrukturierte Interviews mit den Teilnehmenden der Fortbildung durchgeführt. Der Leitfaden (siehe Anhang F) wurde nach dem Verfahren von Helfferich (2019) entwickelt. Dabei wurden *kritische Fälle* (Yin, 2003) ausgewählt, die es erlauben, den Zusammenhang zwischen der Fortbildung und der Art und Weise der Veränderung des Unterrichts zu überprüfen: Tatsächlich hatten trotz der Covid-19-Pandemie und des entsprechend eingeschränkten Schulbetriebs zwei der befragten Lehrkräfte bereits vor Beginn der Schulschließungen entsprechende Inhalte der Fortbildung in nennenswertem Umfang in ihren Unterricht integriert bzw. diese trotz der Anforderungen und Zusatzbelastung des digitalen Unterrichts im Fernunterricht umgesetzt.

Auswertung

Die entsprechenden Interviews wurden zunächst wörtlich transkribiert und in Schriftdeutsch überführt und werden im Folgenden in einer einzelfallbasierten Analyse (*within-case analysis*) ausgewertet, um ein tiefer gehendes Verständnis für die jeweiligen Veränderungen zu entwickeln (*Eisenhardt, 1989*). Anschließend sollen (*cross-case*) zentrale fallübergreifende Merkmale identifiziert werden.

9.3.3 Ergebnisse

Im Folgenden sollen nun die zwei Fälle im Detail beschrieben werden, um anschließend gemeinsame Merkmale der Fälle zu identifizieren.

Lehrkraft I

Lehrkraft I berichtet davon, dass sie bereits kurz nach Ende des Workshops erste Konzepte der Fortbildung zur systematischen Begleitung der Schülerinnen und Schüler umgesetzt hat:

Sobald das möglich war, habe ich das probiert. Das heißt, wir haben dann im Dezember oder Januar eine Rückschau gehalten und [Fehler] kategorisiert nach Typen und uns auch über den Umgang mit Fehlern ausgetauscht. Das war eigentlich so eine Doppelstunde, wo man das als Thema hatte. Die Schüler hatten ein vorgegebenes Programmierprojekt, also das war bereits fertig, und da waren Fehler eingebaut, und die mussten verschiedene Aufgaben dann eben lösen, [...] vom Strichpunkt, der fehlt, bis hin zu semantischen Fehlern, die am Schluss doch noch auftauchen, obwohl das Programm nach außen hin einwandfrei läuft. [LKI, #00:01:19]

Hier wurden also Debuggingaufgaben eingesetzt und insbesondere der Aspekt der Kategorisierung verschiedener Fehlertypen und eines entsprechend unterschiedlichen Vorgehens betont. Generell stellt das Systematisieren von Fehlern für die Schülerinnen und Schüler, auch in der individuellen Unterstützung, eine der zentralen Veränderungen des Unterrichts der Lehrkraft dar. Dabei konnte die Lehrkraft für diesen Unterricht vor dem Covid-19-bedingten Fernunterricht auch noch ihre Einschätzung zu Erfolg und Wirkung auf die Schülerinnen und Schüler geben:

[Wie hilfreich das langfristig ist,] ist im Moment schwierig zu bestätigen oder zu widerlegen. Für die Schwächeren ist es auf jeden Fall eine Hilfe gewesen. Die haben zumindest dieses Systematisieren dankbar angenommen auch. Und haben [...] dann so

ein Glossar geschrieben, und das, denke ich, wieder rausgezogen, wenn sie dann Fehler hatten, die sie jetzt erstmal nicht kannten und an die sie sich erinnern mussten. [LKI, #00:05:56]

Mit der gemeinsamen Fehlersammlung in Form des „Glossars“ wurde also einerseits eine weitere Idee des Unterrichtskonzepts adaptiert und – soweit beurteilbar – erfolgreich eingesetzt. Andererseits zeigt sich, dass tatsächlich insbesondere „schwächere“ Schülerinnen und Schüler durch die Unterstützung profitiert haben. Generell betont die Lehrkraft, dass sie den Umgang mit Fehlern in Zukunft bereits zu Beginn der Programmierunterrichts einführen will.

Darüber hinaus hat die Lehrkraft Debugging auf das Thema Tabellenkalkulation transferiert und auch hier „Debuggingaufgaben“ eingesetzt:

Da war es mal interessant zu sehen, welche Arten von Fehlern gibt es da? Die kommen ja auch oft nicht zurecht. Man muss rumrennen, weil dann zu viele Rauten angezeigt werden, was halt einfach darauf hindeutet, dass die Spalten zu schmal sind. Typische Fehler oder auch Division durch Null und so weiter. Und die Fehlermeldungen sind relativ kryptisch. [LKI, #00:02:35]

Der Umgang mit Fehlern stellt für die Lehrkraft nach der Fortbildung ein übergreifendes Thema über alle Jahrgangsstufen hinweg dar und ist nicht auf die Programmierung beschränkt. Damit wird die allgemeinbildende Bedeutung von Debugging über die Programmierung hinaus betont. Als Konsequenz der Fortbildung nimmt die Lehrkraft dabei insbesondere die Fehlerkultur im Unterricht in den Blick:

Also was wir in dem Zusammenhang noch gemacht haben in der Stunde vor Weihnachten, was ich aber auch in den letzten Jahren schon manchmal gemacht habe, ist, berühmte Softwarefehler zu besprechen. Fehler machen, Ariane und Mars Lunar Voyager und so weiter, diese ganzen Sachen. Das sind spannende Geschichten, die den Schülern auch zeigen, dass auch im Großen mal Mist gebaut wird und dass Fehler einfach passieren. [LKI, #00:04:04]

Nach der Reflexion ihres Unterrichts vermutet sie dabei einen Zusammenhang zwischen der Öffnung von Aufgaben und der Wahrnehmung von sowie dem Umgang mit Fehlern:

Die Aufgaben sind oft so, dass die Schüler sehr gezielt ein Problem lösen müssen. Die Aufgabenstellung ist klar umrissen. Die Schüler müssen das lösen und stolpern dann irgendwo rein in diese Fehler, die sie dann machen, und sind dann enttäuscht, weil es nicht auf Anhieb klappt. [...] Das heißt, man kommt in jeder Stunde an den Punkt, wenn die Schüler Fehler machen: „Jetzt habe ich einen Fehler gemacht. Ich habe das Problem nicht so gelöst, wie es der Lehrer intendiert hat.“ Und der Weg müsste ja eigentlich ein

anderer sein. In dem ganzen Programmierunterricht, nämlich, dass ich kreativer arbeite und den Schülern einfach mehr Freiräume gebe in den Beispielen. [LKI, #00:06:50]

Sie will daher in Zukunft bereits im Anfangsunterricht offenere Aufgabenstellungen verwenden, um zu erproben, welchen Einfluss das auch auf den Umgang mit Fehlern der Schülerinnen und Schüler hat. Dies lässt sich als Ansatz gemäß dem *productive-failure*-Ansatz (vgl. Kapitel 2.6) bezeichnen. Insgesamt hat damit die Bedeutung des Themas für den Unterricht zugenommen und wird auch in der Ausbildung von Referendarinnen und Referendaren multipliziert:

Aus der Praxis kann ich noch sagen, dass ich das jetzt auch mit den Referendaren thematisiere. Ich werde das als Fachsitzung auch machen in den nächsten Wochen, bis jetzt waren wir da auch durch Corona gehindert. Aber jetzt, bis zum Ende des Schuljahres, ist es auch für mich nochmal Ziel, dass wir da Herangehensweisen [für das Debuggen im Unterricht] erkunden. [LKI, #00:10:58]

Darüber hinaus berichtet die Lehrkraft auch davon, dass sich ihr eigenes Vorgehen beim Debuggen im Kontext von Programmierprojekten einer neu erlernten Programmiersprache weiterentwickelt hat:

Da hab ich gelernt, etwas was ich in Java nie gemacht habe und nie gebraucht habe, was wir da auf der Tagung besprochen haben, nämlich das print-Debugging. Das brauche ich ein, zwei Mal bei irgendwelchen Java-Problemen, aber bei den Python-Problemen war es dermaßen massiv. [...] Das heißt, man muss immer schauen, was ist Input, was ist Output, was ist der aktuelle Stand der Variablen? Und da habe ich unwahrscheinlich viel mit print-Debugging gemacht. Was ich sonst eigentlich nie gemacht habe. [LKI, #00:12:28]

Zusammenfassend zeigt sich damit eine bedeutende Veränderung der Unterrichtspraxis der Lehrkraft in Folge der Fortbildung. So wird der Umgang mit Fehlern nun als jahrgangsstufenübergreifender Inhalt aufgefasst, die Schülerinnen und Schülern werden insbesondere durch eine Systematisierung und Sammlung verschiedener Fehler unterstützt und die Lehrkraft versucht, eine positive Fehlerkultur im Unterricht zu schaffen.

Lehrkraft II

Die zweite Lehrkraft berichtet ebenfalls davon, Debugging in Konsequenz der Fortbildung prominenter in ihren Unterricht integriert zu haben. Entsprechend führte sie während des Fernunterrichts erstmalig zwei Debuggingstrategien ein. Zunächst eine „Debug-Klasse“, die die Aufrufe der Methoden eines gegebenen Projektes loggt:

[...] habe ich eine Debug-Klasse eingeführt, weil ich so selber Fehler suche und mit Hilfe dieser Debug-Klasse wurden dann also Traces erzeugt und diese Traces sollten dann wieder in Sequenzdiagrammen dargestellt werden, um in Richtung Modellierungen zu gehen [...], aber auch um grundsätzliche Mechanismen klarzumachen. Das heißt, wir haben jetzt nicht am Fehler gelernt, sondern erstmal am Trace. [LKII, #00:02:11]

Damit wurde diese Debuggingstrategie zunächst nicht im Kontext der Fehlerbehebung, sondern der Modellierung eingeführt und sollte erst darauf aufbauend auch für das Debugging verwendet werden. Darüber hinaus wurde der Debugger analog zur Vorgehensweise des in der Fortbildung vorgestellten Unterrichtskonzepts (aber ohne Skill-Card) anhand eines von der Lehrkraft erstellten Projekts als optionale Aufgabe eingeführt. Die Lehrkraft betonte dabei, dass diese beiden Ansätze aus dem Workshop kommen, da Debugging kein vorgegebenes Thema im Rahmenlehrplan darstellt.

Von dieser Veränderung erhofft sie sich mehr Selbstständigkeit in der Fehlerbehebung durch die Schülerinnen und Schüler, die auch andersartige Unterrichtsprojekte ermöglicht – für die sie aufgrund des Fernunterrichts allerdings keine Erfahrungen sammeln konnte:

Das ist eben die Neuerung in meinem Unterricht oder auch in meinem Anspruch oder meinem Herangehen. Das war ja in der Vergangenheit nicht so der Fokus. Da ging es ja darum, das Projekt grundsätzlich so klein zu halten, dass ich eigentlich davon ausgehe, dass das fehlerfrei stattfindet. [LKII, #00:23:07]

Abschließend fasst die Lehrkraft die Veränderung ihrer Unterrichtspraxis insbesondere bezüglich des Themas Debugging zusammen:

Was ihr gemacht habt, aus meiner Sicht, ist, ihr habt an den Überzeugungen der Kollegen gearbeitet. Und da sehe ich jetzt die Bedeutung höher als vorher. Ich bin tatsächlich der Meinung, dass es ja nicht ganz fair ist, dem Schüler immer vorzuwerfen, er findet die Fehler nicht, wenn ich ihm doch nicht mal gezeigt habe, was ich normalerweise nutze. Das empfinde ich jetzt als unfair. Deshalb ist eben das Minimum, was ich ihm zeigen muss, das, was ich üblicherweise selber benutzte. Ich benutze üblicherweise das Tracing, deswegen ist es bei mir drin. [LKII, #00:41:23]

Damit zeigt sich eine veränderter Blick auf den eigenen Unterricht, der betont, dass die Schülerinnen und Schüler ohne die Vermittlung adäquater Vorgehensweisen eben nicht in der Lage sind, selbstständig mit Fehlern umzugehen. Die Fortbildung hat somit dazu beigetragen, die persönlichen Überzeugungen der Lehrkraft bezüglich des Stellenwerts des Debuggings für den Programmierunterricht zu verändern.

Zusammenfassend zeigt sich damit für diese Lehrkraft, dass insbesondere die Bedeutung des Themas Debugging im Unterricht – obwohl nicht curricular verankert – deutlich zuge-

nommen hat. So führt sie, ausgehend von ihrem eigenen Debuggingvorgehen, nun Debuggingstrategien systematisch ein, um die Selbstständigkeit der Schülerinnen und Schüler zu steigern.

9.3.4 Interpretation

Vergleicht man beide Fälle bezüglich der Veränderung des Unterrichts, ist zunächst der gestiegene **Stellenwert** des Themas festzustellen. Die Bedeutung von Debugging für den eigenen Unterricht – sogar über die Programmierung hinaus – schlägt sich dabei in (gestiegener) Unterrichtszeit nieder, die für die explizite Vermittlung von Debuggingfähigkeiten aufgewendet wird.

Beide Lehrkräfte haben die Materialien aus der Fortbildung dabei nicht direkt übernommen, sondern für ihre eigenen Bedürfnisse **adaptiert**. Dabei haben sie unterschiedliche Schwerpunkte gesetzt: So konzentriert sich Lehrkraft I insbesondere auf den Aspekt der Fehlerkultur und der Systematisierung unterschiedlicher Fehlertypen, während Lehrkraft II vor allem ihr persönliches Debuggingvorgehen den Schülerinnen und Schüler vermittelt.

In beiden Fällen kommt es dabei zu einer **Erweiterung** der Fortbildungsinhalte: So überträgt Lehrkraft I die Konzepte der Fortbildung auch für den Unterricht von Tabellenkalkulationssystemen, während Lehrkraft II die Einführung entsprechender Tracing-Strategien mit der Modellierung kombiniert.

Auch **reflektieren** beide Lehrkräfte in Konsequenz der Fortbildung ihre Unterrichtspraxis bezüglich des Umgangs mit Fehlern: So möchten sie in Zukunft offenere Aufgabenstellungen erproben. Lehrkraft II erhofft sich dabei durch eine Steigerung der Selbstständigkeit der Schülerinnen und Schüler in der Fehlerbehebung, solche offeneren Formate überhaupt einsetzen zu können. Lehrkraft I vermutet einen Zusammenhang zwischen offeneren Aufgabenstellungen und der Wahrnehmung und dem Umgang mit Fehlern gemäß einem *productive-failure*-Ansatz, den sie in Zukunft explorieren möchte.

Grundlage für diese Veränderungen der Unterrichtspraxis stellt die **Erweiterung der professionellen Kompetenz** im Rahmen der Fortbildung dar, die sich in den Fallstudien gezeigt hat: Einerseits haben die Lehrkräfte entsprechendes *content knowledge* erworben, das sogar Einfluss auf ihren persönlichen Debuggingprozess hatte. Darüber hinaus haben die Lehrkräfte verschiedene Möglichkeiten zur Vermittlung von Debuggingfähigkeiten kennen gelernt (*pedagogical content knowledge*). Gleichzeitig haben sich insbesondere die *Einstellungen* der Lehrkräfte gegenüber der expliziten Vermittlung von Debugging verändert. Diese Vermittlung hatte in beiden Fällen vorher kaum eine Rolle in ihrer Unterrichtspraxis gespielt.

