Institut für Informatik und Computational Science

Entwicklung und Evaluation einer prototypischen Lernumgebung für das systematische Debugging logischer Fehler in Quellcode

Masterarbeit

zur Erlangung des akademischen Grades eines

Master of Science

vorgelegt von

Felix Ziemann

Bearbeitungszeit: von 07.09.2023 bis 08.03.2024

Erstgutachter: Prof. em. Dr. Andreas Schwill, Didaktik der Informatik, Universität Potsdam

Zweitgutachter: M. Ed. Florian Reuß, Didaktik der Informatik, Universität Potsdam

Online veröffentlicht auf dem
Publikationsserver der Universität Potsdam:
https://doi.org/10.25932/publishup-63273
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-632734

Danksagung

Während der Anfertigung der vorliegenden Arbeit wurde ich von verschiedenen Personen unterstützt, denen ich an dieser Stelle danken möchte.

Zunächst gebührt mein Dank Florian Reuß, der mich während dieser Zeit überaus interessiert betreut und die Arbeit begutachtet hat. Für die konstruktive Unterstützung, die stetige Erreichbarkeit sowie die stets wertschätzenden, motivierenden und voranbringenden Treffen und Gespräche möchte ich mich ganz herzlich bedanken.

Außerdem bedanke ich mich bei Prof. Dr. Andreas Schwill für das Interesse an meiner Arbeit, die Übernahme ihrer Begutachtung sowie die konstruktiven Hinweise während der Erarbeitung.

Zudem danke ich Alexander Hacke für die hilfreichen Vorschläge hinsichtlich der methodischen Besonderheiten der Datenerhebung im Rahmen der Evaluation.

Ebenfalls möchte ich mich bei Paula Müßigbrodt und Jonas Richter für das Korrekturlesen dieser Arbeit bedanken.

Ein besonderer Dank gilt allen Teilnehmenden, die an der Studie teilgenommen haben. Mein Dank gilt insbesondere ihren informativen sowie umfangreichen Rückmeldungen und Verbesserungsvorschlägen.

Abstract

Wo programmiert wird, da passieren Fehler. Um das Debugging, also die Suche sowie die Behebung von Fehlern in Quellcode, stärker explizit zu adressieren, verfolgt die vorliegende Arbeit das Ziel, entlang einer prototypischen Lernumgebung sowohl ein systematisches Vorgehen während des Debuggings zu vermitteln als auch Gestaltungsfolgerungen für ebensolche Lernumgebungen zu identifizieren. Dazu wird die folgende Forschungsfrage gestellt: Wie verhalten sich die Lernenden während des kurzzeitigen Gebrauchs einer Lernumgebung nach dem *Cognitive Apprenticeship*-Ansatz mit dem Ziel der expliziten Vermittlung eines systematischen Debuggingvorgehens und welche Eindrücke entstehen während der Bearbeitung?

Zur Beantwortung dieser Forschungsfrage wurde orientierend an literaturbasierten Implikationen für die Vermittlung von Debugging und (medien-)didaktischen Gestaltungsaspekten eine prototypische Lernumgebung entwickelt und im Rahmen einer qualitativen Nutzerstudie mit Bachelorstudierenden informatischer Studiengänge erprobt. Hierbei wurden zum einen anwendungsbezogene Verbesserungspotenziale identifiziert. Zum anderen zeigte sich insbesondere gegenüber der Systematisierung des Debuggingprozesses innerhalb der Aufgabenbearbeitung eine positive Resonanz. Eine Untersuchung, inwieweit sich die Nutzung der Lernumgebung längerfristig auf das Verhalten von Personen und ihre Vorgehensweisen während des Debuggings auswirkt, könnte Gegenstand kommender Arbeiten sein.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

CAA	Cognitive Apprenticeship-Ansatz
CATML	Cognitive-affective Theory of Learning with Media
CBR	Case-Based Reasoning
CBT	Computer-Based Training
CLT	Cognitive Load Theory
CTML	Cognitive Theory of Multimedia Learning
E-Learning	Electronic Learning
GeRRI	Gemeinsamer Referenzrahmen Informatik
GI	Gesellschaft für Informatik e. V.
IDE	Integrated Development Environment
ITS	Intelligent Tutoring System
MBR	Model-Based Reasoning
MNU	Verband zur Förderung des MINT-Unterrichts
RLP	Rahmenlehrplan
UI	User Interface
UP	Universität Potsdam
WBT	Web-Based Training

1 Einleitung

1.1 Problemstellung und Motivation

Das Finden und Beheben von Fehlern in Quellcode ist ein überaus relevanter Bestandteil der Entwicklung von Software und kann, wie Studien zeigen, einen wesentlichen Teil der Ressourcen innerhalb der professionellen Softwareentwicklung einnehmen (Zeller, 2009).

Dennoch sind im Hinblick auf den schulischen Informatikunterricht sowohl in dem Rahmenlehrplan (RLP) für die Jahrgangsstufen 7-10 der Länder Berlin und Brandenburg als auch in dem RLP für die gymnasiale Oberstufe der Länder Berlin und Brandenburg weder der explizite Begriff *Debugging* noch andere Formulierungen enthalten, welche die Behebung von Fehlern in Programmen als verbindlichen Unterrichtsinhalt ausweisen würden (MBJS, 2015; MBJS, 2022). Auch in den Curricula der anderen Bundesländer wird das Debugging lediglich selten ausdrücklich benannt (Michaeli & Romeike, 2019a; Michaeli & Romeike, 2019c). Zeitgleich verdeutlicht jedoch der hohe curriculare Stellenwert der Implementierung algorithmischer Lösungswege die Relevanz von Debuggingfähigkeiten, um im Falle von auftretenden Fehlern die Ursachen identifizieren und beheben zu können (Michaeli & Romeike, 2019b).

Die Bedeutung des Debuggings wird auch innerhalb des gemeinsamen Referenzrahmens Informatik (GeRRI) des Verbands zur Förderung des MINT-Unterrichts (MNU) und der Gesellschaft für Informatik (GI) ersichtlich, welcher Mindeststandards für die informatische Bildung formuliert und explizite Kompetenzen innerhalb der Teilkonzepte *Test* und *Fehleranalyse* als Bestandteil des Grundkonzepts *Automatisierung* ausweist, die im Rahmen des Informatikunterrichts erworben werden sollen. Hierin werden unter anderem die Identifikation von Fehlerursachen, das Verständnis und die Interpretation von Fehlermeldungen sowie die Unterscheidung zwischen verschiedenen Fehlerarten genannt (MNU & GI, 2020).

Aufgrund der Absenz der Thematik in den Curricula, des Zeitmangels und fehlender Unterrichtsmaterialien sowie -konzepte werden zumeist keine separaten Unterrichtsstunden für die explizite Vermittlung von Debugging genutzt; stattdessen erfolgt sie häufig nebenher, indem die Lehrkräfte während des Unterrichts aufgrund der Hilflosigkeit und Frustration der Schüler:innen im Falle von auftretenden Fehlern oftmals von Schüler:in zu Schüler:in eilen, um sie in der Behebung des Fehlers zu unterstützen. Häufig wird dies auch als *Turnschuhdidaktik* bezeichnet. Dabei sind unter den Lehrkräften die Bereitschaft und der Wille vorhanden, Debugging im Informatikunterricht stärker zu gewichten (Michaeli & Romeike, 2019a; Michaeli & Romeike, 2019c).

Auch im Hinblick auf die universitäre Lehre am Institut für Informatik und Computational Science der Universität Potsdam (UP) zeigen sich konkrete Anknüpfungspunkte und Potenziale für eine explizite Vermittlung von Debugging. So wird innerhalb des Pflichtmoduls *Praxis der Programmierung* für die Bachelor-Studiengänge Informatik/Computational Science,

Wirtschaftsinformatik sowie Lehramt Informatik eine adäquate Fehlerbehandlung formal ausdrücklich als Qualifikationsziel benannt:

> „Die Studierenden können Fehlerkonzepte unterscheiden und Fehler behandeln“ (UP, 2019, S. 44).

Eine Untersuchung[1] der Kursmaterialien zeigt, dass innerhalb der Vorlesungsmaterialien vor allem auf *wiederkehrende Fehler*, die jeweils zugehörigen *Ursachen* und *Möglichkeiten einer frühen Fehlervermeidung* hingewiesen wird. Außerdem werden die verschiedenen *Fehlerarten* definiert; die Fehlersuche und -behebung jedoch werden für syntaktische sowie semantische Fehler lediglich knapp dargelegt; im Hinblick auf logische Fehler wird sogar ausschließlich deren Vermeidung thematisiert.

Die Übungsblätter hingegen umfassen oftmals Aufgaben zur *Fehlerbehandlung*. Hierbei unterstützen dann die jeweiligen Übungsleiter:innen. Die hier ersichtlich werdende Diskrepanz zwischen der Vermittlung und den späteren Erwartungen an die Studierenden bietet Raum für neue Ansätze, die Debuggingfähigkeiten der Studierenden explizit zu adressieren. Darüber hinaus würde eine veranstaltungsübergreifend einheitliche Vermittlung von Debugging eine unter den Studierenden vergleichbare Wissensbasis schaffen und die Lehre von den Personen entkoppeln, die als Übungsleiter:innen fungieren.

1.2 Ziele, Fragestellung und weiteres Vorgehen

Orientierend an der vorangegangenen Motivation verfolgt die vorliegende Arbeit zum einen das folgende *nutzerorientierte* Ziel:

(1) Förderung eines systematischen Vorgehens im Rahmen des Debuggings, insbesondere im Hinblick auf Programmieranfänger:innen, im schulischen sowie universitären Bereich.

Dazu wird eine prototypische Lernumgebung entwickelt, die sowohl von Lehrenden genutzt werden kann, um Debugging innerhalb von Lehr-Lern-Situationen explizit zu adressieren, als auch Lernenden ermöglicht, ihre Debuggingfähigkeiten eigenständig zu verbessern. Hieraus ergibt sich nachfolgendes *forschungsorientiertes* Ziel:

(2) Identifikation von Gestaltungsfolgerungen im Hinblick auf Lernumgebungen, die ein systematisches Vorgehen während des Debuggings vermitteln sollen.

[1] Anmerkung: Das Vorgehen orientiert sich in Teilen an der zusammenfassenden Inhaltsanalyse nach Mayring (2022). Dabei erfolgt aus den Teilen der Vorlesungs- und Übungsmaterialien, welche das Debugging adressieren, eine induktive Bildung von Kategorien, welche die inhaltliche Schwerpunktsetzung repräsentieren. Der Zugriff auf die Materialien erfolgt am 29.09.2023 über den Moodle-Kurs der Veranstaltung: https://moodle2.uni-potsdam.de/course/view.php?id=36753.

Hier knüpft die folgende Forschungsfrage an, die im Rahmen dieser Arbeit beantwortet werden soll:

Wie verhalten sich die Lernenden während des kurzzeitigen Gebrauchs einer Lernumgebung nach dem *Cognitive Apprenticeship*-Ansatz mit dem Ziel der expliziten Vermittlung eines systematischen Debuggingvorgehens und welche Eindrücke entstehen während der Bearbeitung?

Basierend auf den genannten Zielsetzungen ergibt sich für den weiteren Verlauf dieser Arbeit die folgende Struktur:

Im Rahmen des zweiten Kapitels erfolgt zunächst eine Einführung in das Debugging sowie in die zugehörigen Prozessschritte. Kapitel 3 formuliert orientierend an den Ausführungen in Kapitel 2 Implikationen für die Vermittlung von Debugging und untersucht, inwieweit sie Bestandteil bereits bestehender Tools sind. Kapitel 4 widmet sich wesentlichen (medien-)didaktischen Gestaltungsaspekten von Lernumgebungen. Basierend auf den bis dahin gewonnenen Erkenntnissen wird dann innerhalb des fünften Kapitels die prototypische Lernumgebung entwickelt. Hierbei werden vor allem die Anforderungen, Überlegungen zu Abläufen sowie der Ästhetik und die technische Umsetzung begründend dargelegt. In Kapitel 6 wird die Evaluation der Lernumgebung beschrieben und insbesondere dessen Ergebnisse sowohl dargelegt als auch hinsichtlich ihrer Implikationen und Limitationen diskutiert. Kapitel 7 bildet unter Berücksichtigung der Zielsetzungen und der Forschungsfrage den Abschluss der vorliegenden Arbeit in Form eines Resümees sowie eines Ausblicks.

2 Debugging(-prozess)

Die Entwicklung einer Lernumgebung mit dem Ziel der Förderung eines systematischen Debuggingvorgehens erfordert zunächst einmal eine literaturbasierte Klärung folgender Fragen: Was ist Debugging? Inwiefern wirkt sich die jeweilige Fehlerart hierauf aus? Und welche Teilschritte umfasst der Debuggingprozess? Hierzu erfolgt eingangs eine begriffliche Abgrenzung des Debuggens zum Testen (siehe Kapitel 2.1.1) sowie zum Programmieren (siehe Kapitel 2.1.2). Anschließend erfolgen eine Ausdifferenzierung verschiedener Fehlerarten sowie eine Darlegung des jeweiligen Einflusses auf den Debuggingprozess (siehe Kapitel 2.2). Im weiteren Verlauf werden der Problemlöseprozess und der Troubleshootingprozess sowie hierin relevante Wissensarten zur Einordnung des Debuggingprozesses dargelegt (siehe Kapitel 2.3), bevor dieser schrittweise entlang der Entwicklung eines mentalen Modells (siehe Kapitel 2.4.1), Identifikation fehlerhaften Systemverhaltens (siehe Kapitel 2.4.2), der Fehlerdiagnose (siehe Kapitel 2.4.3), Fehlerbehebung (siehe Kapitel 2.4.4) und der Speicherung erworbener Erfahrungen (siehe Kapitel 2.4.5) ausdifferenziert werden.