Limitationen

Aufgrund der Covid-19-bedingten Unterrichtssituation und der entsprechenden zeitlichen wie organisatorischen Anforderungen berichteten viele Teilnehmende der Fortbildung, die entsprechend vorgesehene Integration von Debugginginhalten in ihren Unterricht noch nicht vorgenommen zu haben. Eine breitere Evaluation der Fortbildung bezüglich des Transfers in die Unterrichtspraxis der Lehrkräfte war so nicht möglich. Allerdings ermöglichen die Fälle der Lehrkräfte, die Debugging bereits vor bzw. trotz des Fernunterrichts in ihrem Unterricht umgesetzt hatten, einen tiefen Einblick in die Veränderungen ihres Unterrichts und lassen damit Rückschlüsse auf Erfolgsfaktoren der Fortbildung zu.

Darüber hinaus konnte das ursprünglich zweite Ziel dieser Untersuchung nicht verfolgt werden: Aufgrund des Covid-19-bedingten Fernunterrichts konnten auch im Rahmen der Fallstudien nur sehr eingeschränkte Erkenntnisse bezüglich der Erfahrungen erhoben werden, die Lehrkräfte mit dem (insbesondere auch langfristigen) Einsatz des Unterrichtskonzepts bzw. ihrer individuellen Adaptionen gemacht hatten. Daher stellt es eine Aufgabe für zukünftige Forschung dar, unter schulischen „Normalbedingungen“ zu untersuchen, wie sich beispielsweise die Selbstständigkeit der Schülerinnen und Schüler, aber auch die „Turnschuhdidaktik“ der Lehrkräfte durch die Integration von Debugging in ihren Unterricht tatsächlich verändert, um im Sinne einer engen Theorie-Praxis-Verschränkung Rückschlüsse auf die Weiterentwicklung des Unterrichtskonzepts bzw. der zugrunde liegenden Gestaltungskriterien ziehen zu können.

9.4 Fazit

Um, aufbauend auf den entwickelten Materialien und Konzepten für das Debugging im Informatikunterricht, zu einer tatsächlichen Weiterentwicklung und Professionalisierung des Informatikunterrichts und damit Praxiswirksamkeit beizutragen, wurde in diesem Kapitel der Transfer der gewonnenen Erkenntnisse und Ergebnisse in die Unterrichtspraxis untersucht. Dazu wurde, geleitet durch die Rückschlüsse auf die *professionelle Kompetenz*, die die zuvor vorgenommene Untersuchung der Perspektive der Lehrkräfte ermöglichte, und existierende Erfahrungen sowie Forschungsergebnisse zur didaktischen Ausgestaltung von Fortbildungen, ein entsprechender Workshop konzipiert und durchgeführt.

Die Evaluation bezüglich der Veränderung der Unterrichtspraxis der Lehrkräfte offenbarte, dass die Fortbildung für die zwei analysierten Fälle die intendierte Wirkung hatte. Dies zeigte sich insbesondere in einer gestiegenen Bedeutung des Themas in der Unterrichtspraxis. Dabei stellten die Materialien der Fortbildung zumeist die Ausgangsbasis für die individuelle Adaption gemäß der persönlichen Bedürfnisse und unterschiedlichen Schwer-

punkte dar. Darüber hinaus wurden Erweiterungen vorgenommen und auch der weitere Unterricht bezüglich der Bedeutung von Fehlern reflektiert und daraus konkrete Veränderungswünsche abgeleitet, die, wie zum Beispiel die geplante Einführung offenerer Aufgaben, über die Vermittlung von Debugging hinausgehen.

Für Praxiswirksamkeit und Transfer in die Unterrichtspraxis erscheint es damit zentral, die Bedeutung des Debugging für den Unterricht zu betonen: Auch wenn nicht explizit in den Curricula gefordert, stellt Debugging einen essentiellen Bestandteil des Programmierprozesses dar, dessen adäquate Vermittlung für die Lehrkräfte Potential zur Weiterentwicklung ihrer Unterrichtspraxis bietet. Die Berichte der Lehrkräfte deuten darauf hin, dass gerade die Reflexion der eigenen Unterrichtspraxis zur Schaffung eines Problembewusstseins beigetragen hat und somit wesentlich für die Veränderung der Einstellungen der Lehrkräfte zum Debugging war. Die dabei entstandenen Überzeugungen wurde durch die konkreten Materialien als mögliche Ansatzpunkte für die Vermittlung von Debuggingfähigkeiten unterstützt: Die Lehrkräfte adaptierten die Ideen der Fortbildung für ihre persönlichen Anforderungen, erweiterten sie und experimentierten mit ihnen – sogar über den Programmierunterricht hinaus.

Zusammenfassend werden damit durch den Transfer im Rahmen der Fortbildung die Forschungsergebnisse dieser Arbeit praxiswirksam, insbesondere da die Bedeutung von Debugging für die Lehrkräfte gestiegen ist, was sich vor allem in einer veränderten Fehlerkultur und -wahrnehmung im Unterricht sowie in der Unterrichtszeit zeigt, die für die explizite Vermittlung von Debuggingfähigkeiten verwendet wird – aufbauend auf dem entwickelten Konzept und den bereitgestellten Materialien.

Teil V:

Teil V: Abschluss

*A computer lets you make more mistakes faster than any invention in human history,
with the possible exceptions of handguns and tequila.*

MITCH RADCLIFFE

10 Fazit

Debuggen ist eine zentrale Tätigkeit – und Notwendigkeit – professioneller Softwareentwicklung. Für den Informatikunterricht stellt Debugging jedoch ein Schlüsselproblem dar: Die Schülerinnen und Schüler müssen sich das systematische Finden und Beheben von Fehlern in der Unterrichtspraxis zumeist selbst aneignen und sehen sich damit einer großen Herausforderung und wiederkehrenden Quelle für Frustration beim Programmierenlernen ausgesetzt. Aber auch die Lehrkräfte stehen vor der enormen Aufgabe, allen Lernenden gleichzeitig gerecht zu werden. Bisher mangelt es an entsprechenden Konzepten zur Vermittlung der benötigten Fähigkeiten, die die Selbstständigkeit der Schülerinnen und Schüler steigern und die Lehrkräfte entsprechend entlasten. Daher war es das Ziel dieser Arbeit, den Prozess Debugging informatikdidaktisch aufzuarbeiten, um auf dieser Basis forschungsgeleitet geeignete Konzepte und Materialien für den Informatikunterricht zu entwickeln. Dementsprechend wurde ein Forschungsprozess, orientiert am Modell der didaktischen Rekonstruktion, ausgestaltet, um dieses Schlüsselproblem des Informatikunterrichts anzugehen.

10.1 Zusammenfassung

Abschließend sollen nun die zentralen Forschungsergebnisse dieser Arbeit, gegliedert nach den einzelnen Forschungsfragen, zusammengefasst werden, die entsprechend dem gewählten Forschungsformat und auf Basis des Forschungsstands abgeleitet wurden.

(RQ1) Was sind relevante Debuggingfähigkeiten für Programmieranfängerinnen und -anfänger?

Im Zuge der *fachlichen Klärung* des Prozesses Debugging wurden ein Korpus aus Monographien, Schul- und Programmierlehrbüchern und Forschungsliteratur ausgewertet und vier Fähigkeiten, die für das Debugging benötigt werden, identifiziert. Die Grundlage stellt dabei die *Anwendung eines systematischen Debuggingvorgehens* dar, also das Verfolgen eines zielgerichteten Plans. Für das Debugging *eigener* Programme bedeutet das vor allem, ausgehend vom Fehlerverhalten des Programms wiederholt Hypothesen über den Fehler aufzustellen, diese experimentell zu überprüfen und gegebenenfalls zu verfeinern oder zu verwerfen. Unterstützt wird dieses Vorgehen durch die *Anwendung von Debuggingstrategien*, wie etwa *print-Debugging* oder *Slicing*, die Informationen liefern und die experimentelle Überprüfung von Hypothesen ermöglichen. Darüber hinaus spielt die bisherige Erfahrung

eine große Rolle: Durch das *Verwenden von Heuristiken und Mustern für typische Fehler* kann der Debuggingprozess häufig abgekürzt werden. Zudem müssen entsprechende *Werkzeuge* wie etwa der *Debugger* oder auch Feedback der IDE adäquat eingesetzt werden. Herausstellungsmerkmal des resultierenden Modells relevanter Debuggingfähigkeiten für Novizen ist dabei, dass die Anwendung eines systematischen Debuggingvorgehens als Grundlage des Debuggingprozesses angeführt wird, während diese in anderen Modellen oftmals lediglich als eine Debuggingstrategie unter vielen gilt. Das Modell und die damit identifizierten Fähigkeiten stellen somit die Basis für die Vermittlung von Debugging im Unterricht dar.

(RQ2) Wie gehen Lehrkräfte mit Programmierfehlern im Unterricht um und wie vermitteln sie Debugging?

Für die Untersuchung der *Perspektive der Lehrkräfte* wurde eine Interviewstudie mit 12 Lehrerinnen und Lehrern aus verschiedenen deutschen Bundesländern durchgeführt. Dabei stellte sich heraus, dass die Lehrkräfte tatsächlich oftmals von einem Arbeitsplatz zum anderen eilen und versuchen, die Schülerinnen und Schülern bei der Fehlerbehebung zu unterstützen, weil insbesondere die „schwächeren“ Schülerinnen und Schüler zumeist hilflos und überfordert sind. Darüber hinaus berichten die Lehrkräfte, dass sie Debugging nur äußerst selten *explizit* unterrichten, insbesondere da es ihnen an Zeit, Konzepten und Materialien fehlt und Debugging oftmals kein expliziter Inhalt des Curriculums ist. So werden vorwiegend unsystematisch bestimmte Debuggingstrategien – oftmals basierend auf dem eigenen Debuggingverhalten – in der individuellen Unterstützung der Schülerinnen und Schüler vorgeführt oder vermittelt. Aus diesen Erfahrungen der Lehrkräfte konnten dann Gestaltungshinweise für die Entwicklung von Konzepten und Materialien für den Unterricht abgeleitet werden.

(RQ3) Welchen Beitrag können Debuggingfähigkeiten zur Allgemeinbildung leisten?

Für die *Klärung gesellschaftlicher Ansprüche* wurden drei mögliche Beiträge des Debuggings zur Allgemeinbildung identifiziert: Zur Erklärung des Phänomens „fehlerhafte Software“ aus der Lebenswelt, bezüglich des Lernens aus Fehlern sowie als Herangehensweise des Computational Thinking für das Troubleshooting. Im Rahmen dieses Kapitels wurde dabei die letzte Dimension eingehend untersucht. Dazu wurden auf Basis des Forschungsstandes Bezüge zum Problemlösen in Form von Troubleshooting herausgearbeitet. So ist Debugging ein Spezialfall allgemeinen Troubleshootings: das Finden und Beheben von Fehlern in der Domäne der Programmierung. Dabei werden vergleichbare Fähigkeiten benötigt: So läuft der allgemeine hypothesengeleitete Troubleshootingprozess analog zum Debuggingprozess ab, wird durch Erfahrung und den Einsatz von speziellen Strategien

unterstützt und domänenspezifische Werkzeuge müssen passgenau eingesetzt werden. Gerade das systematische Debuggingvorgehen sowie (lokale) domänenspezifische Debuggingstrategien können damit einen Beitrag zur Allgemeinbildung in Form eines systematischen Troubleshooting-Vorgehens und entsprechender (globaler) domänenunabhängiger Troubleshootingstrategien leisten.

(RQ4) Welche debuggingspezifischen Lernvoraussetzungen bringen Schülerinnen und Schüler aus ihrem Alltag mit?

Für die Untersuchung der *Perspektive der Lernenden* wurden im Rahmen eines Escape-Rooms Schülerinnen und Schüler beim Bearbeiten verschiedener forschungsgeleitet entwickelter Troubleshooting-Aufgaben beobachtet, um entsprechende Vorerfahrungen im Alltag festzustellen. Damit konnten verschiedene Debugging-Lernvoraussetzungen identifiziert werden, die einerseits zum Verständnis des Debuggingverhaltens von Novizen beitragen und entsprechende Muster erklären können. Andererseits stellen diese Gestaltungshinweise für die Entwicklung von Konzepten und Materialien für den Unterricht dar. So wenden die Schülerinnen und Schüler zwar ein systematisches Vorgehen für das Troubleshooten an und beziehen bisherige Erfahrungen und Muster für typische Fehler mit in diesen Prozess ein, haben aber insbesondere Probleme mit dem Aufstellen von (Alternativ-)Hypothesen oder dem Rückgängigmachen von erfolglosen Änderungen. Darüber hinaus haben sie Schwierigkeiten mit der Anwendung von Strategien wie dem *Testen* oder der *topographischen* Suche, die entsprechend im Informatikunterricht adressiert werden müssen.

(RQ5) Wie sollten Konzepte und Materialien für die Vermittlung von Debuggingfähigkeiten im Unterricht gestaltet werden?

Basierend auf der Untersuchung der *zugrunde liegenden Perspektiven* konnten nun 10 Gestaltungskriterien für Konzepte und Materialien für das Debugging im Unterricht identifiziert werden. Konzepte und Materialien sollten...

- (1) ... Selbstständigkeit fordern und fördern.
- (2) ... einen Fokus darauf legen, das Formulieren von Hypothesen einzuüben.
- (3) ... insbesondere die Bedürfnisse „schwächerer“ Schülerinnen und Schüler adressieren.
- (4) ... Debuggingstrategien explizit vermitteln.
- (5) ... den aktiven Aufbau von Erfahrung unterstützen.

- (6) ... typische Hürden und Probleme von Lernenden beim Debugging adressieren.
- (7) ... unterschiedliches Vorgehensweisen für verschiedene Fehlerarten vermitteln.
- (8) ... der Vielfalt der persönlichen Debuggingvorlieben der Lehrkräfte Rechnung tragen.
- (9) ... variabel bei Bedarf im Unterricht einsetzbar sein.
- (10) ... tatsächliche (relevante) Debuggingfähigkeiten fördern.

Auf Basis dieser Kriterien wurde anschließend ein konkretes integratives Unterrichtskonzept für die Sekundarstufe I und für eine textbasierte Programmiersprache entwickelt.

(RQ6) Welchen Effekt hat die explizite Vermittlung von Debuggingfähigkeiten auf Selbstwirksamkeit und Debuggingleistung?

Anhand der Einführung eines systematischen Debuggingvorgehens gemäß des entwickelten Unterrichtskonzepts wurde die Wirksamkeit der expliziten Vermittlung von Debugging unter den realen Bedingungen der Schulpraxis beforscht. Innerhalb des gewählten Pre-Post-Kontrollgruppen-Test-Designs wurden im Pretest die Selbstwirksamkeitserwartungen sowie Debuggingleistung der Schülerinnen und Schüler erhoben. Als Messinstrument für die Debuggingleistung wurde eine adaptierte Variante traditioneller Debuggingaufgaben entwickelt. Während in der Experimentalgruppe die Intervention erfolgte, bearbeitete die Kontrollgruppe weitere Debuggingaufgaben. Im Posttest zeigt sich sowohl ein signifikanter Anstieg der Selbstwirksamkeitserwartungen als auch der Debuggingleistung in der Experimentalgruppe. Damit hat sich die These der Wirksamkeit der expliziten Vermittlung von Debugging – in Kontrast zum unsystematischen Üben – empirisch verdichtet.

(RQ7) Wie ändert sich der Unterricht der an einer Fortbildung beteiligten Lehrkräfte im Bezug auf Debugging?

Um eine tatsächlichen Weiterentwicklung und Professionalisierung des Informatikunterrichts und damit die Praxiswirksamkeit der Forschungsergebnisse zu erreichen, wurde eine Fortbildung konzipiert, durchgeführt und bezüglich des Transfers der Fortbildungsinhalte in die Unterrichtspraxis der Lehrkräfte evaluiert. In zwei Fallstudien zeigte sich, dass die Bedeutung des Themas Debugging im Unterricht deutlich gestiegen war. Dabei stellten die Materialien der Fortbildung zumeist die Ausgangsbasis für die individuelle Adaption gemäß der persönlichen Bedürfnisse der Lehrkräfte dar, die darüber hinaus Erweiterungen der Fortbildungsinhalte – auch über den Programmierunterricht hinaus – vornahmen und auch ihren weiteren Unterricht bezüglich der Bedeutung von Fehlern reflektiert. Damit

wurden die Forschungsergebnisse dieser Arbeit im Unterricht der Lehrkräfte praxiswirksam, insbesondere, da der Stellenwert von Debugging anstieg, was sich in Unterrichtszeit die für die explizite Vermittlung von Debuggingfähigkeiten – aufbauend auf dem entwickelten Konzept und den Materialien – sowie einer veränderten Fehlerkultur im Unterricht zeigte.

10.2 Beitrag und Ausblick

Im Folgenden soll nun der Beitrag dieser Ergebnisse diskutiert und ein Ausblick auf zukünftige Forschungsdesiderate gegeben werden, die den Rahmen dieser Arbeit überschreiten. Obwohl die informatische und informatikdidaktische Forschung zu Debugging bis auf die Siebzigerjahre zurückgeht, mangelt es an Erkenntnissen, die zentral für die Gestaltung von Konzepten und Materialien für den Informatikunterricht sind. Besonders auffällig ist dabei, dass lediglich eine äußerst geringe Anzahl an Untersuchungen für den schulischen Kontext durchgeführt wurde.

Einleitend wurde die entsprechende Forschungslücke exemplarisch anhand der Perspektiven des didaktischen Dreiecks, *Stoff, Lehrkräfte und Lernende* dargestellt:

Bezüglich des *Stoffes* war unbeantwortet, welche Fähigkeiten Programmieranfängerinnen und -anfänger überhaupt für selbstständiges Debugging benötigen. Diese Basis fachdidaktischen und unterrichtlichen Handelns wurde im Rahmen dieser Arbeit durch die Identifikation relevanter Debuggingfähigkeiten nun gelegt.

Auch fehlte es an Erkenntnissen zu den *Lehrkräften* und damit der unterrichtspraktischen Realität des Debugging im Informatikunterricht. Im Rahmen dieser Arbeit konnten nun ein tiefer Einblick in diese Realität, die bisherigen Ansätze sowie unterrichtspraktische Anforderungen gewonnen werden. Dies ermöglichte es, entsprechende Gestaltungskriterien zu identifizieren, die einerseits inhaltlicher Natur (z.B. bezüglich der konkreten Probleme der Schülerinnen und Schüler) und andererseits zentral für die Akzeptanz in der Unterrichtspraxis sind.

Außerdem mangelte es an Forschungsergebnissen bezüglich der *Lernenden*. Die bisher erfolgte Forschung konzentriert sich auf Studierende und die tertiäre Bildung, die sich in vielerlei Hinsicht von den Anforderungen und der Realität von Informatikunterricht und Schule unterscheidet. So wurden in dieser Arbeit tatsächlich Schülerinnen und Schüler und deren Debugging-Lernvoraussetzungen untersucht. Die Ergebnisse dieser Untersuchung tragen dabei einerseits zum Verständnis und der Erklärung von Verhaltensweisen von Novizen beim Debuggen bei und stellen damit andererseits Aufgaben für die Vermittlung

von Debugging dar, da diese Lernvoraussetzungen dort entsprechend adressiert werden müssen.

Durch die in dieser Arbeit vorgenommenen Untersuchungen wurde damit die Grundlage geschaffen, um Konzepte und Materialien für den Informatikunterricht zu entwickeln. So wurden auf dieser Basis Gestaltungskriterien für die Vermittlung von Debugging identifiziert und ein exemplarisches Unterrichtskonzept entwickelt. Dieses Konzept wurde anschließend empirisch auf Wirksamkeit überprüft und in die Schulpraxis gegeben. Damit wurde in dieser Arbeit für die Frage „Wie kann Debugging im Informatikunterricht vermittelt werden?“ die theoretische Grundlage gelegt sowie eine konkrete Antwort gegeben und evaluiert und somit das Schlüsselproblem Debugging im Informatikunterricht erfolgreich adressiert.

Neben diesen inhaltlichen Beiträgen wurden im Rahmen dieser Arbeit darüber hinaus zwei methodische Beiträge geleistet: Zum einen wurde in Form des Escape-Room-Ansatzes ein innovativer methodischer Ansatz zur Untersuchung von Problemlöse- und Troubleshooting-Verhaltensweisen und -Strategien entwickelt. Im Vergleich zu einer Befragung oder schriftlichen Erhebung der Reaktionen der Teilnehmenden in vorgegebenen Situationen konnte der tatsächliche Prozess in einer natürlichen Umgebung beobachtet werden, einschließlich der Reaktionen, die auftreten, wenn der erste Ansatz nicht funktioniert. Die Kommunikation innerhalb der Gruppen, die in der Gestaltung der Aufgaben aktiv gefördert wurde, stellte sich als geeignet heraus, um das Vorgehen der Teilnehmenden beobachtbar zu machen. Zweitens wurde in Form der prototypenbasierten Debuggingaufgaben ein Instrument zur Messung der Debuggingleistung entwickelt, welches den Einfluss von *Codeverständnis* und *fremdem Code* auf Debuggingprozess und -erfolg der Probandinnen und Probanden reduzieren soll.

Daran schließt sich auch ein Anknüpfungspunkt für zukünftige informatikdidaktische Forschung an: Diese These und damit die Konstruktvalidität des Instruments bedarf einer empirischen Überprüfung im Vergleich zu den anderen entsprechend benannten Ansätzen, um unter Minimierung des Einflusses von *Codeverständnis* Debuggingleistung zu messen bzw. den Debuggingprozess zu untersuchen. Diese bisher oftmals unbeachtete Entwicklung eines geeigneten Messinstruments ist dabei zentral für weitergehende Forschung im Kontext des Debuggings.

Darüber hinaus ist zu untersuchen, welche *langfristigen* Effekte, z.B. hinsichtlich der Selbstständigkeit der Schülerinnen und Schüler sich in der Unterrichtspraxis durch die explizite Vermittlung von Debuggingfähigkeiten ergeben, die in dieser Arbeit auch aufgrund der Covid-19-Pandemie nur eingeschränkt erforscht werden konnten. Einen weiteren Untersuchungsgegenstand stellt dabei der Zusammenhang mit Programmierfähigkeiten dar: Möglicherweise ermöglicht das explizite Unterrichten von Debugging sogar eine wertvolle Gelegenheit, die allgemeinen Programmierfähigkeiten der Schülerinnen und Schüler zu

verbessern: So spielen beispielsweise Tracing-Strategien nicht nur im Debuggingprozess eine wichtige Rolle, sondern sind auch für die Programmierung in Form von allgemeinem Programmverständnis (*Perkins und Martin, 1985*) oder bezüglich des Konzepts der *notional machine* (*Venables, Tan und Lister, 2009*) zentral.

Ein weiteres offenes Forschungsdesiderat stellt die empirische Untersuchung des postulierten Beitrags zur Allgemeinbildung dar: Nachdem in der didaktischen Strukturierung Gestaltungsprinzipien und ein entsprechendes Konzept zur Förderung von Debuggingfähigkeiten entwickelt wurde, kann nun untersucht werden, inwieweit die Förderung von Debuggingfähigkeiten tatsächlich einen Beitrag zur Allgemeinbildung leistet und ein systematisches Debuggingvorgehen und entsprechende Strategien auf andere Kontexte übertragen werden können. Dazu kann beispielsweise ein Escape-Room-Ansatz analog zu dieser Arbeit und ein Studiendesign analog zu *Carver (1986)* (vgl. auch Kapitel 2.6) angewandt werden.

Schließlich muss analysiert werden, wie die Anwendung der entwickelten Gestaltungsprinzipien auch auf die blockbasierte Programmierung als typischen Zugang zur Programmierung übertragen werden kann. So ist zum Beispiel zu klären, was entsprechende Debuggingstrategien und Debuggingwerkzeuge in der blockbasierten Welt sind, wo typische Probleme liegen und wie sich die Gestaltungsprinzipien und Erkenntnisse dort realisieren lassen.

Zusammenfassend leistet diese Arbeit also einen Beitrag zu einem Bereich, der in der informatikdidaktischen Forschung trotz der Bedeutung als unterrichtspraktisches Schlüsselproblem bisher weitestgehend vernachlässigt wurde. Damit wird durch die Aufarbeitung von Debugging als zentralem Prozess in der Programmierung ein essentielles Aufgabengebiet der Informatikdidaktik und des Informatikunterrichts adressiert: Schülerinnen und Schüler zur aktiven und kreativen Gestaltung der sogenannten „digitalen Welt“ zu befähigen.

Verzeichnisse

Literaturverzeichnis

Alle gegebenenfalls angegebenen Weblinks wurden zuletzt überprüft am 7.1.2021.

- Agans, David (2002). *Debugging – The 9 indispensable rules for finding even the most elusive software and hardware problems*. New York, NY, USA: Amacom.
- Ahmadzadeh, Marzieh, Elliman, Dave und Higgins, Colin (2005). „An analysis of patterns of debugging among novice computer science students“. In: *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. New York, NY, USA: ACM, S. 84–88.
- (2007). „The impact of improving debugging skill on programming ability“. In: *Innovation in Teaching and Learning in Information and Computer Sciences 6.4*, S. 72–87.
- Allwood, Carl Martin und Björhag, Carl-Gustav (1990). „Novices’ debugging when programming in Pascal“. In: *International Journal of Man-Machine Studies 33.6*, S. 707–724.
- (1991). „Training of Pascal novices’ error handling ability“. In: *Acta Psychologica 78.1-3*, S. 137–150.
- Altadmri, Amjad und Brown, Neil CC (2015). „37 million compilations: Investigating novice programming mistakes in large-scale student data“. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, S. 522–527.
- Altadmri, Amjad, Kölling, Michael und Brown, Neil CC (2016). „The cost of syntax and how to avoid it: Text versus frame-based editing“. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Bd. 1. Atlanta, GA, USA: IEEE, S. 748–753.
- Anderson, Gary und Page, Bonnie (1995). „Narrative knowledge and educational administration: The stories that guide our practice“. In: *The knowledge base in educational administration: Multiple perspectives*, S. 124–136.
- Anderson, John Robert und Crawford, Jane (1980). *Cognitive psychology and its implications*. San Francisco, CA, USA: Worth Publishers.
- Araki, Keijiro, Furukawa, Zengo und Cheng, Jingde (1991). „A general framework for debugging“. In: *IEEE software 8.3*, S. 14–20.
- Bandura, Albert (1962). *Lernen am Modell. Ansätze zu einer sozial-kognitiven Lerntheorie*. Stuttgart: Klett.
- (1982). „Self-efficacy mechanism in human agency.“ In: *American psychologist 37.2*, S. 122.
- Barnes, David John, Kölling, Michael und Gosling, James (2016). *Objects First with Java: A practical introduction using BlueJ*. 6. Aufl. London, UK: Pearson/Prentice Hall.
- Barzel, Bärbel und Selter, Christoph (2015). „Die DZLM-Gestaltungsprinzipien für Fortbildungen“. In: *Journal für Mathematik-Didaktik 36.2*, S. 259–284.

-
- Baumert, Jürgen und Kunter, Mareike (2006). „Stichwort: Professionelle Kompetenz von Lehrkräften“. In: *Zeitschrift für Erziehungswissenschaft* 9.4, S. 469–520.
- Becker, Arno (2015). *Android 5 : Programmieren für Smartphones und Tablets*. Heidelberg: dpunkt.verlag.
- Beller, Moritz et al. (2018). „On the dichotomy of debugging behavior among programmers“. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, S. 572–583.
- Bereiter, Susan und Miller, Steven (1989). „A field-based study of troubleshooting in computer-controlled manufacturing systems“. In: *IEEE transactions on Systems, Man, and Cybernetics* 19.2, S. 205–219.
- Blömeke, Sigrid, Suhl, Ute und Döhrmann, Martina (2012). „Zusammenfügen was zusammengehört. Kompetenzprofile am Ende der Lehrerausbildung im internationalen Vergleich“. In: *Zeitschrift für Pädagogik* 58.4, S. 422–440.
- Board, Mishap Investigation (1999). *Mars Climate Orbiter Mishap Investigation Board Phase I Report*.
- Boles, Dietrich (1999). *Programmieren spielend gelernt mit dem Java-Hamster-Modell*. Bd. 2. Wiesbaden: Springer.
- Bonar, Jeffrey und Soloway, Elliot (1985). „Preprogramming knowledge: A major source of misconceptions in novice programmers“. In: *Human-Computer Interaction* 1.2, S. 133–161.
- Bönsch, M (2006). *Allgemeine Didaktik. Ein Handbuch zur Wissenschaft vom Unterricht*. Stuttgart: W. Kohlhammer Verlag.
- Borchardt, Andreas und Göthlich, Stephan (2007). „Erkenntnisgewinnung durch Fallstudien“. In: *Methodik der empirischen Forschung*. Hrsg. von Sönke Albers et al. Wiesbaden: Gabler, S. 33–48.
- Borowski, Andreas et al. (2010). „Professionswissen von Lehrkräften in den Naturwissenschaften (ProwiN)–Kurzdarstellung des BMBF-Projekts“. In: *Zeitschrift für Didaktik der Naturwissenschaften* 16, S. 341–349.
- Borrego, Carlos et al. (2017). „Room escape at class: Escape games activities to facilitate the motivation and learning in computer science“. In: *JOTSE* 7.2, S. 162–171.
- Bortz, Jürgen und Döring, Nicola (2006). *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler*. 4. Aufl. Berlin Heidelberg: Springer.
- Böttcher, Axel et al. (2016). „Debugging students’ debugging process“. In: *2016 IEEE Frontiers in Education Conference (FIE)*. Erie, PA, USA: IEEE, S. 1–7.
- Bransford, John, Brown, Ann und Cocking, Rodney (2000). *How people learn*. Washington DC, USA: National Academy Press.
- Brauer, Johannes (2015). *Programming Smalltalk-Object-Orientation from the Beginning*. Wiesbaden: Springer.
- Brennan, Karen und Resnick, Mitchel (2012). „New frameworks for studying and assessing the development of computational thinking“. In: *Proceedings of the 2012 annual meeting of the American Educational Research Association*. Vancouver, Canada, S. 1–25.