2.1 Begriffsbestimmung

2.1.1 Debuggen und Testen

Fehlerhafte Software kann unter Umständen erhebliche sowohl gesundheitliche als auch finanzielle Folgen mit sich bringen. Ein populäres Beispiel ist unter anderem die Explosion der Trägerrakete Ariane 5 im Juni 1996. Diese war im Wesentlichen auf die Wiederverwendung von früherem Quellcode des Vorgängermodells, ohne dass dieser hinreichend an die technischen Abweichungen zwischen den Modellen angepasst wurde, zurückzuführen (Dowson, 1997). Es existieren darüber hinaus zahlreiche weitere Vorfälle mit vergleichbar schwerwiegenden Konsequenzen.[2]

Dies spiegelt den besonderen Stellenwert der Sicherstellung der Softwarequalität innerhalb der Softwareentwicklung wider. Insbesondere um die funktionale Korrektheit von Software, die in der ISO-Norm 25010, welche die Qualitätseigenschaften von Software in einem Qualitätsmodell für Software subsummiert (ISO, 2011), als *functional suitability* aufgeführt wird, zu gewährleisten, sind das Testen und Debuggen integrale Bestandteile des Softwareentwicklungsprozesses; unabhängig von dem spezifischen Vorgehensmodell, das gewählt wird (Hailpern & Santhanam, 2001). Doch ähnlich wie die Prozesse des Testens und Debuggens an verschiedenen Stellen des Softwareentwicklungsprozesses ineinandergreifen (Hailpern & Santhanam, 2001) erfolgt auch die begriffliche Verwendung beider Begriffe nicht immer trennscharf.

[2] Anmerkung: Unter https://en.wikipedia.org/wiki/List_of_software_bugs findet sich eine Übersicht über ausgewählte Softwarefehler mit besonders folgenschweren Auswirkungen, letzter Zugriff am 06.03.2024.

Metzger (2004) definiert die Begriffe Testen und Debuggen und differenziert zwischen ihnen wie folgt:

> „Testing is the process of determining *whether* a given set of inputs causes an unacceptable behavior in a program" (Metzger, 2004, S. 12).
>
> „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" (Metzger, 2004, S. 12).

Typische Testphasen in der Entwicklung eines kommerziellen Softwaresystems sind die Entwicklertests sowie die Freigabetests. Während die Entwicklertests während der Entwicklung von den Programmierer:innen selbst ausgeführt werden, um bereits frühzeitig Fehler in der Implementierung zu identifizieren, sollen Freigabetests überprüfen, ob das System für die externe Nutzung freigegeben werden kann. Normalerweise ist hierfür ein separates Team, das nicht an der vorangegangenen Systementwicklung beteiligt war, verantwortlich (Sommerville, 2012). Diese strikte Rollenteilung verdeutlicht die Verschiedenheit des Testens und Debuggens, wie sie in der begrifflichen Abgrenzung nach Metzger (2004) formuliert wird: Das System ist für die testenden Personen eine Art Blackbox, da sie sich ausschließlich mit dessen Verhalten, basierend auf verschiedenen Ein- und Ausgaben, beschäftigen. Abweichungen werden an die Entwickler:innen weitergegeben, die dann im Rahmen der Fehlerbehebung den Programmfehler lokalisieren und beheben (Sommerville, 2012). Obgleich das Debugging oftmals durch eine vorherige Testung des Programms ausgelöst wird, bleiben die Prozesse weiterhin eng zusammengehörig. So ist das Testen häufig auch ein integraler Bestandteil des Debuggingprozesses (siehe Kapitel 2.4).

Eine weitere Definition des Begriffs Debugging nach Zeller (2009) ist die Folgende:

> „Relating a failure or an infection to a defect (via an infection chain) and subsequent fixing of the defect" (Zeller, 2009, S. 377).[3]

Ein Vergleich der Definitionen nach Metzger (2004) und Zeller (2009) zeigt, dass beide Autoren Debugging als ein Prozess definieren, der sich im Grunde in zwei Schritte unterteilen lässt: Die *Bestimmung des Fehlerursprungs* (jeweils der erste Teil beider Definitionen) und die anschließende *Behebung des Fehlers* (jeweils der zweite Teil beider Definitionen). Das dieser Arbeit zugrundeliegende Verständnis von Debugging entspricht daher dieser Schnittmenge.

[3] Anmerkung: Als *infection* bezeichnet Zeller (2009) einen fehlerhaften Programmzustand, der insbesondere zu einem *failure* führen kann. Unter *failure* versteht Zeller (2009) ein sichtbar fehlerhaftes Programmverhalten. Sie beide sind auf einen *defect*, ein Fehler im Programm, zurückzuführen.

2.1.2 Debuggen und Programmieren

In Ergänzung zu der vorangegangenen Begriffsabgrenzung zum Testen unterscheiden sich auch die für das Debugging relevanten Fähigkeiten von jenen in der Programmierung. Bereits Kessler und Anderson (1986) formulierten dies wie folgt:

> „[…] debugging is a skill that does not immediately follow from the ability to write code. Rather, it consists of several subskills that can and must be taught in addition to instructions about how to write programs" (Kessler & Anderson, 1986, S. 208).

Ahmadzadeh et al. (2005) untersuchten die Programmier- und Debuggingfähigkeiten von Programmieranfänger:innen und stellten fest, dass 66 % der guten Debugger:innen auch gute Programmierer:innen waren. Lediglich 39 % der guten Programmierer:innen erzielten jedoch auch gute Resultate im Debugging.[4] Als wesentliche Hürden hierfür nennen Ahmadzadeh et al. (2005) die inkonsistente Anwendung von Debuggingstrategien sowie Schwierigkeiten im Hinblick auf das Verständnis des nicht selbst verfassten Quellcodes.

Die Programmier- und Debuggingfähigkeiten von Programmieranfänger:innen wurden auch im Rahmen einer Studie von Fitzgerald et al. (2008) verglichen. Die Autor:innen konstatierten ebenfalls, dass ausgeprägte Debuggingfähigkeiten zumeist mit gleichermaßen vorhandenen Programmierfähigkeiten einhergehen, während jedoch ausgeprägte Programmierfähigkeiten nicht zwingend auf gleichermaßen vorhandene Debuggingfähigkeiten schließen lassen (Fitzgerald et al., 2008). Doch woran lässt sich dann ein:e gute:r Debugger:in erkennen? Michaeli und Romeike (2019a) nennen sowohl die Anwendung eines *systematischen Vorgehens*, von *Heuristiken und Pattern* sowie von *Debuggingstrategien* als auch die Verwendung von *Werkzeugen* als relevante Debuggingfähigkeiten und entscheidende Merkmale guter Debugger:innen. Die in Kapitel 2.4 folgende nähere Betrachtung und Ausdifferenzierung des Debuggingprozesses soll sich dieser Fragestellung intensiver annehmen. Zuvor ist es jedoch notwendig, die verschiedenen Fehlerarten im Kontext des Debuggings (siehe Kapitel 2.2) sowie den Problemlöseprozess und den Troubleshootingprozess als übergeordnete Vorgehensweisen in ihren Grundzügen zu beleuchten (siehe Kapitel 2.3).

2.2 Fehlerarten

„Jeder Programmierer macht Fehler" (Gumm & Sommer, 2011, S. 183). Zunehmende Programmiererfahrung geht im Allgemeinen mit dem Auftreten vieler verschiedenartiger Fehler einher, die es im Laufe der Zeit zu beseitigen gilt. Daher sollen im Folgenden die unterschiedlichen Arten von Fehlern definiert sowie deren jeweilige Bedeutung für den Debuggingprozess untersucht werden.

[4] Anmerkung: Als *gute* Debugger:innen bezeichnen Ahmadzadeh et al. (2005) jene Studierende, die im Rahmen einer zu bearbeitenden Debuggingaufgabe sämtliche Fehler finden und erfolgreich korrigieren konnten. Als *gute* Programmierer:innen hingegen gelten die Studierenden, welche mindestens 70 % der zu erwerbenden Punkte in der Klausur zur Einführung in die Programmierung erhielten.

Ein Softwarefehler wird im Allgemeinen wie folgt definiert:

> "The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition" (ISO, 2010, S. 128).

Im Allgemeinen wird in der Softwareentwicklung zwischen syntaktischen, semantischen und logischen Fehlern differenziert (Gumm & Sommer, 2011; Hristova et al., 2003). Nachfolgend werden diese näher erläutert.

- *Syntaktische Fehler* treten auf, insofern ein Verstoß gegen syntaktische Regeln vorliegt. Diese beschreiben, wie aus den lexikalischen Regeln hervorgehende zulässige Wörter zu korrekten Programmen zusammengesetzt werden dürfen. Beispiele sind eine inkorrekte Klammersetzung oder die Nutzung reservierter Schlüsselwörter als Variablen (Gumm & Sommer, 2011).
- *Semantische Fehler* hingegen stellen zwar keine direkten Verstöße gegen die syntaktischen Regeln dar, zeigen jedoch Verletzungen im Hinblick auf die Bedeutung der verwendeten Konstrukte der jeweiligen Programmiersprache auf. Beispiele sind die Division durch Null oder auch der versuchte Zugriff auf einen nicht vorhandenen Index einer Liste (Gumm & Sommer, 2011).
- *Logische Fehler* widersprechen weder den syntaktischen noch den semantischen Regeln. In diesem Falle terminiert das Programm zwar für alle Eingabewerte, während dessen Verhalten dennoch nicht den Vorstellungen der/des Entwickler:in entspricht. Beispiele sind eine Verwechslung von Variablennamen oder die Zuweisung fehlerhafter Werte (Gumm & Sommer, 2011).

Syntaktische Fehler werden seitens des *Compilers* während der Übersetzung des Quellcodes in maschinenlesbare Sprache vor der eigentlichen Ausführung des Programms entdeckt. Moderne integrierte Entwicklungsumgebungen (IDE) zeigen Fehler dieser Art bereits während der Implementierung an. Semantische Fehler werden entweder ebenso zur Kompilierzeit oder aber erst während der Laufzeit ermittelt, während logische Fehler wiederum entlang des Vergleichs des Programms mit dessen Spezifikation ersichtlich werden (Gumm & Sommer, 2011).[5]

Innerhalb des Debuggingprozesses bestimmt die jeweilige Fehlerart vor allem die Verfügbarkeit von Informationen als Ausgangspunkt des Prozesses und wirkt sich somit auch auf dessen Verlauf aus (Michaeli & Romeike, 2019b). Im Falle von syntaktischen Fehlern stellen moderne IDEs gemeinhin die exakte Position und eine Beschreibung des jeweiligen Fehlers sowie häufig sogar bereits Vorschläge zur Fehlerbehebung bereit. So wie die umfassenden Hilfestellungen im Falle von syntaktischen Fehlern bereits nahelegen, bestätigen auch Altadmri und Brown (2015) eine zumeist unproblematische Behebung dieses Fehlertyps.

[5] Anmerkung: Eine weitere Kategorisierung von Fehlern, die sich an dem Zeitpunkt der Fehlererkennung orientiert, findet sich in Claus und Schwill (2003). Als Fehlerarten werden hierin entsprechend *Kompilierzeitfehler*, *Laufzeitfehler* und *logische Fehler* genannt.

Fehler, die nicht bereits während der Kompilierzeit entdeckt werden, beanspruchen deutlich mehr Zeit, um gefunden und behoben zu werden (Fitzgerald et al., 2008; Michaeli & Romeike, 2019b). Während auch im Hinblick auf semantische Fehler im Normalfall Fehlermeldungen zur Verfügung stehen, ist die Suche nach logischen Fehlern insofern besonders herausfordernd, als dass sie für gewöhnlich ohne jedwede systemseitige Unterstützung beginnt. In diesem Falle ermöglicht lediglich die fehlerhafte Ausgabe eine erste Orientierung für die Fehlersuche. Unter Berücksichtigung dieser Besonderheit, des je nach Fehlerart variierenden Debuggingprozesses sowie des begrenzten Umfangs dieser Arbeit beschränkt sich der weitere Verlauf auf *logische Fehler*.

2.3 Problemlösen und Troubleshooten

Martinez (1998) definiert Problemlösen wie folgt:

> „Problem solving is the process of moving toward a goal when the path to that goal is uncertain" (Martinez, 1998, S. 605).

Zu Beginn des Problemlöseprozesses nach Gick (1986) erfolgen das Verständnis des Problems und die Erstellung einer Problemrepräsentation. Außerdem wird versucht, das vorliegende Problem entlang von Erfahrungswissen mit bereits bestehenden Schemata vorangegangener Probleme zu verknüpfen.[6] Falls bereits ein solches Schema vorliegt, wird anschließend die Lösung des Problems in das System integriert. Andernfalls erfolgt zunächst die Ermittlung einer adäquaten Lösung des Problems. Schlägt die Integration der Lösung fehl, folgt eine Rückkopplung zu früheren Schritten des Problemlöseprozesses (Gick, 1986) (siehe Abbildung 1).

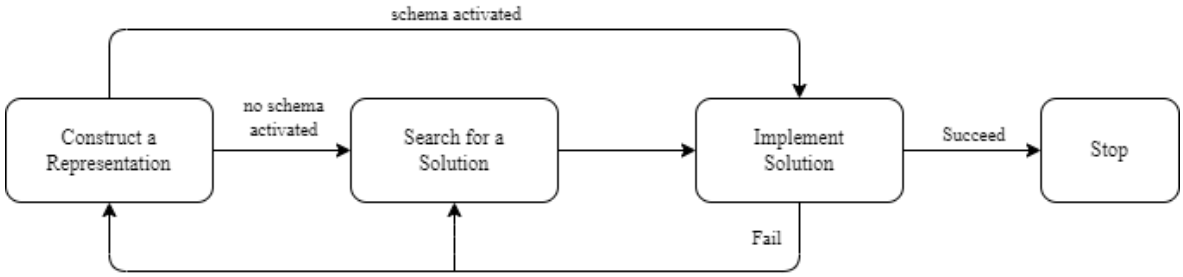


Abbildung 1: Schematische Darstellung des Problemlöseprozesses (Gick, 1986, S. 101)

Jonassen und Hung (2006) beschreiben *troubleshooting* als eine Form des Problemlösens, welche in erster Linie mit der Reparatur physikalischer, mechanischer oder elektronischer Systeme assoziiert wird, und definieren sie folgendermaßen:

> „Troubleshooting attempts to isolate fault states in a system and repair or replace the faulty components in order to reinstate the system to normal functioning" (Jonassen & Hung, 2006, S. 78).

[6] Anmerkung: Als Schema ist an dieser Stelle ein Wissensverbund gemeint, der unter anderem typische Ziele, Bedingungen und Lösungen für ein spezifisches Problem umfasst (Gick, 1986).

Folgende Wissensarten gelten nach Jonassen und Hung (2006) als besonders relevant für den Prozess des *troubleshootings*:

- Der Einstieg in einen Troubleshootingprozess setzt ein gewisses Verständnis übergeordneter Theorien und Prinzipien, denen das jeweilige System unterliegt, in Form von *domain knowledge* voraus. Darüber hinaus ist es insbesondere für den Transfer hin zu anderen Systemen und ein tiefgehendes systemisches Verständnis entscheidend (Jonassen & Hung, 2006).
- *System/Device knowledge* stellt ein wesentliches Unterscheidungsmerkmal im Hinblick auf den Prozess des *troubleshootings* von Noviz:innen sowie Expert:innen dar. Es unterteilt sich in *topographisches* und *funktionales Wissen* über das System. Während *topographisches Wissen* im Wesentlichen die Struktur des Systems abbildet, umfasst *funktionales Wissen* hingegen ein Verständnis der Funktionsweise der Systemkomponenten; sowohl isoliert betrachtet als auch in Interaktion miteinander (Jonassen & Hung, 2006; Kieras, 1988).
- *Performance/Procedural knowledge* beschreibt das Wissen darüber, wie spezifische Teilschritte im Troubleshootingprozess ausgeführt werden. Es ist kontextsensitiv im Hinblick auf das System und die genutzten Werkzeuge (Jonassen & Hung, 2006).
- *Strategic knowledge* erlaubt eine schrittweise Formulierung, Testung und Auswertung von Hypothesen mit dem Ziel, Fehler im System auszumerzen. Es wird zwischen *globalen* und *lokalen Strategien* differenziert. *Globale Strategien* sind dabei kontextunabhängig, in beliebigen Systemen anwendbar und sollen den Problemraum reduzieren. *Lokale Strategien* hingegen sind systemspezifische Strategien, welche die Durchführung dieses Reduktionsprozesses unterstützen (Jonassen & Hung, 2006; Schaafstal & Schraagen, 1993).

Mit zunehmendem *experiential knowledge* aus vorangegangenen Troubleshootingprozessen liegt stetig mehr Wissen über typische Probleme und zugehörige Lösungen vor, das, ähnlich wie im Problemlöseprozess nach Gick (1986) dargestellt, als Erfahrungswissen in zukünftige Troubleshootingprozesse miteinfließt. Erfahrungswissen wird überaus häufig im Rahmen des *troubleshootens* herangezogen und nimmt eine entscheidende Rolle innerhalb des Troubleshootingprozesses ein (Jonassen & Hung, 2006).

Jonassen und Hung (2006) beschreiben den Troubleshootingprozess als iterativen Prozess, bestehend aus fünf Schritten. Nachfolgend werden diese Schritte näher erläutert.

(1) *Construct problem space:* Analog zum Problemlöseprozess nach Gick (1986) wird zu Beginn des Troubleshootingprozesses ein mentales Modell des Systems erstellt, das neben den Komponenten und ihren Verknüpfungen untereinander unter anderem auch das beabsichtigte sowie das tatsächliche Verhalten des Systems umfasst.

(2) *Identify fault symptoms:* Anschließend folgt die Identifikation fehlerhaften Systemverhaltens[7] entlang von Diskrepanzen, die sich aus dem Vergleich des beabsichtigten mit dem tatsächlichen Systemverhalten ergeben.

(3) *Diagnose fault(s):* Im Rahmen der Fehlerdiagnose wird dann zunächst geprüft, ob zuvor bereits ein vergleichbares Problem, welches dem vorliegenden System und dessen fehlerhaften Verhalten ähnelt, gelöst wurde. In einem solchen Falle wird das vorhandene *experience knowledge* für die Problemlösung verwendet; erneut analog zum Problemlöseprozess nach Gick (1986). Andernfalls werden Hypothesen über den Fehlerursprung formuliert und überprüft. Hierzu kann zunächst eine Eingrenzung potenzieller Fehlerursachen entlang der Identifikation und des Ausschlusses irrelevanter Systemkomponenten erfolgen. Im weiteren Verlauf der Fehlerdiagnose sind die Konkretisierung bestätigter Hypothesen sowie die Abwendung von Hypothesen, die sich als inkorrekt herausgestellt haben, für ein effektives *troubleshooten* entscheidend.

(4) *Generate and verify solutions:* Im Anschluss werden mögliche Lösungen generiert und validiert. Die Auswahl der plausibelsten Lösung erfolgt entweder erfahrungsbasiert oder durch iteratives Testen.

(5) *Remember experience:* Schließlich wird der abgeschlossene Troubleshootingprozess implizit dem *experience knowledge* hinzugefügt und steht fortan für das *troubleshooten* vergleichbarer Probleme unterstützend zur Verfügung (Jonassen & Hung, 2006).

Debugging gilt als eine spezielle Form des *troubleshootings* (Ardimento et al., 2019; Katz & Anderson, 1987; Li et al., 2019; Michaeli & Romeike, 2021). Der Debuggingprozess soll daher im weiteren Verlauf in den Troubleshootingprozess eingebettet und weiter ausgeführt werden.

2.4 Debuggingprozess[8]

2.4.1 Entwicklung eines mentalen Modells

Im Rahmen des Debuggingprozesses entspricht die Entwicklung eines mentalen Modells des Systems dem ersten Schritt des Troubleshootingprozesses nach Jonassen und Hung (2006), *construct problem space* (Li et al., 2019). Hierbei kommen vor allem das *domain knowledge* und das *system/device knowledge* zum Tragen (siehe Kapitel 2.3).

Mentale Modelle von Individuen bilden Aspekte ihrer Umgebungen als mentale Strukturen ab und erlauben es ihnen, den Zweck sowie die Funktionsweise der ihnen zugrunde liegenden Systeme zu verstehen und zukünftige Systemzustände vorherzusagen (Sorva, 2013).

[7] Anmerkung: Die englischsprachige Formulierung *identify fault symptoms* entspricht im weiteren Verlauf dieser Arbeit der *Identifikation fehlerhaften Systemverhaltens* und wird gleichbedeutend verwendet.
[8] Anmerkung: Die in diesem Kapitel referenzierte Literatur wurde oftmals bereits vor mehr als 20 Jahren veröffentlicht. Dies ist darauf zurückzuführen, dass ein breiter Korpus an Literatur über Debuggingstrategien aus dieser Zeit stammt, auf den jedoch auch in aktuellen Publikationen weiterhin verwiesen wird; dargelegt auch in Murphy et al. (2008).

Konkret beinhaltet das mentale Modell eines Systems zum einen topographisches Wissen in Form der Struktur des Programms und Wissen über dessen Funktionalität (Decasse & Emde, 1988; Klahr & Carver, 1988; Yoon & Garcia, 1998). Zum anderen umfasst es jedoch auch Wissen über sowohl das intendierte als auch das tatsächlich beobachtbare Verhalten des Programms, insbesondere dessen Output (Decasse & Emde, 1988; Gugerty & Olson, 1986; Yoon & Garcia, 1998). Die Entwicklung eines solchen mentalen Modells bedarf darüber hinaus ein Verständnis der Programmiersprache sowie allgemeine Programmiererfahrung (Decasse & Emde, 1988; Yoon & Garcia, 1998).

Die Entwicklung eines mentalen Modells eines Programms erfolgt häufig entlang des *code tracings* (McCauley et al., 2008; Murphy et al., 2008; Yoon & Garcia, 1998). *Code tracing* beschreibt ein Vorgehen, in dem die schrittweise Ausführung eines Programms mental nachvollzogen wird und auf verschiedene Weisen dokumentiert werden kann. Die Möglichkeiten reichen von einer Beschreibung des Algorithmus über die Angabe, wie häufig ausgewählte Kontrollstrukturen ausgeführt werden bis hin zu der Erstellung von *trace tables*, die es erlauben, Veränderungen der Werte einer oder mehrerer Variablen entlang der Programmausführung tabellarisch darzustellen (Lister et al., 2004; Ruf et al., 2015).

Nach Vessey (1985) ist darüber hinaus auch das *program chunking*, also die Fähigkeit, ein Programm in dessen Bestandteile, wie zum Beispiel Funktionen oder Kontrollstrukturen, zu unterteilen, um es anschließend modular betrachten zu können, ein entscheidender Faktor für das Verständnis des Programms sowie einen erfolgreichen Debuggingprozess im Allgemeinen.

Dem Verständnis des zu debuggenden Programms in Form eines entsprechenden mentalen Modells wird unter anderem von Ahmadzadeh et al. (2005) sowie Decasse und Emde (1988) eine besondere Bedeutung für ein erfolgreiches Debugging zugeschrieben. Darüber hinaus wenden erfahrene Programmierer:innen mehr Zeit für das Verständnis des vorliegenden Programms auf (Jeffries, 1982, zitiert nach McCauley et al., 2008) und verfügen weiterhin über detailliertere mentale Repräsentationen als die noch unerfahrenen Programmierer:innen (Fix et al., 1993),[9] welche zudem dazu neigen, die Zeilen des Quellcodes, ähnlich wie gewöhnliche Texte, von oben nach unten statt in der Reihenfolge der Ausführung zu lesen (Nanja & Cook, 1987, zitiert nach McCauley et al., 2008).[10]

Katz und Anderson (1987) beschreiben außerdem, dass die aufgewendete Zeit für die Entwicklung eines mentalen Modells des Programms variiert, je nachdem, ob es sich um selbst- oder fremdgeschriebenen Quellcode handelt. Dies scheint naheliegend, da die Debugger:innen selbstverfassten Quellcodes auf das bereits während der Implementierung entwickelte mentale

[9] Anmerkung: Nach McCauley et al. (2008) definiert Jeffries (1982) erfahrene Programmierer:innen als *advanced graduate students*. Fix et al. (1993) verstehen unter erfahrenen Programmierer:innen hingegen *professional programmer* mit durchschnittlich sieben Jahren Erfahrung. Absolvierende des ersten Kurses in der Programmierung bezeichnen Jeffries (1982) sowie auch Fix et al. (1993) als unerfahrene Programmierer:innen.

[10] Nach McCauley et al. (2008) definieren Nanja und Cook (1987) Studierende am Ende des zweiten Semesters als unerfahrene Programmierer:innen und Absolvierende hingegen als erfahrene Programmierer:innen.

Modell zurückgreifen können und daher weniger Zeit für das Verständnis des Programms benötigen als Personen, die fremdverfassten Quellcode debuggen. Der Ursprung des Programms wirkt sich des Weiteren auch auf die verwendete Debuggingstrategie aus (siehe Kapitel 2.4.3).

2.4.2 Identifikation fehlerhaften Systemverhaltens

Mit Hilfe des zuvor entwickelten Verständnisses des Programms erfolgt dann die Identifikation fehlerhaften Programmverhaltens. Hierzu wird das Programm häufig getestet, um Diskrepanzen zwischen dem intendierten und tatsächlichen Programmverhalten festzustellen (Carver & Risinger, 1987; Katz & Anderson, 1987; Kessler & Anderson, 1986). Eine solche Anomalie innerhalb des Systemverhaltens stellt den Ausgangspunkt für alle nachfolgenden Schritte des Debuggingprozesses dar.

Hier wird erneut die in Kapitel 2.1.1 beschriebene enge Zusammengehörigkeit von Debuggen und Testen ersichtlich, deren stellenweises Ineinandergreifen sich auch entlang der Fehlerbehebung (siehe Kapitel 2.4.4) zeigt.

2.4.3 Fehlerdiagnose

Im Rahmen der Fehlerdiagnose wird zunächst *experiential knowledge* herangezogen, um das vorliegende fehlerhafte Programmverhalten, falls vorhanden, mit vergleichbaren Beobachtungen früherer Fehler zu verknüpfen (Katz & Anderson, 1987) (siehe Kapitel 2.3). Decasse und Emde (1988) bezeichnen in einem spezifischen Kontext, jedoch über verschiedene Programme hinweg wiederholt auftretende Symptome und zugehörige Fehler als *stereotyped bugs*. Insofern ein solcher Fehler identifiziert wird, ermöglicht dessen Gleichartigkeit die Einleitung einer unmittelbaren Fehlerbehebung. Wachsendes *experience knowledge* (siehe Kapitel 2.4.5) erleichtert zudem diesen Prozess:

> „As time goes on, people may learn more and more of these mappings between bugs und behavior, until most of their debugging looks like simple mapping" (Katz & Anderson, 1987, S. 365).