-
- Briggs, Jason und Haxsen, Volker (2016). *Python kinderleicht! : Einfach programmieren lernen – nicht nur für Kids*. Heidelberg: dpunkt.verlag.
- Brown, Neil und Altadmri, Amjad (2014). „Investigating novice programming mistakes: educator beliefs vs. student data“. In: *Proceedings of the tenth annual conference on International computing education research*. New York, NY, USA: ACM, S. 43–50.
- Brown, Neil et al. (2014). „Restart: The resurgence of computer science in UK schools“. In: *ACM Transactions on Computing Education (TOCE)* 14.2, S. 1–22.
- BSI, A (1997). *Definition of Year 2000 Conformity Requirements*. Techn. Ber. DISC-PD2000-1, British Standards Institution, London.
- Carter, Elizabeth Emily (2014). „An intelligent debugging tutor for novice computer science students“. Diss. Lehigh University.
- Carver, Sharon (1986). „Transfer of LOGO Debugging Skill: Analysis, Instruction, and Assessment.“ Diss. Carnegie Mellon University.
- Carver, Sharon und Klahr, David (1986). „Assessing children’s LOGO debugging skills with a formal model“. In: *Journal of educational computing research* 2.4, S. 487–525.
- Carver, Sharon und Risinger, Sally (1987). „Improving children’s debugging skills“. In: *Empirical studies of programmers: Second workshop*. Ablex Publishing Corp., S. 147–171.
- Chen, Monchu und Lim, Veraneka (2013). „Eye gaze and mouse cursor relationship in a debugging task“. In: *Proceedings of the International Conference on Human-Computer Interaction*. Berlin Heidelberg: Springer, S. 468–472.
- Chmiel, Ryan und Loui, Michael C (2004). „Debugging: from novice to expert“. In: *ACM SIGCSE Bulletin* 36.1, S. 17–21.
- Clarke, Samantha et al. (2017). „escapED: a framework for creating educational escape rooms and Interactive Games For Higher/Further Education“. In: *International Journal of Serious Games* 4.3, S. 73–86.
- Claus, Volker und Schwill, Andreas (2003). *Duden Informatik – Ein Fachlexikon für Studium und Praxis*. Berlin: Bibliographisches Institut.
- Cohen, Jacob (1988). *Statistical power analysis for the behavioural sciences*. 2. Aufl. Hillsdale, NJ, USA: Lawrence Erlbaum.
- Cunningham, Kathryn et al. (2017). „Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw“. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. New York, NY, USA: AMC, S. 164–172.
- Danielsiek, Holger, Toma, Laura und Vahrenhold, Jan (2018). „An instrument to assess self-efficacy in introductory algorithms courses“. In: *ACM Inroads* 9.1, S. 56–65.
- Darling-Hammond, Linda, Hyler, Maria und Gardner, Madelyn (2017). *Effective teacher professional development*. Palo Alto, CA, USA: Learning Policy Institute.
- Deci, Edward und Ryan, Richard (2012). „Motivation, personality, and development within embedded social contexts: An overview of self-determination theory“. In: *The Oxford handbook of human motivation*. New York, NY, USA: Oxford University Press, S. 85–107.

-
- DeLiema, David et al. (2019). „Debugging as a context for fostering reflection on critical thinking and emotion“. In: *Deeper Learning, Dialogic Learning, and Critical Thinking: Research-based Strategies for the Classroom*. Hrsg. von Emmanuel Manalo. New York: Routledge, S. 209–228.
- Denning, Peter J (2017). „Remaining trouble spots with computational thinking“. In: *Communications of the ACM* 60.6, S. 33–39.
- Détienne, Françoise und Soloway, Elliot (1990). „An empirically-derived control structure for the process of program understanding“. In: *International Journal of Man-Machine Studies* 33.3, S. 323–342.
- Dewey, John (1986). „Experience and education“. In: *The Educational Forum*. Bd. 50. 3. Taylor & Francis, S. 241–252.
- Diethelm, Ira, Hubwieser, Peter und Klaus, Robert (2012). „Students, teachers and phenomena: educational reconstruction for computer science education“. In: *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. New York, NY, USA: ACM, S. 164–173.
- Diethelm, Ira et al. (2011). „Die Didaktische Rekonstruktion für den Informatikunterricht“. In: *Informatik in Bildung und Beruf - 14. GI-Fachtagung Informatik und Schule - INFOS 2011*. Hrsg. von Marco Thomas. Bd. P-189. LNI. Bonn: GI, S. 77–86.
- Döbeli Honegger, Beat (2016). *Mehr als 0 und 1: Schule in einer digitalisierten Welt*. Bern: Hep Verlag.
- Dohmen, Günther (2000). „Das Übergreifende denken, das Praktische erkunden, das Vernünftige tun“. In: *Literatur- und Forschungsreport Weiterbildung* 2.45, S. 55–72.
- Dörn, Sebastian (2019). *Java lernen in abgeschlossenen Lerneinheiten*. Wiesbaden: Springer Vieweg.
- (2020). „Python lernen in abgeschlossenen Lerneinheiten“. In:
- Dounas-Frazer, Dimitri et al. (2016). „Investigating the role of model-based reasoning while troubleshooting an electric circuit“. In: *Physical Review Physics Education Research* 12 (1), S. 010137-01 –010137-20.
- Ducasse, M und Emde, A-M (1988). „A review of automated debugging systems: knowledge, strategies and techniques“. In: *Proceedings.[1989] 11th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE, S. 162–171.
- Duit, Reinders, Gropengießer, Harald und Kattmann, Ulrich (2005). „Towards science education research that is relevant for improving practice: The model of educational reconstruction“. In: *Developing standards in research on science education edition*. Hrsg. von H.E. Fischer. London, UK: Taylor & Francis, S. 1–10.
- Edwards, Stephen H (2004). „Using software testing to move students from trial-and-error to reflection-in-action“. In: *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, S. 26–30.
- Eisenhardt, Kathleen M (1989). „Building theories from case study research“. In: *Academy of management review* 14.4, S. 532–550.

-
- Eisenstadt, Marc (1997). „My hairiest bug war stories“. In: *Communications of the ACM* 40.4, S. 30–37.
- Eranki, Kiran LN und Moudgalya, Kannan M (2016). „Program slicing technique: a novel approach to improve programming skills in novice learners“. In: *Proceedings of the 17th Annual Conference on Information Technology Education*. New York, NY, USA: ACM, S. 160–165.
- Erbs, Heinz-Erich und Stolz, Otto (1984). *Einführung in die Programmierung mit PASCAL*. Wiesbaden: Vieweg & Teubner.
- Ettles, Andrew, Luxton-Reilly, Andrew und Denny, Paul (2018). „Common logic errors made by novice programmers“. In: *Proceedings of the 20th Australasian Computing Education Conference*. New York, NY, USA: ACM, S. 83–89.
- Eukel, Heidi, Frenzel, Jeanne und Cernusca, Dan (2017). „Educational Gaming for Pharmacy Students—Design and Evaluation of a Diabetes-themed Escape Room“. In: *American journal of pharmaceutical education* 81.7, S. 6265.
- Farmer, Jeff, Gerretson, Helen und Lassak, Marshall (2003). „What teachers take from professional development: Cases and implications“. In: *Journal of Mathematics Teacher Education* 6.4, S. 331–360.
- Fields, Deborah A, Searle, Kristin A und Kafai, Yasmin B (2016). „Deconstruction kits for learning: Students’ collaborative debugging of electronic textile designs“. In: *Proceedings of the 6th Annual Conference on Creativity and Fabrication in Education*. New York, NY, USA: ACM, S. 82–85.
- Fields, Deborah und Kafai, Yasmin (2020). „Debugging by Design: Students’ Reflections on Designing Buggy E-Textile Projects“. In: *Proceedings of the 2020 Constructionism Conference*.
- Fincher, Sally A., Kolikant, Yifat Ben-David und Falkner, Katrina (2019). „Teacher Learning and Professional Development“. In: *The Cambridge Handbook of Computing Education Research*. Hrsg. von Sally A. Fincher und Anthony V. Robins. Cambridge Handbooks in Psychology. Cambridge, UK: Cambridge University Press, S. 727–748.
- Fitzgerald, Sue et al. (2008). „Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers“. In: *Computer Science Education* 18.2, S. 93–116.
- Fitzgerald, Sue et al. (2009). „Debugging from the student perspective“. In: *IEEE Transactions on Education* 53.3, S. 390–396.
- Frerichs, Vera (1999). „Schülervorstellungen und wissenschaftliche Vorstellungen zu den Strukturen und Prozessen der Vererbung: ein Beitrag zur didaktischen Rekonstruktion“. Diss. Universität Oldenburg.
- Friedrich, Cheri et al. (2019). „Escaping the professional silo: an escape room implemented in an interprofessional education curriculum“. In: *Journal of interprofessional care* 33.5, S. 573–575.
- Geldreich, Katharina, Talbot, Mike und Hubwieser, Peter (2018). „Off to new shores: preparing primary school teachers for teaching algorithmics and programming“. In: *Proceedings*

-
- of the 13th Workshop in Primary and Secondary Computing Education. New York, NY, USA: ACM, S. 1–6.
- Gesellschaft für Informatik (2008). „Grundsätze und Standards für die Informatik in der Schule. Bildungsstandards Informatik für die Sekundarstufe I“. In: *LOG IN* 28.150/151.
- Gibson, Paul und O’Kelly, Jackie (2005). „Software engineering as a model of understanding for learning and problem solving“. In: *Proceedings of the first international workshop on Computing education research*. New York, NY, USA: ACM, S. 87–97.
- Gick, Mary L (1986). „Problem-solving strategies“. In: *Educational psychologist* 21.1-2, S. 99–120.
- Gilmore, David J (1991). „Models of debugging“. In: *Acta psychologica* 78.1-3, S. 151–172.
- Göhner, Maximilian und Krell, Moritz (2020). „Qualitative Inhaltsanalyse in naturwissenschaftsdidaktischer Forschung unter Berücksichtigung von Gütekriterien: Ein Review“. In: *Zeitschrift für Didaktik der Naturwissenschaften*.
- Gould, John D (1975). „Some psychological evidence on how people debug computer programs“. In: *International Journal of Man-Machine Studies* 7.2, S. 151–182.
- Gould, John und Drongowski, Paul (1974). „An exploratory study of computer program debugging“. In: *Human Factors* 16.3, S. 258–277.
- Gräsel, Cornelia (2011). „Die Verbreitung von Innovationen als Aufgabe der Unterrichtsforschung“. In: *Stationen Empirischer Bildungsforschung: Traditionslinien und Perspektiven*. Wiesbaden: VS Verlag für Sozialwissenschaften, S. 320–328.
- Gräsel, Cornelia und Parchmann, Ilka (2004). „Implementationsforschung-oder: der steinige Weg, Unterricht zu verändern“. In: *Unterrichtswissenschaft* 32.3, S. 196–214.
- Grillenberger, Andreas (2019). „Von Datenmanagement zu Data Literacy: Informatikdidaktische Aufarbeitung des Gegenstandsbereichs Daten für den allgemeinbildenden Schulunterricht“. Diss. Freie Universität Berlin.
- Grillenberger, Andreas, Przybylla, Mareen und Romeike, Ralf (2016). „Bringing CS innovations to the classroom: A process model of educational reconstruction“. In: *ISSEP 2016*, S. 31.
- Gruber, Hans und Stöger, Heidrun (2011). „Experten-Novizen-Paradigma“. In: *Basiswissen Unterrichtsgestaltung. Bd. 2. Unterrichtsgestaltung als Gegenstand der Wissenschaft*. Hrsg. von Ewald Kiel und Klaus Zierer. Baltmannsweiler: Schneider-Verl. Hohengehren, S. 247–264.
- Gugerty, Leo und Olson, Gary M (1986). „Comprehension differences in debugging by skilled and novice programmers“. In: *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, S. 13–27.
- Guzdial, Mark (2015). „Learner-centered design of computing education: Research on computing for everyone“. In: *Synthesis Lectures on Human-Centered Informatics* 8.6, S. 1–165.
- Hacke, Alexander (2019). „Computer Science Problem Solving in the Escape Game “Room-X”“. In: *Informatics in Schools. New Ideas in School Informatics*. Hrsg. von Sergei N. Pozdniakov und Valentina Dagiè. Cham: Springer International Publishing, S. 281–292.

-
- Hahn, Steffen und Prediger, Susanne (2008). „Bestand und Änderung—Ein Beitrag zur Didaktischen Rekonstruktion der Analysis“. In: *Journal für Mathematik-Didaktik* 29.3-4, S. 163–198.
- Hall, Morgan et al. (2012). „An empirical study of programming bugs in CS1, CS2, and CS3 homework submissions“. In: *Journal of Computing Sciences in Colleges* 28.2, S. 87–94.
- Harms, Ute und Riese, Josef (2018). „Professionelle Kompetenz und Professionswissen“. In: *Theorien in der naturwissenschaftsdidaktischen Forschung*. Hrsg. von Dirk Krüger, Ilka Parchmann und Horst Schecker. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 283–298.
- Helfferich, Cornelia (2019). „Leitfaden- und Experteninterviews“. In: *Handbuch Methoden der empirischen Sozialforschung*. Springer, S. 669–686.
- Henderson, Charles, Beach, Andrea und Finkelstein, Noah (2011). „Facilitating change in undergraduate STEM instructional practices: An analytic review of the literature“. In: *Journal of research in science teaching* 48.8, S. 952–984.
- Heymann, Hans Werner (1996). *Allgemeinbildung und Mathematik*. Weinheim: Bertz.
- Hopper, Grace (Mai 1954). „A Glossary of Computer Terminology“. In: *Computers and Automation* 3.5, S. 16.
- House, Ernest R (1974). *The politics of educational innovation*. Berkeley, CA: McCutchan.
- Hristova, Maria et al. (2003). „Identifying and correcting Java programming errors for introductory computer science students“. In: *ACM SIGCSE Bulletin* 35.1, S. 153–156.
- Hsieh, Hsiu-Fang und Shannon, Sarah E (2005). „Three approaches to qualitative content analysis“. In: *Qualitative health research* 15.9, S. 1277–1288.
- Hubwieser, Peter et al. (2015). „A global snapshot of computer science education in K-12 schools“. In: *Proceedings of the 2015 ITiCSE on working group reports*. New York, NY, USA: ACM, S. 65–83.
- ISO, IEC (2010). „IEEE, Systems and Software Engineering–Vocabulary“. In: *IEEE computer society, Piscataway, NJ* 8, S. 9.
- Jadud, Matthew C (2005). „A first look at novice compilation behaviour using BlueJ“. In: *Computer Science Education* 15.1, S. 25–40.
- Jahn, Dirk (2017). „Entwicklungsforschung aus einer handlungstheoretischen Perspektive: Was Design Based Research von Hannah Arendt lernen könnte.“ In: *EDeR. Educational Design Research* 1.2, S. 1–17.
- Järveläinen, Jonna und Paavilainen-Mäntymäki, Eriikka (2019). „Escape Room as Game-Based Learning Process: Causation-Effectuation Perspective“. In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*. Waikoloa Village, HI, USA: ScholarSpace 2019.
- Jayathirtha, Gayithri, Fields, Deborah und Kafai, Yasmin (2020). „Pair Debugging of Electronic Textiles Projects: Analyzing Think-Aloud Protocols for High School Students’ Strategies and Practices While Problem Solving“. In: *ICLS 2018 Proceedings*. International Society of the Learning Sciences (ISLS).