Insofern das identifizierte fehlerhafte Systemverhalten nicht allein mit Hilfe des verfügbaren *experience knowledge* interpretiert und behoben werden kann, gilt es anschließend, die Möglichkeiten des Fehlerursprungs weitestgehend zu reduzieren. Hierbei ist das *strategic knowledge* von übergeordneter Bedeutung (siehe Kapitel 2.3). Nach Klahr und Carver (1988) sowie Carver und Risinger (1987) wirken sich außerdem die vorangegangenen Schritte in Form des Programmverständnisses (siehe Kapitel 2.4.1) und der Präzision der Fehlerbeschreibung (siehe Kapitel 2.4.2) auf die initiale Anzahl möglicher Fehlerursprünge und somit auf die Effizienz der Fehlerlokalisierung aus.

Analog zu der Unterscheidung zwischen globalen und lokalen Strategien im Kontext des *strategic knowledge* nach Jonassen und Hung (2006) wird auch der Debuggingprozess sowohl auf einer Makroebene betrachtet, die Vorgehensweisen umfasst, welche den Prozess als Ganzes abstrahieren und strukturieren, als auch auf einer Mikroebene beleuchtet, die wiederum

spezifische Methoden und Techniken für Teilschritte dieser allgemeinen Vorgehensweisen bereithält.

Yoon und Garcia (1998) differenzieren im Wesentlichen zwischen folgenden allgemeinen Vorgehensweisen (Globale Strategien): *Comprehension*-Strategie und *Isolation*-Strategie.[11]

Im Rahmen der *Comprehension*-Strategie werden die definierten Anforderungen an das Programm in Form des intendierten Programmverhaltens schrittweise verifiziert, indem es dem beobachtbaren Verhalten des tatsächlichen Programms gegenübergestellt wird, sodass potenzielle Diskrepanzen sichtbar werden. Dieser *top-down*-Prozess erfolgt zumeist entlang des *code tracings*, durch welches sowohl statische als auch dynamische Informationen über das Programm(-verhalten) gesammelt werden (Yoon und Garcia, 1998) (siehe Abbildung 2).

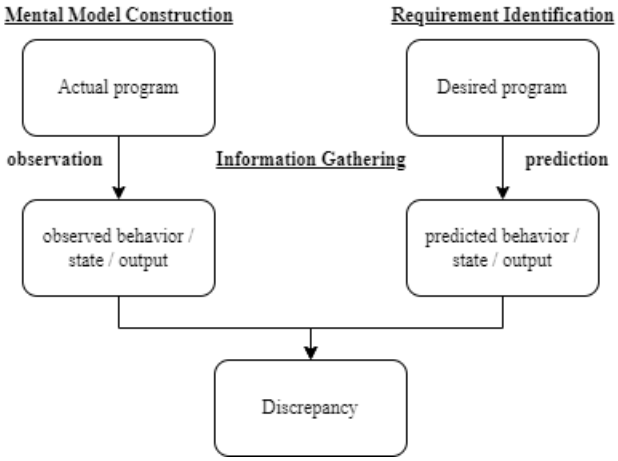


Abbildung 2: Schematische Darstellung der *Comprehension*-Strategie (Yoon & Garcia, 1998, S. 161)

Die Entwicklung eines mentalen Modells zu Beginn des Debuggingprozesses (siehe Kapitel 2.4.1), die Identifikation fehlerhaften Systemverhaltens (siehe Kapitel 2.4.2) und die Lokalisierung des Fehlerursprungs entlang der Fehlerdiagnose (siehe Kapitel 2.4.3) werden hier als separate, aufeinanderfolgende Prozessschritte dargestellt. Jedoch kann eine Verwendung der *Comprehension*-Strategie diese Schritte in Teilen miteinander verschmelzen lassen. Hierzu werden beispielsweise bereits während der Entwicklung eines Programmverständnisses entlang einer mentalen sequenziellen Ausführung des vorliegenden Programms Fehler und zugehörige Fehlerursachen identifiziert (Katz & Anderson, 1987).

Die *Isolation*-Strategie hingegen beschreibt einen *bottom-up*-Prozess, der sich im Wesentlichen durch die iterative Entwicklung, Überprüfung und gegebenenfalls Anpassung von Hypothesen über den Fehlerursprung auszeichnet. Fehlermeldungen sowie fehlerhaftes Systemverhalten sind mögliche Auslöser dieses Prozesses, welcher schließlich die Region des Fehlerursprungs, von Yoon und Garcia (1998) *focus of attention* genannt, fortlaufend weiter eingrenzen soll (Yoon & Garcia, 1998).

[11] Anmerkung: In der Literatur werden die *Comprehension*-Strategie und *Isolation*-Strategie oftmals auch als *forward reasoning* und *backward reasoning* bezeichnet (Katz & Anderson, 1987; Li et al., 2019).

Die *Comprehension*- und *Isolation*-Strategie greifen teilweise ineinander, um den Fehlerursprung zu bestimmen, indem die *Isolation*-Strategie für die Bestimmung des *focus of attention* und die *Comprehension*-Strategie anschließend für eine detaillierte Untersuchung dieser Region genutzt werden (Yoon & Garcia, 1998).

Zeller (2009) verwendet auch die Bezeichnung *scientific debugging* für die *Isolation*-Strategie, da sie im Grunde der wissenschaftlichen Methode für die Interpretation natürlicher Phänomene aus den Naturwissenschaften entspricht (Michaeli & Romeike, 2019c). Das fehlerhafte Programm fungiert hierbei als zu interpretierendes Phänomen. Nach Zeller (2009) untergliedert sich das *scientific debugging* in die folgenden fünf Schritte:

(1) Beobachtung eines Fehlers im Programm.
(2) Formulierung einer Hypothese über die Fehlerursache.
(3) Entwicklung von Annahmen, basierend auf der vorangegangenen Hypothese.
(4) Überprüfung und Modifikation der Hypothese (aus Schritt 2):
    a. Konkretisierung der Hypothese, falls die Annahmen verifiziert werden.
    b. Formulierung einer alternativen Hypothese, falls die Annahmen falsifiziert werden.
(5) Wiederholung der Schritte 3 und 4, bis die Hypothese weder erneuert noch weiter konkretisiert werden muss (Zeller, 2009).

Abbildung 3 veranschaulicht diesen iterativen Prozess, welcher im Anschluss hin zu einer finalen Diagnose überleitet, auf die schließlich die Behebung des Fehlers folgt. Nach Zeller (2009) erfordern die Herleitung sowie Anpassung einer Hypothese, die er als den kreativen Teil des Debuggings bezeichnet, neben des innerhalb des ersten Prozessschrittes (siehe Kapitel 2.4.1) generierten *system knowledge* (siehe Kapitel 2.3) und der Berücksichtigung vorangegangener Hypothesen unter anderem auch eine prägnante Beschreibung des Problems.

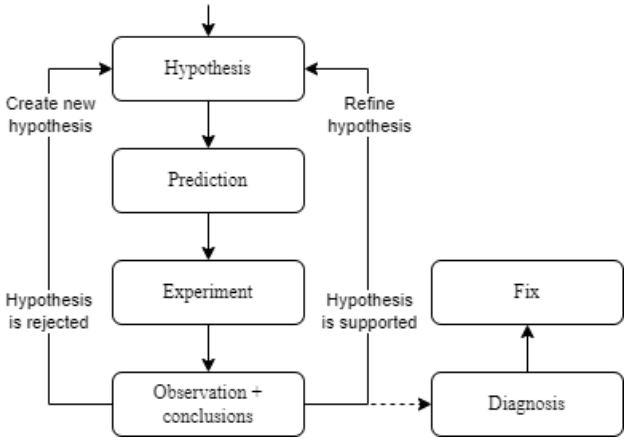


Abbildung 3: Schematische Darstellung des *scientific debuggings* (eigene Darstellung, in Anlehnung an Zeller, 2009, S. 131)

Katz und Anderson (1987) zeigen entlang einer Studie, in der die Teilnehmenden sowohl selbst- als auch fremdgeschriebene Programme debuggen sollten, dass die Wahl der globalen Strategie häufig durch den Ursprung des Programms determiniert wird. So tendieren Personen, die

selbstgeschriebenen Quellcode debuggen und daher bereits ein entsprechendes mentales Modell entwickeln konnten, dazu, die *Isolation*-Strategie zu verwenden. Hingegen ziehen Personen für das Debuggen fremdgeschriebenen Quellcodes eher die *Comprehension*-Strategie heran.

Darüber hinaus existieren zahlreiche spezifische Methoden und Techniken, die unabhängig von der jeweiligen allgemeinen Vorgehensweise im Rahmen der Fehlerdiagnose genutzt werden (Lokale Strategien). Hierzu gehören unter anderem der Gebrauch von *print*-Statements, des Debuggers, um die tatsächlichen Programmzustände zu spezifischen Zeitpunkten während der Ausführung zu überprüfen (Fitzgerald et al., 2008), oder auch die Berücksichtigung konkreter Fehlermeldungen für die Fehlersuche (Katz & Anderson, 1987). Ebenso unterstützen aussagekräftige Variablenbezeichnungen und Kommentare innerhalb des Quellcodes dessen Verständlichkeit und folglich auch die Fehlerdiagnose (Gould, 1975).

Die Suche nach dem Fehlerursprung eines fehlerhaften Programms wird häufig auch als der schwierigste Teil des Debuggings bezeichnet (Gould, 1975; Sedlmeyer et al., 1983). Insbesondere den leistungsschwächeren Debugger:innen fällt das Verwerfen sich als inkorrekt herausgestellter Hypothesen und damit einhergehend die Entwicklung alternativer Ideen über den Fehlerursprung schwer (Vessey, 1985). Je mehr Erfahrung eine Person jedoch aus vergangenen Debuggingprozessen sammeln konnte, desto einfacher fallen und schneller gelingen ihr die Identifizierung potenzieller Fehlerursachen und die Entwicklung akkurater Hypothesen (Gugerty & Olson, 1986; Zeller, 2009) (siehe Kapitel 2.4.5).

2.4.4 Fehlerbehebung

An die vorangegangene Fehlerdiagnose knüpft die Behebung des Fehlers an, indem der fehlerhafte Programmabschnitt korrigiert wird. Anschließend wird geprüft, ob das korrigierte Programm (1) den entsprechenden Fehler ausmerzen konnte und (2) keine weiteren, neu hinzugekommenen Fehler aufweist (Katz & Anderson, 1987; Klahr & Carver, 1988). Insofern sich das überarbeitete Programm weiterhin als fehlerhaft herausstellt, erfolgt eine Rückkopplung innerhalb des Debuggingprozesses hin zur Fehlerdiagnose (siehe Kapitel 2.4.3), um auch die übrigen Fehler zu beheben (Klahr & Carver, 1988). Hierzu wird das nun überarbeitete Programm erneut getestet, analog wie zur Identifikation fehlerhaften Systemverhaltens (siehe Kapitel 2.4.2) (Fitzgerald et al., 2008, Murphy et al., 2008).

Außerdem stellten Gugerty und Olson (1986) entlang einer Studie, in welcher das Vorgehen von Programmieranfänger:innen und erfahrenen Programmierer:innen während des Debuggings verglichen wurde, fest, dass die Anfänger:innen während der Fehlerbehebung unbeabsichtigt regelmäßig neue Fehler in das zu debuggende Programm integrierten.[12] Dies ließ sich

[12] Anmerkung: Die Klassifizierung von Programmieranfängerinnen und -anfängern und erfahrenen Programmiererinnen und Programmierern nach Gugerty und Olson (1986) entspricht jener von Jeffries (1982). Absolvierende des ersten Kurses in der Programmierung gelten als Programmieranfänger:innen. Unter erfahrenen Programmiererinnen und Programmierern werden *advanced graduate students* verstanden.

darauf zurückführen, dass sie im Falle eines Wechsels hin zu einer neuen Hypothese die bereits getätigten Anpassungen im Programm beibehielten statt sie zu widerrufen (Gugerty & Olson, 1986).

2.4.5 Speicherung erworbener Erfahrungen

Zuletzt wird das *experiential knowledge* um die während des Debuggingprozesses gewonnenen Erfahrungen erweitert (siehe Kapitel 2.3). Dem Lernen aus Fehlern wird im Kontext des Debuggings gemeinhin eine besondere Bedeutung zugesprochen. So widmet unter anderem Zeller (2009) dieser Thematik ein eigenständiges Kapitel, in welchem er ausführlich darlegt, wie die systematische Sammlung und Interpretation früherer Fehler das Risiko künftiger Fehler mindern sowie den Umgang mit ihnen erleichtern. Wie bereits in Kapitel 2.1.2 erwähnt, bezeichnen auch Michaeli und Romeike (2019a) die Nutzung von Erfahrungswissen in Form der Anwendung von *Heuristiken und Pattern* für die Fehlerdiagnose und -behebung als eine Charakteristik guter Debugger:innen.

3 Vermittlung von Debugging

Nachdem nun eine Einführung in das Debugging und eine schrittweise Strukturierung des Debuggingprozesses erfolgte (siehe Kapitel 2), widmet sich dieses Kapitel der Vermittlung von Debugging und soll zugehörige lehrspezifische Implikationen im Hinblick auf die Entwicklung einer Lernumgebung für ein systematisches Debugging (siehe Kapitel 5) formulieren. Dies erfordert die Klärung folgender Fragen: Welche konkreten Schlussfolgerungen für die Lehre lassen sich aus der vorangegangenen Betrachtung des Debuggingprozesses ableiten? Sind sie Bestandteil gegenwärtiger Tools und falls ja, wie erfolgt die Umsetzung? Hierzu werden zunächst Implikationen für die Vermittlung von Debugging(-fähigkeiten) definiert, die aus der vorangegangen Ausführung der Schritte innerhalb des Debuggingprozesses resultieren (siehe Kapitel 3.1). Anschließend erfolgt eine Untersuchung existierender Tools, die das Ziel verfolgen, Debugging zu vermitteln; nach einer Darlegung der Ausgangslage (siehe Kapitel 3.2.1), wird zunächst das Vorgehen beschrieben (siehe Kapitel 3.2.2), bevor schließlich Ergebnisse sowie hiermit zusammenhängende Rückschlüsse formuliert werden (siehe Kapitel 3.2.3).

3.1 Implikationen des Debuggingprozesses

Im Folgenden werden zunächst in Anknüpfung an die zuvor dargelegten Teilschritte des Debuggingprozesses vermittlungsspezifische Implikationen erläutert, die dann anschließend im Rahmen der Untersuchung bestehender Tools als denkbare Vermittlungsschwerpunkte von Debugging herangezogen werden.

Die Entwicklung eines mentalen Modells ist für das Verständnis des zu debuggenden Programms und infolgedessen für den Debuggingprozess als Ganzes essenziell (Ahmadzadeh et al., 2005; Decasse und Emde, 1988). Sowohl das *code tracing* als auch das *program chunking* nehmen hierbei entscheidende Rollen ein (McCauley et al., 2008; Murphy et al., 2008; Yoon & Garcia, 1998) und sollten daher gezielt gefördert werden (McCauley et al., 2008). Darüber hinaus sollten Programmieranfänger:innen mehr Zeit für das Programmverständnis aufwenden sowie darin unterstützt werden, den Quellcode in der Reihenfolge der Ausführung zu lesen (McCauley et al., 2008). Ebenso erleichtern adäquate Variablenbezeichnungen und Kommentare das Verständnis des Programms (Gould, 1975) (siehe Kapitel 2.4.1).

Innerhalb der Fehlerdiagnose werden zunächst vergleichbare Beobachtungen früherer Fehler herangezogen, welche bestenfalls die Einleitung einer unmittelbaren Fehlerbehebung ermöglichen (Decasse und Emde, 1988; Katz & Anderson, 1987). Andernfalls werden sowohl globale Strategien, entweder in Form der *Comprehension*-Strategie, *Isolation*-Strategie oder einer Kombination beider Strategien (Yoon und Garcia, 1998), als auch lokale Strategien, zu denen unter anderem der Gebrauch von *print*-Statements oder des Debuggers zählen, verwendet, um den Fehlerursprung schrittweise weiter einzugrenzen (Fitzgerald et al., 2008). Die Suche nach dem Fehlerursprung gilt als besonders herausfordernd (Gould, 1975; Sedlmeyer et al., 1983). Insbesondere das Verwerfen sich als inkorrekt herausgestellter Hypothesen sowie die

anschließende Entwicklung alternativer Ideen über den Fehlerursprung bereiten leistungsschwächeren Debugger:innen Schwierigkeiten (Vessey, 1985) und sollten daher ebenso in der Lehre adressiert werden (McCauley et al., 2008). Außerdem unterstützt auch wachsendes Erfahrungswissen die Hypothesenentwicklung sowie -anpassung (Gugerty & Olson, 1986; Zeller, 2009) (siehe Kapitel 2.4.3).

Im Rahmen der Fehlerbehebung erfolgen die Korrektur des fehlerhaften Programmabschnitts und eine anschließende Testung des Programms, die im Gegensatz zur Identifikation fehlerhaften Systemverhaltens (siehe Kapitel 2.4.2) außerdem prüfen soll, dass in Folge der Korrektur keine neuen Fehler in das Programm integriert wurden (Katz & Anderson, 1987; Klahr & Carver, 1988). Insbesondere Programmieranfänger:innen neigen dazu, während der Fehlerbehebung unbeabsichtigt neue Fehler zum jeweiligen Programm hinzuzufügen (Gugerty & Olson, 1986). Daher sollten auch das Widerrufen von Änderungen im Programm, die aufgrund sich als fehlerhaft herausgestellter Hypothesen vorgenommen wurden, sowie die allgemeine Förderung des Denkens in Form eines *hypothesis testing modes* innerhalb der Vermittlung von Debugging berücksichtigt werden (McCauley et al., 2008) (siehe Kapitel 2.4.4).

Das Lernen aus Fehlern nimmt eine entscheidende Rolle im Kontext des Debuggings ein (Zeller, 2004). Verschiedene Autor:innen schlagen hierzu die Dokumentation des Debuggingprozesses mit Hilfe eines Logbuchs vor, das unter anderem eine Historie der aufgestellten Hypothesen und Fehler, inklusive der zugehörigen Fehlerursachen, umfasst. Ein solches Logbuch wäre als ständiger Begleiter für das Debugging denkbar, indem es kontinuierlich aktualisiert und im Rahmen späterer Debuggingprozesse herangezogen wird (Siegmund et al., 2014; Yoon & Garcia, 1998) (siehe Kapitel 2.4.5).

3.2 Untersuchung bestehender Tools

3.2.1 Ausgangslage

Li et al. (2019) untersuchten neun Tools, die das Ziel verfolgen, Aspekte des Debuggings zu lehren. Hierzu ordneten sie die Tools jeweils den Wissenskategorien des Troubleshootingprozesses nach Jonassen und Hung (2006) (siehe Kapitel 2.3) zu, die sie schwerpunktmäßig adressieren (siehe Tabelle 1). Dabei ergänzten sie die bisherigen Wissenskategorien um die Kategorie *iterative knowledge*, welche auf den sich wiederholenden Charakter des Debuggingprozesses in Form der Entwicklung, Überprüfung und gegebenenfalls Anpassung von Hypothesen über den Fehlerursprung (siehe Kapitel 2.4.3) verweist.

Knowledge	Detail	Research
Domain	Read/write code	[2, 16–19, 26]
System	Functional	[11, 16, 17, 19]
	Topographical	[17]
Strategic Global	Forward trace	[2, 9, 14, 16–19, 26]
	Backward trace	[17]
	Breadth-first	[26]
	Discrepancy detect	[2, 9, 11, 14, 16–19, 26]
Strategic Local	Print/debugger	[17, 26]
	Observe output	[2, 16–18, 26]
	Error message	[2, 14, 16, 18]
	Test cases	[26]
Experience		[2, 16, 18]
Iterative		[11]

Tabelle 1: Einordnung bestehender Debugging-Tools (Li et al., 2019, S. 84)

Tabelle 1 zeigt, dass zwei Drittel der untersuchten Tools das Lesen und/oder Schreiben von Quellcode umfassen. Damit adressiert ein Großteil der Tools das *domain knowledge* der Lernenden. Außerdem stellen die Autor:innen fest, dass vier der neun Tools das *system knowledge* fördern, indem sie beispielsweise Erläuterungen oder alternative Darstellungsformen des Quellcodes, unter anderem in Form von Visualisierungen der Programmzustände sowie -struktur, bereitstellen. Im Hinblick auf das *strategic knowledge* wird ersichtlich, dass zwar beinahe alle der untersuchten Tools eine Anwendung der *Comprehension*-Strategie erlauben. Jedoch unterstützt lediglich eines der Tools die *Isolation*-Strategie, indem die Programmausführung in umgekehrter Richtung mit Hilfe eines Debuggers nachvollzogen werden kann.[13] Darüber hinaus adressieren die Tools verschiedene lokale Strategien; zum Beispiel die Nutzung von *print*-Statements, das Setzen von Breakpoints oder die Interpretation von Fehlermeldungen. Sowohl das *experience knowledge* als auch das *iterative knowledge* werden jedoch von den Tools kaum berücksichtigt (Li et al., 2019).

Im Hinblick auf die spärliche Unterstützung der *Isolation*-Strategie innerhalb der von Li et al. (2019) untersuchten Tools bei einem gleichzeitig hohen Stellenwert dieser globalen Vorgehensweise, insbesondere für das Debuggen selbstgeschriebenen Quellcodes (siehe Kapitel 2.4.3), soll die Konzeption der Lernumgebung (siehe Kapitel 5) schwerpunktmäßig die Vermittlung eines systematischen Vorgehens entlang der *Isolation*-Strategie als übergeordnetes Ziel anstreben. Die Förderung dieser Strategie in Form einer sich wiederholenden Entwicklung, Überprüfung und gegebenenfalls Anpassung von Hypothesen über den Fehlerursprung bietet außerdem das Potenzial, das von den Tools ebenso kaum berücksichtigte *iterative knowledge* zu adressieren. In Anbetracht der Relevanz des Lernens aus Fehlern innerhalb des Debuggings (siehe Kapitel 2.4.5) und dessen lediglich seltener Berücksichtigung in den untersuchten Tools plädieren Li et al. (2019) darüber hinaus für eine stärkere Einbeziehung des *experience knowledge*,

[13] Anmerkung: Li et al. (2019) bezeichnen die *Comprehension*-Strategie als *forward trace* und die *Isolation*-Strategie als *backward trace*.

insbesondere entlang des Erinnerns an und der Verwendung von *stereotyped bugs* (siehe Kapitel 2.4.3). Da das *experience knowledge* eng mit der Fehlerdiagnose verbunden ist, indem unter anderem fehlerhaftes Programmverhalten zunächst mit, falls vorhanden, vergleichbaren Beobachtungen früherer Fehler verglichen wird und sich wachsendes Erfahrungswissen auch auf die Entwicklung akkurater Hypothesen auswirkt (siehe Kapitel 2.4.3), ergibt sich basierend auf der gewählten Schwerpunktsetzung ebenso der Anspruch, das *experience knowledge* als begleitenden Einflussfaktor der *Isolation*-Strategie mit zu fördern.

3.2.2 Vorgehen

Das Ziel dieser Untersuchung ist eine detaillierte Betrachtung bereits verfügbarer Tools, die sowohl das Ziel verfolgen, Debugging(-fähigkeiten) zu vermitteln als auch die *Isolation*-Strategie, das *experience knowledge* oder das *iterative knowledge* zu adressieren. So sollen im Hinblick auf die spätere Entwicklung der Lernumgebung (siehe Kapitel 5) unter Berücksichtigung der begründeten Wahl der Schwerpunktsetzung sowohl *Redundanzen* vermieden als auch *Potenziale* identifiziert werden, die in existierenden Tools noch nicht ausgeschöpft werden. Die sich hieraus ergebenden Alleinstellungsmerkmale verdeutlichen zudem die (Praxis-)Relevanz der Lernumgebung.

Unter Einbeziehung der Kategorisierung von Debuggingtools nach Li et al. (2019) entsprechen die folgenden fünf Tools dieser Auswahl:

- ViLLE (Laakso et al., 2008),
- ITS-Debug (Carter, 2014),
- Intelligent Tutoring System (Kumar, 2002),
- DebugIt (Lee & Wu, 1999),
- PLATO FAULT (Johnson et al., 1982).

Zu all diesen Tools liegen wissenschaftliche Veröffentlichungen vor. Die Verfügbarkeit des jeweiligen Tools zur eigenständigen Nutzung beschränkt sich auf *ViLLE*; ein direkter Zugriff auf die übrigen Tools war nicht möglich. Die Betrachtung der Tools *ITS-Debug*, *Intelligent Tutoring System*, *DebugIt* und *PLATO FAULT* erfolgt daher ausschließlich auf Grundlage der zugehörigen Veröffentlichungen.

Zur Sicherstellung der Vergleichbarkeit der Ergebnisse werden die genannten Tools entlang desselben Vorgehens untersucht: Hierzu werden die Tools zwecks Einordnung zunächst grundlegend hinsichtlich ihrer Idee umrissen. Anschließend wird beschrieben, *wie* die Tools die jeweilige Wissenskategorie, der sie zugeordnet wurden (*strategic knowledge* in Form der *Isolation*-Strategie, *experience knowledge* oder *iterative knowledge*) adressieren. Hierbei wird insbesondere verglichen, inwieweit die lehrspezifischen Implikationen des Debuggingprozesses (siehe Kapitel 3.1) innerhalb gegenwärtiger Tools bereits realisiert werden.

3.2.3 Ergebnisse und Implikationen

Eines der Tools, welches nach Li et al. (2019) insbesondere das *strategic knowledge* in Form der *Comprehension*- sowie *Isolation*-Strategie adressiert und von Laakso et al. (2008) entwickelt wurde, ist *ViLLE*. Der Schwerpunkt des Tools ist die Vermittlung von Debugging entlang einer gezielten Förderung des Programmverständnisses. Hierzu kombiniert *ViLLE* zahlreiche Visualisierungshilfen mit einer Unterstützung des Konzeptes der *roles of variables*, welches Variablen wiederkehrenden Mustern in Form von stereotypen Rollen zuordnet. Häufig auftretende Rollen sind zum Beispiel Variablen, deren jeweiliger Wert nach der Initialisierung nicht mehr geändert wird (*constant*) oder Variablen, welche für das Iterieren durch Datenstrukturen genutzt werden und hierfür die Indizes zwischenspeichern (*stepper*) (Sajaniemi, 2002). *ViLLE* kann nach Laakso et al. (2008) sowohl im Rahmen des Unterrichts als auch für das Selbstlernen verwendet werden und richtet sich dabei vor allem an Programmieranfänger:innen.

Konkrete Funktionalitäten des Tools sind unter anderem die Bereitstellung von Erläuterungen zu allen Zeilen des Quellcodes, welche darüber hinaus auch Informationen über die jeweiligen Rollen der Variablen umfassen, eine Übersetzung des Quellcodes in andere Programmiersprachen und Pseudocode sowie eine Visualisierung des Aufrufstapels des Programms. Darüber hinaus können Pop-Ups zur Wissensabfrage genutzt werden. Die Unterstützung der *Isolation*-Strategie erfolgt primär entlang der Möglichkeit, die Programmausführung sowohl vor- als auch rückwärts nachzuverfolgen (Laakso et al., 2008) (siehe Abbildung 4).