-
- Jeffries, Robin (1982). „A comparison of the debugging behavior of expert and novice programmers“. In: *Proceedings of AERA annual meeting*. New York, NY, USA.
- Johnson, W Lewis et al. (1983). *Bug catalogue: I*. Techn. Ber. Yale University.
- Jonassen, David H (2000). „Toward a design theory of problem solving“. In: *Educational technology research and development* 48.4, S. 63–85.
- Jonassen, David H und Hung, Woei (2006). „Learning to troubleshoot: A new theory-based design architecture“. In: *Educational Psychology Review* 18.1, S. 77–114.
- Kaddoura, Mahmoud (2013). „Think pair share: A teaching learning strategy to enhance students' critical thinking.“ In: *Educational Research Quarterly* 36.4, S. 3–24.
- Kapur, Manu (2008). „Productive failure“. In: *Cognition and instruction* 26.3, S. 379–424.
- Kattmann, Ulrich (2007). „Didaktische Rekonstruktion – eine praktische Theorie“. In: *Theorien in der biogiedidaktischen Forschung – Ein Handbuch für Lehramtsstudenten und Doktoranden*. Berlin Heidelberg: Springer, S. 93–104.
- Kattmann, Ulrich et al. (1997). „Das Modell der Didaktischen Rekonstruktion“. In: *Zeitschrift für Didaktik der Naturwissenschaften* 3.3, S. 3–18.
- Katz, Irvin und Anderson, John (1987). „Debugging: An analysis of bug-location strategies“. In: *Human-Computer Interaction* 3.4, S. 351–399.
- Kazimoglu, Cagin et al. (2012). „Learning programming at the computational thinking level via digital game-play“. In: *Procedia Computer Science* 9, S. 522–531.
- Kessler, Claudius M und Anderson, John R (1986). „A model of novice debugging in LISP“. In: *Proceedings of the First Workshop on Empirical Studies of Programmers*. Norwood, NJ, USA: Ablex, S. 198–212.
- Kim, ChanMin et al. (2018). „Debugging during block-based programming“. In: *Instructional Science* 46.5, S. 767–787.
- Kinnunen, Paivi und Simon, Beth (2010). „Experiencing Programming Assignments in CS1: The Emotional Toll“. In: *Proceedings of the Sixth International Workshop on Computing Education Research*. ICER '10. New York, NY, USA: ACM, S. 77–86.
- Klafki, Wolfgang (1993). *Neue Studien zur Bildungstheorie und Didaktik: zeitgemäße Allgemeinbildung und kritisch-konstruktive Didaktik*. Weinheim: Beltz.
- Klahr, David und Carver, Sharon (1988). „Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer“. In: *Cognitive Psychology* 20.3, S. 362–404.
- Klein, Bernd (2013). *Einführung in Python 3*. München: Carl Hanser.
- Klein, Gary (1989). „Recognition-primed decisions“. In: *Advances in man-machine system research* 5, S. 47–92.
- Klima, Robert und Selberherr, Siegfried (2010). *Programmieren in C*. Vienna: Springer.
- Knobelsdorf, Maria und Schulte, Carsten (2005). „Computer Biographies-A Biographical Research Perspective on Computer Usage and Attitudes Toward Informatics“. In: *Proc. of the Koli Calling 2005 Conference on Computer Science Education*. New York, NY, USA: ACM, S. 139–142.

-
- Ko, Andrew J. und Fincher, Sally A. (2019). „A Study Design Process“. In: *The Cambridge Handbook of Computing Education Research*. Hrsg. von Sally A. Fincher und Anthony V.Editors Robins. Cambridge, UK: Cambridge University Press, S. 81–101.
- Ko, Andrew und Myers, Brad A (2005). „A framework and methodology for studying the causes of software errors in programming systems“. In: *Journal of Visual Languages & Computing* 16.1-2, S. 41–84.
- Kolikant, Yifat Ben-David (2001). „Gardeners and cinema tickets: High school students' preconceptions of concurrency“. In: *Computer Science Education* 11.3, S. 221–245.
- Kölling, Michael (2000). *The BlueJ Tutorial*. Techn. Ber. Monash University.
- Kölling, Michael et al. (2003). „The BlueJ system and its pedagogy“. In: *Computer Science Education* 13.4, S. 249–268.
- Konradt, Udo (1995). „Strategies of failure diagnosis in computer-controlled manufacturing systems: empirical analysis and implications for the design of adaptive decision support systems“. In: *International journal of human-computer studies* 43.4, S. 503–521.
- Krebs, Dagmar und Menold, Natalja (2014). „Gütekriterien quantitativer Sozialforschung“. In: *Handbuch Methoden der empirischen Sozialforschung*. Wiesbaden: Springer Fachmedien, S. 425–438.
- Krienke, Rainer (2002). *Programmieren in Perl*. München: Carl Hanser.
- Kumar, Amruth (2002). „Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug c++ programs“. In: *International Conference on Intelligent Tutoring Systems*. Heidelberg: Springer, S. 792–801.
- Kundu, Arup und Ghose, Aditi (2016). „The relationship between attitude and self-efficacy in mathematics among higher secondary students“. In: *Journal of Humanities and Social Science* 21.4, S. 25–31.
- Kunter, Mareike et al. (2011). „Die Entwicklung professioneller Kompetenz von Lehrkräften“. In: *Professionelle Kompetenz von Lehrkräften. Ergebnisse des Forschungsprogramms COACTIV*. Bd. 1. Münster: Waxmann, S. 55–68.
- Lahtinen, Essi, Ala-Mutka, Kirsti und Järvinen, Hannu-Matti (2005). „A study of the difficulties of novice programmers“. In: *Acm Sigcse Bulletin* 37.3, S. 14–18.
- Lee, Greg C und Wu, Jackie C (1999). „Debug it: A debugging practicing system“. In: *Computers & Education* 32.2, S. 165–179.
- Lee, Michael et al. (2014). „Principles of a debugging-first puzzle game for computing education“. In: *2014 IEEE symposium on visual languages and human-centric computing (VL/HCC)*. IEEE, S. 57–64.
- Lee, Michael und Ko, Andrew (2011). „Personifying programming tool feedback improves novice programmers' learning“. In: *Proceedings of the seventh international workshop on Computing education research*. New York, NY, USA: ACM, S. 109–116.
- Lewandowski, Gary et al. (2007). „Commonsense computing (episode 3) concurrency and concert tickets“. In: *Proceedings of the third international workshop on Computing education research*. New York, NY, USA: ACM, S. 133–144.

-
- Li, Chen et al. (2019). „Towards a Framework for Teaching Debugging“. In: *Proceedings of the Twenty-First Australasian Computing Education Conference*. New York, NY, USA: ACM, S. 79–86.
- Lions, Jacques-Louis et al. (1996). *Ariane 5 flight 501 failure report by the inquiry board*.
- Lipowsky, Frank (2004). „Was macht Fortbildungen für Lehrkräfte erfolgreich“. In: *Die Deutsche Schule* 96.4, S. 462–479.
- Lipowsky, Frank und Rzejak, Daniela (2017). „Fortbildungen für Lehrkräfte wirksam gestalten–erfolgsverprechende Wege und Konzepte aus Sicht der empirischen Bildungsforschung“. In: *Bildung und Erziehung* 70.4, S. 379–400.
- Lister, Raymond et al. (2004). „A multi-national study of reading and tracing skills in novice programmers“. In: *ACM SIGCSE Bulletin* 36.4, S. 119–150.
- Liu, Zhongxiu et al. (2017). „Understanding problem solving behavior of 6–8 graders in a debugging game“. In: *Computer Science Education* 27.1, S. 1–29.
- Long, Ju (2007). „Just For Fun: using programming games in software programming training and education“. In: *Journal of Information Technology Education: Research* 6.1, S. 279–290.
- Lowe, T. (2019). „Debugging: The key to unlocking the mind of a novice programmer?“ In: *2019 IEEE Frontiers in Education Conference (FIE)*. Covington, KY, USA: IEEE, S. 1–9.
- Lui, Debora et al. (2017). „Learning by fixing and designing problems: A reconstruction kit for debugging e-textiles“. In: *Proceedings of the 7th Annual Conference on Creativity and Fabrication in Education*. New York, NY, USA: ACM, S. 1–8.
- Magnusson, Shirley, Krajcik, Joseph und Borko, Hilda (1999). „Nature, sources, and development of pedagogical content knowledge for science teaching“. In: *Examining pedagogical content knowledge*. Dordrecht: Springer, S. 95–132.
- Malmi, Lauri, Utting, Ian und Ko, Andrew J. (2019). „Tools and Environments“. In: *The Cambridge Handbook of Computing Education Research*. Hrsg. von Sally A. Fincher und Anthony V. Editors Robins. Cambridge, UK: Cambridge University Press, S. 639–662.
- Mayring, Philipp (1985). „Qualitative Inhaltsanalyse“. In: *Qualitative Forschung in der Psychologie : Grundfragen, Verfahrensweisen, Anwendungsfelder*. Hrsg. von Gerd Jüttemann. Weinheim: Beltz, S. 187–211.
- (2001). „Combination and integration of qualitative and quantitative analysis“. In: *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*. Bd. 2. 1.
 - (2002). *Einführung in die qualitative Sozialforschung*. Weinheim: Beltz.
 - (2014). *Qualitative content analysis: theoretical foundation, basic procedures and software solution*. 2014. Klagenfurt: GESIS–Leibniz Institute for the Social Sciences.
- Mayring, Philipp, Gläser-Zikuda, Michaela und Ziegelbauer, Sascha (2005). „Auswertung von Videoaufnahmen mit Hilfe der Qualitativen Inhaltsanalyse–ein Beispiel aus der Unterrichtsforschung“. In: *MedienPädagogik* 9, S. 1–17.

-
- McCall, Davin und Kölling, Michael (2014). „Meaningful categorisation of novice programmer errors“. In: *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. Madrid, Spain: IEEE, S. 1–8.
- McCauley, Renee et al. (2008). „Debugging: a review of the literature from an educational perspective“. In: *Computer Science Education* 18.2, S. 67–92.
- Metzger, Robert C (2004). *Debugging by thinking: A multidisciplinary approach*. Burlington, MA: Elsevier Digital Press.
- Meyer, Jeanine (2018). *Programming 101: The How and Why of Programming Revealed Using the Processing Programming Language*. Berkeley, CA, USA: Apress.
- Michaeli, Tilman und Romeike, Ralf (2017). „Addressing teaching practices regarding software quality: Testing and debugging in the classroom“. In: *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. New York, NY, USA: ACM, S. 105–106.
- (2019a). „Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study“. In: *2019 IEEE Global Engineering Education Conference (EDUCON)*. Dubai, VAE: IEEE, S. 1030–1038.
 - (2019b). „Debuggen im Unterricht–Ein systematisches Vorgehen macht den Unterschied“. In: *INFOS 2019, 18. GI-Fachtagung Informatik und Schule*. Bonn: Gesellschaft für Informatik, S. 129–138.
 - (2019c). „Improving Debugging Skills in the Classroom: The Effects of Teaching a Systematic Debugging Process“. In: *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*. New York, NY, USA: ACM, S. 1–7.
- Miljanovic, Michael und Bradbury, Jeremy (2017). „Robobug: a serious game for learning debugging techniques“. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. New York, NY, USA: ACM, S. 93–100.
- Miller, Lance A (1981). „Natural language programming: Styles, strategies, and contrasts“. In: *IBM Systems Journal* 20.2, S. 184–215.
- Morris, Nancy M und Rouse, William B (1985). „Review and evaluation of empirical research in troubleshooting“. In: *Human factors* 27.5, S. 503–530.
- Mueller, John Paul (2018). *Python programmieren lernen für Dummies*. 2. Aufl. Weinheim: Wiley.
- Munson, Jonathan P und Schilling, Elizabeth A (2016). „Analyzing novice programmers’ response to compiler error messages“. In: *Journal of Computing Sciences in Colleges* 31.3, S. 53–61.
- Murphy, Laurie et al. (2008). „Debugging: the good, the bad, and the quirky—a qualitative analysis of novices’ strategies“. In: *ACM SIGCSE Bulletin* 40.1, S. 163–167.
- Nanja, Murthi und Cook, Curtis R (1987). „An analysis of the on-line debugging process“. In: *Empirical studies of programmers: second workshop*. Ablex. Norwood, NJ, USA, S. 172–184.

-
- Nicholson, Scott (2015). *Peeking behind the locked door: A survey of escape room facilities*. Techn. Ber. Wilfrid Laurier University.
- (2018). „Creating engaging escape rooms for the classroom“. In: *Childhood Education* 94.1, S. 44–49.
- Niemeyer, Gerhard (2020). *Einführung in das Programmieren in PASCAL*. Berlin, Boston: De Gruyter.
- Niermann, Anne (2016). *Professionswissen von Lehrerinnen und Lehrern des Mathematik- und Sachunterrichts: „... man muss schon von der Sache wissen.“* Bad Heilbrunn: Julius Klinkhardt.
- Onorato, Lisa und Schvaneveldt, Roger W (1987). „Programmer-nonprogrammer differences in specifying procedures to people and computers“. In: *Journal of Systems and Software* 7.4, S. 357–369.
- Oser, Fritz, Hascher, Tina und Spychiger, Maria (1999). „Lernen aus Fehlern Zur Psychologie des „negativen“ Wissens“. In: *Fehlerwelten*. Hrsg. von Wolfgang Althof. Wiesbaden: VS Verlag für Sozialwissenschaften, S. 11–41.
- Padmanabhan, Tattamangalam R (2016). *Programming with python*. Singapore: Springer.
- Pajares, Frank (1996). „Self-efficacy beliefs and mathematical problem-solving of gifted students“. In: *Contemporary educational psychology* 21.4, S. 325–344.
- Pan, Rui, Lo, Henry und Neustaedter, Carman (2017). „Collaboration, awareness, and communication in real-life escape rooms“. In: *Proceedings of the 2017 Conference on Designing Interactive Systems*. ACM. New York, NY, USA, S. 1353–1364.
- Papert, Seymour (1980). *Mindstorms; Children, Computers and Powerful Ideas*. New York, NY, USA: Basic Book.
- Passey, Don (2017). „Computer science (CS) in the compulsory education curriculum: Implications for future research“. In: *Education and Information Technologies* 22.2, S. 421–443.
- Passig, Kathrin und Jander, Johannes (2013). *Weniger schlecht programmieren*. Sebastopol, CA, USA: O’Reilly.
- Patton, Michael Quinn (2002). *Qualitative research and evaluation methods*. Thousand Oaks: Sage Publications.
- Perkins, David N et al. (1986). „Conditions of learning in novice programmers“. In: *Journal of Educational Computing Research* 2.1, S. 37–55.
- Perkins, David und Martin, Fay (1985). *Fragile Knowledge and Neglected Strategies in Novice Programmers*. IR85-22. Techn. Ber. Educational Technology Center.
- Perscheid, Michael et al. (2017). „Studying the advancement in debugging practice of professional software developers“. In: *Software Quality Journal* 25.1, S. 83–110.
- Peterson, C, Maier, SF und Seligman, ME (1993). *Learned helplessness: A theory for the age of personal control*. New York: Oxford University Press.
- Phillips, Denis C (1995). „The good, the bad, and the ugly: The many faces of constructivism“. In: *Educational researcher* 24.7, S. 5–12.