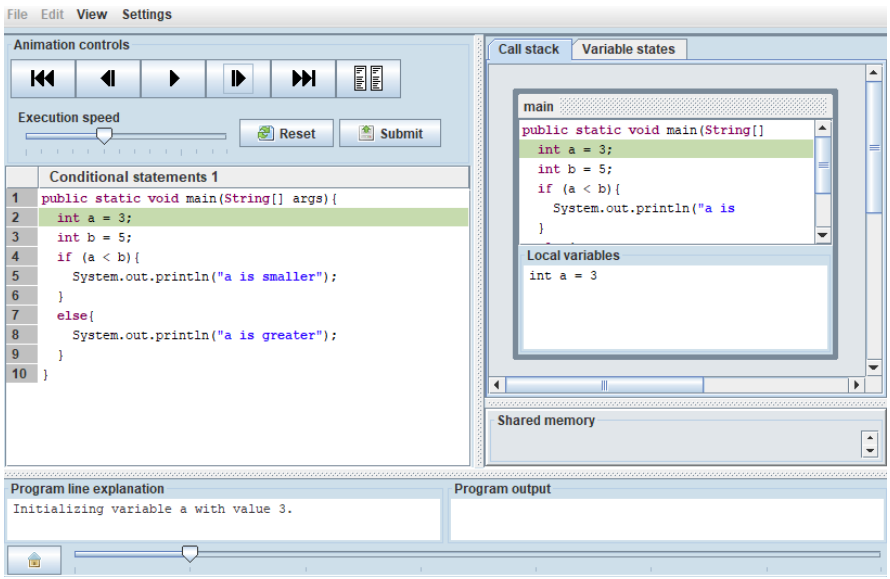


Abbildung 4: Nutzung von ViLLE[14] (eigene Darstellung)

Nach Li et al. (2019) adressieren drei der untersuchten Tools das *experience knowledge*. Eines dieser Tools ist *ITS-Debug*, ein *intelligent tutoring system* (ITS) für Programmieranfänger:innen mit dem Schwerpunkt, Debugging entlang einer Nutzung des *case-based reasoning* (CBR)

[14] Anmerkung: Die Nutzung von ViLLE erfolgte mit der Version 3.0.

zu vermitteln (Carter, 2014). Die Idee von CBR ist, dass, ähnlich zu den innerhalb des Kapitel 2.4.3 beschriebenen *stereotyped bugs*, für die Lösung gegenwärtiger Probleme frühere Lösungen vergleichbarer Probleme herangezogen und schließlich um das neu aufgetretene Problem ergänzt und aktualisiert werden (Mousavinasab et al., 2021).

Dies ermöglicht *ITS-Debug*, spezifische sowie individuelle Rückmeldungen bereitzustellen, welche die Lernenden in der Fehlerbehebung unterstützen sollen. Das *experience knowledge* der Lernenden wird hierbei insbesondere dahin gehend adressiert, dass insofern der/die Lernende bereits früher auf einen vergleichbaren Fehler gestoßen ist, ihr/ihm eine Zusammenfassung dieser Begegnung zur Verfügung gestellt wird (Carter, 2014).

Intelligent Tutoring System ist ein weiteres ITS, welches nach Li et al. (2019) das *experience knowledge* berücksichtigt und hierbei *model-based reasoning* (MBR) nutzt. In MBR wird zunächst ein Modell der jeweiligen Domäne konstruiert und anschließend verwendet, um Verhalten innerhalb dieser Domäne zu simulieren. Im Kontext des konkreten Tools stellt die Programmiersprache C++ die Domäne dar, dessen Modell genutzt wird, um das Verhalten während der Ausführung verschiedener Programme vorherzusagen (Kumar, 2002).

Die Verwendung von MBR sowie von *templates* für die Generierung von Problemen erlaubt *Intelligent Tutoring System* eine praktisch unbegrenzte Anzahl von Problemfällen zu erzeugen. Dies führt wiederum dazu, dass die Lernenden mit zahlreichen verschiedenen Problemen konfrontiert werden und somit unter anderem auch ihr *experience knowledge* erweitern können (siehe Kapitel 2.4.5) (Kumar, 2002).

Ein weiteres Tool, welches nach Li et al. (2019) das *experience knowledge* adressiert, ist *DebugIt*. Auch hierin berücksichtigen die Autoren die Relevanz von häufigen Übungen und des hiermit verbundenen wachsenden Erfahrungswissens für die Verbesserung der Debuggingfähigkeiten, indem *DebugIt* zahlreiche Fehler umfasst, die häufig während der Verwendung von Schleifen und insbesondere unter den Programmieranfängerinnen und -anfängern auftreten (Lee & Wu, 1999).

Das Tool *PLATO FAULT*, welches die Fehlersuche sowie -behebung innerhalb des Troubleshootingprozesses entlang verschiedener spielerischer Elemente vermitteln soll, ist nach Li et al. (2019) das Einzige unter den untersuchten Tools, welches das *iterative knowledge* adressiert. Hierzu erlaubt *PLATO FAULT* den Lernenden, sich abhängig von den vorangegangenen Schritten den verbleibenden Problemraum in Form der Anzahl der noch übrigen potenziellen Fehlerquellen anzeigen zu lassen (Johnson et al., 1982).

Die Untersuchung zeigt auf, dass obgleich einige Tools existieren, die sich der Vermittlung von Debugging widmen, nicht alle der innerhalb des Debuggingprozesses relevanten Wissensarten gleichermaßen von ihnen abgedeckt werden. Eine nähere Betrachtung der Tools, welche das *strategic knowledge* in Form der *Isolation*-Strategie, das *experience knowledge* oder das *iterative knowledge* adressieren, zeigt außerdem, dass lehrspezifische Implikationen der einschlägigen Literatur für die Vermittlung von Debugging in existierenden Tools nicht immer

berücksichtigt werden und offenbart damit zugleich Potenziale im Hinblick auf die spätere Entwicklung der Lernumgebung (siehe Kapitel 5).

So werden beispielsweise weder das Vorgehen innerhalb der *Isolation*-Strategie in Form einer iterativen Entwicklung, Überprüfung und gegebenenfalls Anpassung von Hypothesen über den Fehlerursprung (siehe Kapitel 2.4.3) noch die hiermit verbundenen Hürden für leistungsschwächere Debuggerinnen und Debugger (siehe Kapitel 3.1) adressiert.

Des Weiteren wird im Hinblick auf das *experience knowledge* ersichtlich, dass eine Berücksichtigung innerhalb der vorgestellten Tools vor allem entlang einer Bereitstellung möglichst vieler Übungsgelegenheiten erfolgt. Eine Integration von Logbüchern, die von verschiedenen Autor:innen zur Unterstützung während des Debuggings vorgeschlagen wird (siehe Kapitel 3.1), ist lediglich im Ansatz zu erkennen, indem *ITS-Debug* Zusammenfassungen früherer Fehlerbegegnungen bereitstellt, insofern vergleichbare Fehler erneut auftreten.

4 Gestaltung digitaler Lernumgebungen

Im Anschluss an die vorangegangenen Betrachtungen hinsichtlich der Entwicklung einer Lernumgebung für das systematische Debugging aus informatischer Perspektive (siehe Kapitel 2, Kapitel 3) wird das folgende Kapitel aus einer (medien-)didaktischen Betrachtungsweise heraus sowohl die innere Vermittlungsstrategie als auch die äußere Gestaltung digitaler Lernumgebungen betrachten. Hierbei sollen folgende Fragen geklärt werden: Welche innere Vermittlungsstrategie eignet sich für die zu entwickelnde Lernumgebung und welche Aspekte sollen im Hinblick auf dessen äußere Gestaltung berücksichtigt werden? Hierzu erfolgt zunächst eine Einordnung digitaler Lernumgebungen (siehe Kapitel 4.1), bevor über die Lerntheorien (siehe Kapitel 4.2.1) hin zum *Cognitive Apprenticeship*-Ansatz (CAA) als Instruktionsansatz zur Gestaltung von Lernumgebungen geleitet wird (siehe Kapitel 4.2.2). Im weiteren Verlauf wird die *Cognitive Theory of Multimedia Learning* (CTML), dessen Zusammenhang mit der *Cognitive Load Theory* (CLT) (siehe Kapitel 4.3.1), ihre Annahmen, Prozesse (siehe Kapitel 4.3.2) und Empfehlungen in Form von verschiedenen Prinzipien multimedialen Designs beschrieben (siehe Kapitel 4.3.3).

4.1 Einordnung

Im Zuge der voranschreitenden Digitalisierung gewinnt die Nutzung digitaler Lehr- und Lernmittel für die Vermittlung von Wissen stetig an Bedeutung. Die zugehörige Bezeichnung *electronic learning* (E-Learning) wird in der Literatur teils mehr, teils minder restriktiv verwendet. Während eine eher weite Definition von E-Learning die Nutzung sämtlicher elektronischer Medien für das Lernen umfasst, definieren Vertreter:innen eines eher engen Begriffsverständnisses E-Learning als ein computerunterstütztes und/oder webbasiertes Lernen (Flindt, 2005; Moore et al., 2011). Digitale Lernumgebungen in Form von Lernsoftware bzw. Lernprogrammen sind nach beiden Definitionen ein Bestandteil von E-Learning (Flindt, 2005). Hierbei wird häufig zwischen *computer-based training* (CBT) und *web-based training* (WBT) differenziert.

CBT beschreibt eine computerbasierte Unterstützung des Lernprozesses entlang lokal bzw. offline betriebener Lernsoftware (Kumar Basak et al., 2018; Mandl & Winkler, 2004). WBT hingegen stellt nach Mandl und Winkler (2004) sowie Flindt (2005) eine Weiterentwicklung des CBT dar, indem die Verteilung der sowie der Zugriff auf die Lernsoftware netzwerkbasiert, zum Beispiel über das Internet, erfolgt (Kumar Basak et al., 2018; Mandl & Winkler, 2004). Dies bringt wiederum verschiedene Vorteile mit sich; darunter unter anderem eine weltweite, ortsunabhängige Verfügbarkeit und eine plattformübergreifende Nutzung des Dienstes (Manochehr, 2006), insofern ein Internetzugang vorhanden ist.

4.2 Lehr-Lerntheoretische Grundlagen

4.2.1 Lerntheorien

Die lehr-lernpsychologische Forschung differenziert im Wesentlichen zwischen drei Paradigmen, die jeweils unterschiedliche Vorstellungen des menschlichen Lernprozesses vertreten.

Das Lernen nach *behavioristischem* Verständnis entspricht einer Veränderung des beobachtbaren Verhaltens als Reaktion auf einen vorangegangenen Reiz, der von außen auf das Individuum einwirkt. Neben dieser *klassischen Konditionierung* nimmt hierbei insbesondere auch die *operante Konditionierung* eine bedeutende Rolle ein, indem sie positive und negative Konsequenzen nutzt, um erwünschte Reaktionen zu verstärken sowie unerwünschte zu schwächen (Gräsel & Gniewosz, 2011; Mietzel, 2017).

Der *Kognitivismus* hingegen versteht Lernen als Prozess der Informationsverarbeitung. Einen entsprechend hohen Stellenwert nehmen daher kognitive Strukturen und Prozesse innerhalb kognitivistischer Ansätze ein. Die entlang des Behaviorismus beschriebene, externe Steuerbarkeit von Lernprozessen bleibt erhalten, während die Rolle der Lernenden innerhalb des Lernprozesses nun jedoch über die bloße Reaktion auf äußere Reize hinausgeht (Gräsel & Gniewosz, 2011; Mietzel, 2017).

Die *konstruktivistische* Sichtweise beschreibt Lernen als eine aktive sowie selbstgesteuerte Konstruktion von Wissen, die stark von der Individualität der Lernenden, zum Beispiel in Form von unterschiedlichen Vorerfahrungen, geprägt wird. Lernenden werden innerhalb dieses Paradigmas eine zentrale Rolle zugeschrieben, da das Wissen entgegen den vorangegangenen Paradigmen nicht an sie herangetragen, sondern erst durch sie konstruiert wird (Gräsel & Gniewosz, 2011; Mietzel, 2017).

Siemens (2005) argumentiert, dass sich das Lernen in der sich heute rasch verändernden Informationsgesellschaft gewandelt hat und ergänzt die klassischen Lerntheorien um einen neuen, noch jungen Ansatz, den *Konnektivismus*. Dessen Grundannahme besteht darin, dass Lernen maßgeblich durch die Vernetzung von Individuen in Form von Netzwerken geprägt wird (Siemens, 2005).

4.2.2 *Cognitive Apprenticeship*-Ansatz

Einordnung

Ein konstruktivistisch geprägter Instruktionsansatz zur Gestaltung von Lernumgebungen ist der CAA nach Collins et al. (1987). Dieser balanciert das Verhältnis der Aktivitäten von Lehrenden und Lernenden kontinuierlich neu, mit dem Ziel, strategisches Wissen zur Lösung variierender Problemstellungen zu vermitteln (Gerstenmaier & Mandl, 1995; Sonntag & Stegmaier, 2007).

Der CAA gilt als eine Methode des *situated learning*, das Wissen als stark kontextabhängig begreift und daher eine enge Anbindung der Lernprozesse an reale Anwendungssituationen fordert (Anderson et al., 1996). Weitere populäre Methoden des *situated learning* sind der

Anchored Instruction- sowie *Cognitive Flexibility*-Ansatz. Während der *Anchored Instruction*-Ansatz eine authentische Gestaltung von Problemsituationen entlang einer narrativen Darstellung des jeweiligen Problems verfolgt, um das Interesse der Lernenden zu wecken, strebt der *Cognitive Flexibility*-Ansatz multiple Wissensrepräsentationen an, die je nach Problemsituation flexibel herangezogen und miteinander kombiniert werden können (Gerstenmaier & Mandl, 1995; Sonntag & Stegmaier, 2007).

Methoden und Prinzipien

Collins et al. (1987) definieren sechs Phasen des CAA, die im Folgenden näher erläutert werden:

(1) *Modelling:* Zunächst entwickeln die Lernenden ein konzeptuelles Modell der Vorgehensweise zum Problemlösen, indem die kognitiven Prozesse und Aktivitäten einer/eines Expert:in während der Problembewältigung externalisiert und von den Lernenden beobachtet werden.
(2) *Coaching:* Das Vorgehen der Lernenden soll sich dem der/des Expert:in möglichst annähern. Hierzu werden die Lernenden begleitend zum Problemlöseprozess entlang von interaktiven sowie individuellen Hinweisen in Form von Feedback und Vorschlägen betreut.
(3) *Scaffolding:* Hierunter fallen Maßnahmen, welche über eine Betreuung der Lernenden hinausgehen und sie gezielt im Problemlösen unterstützen sollen; auch eine Reduktion des Problemlöseprozesses um bestimmte Aspekte ist denkbar. Die Lernenden sollen möglichst eigenständig arbeiten. Daher erfolgt die Unterstützung stets bedarfsgeleitet und wird sukzessive verringert (*fading*).
(4) *Articulation:* Die Artikulation des Wissens sowie Vorgehens während des Problemlösens seitens der Lernenden ermöglicht ihnen, ein Bewusstsein für die eigenen Problemlösefähigkeiten zu schaffen.
(5) *Reflection:* Ein Vergleich des Problemlösens der Lernenden mit jenen der anderen Lernenden oder einer/eines Expert:in regt eine Reflexion über den eigenen Problemlöseprozess an.
(6) *Exploration*: Entlang der allmählich abnehmenden Unterstützung gewinnen die Lernenden in ihrem Problemlösen schrittweise an Selbständigkeit, bis sie schließlich autonom agieren und lernen (Collins et al., 1987).

Modelling, *coaching*, *scaffolding* sowie *fading* repräsentieren hierbei den Kern des Lernprozesses, worin Lernende entlang von Beobachtung und unterstützter Anwendung die grundlegenden Fähigkeiten für die Problemlösung erwerben (Collins et al., 1987).

Collins et al. (1987) formulieren darüber hinaus drei Prinzipien, welche im Hinblick auf die Abfolge der Lernaktivitäten berücksichtigt werden sollten. Zum einen wird empfohlen, dass die Komplexität der Aufgaben stetig weiter zunimmt. Zum anderen soll ebenso die Anzahl der hierin benötigten Fähigkeiten zunehmend steigen. Des Weiteren betonen Collins et al. (1987), dass die frühe Entwicklung einer Vorstellung von dem beabsichtigten Lernziel entlang einer

Darstellung des Gesamtprozesses sowohl das Verständnis der Zusammenhänge spezifischer Teilschritte fördert als auch das Auftreten von Verständnisfehlern verringert.

Im Hinblick auf soziologische Aspekte heben Collins et al. (1987) außerdem die Bedeutung eines Realitätsbezugs der Lernumgebung sowie des Austauschs und der Zusammenarbeit hervor, die nicht zuletzt die Motivation der Lernenden steigern sollen.

Anwendung

Es existieren zahlreiche Publikationen, die eine Integration des CAA in digitale Lernumgebungen beschreiben. So untersuchten Gräsel und Mandl (1993) die Förderung diagnostischer Strategien unter Medizinstudierenden entlang eines fallbasierten Lernprogramms und stellten hierbei fest, dass eine ergänzende instruktionale Unterstützung, die sich an dem CAA orientiert, das strategische Vorgehen während der Lösung eines Transferfalls positiv beeinflusst.

Auch Elting (1996) zeigt entlang eines Lernprogramms für die Vermittlung des Sortierens mittels AVL-Bäume, dessen Gestaltung sich stark an den Phasen des CAA (siehe Kapitel 4.2.2, Methoden und Prinzipien) orientiert, dass eine Umsetzung dieses Ansatzes im Rahmen einer digitalen Lernumgebung sowohl auf große Akzeptanz der Nutzer:innen führen kann als auch das Potenzial besitzt, deren Lernzuwachs deutlich zu steigern.

Liu (2005) beschreibt eine Integration des CAA in die Lehrerbildung, indem den Lernenden mit Hilfe multimedialer Unterstützung das Vorgehen in der Unterrichtsplanung zunächst demonstriert und von ihnen anschließend angeleitet eingeübt wird. Neben einer Förderung des Verständnisses für die Integration von Technologie in die Lehre, zeigt auch diese Arbeit die lernförderliche Wirkung des Ansatzes in Form einer Leistungssteigerung sowie einer positiveren Haltung gegenüber der Unterrichtsplanung unter den Lernenden.

Saadati et al. (2014) führen das *Internet-Based Cognitive Apprenticeship Model* ein, welches die ursprünglichen sechs Phasen des CAA (siehe Kapitel 4.2.2, Methoden und Prinzipien) innerhalb der drei Phasen *Handling*, *Supporting* und *Self-Exploring* zusammenfasst. Die Integration dieses Modells in einen Blended Learning-Kurs zum Thema Statistik führte zu einer signifikanten Steigerung der Leistungen von Lernenden im statistischen Problemlösen.

Obgleich die dargelegten Studien lediglich einen kleinen Ausschnitt aus der Literatur, welche sich der Integration des CAA in digitale Lernumgebungen widmet, repräsentieren, zeigen sich bereits hier die disziplinenübergreifenden positiven Effekte auf den Lernprozess; insbesondere auf die Leistungsfähigkeit der Lernenden. Daher wird sich ebenso die Vermittlung von Debugging entlang der im Rahmen dieser Arbeit zu entwickelnden Lernumgebung (siehe Kapitel 5) an dem CAA orientieren.

4.3 Cognitive Theory of Multimedia Learning

4.3.1 Cognitive Load Theory

Die Gedächtnisforschung differenziert zwischen einer kurzfristigen Informationsspeicherung (Sensorisches Gedächtnis und Arbeitsgedächtnis) sowie einer langfristigen Informationsspeicherung (Langzeitgedächtnis) (Gruber, 2018). Innerhalb des Wissenserwerbs nimmt das Arbeitsgedächtnis eine besondere Rolle ein; dessen Kapazität ist jedoch auf sieben plus/minus zwei Einträge limitiert (Miller, 1956), die für etwa 20 Sekunden aufrechterhalten werden können (Chandler & Sweller, 1991).[15] Die CLT nach Chandler und Sweller (1991) knüpft hieran an, indem sie die verschiedenen Arten kognitiver Belastung definiert und ihren jeweiligen Einfluss auf den Wissenserwerb beschreibt. Sie werden im Folgenden näher beschrieben.

Die *intrinsic cognitive load* wird von der Komplexität der Lerninhalte bestimmt. Je stärker die inneren Zusammenhänge der Lerninhalte, desto größer ist die *intrinsic cognitive load*. Die *extraneous cognitive load* hingegen hängt von der Art der Vermittlung und Darstellung der Lerninhalte ab (Paas et al., 2010). Beide Arten kognitiver Belastung sollten stets möglichst gering gehalten werden; insbesondere aber wenn die *intrinsic cognitive load* groß ist, ist eine geringe *extraneous cognitive load* wichtig, um die Kapazität des Arbeitsgedächtnisses nicht zu überschreiten. Auf diese Weise stehen der *germane cognitive load*, welche die lernbezogene Belastung, die für das Verständnis der Lerninhalte aufgewendet wird, beschreibt, mehr Kapazitäten zur Verfügung. Im Gegensatz zur *intrinsic cognitive load* und *extraneous cognitive load* wirkt sie sich positiv auf den Lernprozess aus und sollte entsprechend gefördert werden (Paas et al., 2010).

Die CTML orientiert sich an den Erkenntnissen der CTL, weicht jedoch teilweise begrifflich leicht ab. So werden in der CTML die *intrinsic cognitive load* als *essential processing*, die *extraneous cognitive load* als *extraneous processing* und die *germane cognitive load* als *generative processing* bezeichnet (Mayer, 2020).

4.3.2 Annahmen und Prozesse

Mayer (2020) formuliert drei Vorannahmen, die das Fundament der CTML bilden. Sie werden nachfolgend näher dargelegt.

Die *Dual-Channels*-Annahme besagt, dass Menschen über zwei separate Kanäle für die Informationsverarbeitung verfügen. Einer der Kanäle wird hierbei für die Verarbeitung visueller Informationen in Form von Bildern und des geschriebenen Wortes und der andere für die Verarbeitung auditiver Informationen in Form des gesprochenen Wortes verwendet (Mayer, 2020). In Abbildung 5 wird die *Dual-Channels*-Annahme entlang der Darstellung sowohl eines

[15] Anmerkung: Im Rahmen der *Cognitive Load Theory* nach Mayer (2020) wird dies ebenso innerhalb der *Limited Capacity*-Annahme beschrieben.

auditiven (oben) als auch eines visuellen Kanals der Informationsverarbeitung (unten) innerhalb des sensorischen Gedächtnisses sowie des Arbeitsgedächtnisses ersichtlich.

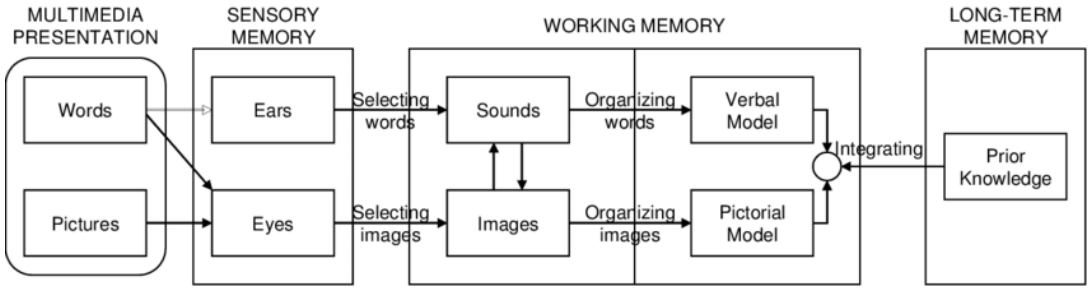


Abbildung 5: Schematische Darstellung der CTML (Mayer, 2020, S. 40)

Die *Limited Capacity*-Annahme beschreibt analog zu Miller (1956) sowie Chandler und Sweller (1991), dass die Menge der Informationen, die in beiden Kanälen zur selben Zeit verarbeitet werden können, limitiert ist (Mayer, 2020).

Die *Active Processing*-Annahme hebt die aktive Rolle der Lernenden in der Wissensverarbeitung hervor, indem sie relevante Informationen identifizieren, auswählen, innerhalb kohärenter mentaler Repräsentationen organisieren und sowohl miteinander verbinden als auch in das bestehende Vorwissen integrieren (Mayer, 2020). Diese kognitiven Prozesse werden in Abbildung 5 mit Hilfe der beschrifteten Pfeile *Selecting words/images*, *Organizing words/images* und *Integrating* dargestellt.

4.3.3 Prinzipien multimedialen Designs

Die CTML und die hierauf basierenden multimedialen Gestaltungsprinzipien haben sich über die Jahre kontinuierlich weiterentwickelt (Mayer, 2009; Mayer, 2020). So wurden zu Beginn vor allem kognitive Faktoren multimedialen Lernens betrachtet, wodurch verschiedene Erweiterungen der CTML entwickelt wurden, welche die affektiven Aspekte des multimedialen Lernens betonen; darunter die *Cognitive-affective Theory of Learning with Media* (CATML) nach Moreno und Mayer (2007). Sie besagt unter anderem, dass auch motivationale und metakognitive Faktoren sowie individuelle Unterschiede der Lernenden den Lernprozess beeinflussen (Moreno & Mayer, 2007, zitiert nach Schneider et al., 2022).

Basierend auf diesen Erkenntnissen formuliert Mayer (2020) 15 Prinzipien multimedialen Designs, die auch soziale, affektive, motivationale und metakognitive Aspekte berücksichtigen, mit dem Ziel, das multimediale Lernen zu fördern; darunter fünf Prinzipien zur Reduktion des *extraneous processing*, drei Prinzipien zum Umgang mit *essential processing* und sieben Prinzipien zur Förderung des *generative processing*, an denen sich die anschließende Entwicklung der Lernumgebung (siehe Kapitel 5) orientieren soll und welche nachfolgend näher beschrieben werden.

Zur Reduktion des *extraneous processing* soll(en)

- ausschließlich lernrelevante Materialien verwendet und eine ergänzende Darstellung überflüssiger Materialien in Form von Wörtern, Grafiken oder auch von Hintergrundmusik, die keine lernunterstützende Wirkung aufweisen, vermieden werden (*Coherence Principle*).
- relevante Materialien entweder auditiv oder visuell hervorgehoben werden, um die Lernenden in deren Identifikation zu unterstützen (*Signaling Principle*).
- auf eine ergänzende visuelle Darstellung der auditiven Beschreibung einer Grafik verzichtet werden (*Redundancy Principle*).
- zusammengehörige Texte und Grafiken nah beieinander anstatt voneinander entfernt präsentiert werden (*Spatial Contiguity Principle*).
- zusammengehörige Texte und Grafiken zur selben Zeit anstatt sukzessive präsentiert werden (*Temporal Contiguity Principl.*

Zum Umgang mit *essential processing* sollen

- insbesondere komplexe Inhalte auf mehrere Abschnitte verteilt werden, die dann im Sinne eines selbstgesteuerten Lernens eigenständig von den Lernenden ausgewählt und bearbeitet werden können (*Segmenting Principle*).
- die Lernenden bereits die Bedeutung der Begriffe, die innerhalb der zu vermittelnden Konzepte verwendet werden, kennen, sobald sie die digitale Lernumgebung nutzen (*Pre-training Principle*).
- zu Grafiken gehörige Texte eher auditiv anstatt visuell präsentiert werden (*Modality Principle*).