-
- Prenzel, I et al. (2010). „Transfer und Transferforschung im Bildungsbereich“. In: *Zeitschrift für Erziehungswissenschaft* 13.1, S. 3–5.
- Prenzel, Manfred und Doll, Jörg (2002). *Bildungsqualität von Schule: Schulische und außerschulische Bedingungen mathematischer, naturwissenschaftlicher und überfachlicher Kompetenzen*. Weinheim: Beltz.
- Przybylla, Mareen (2018). „From Embedded Systems to Physical Computing: Challenges of the “Digital World” in Secondary Computer Science Education“. Doctoral Thesis. Universität Potsdam.
- Rädiker, Stefan und Kuckartz, Udo (2019). „Intercoder-Übereinstimmung analysieren“. In: *Analyse qualitativer Daten mit MAXQDA*. Wiesbaden: Springer Fachmedien, S. 287–303.
- Rasch, Björn et al. (2010). *Quantitative Methoden 2: Einführung in die Statistik für Psychologen und Sozialwissenschaftler*. 3. Aufl. Berlin: Springer.
- Rasmussen, Jens und Jensen, Aage (1974). „Mental procedures in real-life tasks: A case study of electronic trouble shooting“. In: *Ergonomics* 17.3, S. 293–307.
- Ratz, Dietmar et al. (2018). *Grundkurs Programmieren in Java*. München: Carl Hanser.
- Reed, Nancy und Johnson, Paul (1993). „Analysis of expert reasoning in hardware diagnosis“. In: *International Journal of Man-Machine Studies* 38.2, S. 251–280.
- Reinders, Heinz (2016). *Qualitative Interviews mit Jugendlichen führen: Ein Leitfaden*. Berlin: De Gruyter.
- Reiss, Kristina und Ufer, Stefan (2009). „Fachdidaktische Forschung im Rahmen der Bildungsforschung. Eine Diskussion wesentlicher Aspekte am Beispiel der Mathematikdidaktik“. In: *Handbuch Bildungsforschung*. Hrsg. von Rudolf Tippelt und Bernhard Schmidt. Wiesbaden: VS Verlag für Sozialwissenschaften, S. 199–213.
- Rich, Kathryn M et al. (2019). „A K-8 Debugging Learning Trajectory Derived from Research Literature“. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, S. 745–751.
- Richardson, Virginia und Placier, Peggy (2001). „Teacher change“. In: *Handbook of Research on Teaching*. Hrsg. von Virginia Richardson. 4. Aufl. Washington, DC, USA: American Educational Research Association, S. 905–947.
- Röhner, Gerhard et al. (2016). „Bildungsstandards Informatik für die Sekundarstufe II“. In: *Beilage zu LOG IN* 183/184, S. 88.
- Sande, Warren und Sande, Carter (2014). *Hello World!: Programmieren für Kids und andere Anfänger*. München: Carl Hanser.
- Sander, Elke (2018). „Natur, Mensch und "biologisches Gleichgewicht": didaktische Rekonstruktion der Sichtweisen von Lernenden und Wissenschaftlern“. Diss. Universität Oldenburg.
- Schaafstal, Alma, Schraagen, Jan Maarten und Van Berl, Marcel (2000). „Cognitive task analysis and innovation of training: The case of structured troubleshooting“. In: *Human factors* 42.1, S. 75–86.

-
- Schmalfeldt, Thomas (2019). „Einsatz von Skill Cards und Story Cards für einen kreativitätsfördernden Informatikunterricht auf der Sekundarstufe I“. In: *Informatik für alle*. Bonn: Gesellschaft für Informatik.
- Schmidt, Silvia (2011). „Didaktische Rekonstruktion des Basiskonzepts Stoff-Teilchen für den Anfangsunterricht nach Chemie im Kontext“. Diss. Universität Oldenburg.
- Schreier, Margrit (2014). „Varianten qualitativer Inhaltsanalyse: ein wegweiser im dickicht der Begrifflichkeiten“. In: *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*. Bd. 15. 1. DEU, S. 27.
- Schulqualität und Bildungsforschung, Staatsinstitut für (2009). *Lehrplan des achtjährigen Gymnasiums in Bayern*.
- Searle, Kristin, Litts, Breanne und Kafai, Yasmin (2018). „Debugging open-ended designs: High school students' perceptions of failure and success in an electronic textiles design activity“. In: *Thinking Skills and Creativity* 30, S. 125–134.
- Seegerer, Stefan, Michaeli, Tilman und Romeike, Ralf (2019). „Informatik für alle-Eine Analyse von Argumenten und Argumentationsschemata für das Schulfach Informatik“. In: *INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik–Informatik für Gesellschaft*, S. 617–630.
- Seehorn, Deborah et al. (2011). *CSTA K–12 Computer Science Standards: Revised 2011*. New York, NY, USA.
- Shapiro, Fred R (1987). „Etymology of the computer bug: History and folklore“. In: *American Speech* 62.4, S. 376–378.
- Shulman, Lee (1987). „Knowledge and teaching: Foundations of the new reform“. In: *Harvard educational review* 57.1, S. 1–23.
- Simon, Beth et al. (2006). „Commonsense computing: what students know before we teach (episode 1: sorting)“. In: *Proceedings of the second international workshop on Computing education research*. New York, NY, USA: ACM, S. 29–40.
- Simon, Beth et al. (2008). „Common sense computing (episode 4): Debugging“. In: *Computer Science Education* 18.2, S. 117–133.
- Sipitakiat, Arnan und Nusen, Nusarin (2012). „Robo-Blocks: designing debugging abilities in a tangible programming system for early primary school children“. In: *Proceedings of the 11th International Conference on Interaction Design and Children*. New York, NY, USA: ACM, S. 98–105.
- Soloway, W Lewis Johnson-Elliot und Johnson, WL (1984). „Intention-based diagnosis of programming errors“. In: *Proceedings of the 5th national conference on artificial intelligence*. Austin, TX, USA: AAAI, S. 162–168.
- Spinellis, Diomidis (2018). „Modern debugging: the art of finding a needle in a haystack“. In: *Communications of the ACM* 61.11, S. 124–134.
- Spohrer, James G. und Soloway, Elliot (1986). „Analyzing the High Frequency Bugs in Novice Programs“. In: *Papers Presented at the First Workshop on Empirical Studies of Pro-*

-
- grammers on *Empirical Studies of Programmers*. Washington, D.C., USA: Ablex Publishing Corp., S. 230–251.
- Staatsinstitut für Bildungsforschung München (2008). *Handreichung Informatik am Naturwissenschaftlich-technologischen Gymnasium, Jahrgangsstufe 10*.
- Stackoverflow (2019). *Developer Survey Results 2019*. URL: <https://insights.stackoverflow.com/survey/2019>.
- Stamouli, Ioanna und Huggard, Meriel (2006). „Object oriented programming and program correctness: the students’ perspective“. In: *Proceedings of the second international workshop on Computing education research*. New York, NY, USA: ACM, S. 109–118.
- Strübing, Jörg et al. (2018). „Gütekriterien qualitativer Sozialforschung. Ein Diskussionsanstoß“. In: *Zeitschrift für Soziologie* 47.2, S. 83–100.
- Tamir, Pinchas (1988). „Subject matter and related pedagogical knowledge in teacher education“. In: *Teaching and teacher education* 4.2, S. 99–110.
- Uebornickel, Falk und Brenner, Walter (2016). „Design thinking“. In: *Business Innovation: Das St. Galler Modell*. Wiesbaden: Springer Fachmedien, S. 243–265.
- Ullenboom, Christian (2020). *Java ist auch eine Insel*. 15. Aufl. Bonn: Rheinwerk.
- van Dijk, Esther M. und Kattmann, Ulrich (2007). „A research model for the study of science teachers’ PCK and improving teacher education“. In: *Teaching and Teacher Education* 23.6, S. 885–897.
- VanDeGrift, Tammy et al. (2010). „Commonsense computing (episode 6) logic is harder than pie“. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. New York, NY, USA: ACM, S. 76–85.
- VanLehn, Kurt et al. (2003). „Why do only some events cause learning during human tutoring?“ In: *Cognition and Instruction* 21.3, S. 209–249.
- Venables, Anne, Tan, Grace und Lister, Raymond (2009). „A closer look at tracing, explaining and code writing skills in the novice programmer“. In: *Proceedings of the fifth international workshop on Computing education research workshop*. New York, NY, USA: ACM, S. 117–128.
- Vessey, Iris (1985). „Expertise in debugging computer programs: A process analysis“. In: *International Journal of Man-Machine Studies* 23.5, S. 459–494.
- Vogel, Sara, Santo, Rafi und Ching, Dixie (2017). „Visions of computer science education: Unpacking arguments for and projected impacts of CS4All initiatives“. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, S. 609–614.
- Vörös, Alpár István Vita und Sárközi, Zsuzsa (2017). „Physics escape room as an educational tool“. In: *AIP Conference Proceedings*. Bd. 1916. 1. AIP Publishing, S. 050002.
- Wahl, Diethelm (2013). *Lernumgebungen erfolgreich gestalten: vom trägen Wissen zum kompetenten Handeln*. Bad Heilbrunn: Julius Klinkhardt.
- Weiser, Mark (1984). „Program slicing“. In: *IEEE Transactions on software engineering* 4, S. 352–357.

-
- Wertz, Harald (1982). „Stereotyped program debugging: an aid for novice programmers“. In: *International Journal of Man-Machine Studies* 16.4, S. 379–392.
- Williams, Patrick (2018). „Using escape room-like puzzles to teach undergraduate students effective and efficient group process skills“. In: *2018 IEEE Integrated STEM Education Conference (ISEC)*. Princeton, NJ, USA: IEEE, S. 254–257.
- Wing, Jeanette (2011). „Research notebook: Computational thinking—What and why“. In: *The link magazine* 6, S. 1–32.
- Wing, Jeannette M (2006). „Computational thinking“. In: *Communications of the ACM* 49.3, S. 33–35.
- Yadav, Aman et al. (2011). „Introducing Computational Thinking in Education Courses“. In: *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*. SIGCSE '11. New York, NY, USA: ACM, S. 465–470.
- Yen, Ching-Zon, Wu, Ping-Huang und Lin, Ching-Fang (2012). „Analysis of experts' and novices' thinking process in program debugging“. In: *Engaging Learners Through Emerging Technologies*. Berlin, Heidelberg: Springer, S. 122–134.
- Yin, Robert (2003). *Case study research: Design and methods*. 3. Aufl. London, UK: Sage.
- Yoon, Byung-do und Garcia, Oscar (1996). „Hierarchical problem-solving in software debugging“. In: *Proceedings Third Annual Symposium on Human Interaction with Complex Systems*. HICS'96. Dayton, OH, USA: IEEE, S. 147.
- (1998). „Cognitive activities and support in debugging“. In: *Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems*. Dayton, OH, USA: IEEE, S. 160–169.
- Zehetmeier, Daniela et al. (2015). „Development of a classification scheme for errors observed in the process of computer programming education“. In: *1st international conferece on higher education advances (HEAD'15)*. Valencia, Spain: Editorial Universitat Politècnica de València, S. 475–484.
- Zeller, Andreas (2009). *Why programs fail: a guide to systematic debugging*. Amsterdam, The Netherlands: Elsevier.

Abbildungsverzeichnis

1.1	Fachdidaktisches Triplet nach <i>Kattmann et al. (1997)</i>	8
1.2	Didaktische Rekonstruktion für die Informatik nach <i>Diethelm et al. (2011)</i>	11
1.3	Adaptiertes Modell der Didaktischen Rekonstruktion für die Informatik (<i>Grillenberger, Przybylla und Romeike, 2016</i>)	13
1.4	Struktur dieser Arbeit nach dem Modell der didaktischen Rekonstruktion	16
2.1	Modell des Debuggingprozesses nach <i>Gould (1975)</i>	24
2.2	Modell des Debuggingprozesses nach <i>Gilmore (1991)</i>	25
2.3	<i>Comprehension</i> -Strategie nach <i>Yoon und Garcia (1998)</i>	27
2.4	<i>Isolation</i> -Strategie nach <i>Yoon und Garcia (1998)</i>	27
2.5	Modell des Debuggingprozesses nach <i>Araki, Furukawa und Cheng (1991)</i>	28
2.6	Modell des Debuggingprozesses nach <i>Spinellis (2018)</i> (Ausschnitt)	28
2.7	Modell des Debuggingprozesses in LOGO nach <i>Klahr und Carver (1988)</i>	30
2.8	Debuggingstrategien der Studierenden bei <i>Fitzgerald et al. (2009)</i>	35
2.9	Debuggingvorgehen nach <i>Böttcher et al. (2016)</i>	52
2.10	Schrittweises Debuggingvorgehen nach <i>Carver und Risinger (1987)</i>	53
2.11	Exemplarische Aufgabe zum Messen der Transferleistung (<i>Klahr und Carver, 1988</i>)	57
3.1	Modell der Debuggingfähigkeiten für Novizen	73
3.2	Learning Trajectory (<i>Rich et al., 2019</i>)	75
5.1	Problemlöseprozess nach <i>Gick (1986)</i>	100
5.2	Visualisierung des Troubleshooting-Prozesses von <i>Dounas-Frazer et al. (2016)</i> , basierend auf der <i>cognitive task analysis</i> nach <i>Schaafstal, Schraagen und Van Berl (2000)</i>	102
6.1	Kabelsalat	115
6.2	Telefon abhören	116
6.3	Tal der Könige	116
6.4	Mr. X	117
7.1	Plakat zum vermittelten Vorgehen und Skill-Cards	147
7.2	Exemplarische ausgefüllte Skill-Card „Debugger“	148
8.1	Untersuchungsdesign	153

9.1	Modell professioneller Kompetenz von Lehrkräften nach <i>Blömeke, Suhl und Döhrmann (2012)</i>	162
9.2	Didaktische Rekonstruktion für die Lehrkräftebildung nach <i>van Dijk und Kattmann (2007)</i>	164
9.3	Persona aus dem Workshop	169

Tabellenverzeichnis

2.1	Übersicht der Ergebnisse zum Vorgehen von Novizen	33
3.1	Resultierendes Kategoriensystem der inhaltlich-strukturierenden qualitativen Inhaltsanalyse.	69
4.1	Kategoriensystem für die qualitative Inhaltsanalyse	82
6.1	Übersicht der resultierenden Aufgaben	117
6.2	Auszug des Kategoriensystem für die strukturierende qualitative Inhaltsanalyse (Schritt 1)	119
8.1	Einfluss auf Selbstwirksamkeitserwartungen	155
8.2	Einfluss auf Debuggingleistung	156
8.3	Mittelwerte für <i>Aufgaben gut lösbar</i>	156

Anhang

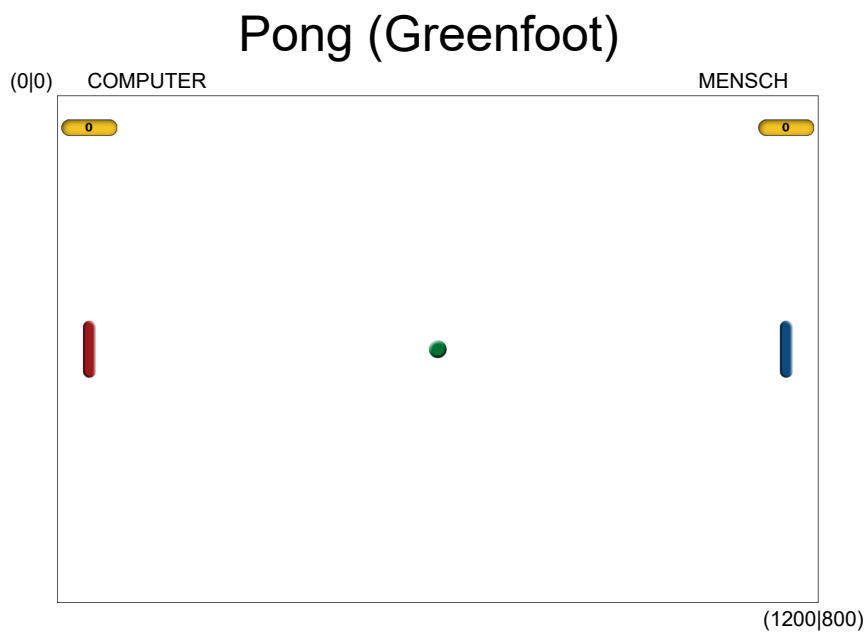
Anhang A: Interviewleitfaden Perspektive der Lehrkräfte

Leitfrage/Erzählaufforderung	Stichworte – nur erfragen wenn nicht von allein thematisiert	Nachfragen mit obligatorischer Formulierung	Erläuterung
<p>Wie liefen Ihre letzten Programmierphasen im Unterricht ab, die Sie im Kopf haben?</p>	<p>Typische Fehler und Probleme</p> <p><i>(Kontext klären - welche Beispiele? - welche Klasse?)</i></p>	<p>Was passiert, wenn eine Schülerin bzw. ein Schüler ein Problem oder Fehler in seinem Code hat? Wie reagieren Schülerinnen bzw. Schüler und Lehrkraft?</p> <p>Insbesondere welche Fragen stellen die Schülerinnen und Schüler, wenn die Lehrkraft zum PC kommt?</p> <p>Unterscheidet eine Schülerin bzw. ein Schüler zwischen dem Kompilieren und der „Korrektheit eines Programmes“, bzw. ab wann?</p> <p>Wie läuft es ab, wenn eine Schülerin bzw. ein Schüler in einer Programmierphase im Unterricht seinen Code „fertig“ gestellt hat?</p> <p>Wie und wann wird in Projektarbeit die korrekte Funktionsweise des Programms durch die Schülerinnen und Schüler sichergestellt?</p>	<p>Erfahrungen und Probleme aus dem Unterricht?</p>
<p>Wie lösen die Schülerinnen und Schüler ihre Probleme beim Programmieren? Was geben Sie ihnen dazu an die Hand?</p>	<p>Vorgehen beim Debuggen, Debugginstrategien (Auskommentieren, Undo, Tracing und Tracetabellen, ...), typische Fehler, Gebrauch des Debuggers</p> <p>Testen, „ausprobieren“, „erproben“, Inspektor in BlueJ</p>	<p>Wie und wann werden die jeweiligen Inhalte/Vorgehensweisen vermittelt?</p> <p>Werden diese Inhalte/Vorgehensweisen systematisch über den Verlauf des Schuljahres ausgebaut?</p>	<p>Was wird wie vermittelt?</p>

Wie bewerten Sie das? Sind Sie zufrieden damit?		Vermitteln Sie die Inhalte/Vorgehensweisen schon immer? Was war Ihre Motivation, damit anzufangen/ die Inhalte auszubauen? Würden sie gerne mehr dazu in Ihren Unterricht integrieren?	Bilanzieren und Bewerten: Warum bzw. Warum nicht
Was fehlt für den Unterricht?	Materialien, Aktivitäten zum Üben Werkzeugsupport		Missing/Wanted

Anhang B: Unterrichtsmaterialien

B.1 Einführung des systematischen Vorgehens in Greenfoot



Im bereitgestellten Verzeichnis findest du mehrere Prototypen eines Pong-Spiels. In jedem Prototypen wurde eine weitere *User Story* umgesetzt.




Die Bewegung des Balles wird durch eine Änderung der x- und y-Koordinate des Balles realisiert. In jedem Durchlauf der `act()`-Methode werden die beiden Koordinaten des Balles um ein `deltaX` bzw. `deltaY` (in positiver oder negativer Richtung) angepasst:




Im Folgenden findest du die jeweiligen *User Stories* sowie die jeweiligen Klassenkarten. Leider haben sich in jedem Prototyp Fehler eingeschlichen.

Prototyp 1: Ball

 Ball
- deltaX: int - deltaY: int
+ act(): void + ueberpruefeWand(): void + ueberpruefeTor(): void

User Story: Der Ball bewegt sich und prallt von allen Wänden ab.

 **Beschreibe die Fehler, und wie du sie gefunden hast!**

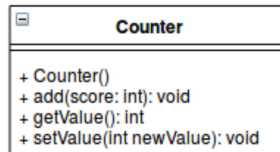
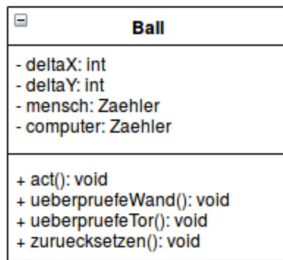
1. Fehlerbeschreibung: _____

so habe ich den Fehler gefunden: _____


2. Fehlerbeschreibung: _____

so habe ich den Fehler gefunden: _____

Prototyp 2: Punkte



User Story: Wenn der Ball ein "Tor" trifft, werden die Punkte des jeweils anderen Spielers hochgezählt und der Ball wird in die Mitte des Spielfeldes zurückgesetzt.

 **Beschreibe die Fehler, und wie du sie gefunden hast!**

1. Fehlerbeschreibung: _____

so habe ich den Fehler gefunden: _____

2. Fehlerbeschreibung: _____

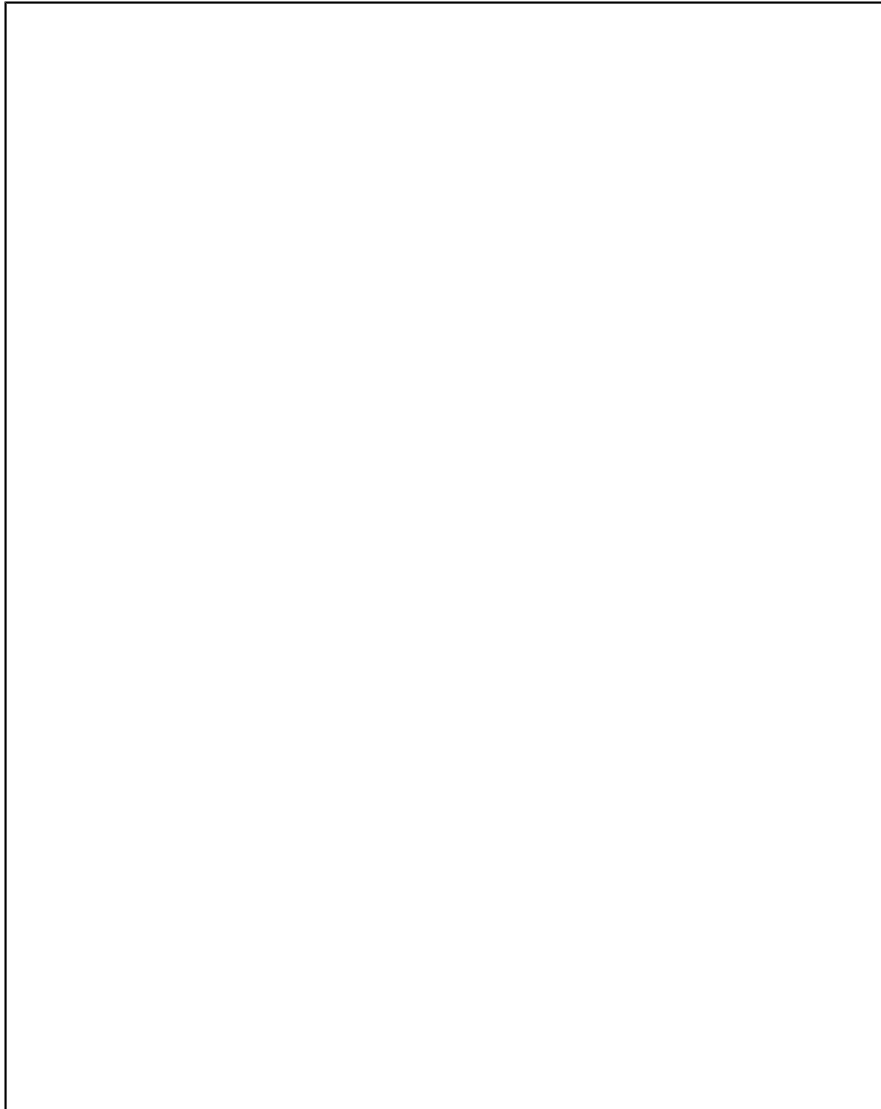
so habe ich den Fehler gefunden: _____

3. Fehlerbeschreibung: _____

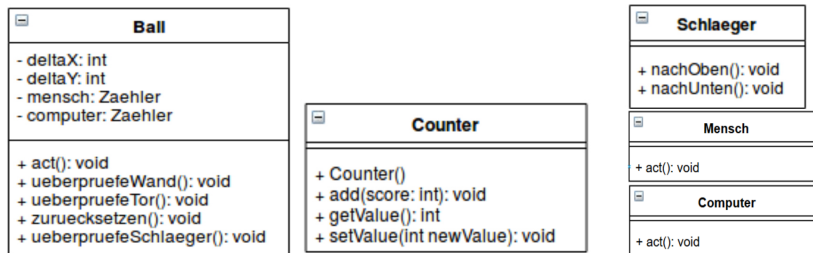
so habe ich den Fehler gefunden: _____

Überlege dir, wie man die bisher gefundenen Fehler in Gruppen einteilen könnte. Gib jeder Gruppe einen Namen und beschreibe sie in Stichpunkten!

Hinweis: Überlege dir, wie du auf den jeweiligen Fehler aufmerksam geworden bist.

A large, empty rectangular box with a thin black border, intended for students to write their answers to the task above. The box is oriented vertically and occupies most of the lower half of the page.

Prototyp 3: Schläger



User Story: Der rechte Schläger lässt sich nach oben (C) und unten (C) bewegen. Der linke Schläger wird vom Computer gesteuert, der den Schläger immer auf die Höhe des Balles bewegt. Wenn der Ball auf den Schläger trifft, prallt er ab.

✎ Beschreibe die Fehler, und wie du sie gefunden hast!

1. Fehlermeldung: _____

hier war der Fehler und so habe ich ihn behoben:

2. erwartetes Verhalten: _____

tatsächliches Verhalten: _____

Vermutung: _____

hier war der Fehler und so habe ich ihn behoben:

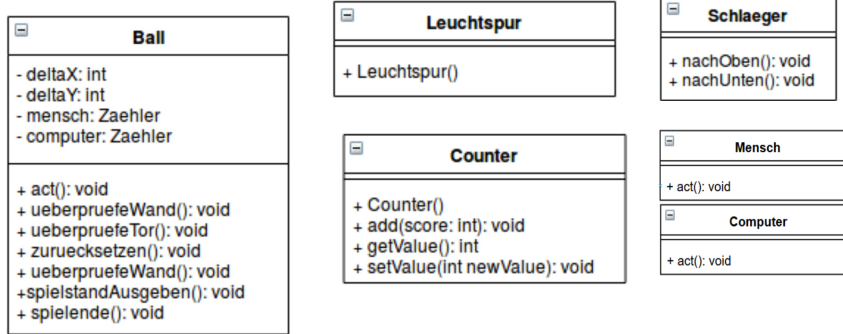
3. erwartetes Verhalten: _____

tatsächliches Verhalten: _____

Vermutung: _____

hier war der Fehler und so habe ich ihn behoben:

Prototyp 4: Spielende und Leuchtspur



User Story: Sobald ein Spieler 3 Punkte erzielt hat, wird das Spiel beendet und ein Endbildschirm angezeigt. Der Ball zieht eine Leuchtspur hinter sich her.

✎ Beschreibe die gefundenen Fehler, und wie du sie gefunden hast!

1. Fehlermeldung: _____

hier war der Fehler und so habe ich ihn behoben:

2. Fehlermeldung: _____

Vermutung: _____

hier war der Fehler und so habe ich ihn behoben:

3. erwartetes Verhalten: _____

tatsächliches Verhalten: _____

Vermutung: _____

hier war der Fehler und so habe ich ihn behoben:

B.2 Einführung des Debuggers in BlueJ

Arbeitsblatt Debugger (BlueJ)

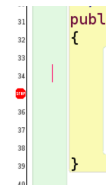
Du willst mit deinen Klassenkameraden verschlüsselt kommunizieren, ohne dass eure Lehrerin mitlesen kann. Wie ihr bereits wisst, ist die sog. Caesar-Chiffre, bei der jeder Buchstabe um eine bestimmte Zahl im Alphabet verschoben wird, aber einfach zu knacken.

Daher habt ihr euch für folgenden Algorithmus entschieden, der das Knacken erschwert:

1. Tausche den ersten mit dem letzten und den zweiten mit dem vorletzten Buchstaben.
z.B. wird aus „geheim“ → „miheeg“
2. Verschiebe jedes Zeichen im Alphabet um einen Schlüssel, der sich in Abhängigkeit von der Länge des Klartextes ergibt ((Länge des Klartextes modulo 4) +1).
z.B. wird aus „miheeg“ → „plkhhj“
3. Wie du weißt, können chars durch ASCII-Zeichen und damit eine Zahl kodiert werden. Verschiebe jedes „gerade“ Zeichen um 6 (also b wird zu h, d wird zu j, ...)
z.B. wird aus „plkhhj“ → „vrknp“

Aufgabe 1: Öffne die Klasse „*Nachrichtenaustausch*“ des gleichnamigen bereitgestellten Projekts mit BlueJ.

Indem du auf eine Zeilennummer klickst, kannst du sogenannte *Haltepunkte* (engl: *Breakpoints*) an beliebigen Stellen im Code einfügen:



→**Setze einen Haltepunkt in Zeile 13 und in Zeile 16.**

Erzeuge nun ein Objekt *Nachrichtenaustausch* und führe die Methode *verUndEntschlüsseln()* mit einem beliebigen String als Eingabe aus. Es öffnet sich ein neues Fenster, der *Debugger*. **Finde heraus, welche Bedeutung die folgenden Schaltflächen haben** (Hinweis: Setze dazu auch weitere Haltepunkte).



Aufgabe 2: Beantworte nun mit Hilfe des Debuggers und passend gesetzten Haltepunkten folgende Fragen, jeweils für die Eingabe „geheim“:

- 1) Welchen Wert hat die Variable *schluessel* in der Klasse *Nachrichtenaustausch* nach Ausführung der Zeile 33 (`int schluessel = text.length() % 4`)? _____
- 2) Im Rahmen des Algorithmus zur Verschlüsselung werden Zahlen (integers) auf Buchstaben (chars) addiert. Beobachte mit Hilfe des Debuggers, was dabei passiert (z.B. Zeile 17 in der Klasse *Verschlüsselung*): _____
- 3) Wie oft wird die Alternative in Zeile 22 der Klasse *Verschlüsselung* betreten? _____
- 4) Beschreibe den zugrunde liegenden Fehler und behebe ihn.

Aufgabe 3: Offensichtlich funktioniert die Ver- und Entschlüsselung trotzdem noch nicht richtig. Finde mit Hilfe des Debuggers alle weiteren Fehler und behebe sie!

Fehler 1: _____

Fehler 2: _____

(Hinweis: Es bietet sich an, Haltepunkte nach jedem Schritt des Algorithmus zu setzen, um zu überprüfen, ob die einzelnen Schritte richtig durchgeführt wurden.)

Der Debugger

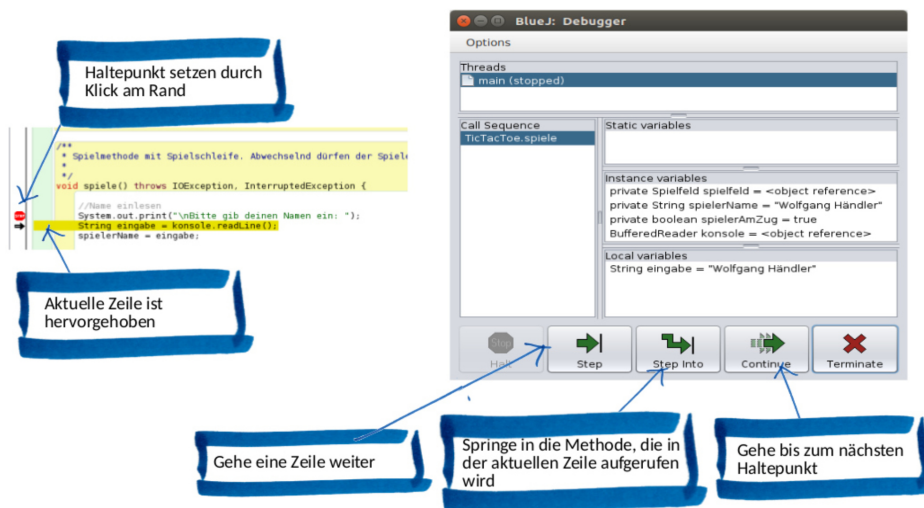
Mit Hilfe des Debuggers lassen sich folgende Fragen beantworten:

- _____
- _____

Das hilft mir bei:

- Kompilierzeitfehlern Laufzeitfehlern logischen Fehlern

So gehe ich dabei vor:



Hinweise:

- Haltepunkt stoppt _____ Ausführung der Zeile, in der er gesetzt ist.
- In BlueJ werden die Haltepunkte beim erneuten Übersetzen entfernt.

B.3 Einführung von Print-Debugging in BlueJ

Arbeitsblatt Print-Debugging (BlueJ)

Aufgabe 1: Öffne das bereitgestellte Hangman-Projekt.

Verwenden den dir bereits bekannten Objektinspektor, um folgende Fragen zu beantworten:

- 1) Welchen Wert hat das Attribut *leben* nach Aufruf des Konstruktors *Hangman()*? _____

Welchen Wert hat das Attribut *leben* nach Aufruf des Konstruktors *Hangman(String: wort)*? _____

- 2) Behebe den Fehler in *Hangman()*.
- 3) Inspiziere auch die Attribute *wort*, *standardwoerter* und *gefunden* nach Aufruf eines der beiden Konstruktoren (Hinweis: auf den Pfeil klicken). Stelle eine Vermutung auf: Wofür dient das boolean-Feld *gefunden*? (Hinweis: Vergleiche *gefunden* und *wort* für verschiedene Eingaben.)

- 4) Irgendetwas scheint in der Spielschleife noch nicht zu funktionieren. Beschreibe, warum du folgende Frage nicht mit dem Objektinspektor beantworten kannst:

Welchen Wert hat das Attribut *leben* und welchen Wert gibt die Methode *alleGefunden()* jeweils zurück, wenn die bedingte Wiederholung in Zeile 30 betreten wird?

- 5) Wie könntest du obige Frage stattdessen beantworten?

→ weiter auf der Rückseite

Falls du Frage 5 nicht beantworten konntest, hier zur Erinnerung:

Mit Hilfe des Befehls `System.out.println()` lassen sich in Java Ausgaben auf die Konsole schreiben. So ließe sich z.B. der Wert des Attributes *leben* in Zeile 30 ausgeben:

```
System.out.println(„Leben: “ + leben);
```

6. Gib damit den Wert der Variable *leben* zu Beginn jedes Durchlaufs der Wiederholung in Zeile 30 aus. Was fällt auf?

7. Behebe den zugrunde liegenden Fehler!

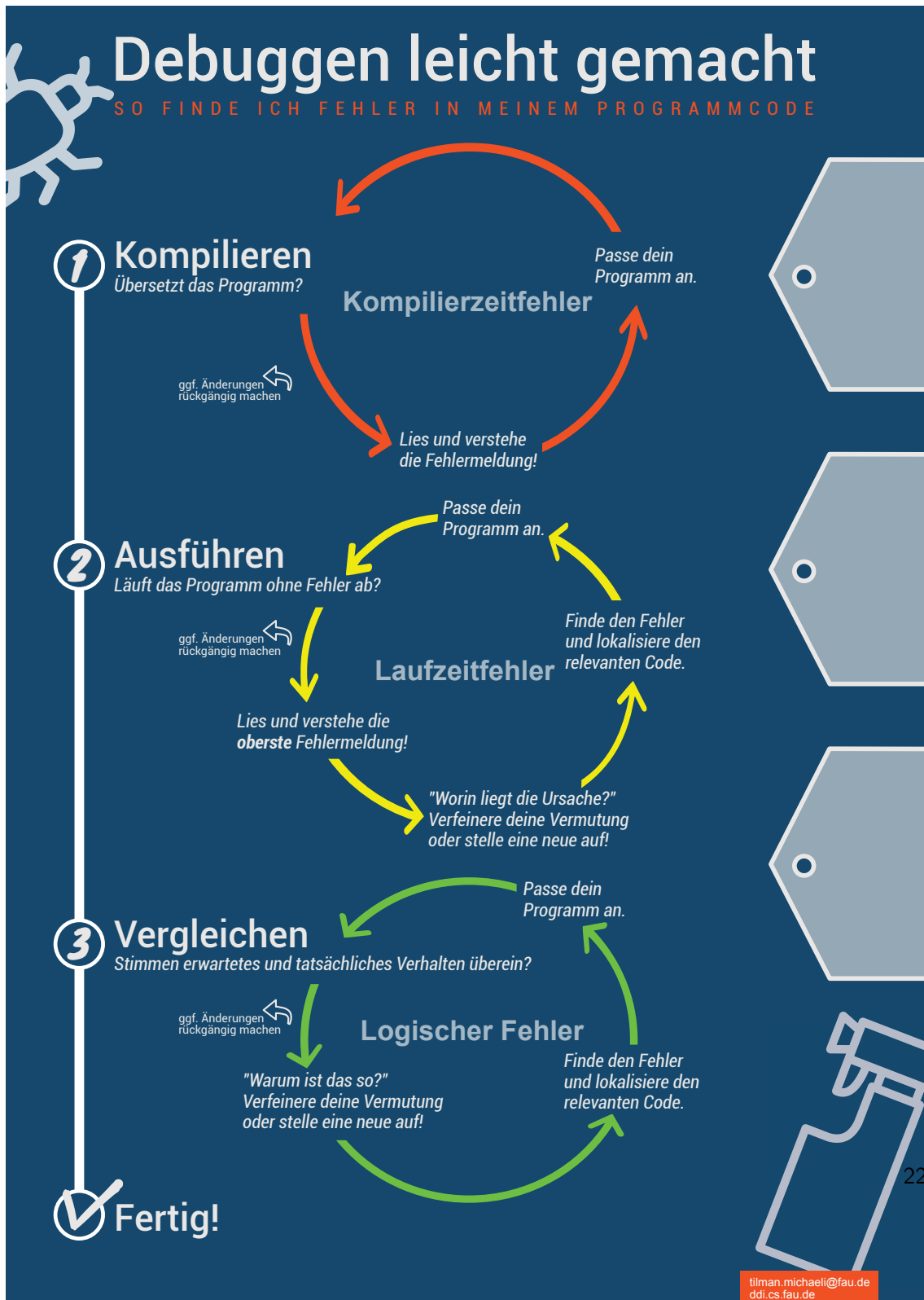
8. Spiele mehrere Runden Hangman.

Beim Spielen ist zu erkennen, dass das letzte Zeichen nie angezeigt wird. Finde mit Hilfe von `System.out.println()` den Fehler, indem du das Attribut *wort* und die Methode `erratenAusgeben()` untersuchst.

Fehler: _____

9. Vergleiche in eigenen Worten den Objektinspektor und `System.out.println()` bei der Fehlersuche:

Anhang C: Poster „Debuggen leicht gemacht“







Anhang D: Fragebogen Schülerinnen und Schüler

D.1 Pretest

Selbsteinschätzung


































































Wie sehr stimmst du den folgenden Aussagen zu? (ankreuzen)

	stimme zu	stimme eher zu	stimme teils zu	stimme kaum zu	stimme nicht zu
Ich kann selbstständig Fehler in Programmcode finden und beheben.					
Ich denke, ich bin in der Schule ziemlich gut, im Vergleich zu anderen.					
Mir macht der Informatikunterricht viel Spaß.					
Ich kann die Ausführung eines Stücks Programmcode schrittweise nachvollziehen, ohne es auszuführen.					
Ich bin mit meiner Leistung in der Schule zufrieden.					
Wenn mein Programm sich nicht verhält, wie es soll, weiß ich, wie ich vorgehen kann.					
Ich kann gut programmieren.					
Ich kann Fehler in Programmcode beheben, wenn mir jemand bei der Fehlersuche hilft.					

D.2 Posttest

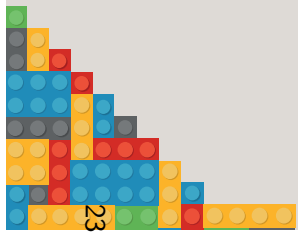
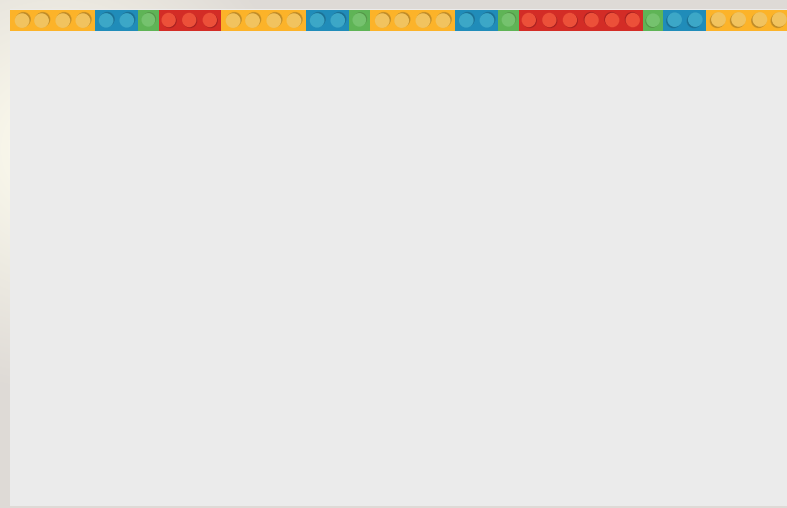
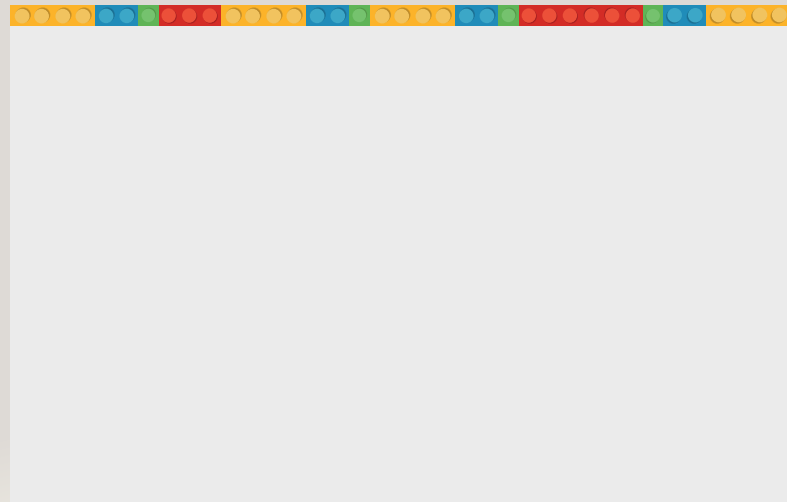
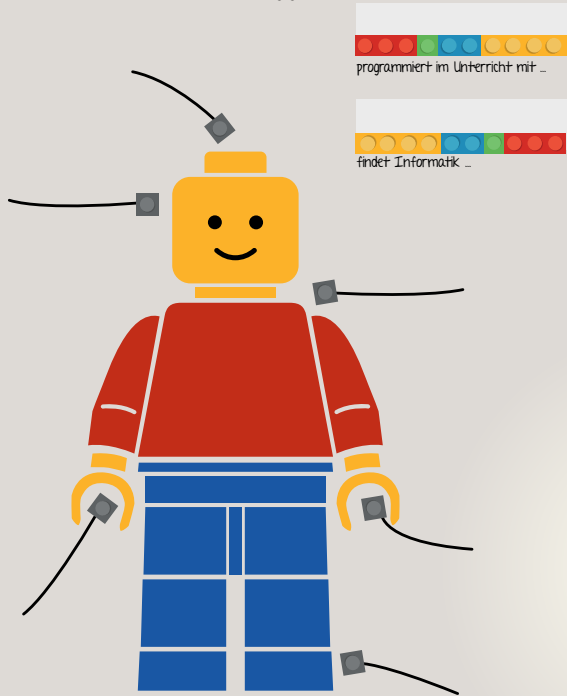
Fragebogen zur Unterrichtseinheit

Wie sehr stimmst du den folgenden Aussagen zu? (ankreuzen)

	stimme zu	stimme eher zu	stimme teils zu	stimme kaum zu	stimme nicht zu
Ich denke, ich habe mich bei dieser Aktivität im Vergleich zu anderen Schülern ziemlich gut geschlagen.					
Ich kann gut programmieren.					
Aufgabe 1 war für mich gut lösbar.					
Aufgabe 2 war für mich gut lösbar.					
Ich habe das Vorgehen verstanden und kann es anwenden.					
Ich kann die Ausführung eines Stücks Programmcode schrittweise nachvollziehen, ohne es auszuführen.					
Ich fand das Suchen und Beheben von Fehlern langweilig.					
Ich kann selbstständig Fehler in Programmcode finden und beheben.					
Ich bin mit meiner Leistung bei den Aufgaben zufrieden.					
Ich kann Fehler in Programmcode beheben, wenn mir jemand bei der Fehlersuche hilft.					
Ich habe mich bei der Bearbeitung der Aufgaben sehr bemüht.					
Wenn mein Programm sich nicht verhält, wie es soll, weiß ich, wie ich vorgehen kann.					
Mir hat das Suchen und Beheben von Fehlern viel Spaß gemacht.					

Anhang E: Personas

■ Unser/e prototypische/r Schüler/in
beim Fehlersuchen & Debuggen



Name

Anhang F: Interviewleitfaden Fortbildungsevaluation

Leitfrage/Erzählaufforderung	Stichworte - nur erfragen, wenn nicht von allein thematisiert	Nachfragen mit obligatorischer Formulierung
Die Fortbildung ist jetzt 5 Monate her, hat sich etwas an deinem Unterricht verändert?	Wahrnehmung der Probleme der SuS Fehlerkultur im Klassenzimmer	Reagierst du jetzt "anders" als vorher auf die Probleme der SuS? "Rennst" du weniger (und wenn ja: warum)?
Hast du die einzelnen Methoden ausprobiert?	Systematisches Vorgehen, Debuggingstrategien, Debugging Logs	Welche Erfahrungen hast du dabei gemacht? Wie hat das funktioniert? Wie hast du die einzelnen Ideen aus der Fortbildung ggf. für dich angepasst?
Welche Aspekte der Fortbildung waren besonders gewinnbringend für dich?	Fachvortrag, Debuggingbiographien, Lego-Schüler, Ausprobieren	Was hast du inhaltlich/fachlich gelernt? Wie bewertest du die in der Fortbildung eingesetzten Methoden? Was hat dir gefehlt?
Abschließend: Was hast du für deine Unterrichtspraxis mitgenommen aus der Fortbildung?		