Zur Förderung des *generative processing* soll(en)

- sowohl Grafiken als auch Texte für die Wissensvermittlung verwendet werden (*Multimedia Principle*).
- Texte dialogisch und persönlich aus einer Erste-Person- oder Zweite-Person-Perspektive anstatt formal formuliert werden (*Personalization Principle*).
- auditive Texte von einer freundlichen menschlichen Stimme anstatt von einer Computerstimme gesprochen werden (*Voice Principle*).
- auf eine visuelle, statische Einbindung sprechender Personen verzichtet werden (*Image Principle*).
- visuell dargestellte Sprecher:innen eine ausgeprägte Gestik sowie Mimik nutzen (*Embodiment Principle*).
- immersive Technologien, zum Beispiel Virtual Reality, sehr bedacht verwendet werden, da sie sich nicht zwangsweise positiv auf den Lernprozess auswirken (*Immersion Principle*).
- die Lernenden zu einer aktiven Beschäftigung mit dem Lerngegenstand motiviert werden; zum Beispiel in Form von Zusammenfassungen oder Zuordnungen spezifischer Inhalte (*Generative Activity Principle*) (Mayer, 2020).

5 Praktische Umsetzung

Im Folgenden sollen sowohl die informatische (siehe Kapitel 2, Kapitel 3) als auch die (medien-)didaktische Perspektive (siehe Kapitel 4) auf eine Lernumgebung für ein systematisches Debugging miteinander vereint werden und hierauf basierend die Entwicklung des Prototyps erfolgen. Dies umfasst neben einer Anforderungsdefinition (siehe Kapitel 5.1), der Gestaltung von Entwurfsskizzen und Festlegung von Systemabläufen (siehe Kapitel 5.2) hinsichtlich individualisierter Profile (siehe Kapitel 5.2.1), einer Übersicht und Aufgabenauswahl (siehe Kapitel 5.2.2), der Aufgabenbearbeitung (siehe Kapitel 5.2.3), des Logbuchs (siehe Kapitel 5.2.4), der Einführung und Lösungen (siehe Kapitel 5.2.5) sowie des Inhaltsumfangs (siehe Kapitel 5.2.6) auch Überlegungen im Hinblick auf die Ästhetik der Lernumgebung (siehe Kapitel 5.3) sowie eine Betrachtung der Umsetzung aus technischer Sicht (siehe Kapitel 5.4).

5.1 Anforderungen

Eine Vielzahl populärer Vorgehensmodelle in der Softwareentwicklung beinhalten zu Beginn eine Phase, in der die Anforderungen an das zu entwickelnde System formuliert werden. Innerhalb des Wasserfallmodells erfolgt dies im Rahmen der Analyse, innerhalb des V-Modells entlang der Anforderungsdefinition und im Falle der Verwendung von Scrum werden die Anforderungen im *Product Backlog* festgehalten (Brandt-Pook & Kollmeier, 2020).

Es wird zwischen verschiedenen Arten von Anforderungen entschieden. *Funktionale* Anforderungen beschreiben, „[…] was das System leisten soll“ (Hicking & Völkel, 2022, S. 252). Hierzu zählen unter anderem die Benutzerschnittstelle und die Anwendungsfälle. *Nicht-funktionale* Anforderungen hingegen beschreiben, „[…] wie das System die Leistung erbringen soll“ (Hicking & Völkel, 2022, S. 252). Hierunter werden unter anderem die Performance und Sicherheit des Systems gefasst. Des Weiteren beeinflussen Rahmenbedingungen, wie zum Beispiel die Kosten oder Infrastruktur, den Softwareentwicklungsprozess (Hicking & Völkel, 2022).

Insbesondere im Hinblick auf interaktive Systeme ist hierbei die Berücksichtigung der Bedürfnisse der Nutzer:innen von entscheidender Bedeutung. Broy und Kuhrmann (2021) formulieren dies wie folgt:

> „Die nutzerzentrierte Ergänzung der Funktionalität des Systems als Beitrag zur Aufgabe und die Vorgehensweise des Nutzers zur Durchführung seiner Aufgabe müssen optimal miteinander integriert sein (Broy & Kuhrmann, 2021, S. 201).“

Daher sollen im Folgenden die konkreten Bedürfnisse der späteren Nutzer:innen betrachtet werden, sodass sowohl die Erkenntnisse der innerhalb des Kapitel 3.2 beschriebenen Betrachtung existierender Systeme, die einer „Ist“-Analyse nach Broy und Kuhrmann (2021) entspricht, als auch die begründete Schwerpunktsetzung der zu entwickelnden Lernumgebung sowie die zugehörigen, in Kapitel 3.1 beschriebenen Ansätze zur Vermittlung von Debugging(-fähigkeiten) im weiteren Verlauf auf sie abgestimmt werden können.

Eine Betrachtung der Bedürfnisse der späteren Nutzer:innen erfordert zunächst eine Definition der Zielgruppe der zu entwickelnden Lernumgebung (Semler, 2016). Die primäre Zielgruppe der Lernumgebung bilden Studierende informatischer Studiengänge am Institut für Informatik und Computational Science der UP, welche die Veranstaltungen *Grundlagen der Programmierung* und/oder *Praxis der Programmierung* belegen (siehe Kapitel 1.1).[16] Hierzu zählen Studierende der Bachelor-Studiengänge Informatik/Computational Science, Wirtschaftsinformatik sowie Lehramt Informatik. Die Veranstaltungen stehen zudem auch weiteren Studiengängen als Wahlpflichtveranstaltungen zur Verfügung. Darüber hinaus ist auch eine Verwendung der Lernumgebung seitens Schüler:innen, Auszubildenden, Studierenden anderer Studiengänge und Forschungseinrichtungen sowie Privatpersonen für eine Förderung ihrer Debuggingfähigkeiten denkbar, insofern sowohl die Programmiersprache als auch die Themen und der Schwierigkeitsgrad der Aufgaben in etwa ihren Vorerfahrungen entsprechen.

Im Anschluss an diese Zielgruppendefinition empfiehlt sich eine detaillierte Sicht auf die Nutzer:innen, um unter anderem Wünsche und Erwartungen gegenüber der Lernumgebung zu sammeln sowie ihre konkreten Lebenslagen zu verstehen. Hierbei ist es auch möglich, Personen der definierten Zielgruppe beispielsweise in Form von Fokusgruppen-Untersuchungen bereits zu Beginn des Entwicklungsprozesses explizit mit einzubeziehen (Broy & Kuhrmann, 2021; Jacobsen, 2014). Insbesondere aufgrund der innerhalb der Kapitel 2.4 und Kapitel 3.1 identifizierten Schwierigkeiten von (häufig eher unerfahrenen) Programmierer:innen während des Debuggings sowie der persönlichen Erfahrungen des Autors entlang einer früheren Zugehörigkeit zu der Zielgruppe wird auf eine weitergehende Einbindung späterer Nutzer:innen in diesem Entwicklungsschritt verzichtet.

Es existieren verschiedene Praktiken der Anforderungserhebung. Hierunter zählen unter anderem *Personas*, die basierend auf einer abstrahierten Darstellung realer Personen relevante Anwendungsfälle identifizieren sollen (Broy & Kuhrmann, 2021). Nach Hicking und Völkel (2022) ist hierbei darauf zu achten, dass sowohl persönliche Informationen und die Tätigkeit als auch Probleme und Herausforderungen sowie Wünsche und Ziele mitaufgenommen werden, sodass eine möglichst realistische Beschreibung der/des fiktiven Nutzer:in erreicht wird. Hierbei ist anzumerken, dass *Personas*, ähnlich wie *User Stories*, lediglich die Bedürfnisse einzelner Personen(-gruppen) widerspiegeln (Broy & Kuhrmann, 2021); die nachfolgende Persona sowie die anschließende Formulierung einiger Anforderungen haben daher nicht den Anspruch, vollständig zu sein. Darüber hinaus stehen hierbei vor allem funktionale Anforderungen im Vordergrund; nicht-funktionale Anforderungen werden eher nicht erfasst (Broy & Kuhrmann, 2021). Da es sich in dieser Arbeit um die Entwicklung und Evaluation einer *prototypischen* Lernumgebung handelt, in der die Funktionalität des Systems und dessen

[16] Da die Veranstaltung *Praxis der Programmierung* an die in der Veranstaltung *Grundlagen der Programmierung* vermittelten Inhalte und Kompetenzen anknüpft, wird sich im weiteren Verlauf dieser Arbeit an den Rahmenbedingungen der Veranstaltung *Grundlagen der Programmierung* orientiert, um zu gewährleisten, dass die Lernumgebung von Studierenden beider Veranstaltungen genutzt werden kann.

Nutzungsschnittstellen im Vordergrund stehen, ist eine schwerpunktmäßige Betrachtung funktionaler Anforderungen jedoch vertretbar.

Im Folgenden wird eine *Persona* für eine fiktive Person aus der primären Zielgruppe formuliert.

Emma ist 20 Jahre alt und seit kurzem Erstsemesterstudentin im Bachelor-Studiengang Informatik/Computational Science an der UP. Ihre Entscheidung für ein Informatikstudium wurde maßgeblich durch eine spontane Teilnahme an der Sommeruni der Freien Universität Berlin im Rahmen der Initiative *Komm, mach MINT* und ihrer hierbei empfundenen, bislang ungeahnten Begeisterung für informatische Sachverhalte beeinflusst. Nennenswerte Vorerfahrungen in der Programmierung brachte sie zu Beginn des Studiums nicht mit. Aktuell belegt sie die Veranstaltung *Grundlagen der Programmierung*. Während ihr die Programmierung und insbesondere die hierbei auftretenden Erfolgsmomente, wenn ein Programm fehlerfrei funktioniert, zwar einerseits Freude bereiten, frustrieren sie andererseits die teils langen Suchen nach Fehlern in ihren selbstgeschriebenen Programmen. Insbesondere da sie das Gefühl hat, dass ihre Kommiliton:innen aufgrund ihrer in der Schule oder Freizeit gesammelten, häufig umfangreicheren Programmiererfahrungen deutlich schneller und reibungsloser fehlerfreie Programme verfassen. Emma ist zum einen besorgt, dass sie die Modulabschlussprüfung nicht bestehen wird, zum anderen wird sie bereits bei dem Gedanken an künftige Veranstaltungen unruhig, innerhalb derer sie wieder programmieren muss. Daher ist sie intrinsisch motiviert, ihre Debuggingfähigkeiten langfristig zu verbessern. Während der COVID-19-Pandemie wurde ihr bewusst, dass sie gerne selbständig lernt. Außerdem hat sie zu dieser Zeit mit zahlreichen Online-Lernplattformen gearbeitet und fühlt sich mittlerweile im Umgang mit ihnen sehr sicher. Entsprechend hat sie bereits nach geeigneten Online-Ressourcen gesucht, die ihr in ihrer jetzigen Situation weiterhelfen könnten. Jedoch stieß sie dabei auf das Problem, dass häufig Konzepte der Programmierung verwendet werden, die sie bislang noch nicht kennengelernt hat. Der Umstand, dass die Materialien häufig in Englisch verfasst sind und sich nicht ausschließlich auf die Programmiersprache Python beschränken, erschwerte das Verständnis zusätzlich.

Basierend auf der vorangegangenen Persona und den Erkenntnissen aus den früheren Kapiteln sollen im Folgenden einige wesentliche Anforderungen an die zu entwickelnde Lernumgebung formuliert werden.

(1) Die Lernumgebung muss sich in ihrer Ausgestaltung an den Rahmenbedingungen der Lehrveranstaltungen der Zielgruppe orientieren (siehe Persona). Hierzu zählen insbesondere die inhaltlichen Themen sowie die Wahl der Programmiersprache.
(2) Die Lernumgebung muss ein systematisches Vorgehen entlang der *Isolation*-Strategie und einer Berücksichtigung des *iterative knowledge* und *experience knowledge* fördern (siehe Kapitel 3.2). Hierbei soll auf entsprechende Implikationen der Literatur zurückgegriffen werden (siehe Kapitel 3.1).

(3) Die Vermittlungsstrategie der Lernumgebung muss außerdem (medien)didaktisch fundiert sein, indem sie sowohl in den CAA (siehe Kapitel 4.2.2) als auch in die multimedialen Gestaltungskriterien eingebettet wird (siehe Kapitel 4.3.3).
(4) Die Lernumgebung soll Personen unterschiedlicher Vorerfahrungen zur Verfügung stehen und somit auch parallel zum Erlernen des Programmierens genutzt werden können (siehe Persona).

Im Folgenden werden diese Anforderungen in konkrete Systemelemente übersetzt (Brandt-Pook & Kollmeier, 2020). Das Hauptaugenmerkt liegt dabei auf den für die Nutzer:innen sichtbaren bzw. wahrnehmbaren Komponenten und Abläufen; technische Aspekte werden in Kapitel 5.4 dargelegt.

5.2 Entwurf und Abläufe

5.2.1 Individualisierte Profile

Die Orientierung am CAA und die damit einhergehende, insbesondere in den Phasen *Coaching* und *Scaffolding* beschriebene, individuelle sowie bedarfsgerechte Unterstützung der Lernenden (siehe Kapitel 4.2.2) setzt eine längerfristige Speicherung des jeweiligen Fortschritts der Nutzer:innen voraus. Der Bedarf einer Unterstützung verschiedener Nutzerprofile zeigt sich zudem auch in der Berücksichtigung des *experience knowledge* entlang einer Dokumentation des individuellen Debuggingprozesses (siehe Kapitel 3.1). Die innerhalb der Phase *Modelling* des CAA beschriebene Entwicklung eines konzeptuellen Modells der Vorgehensweise zum Problemlösen der Lernenden (siehe Kapitel 4.2.2) wird in der Lernumgebung mittels einer Einführung[17] umgesetzt. Sie beinhaltet neben einer allgemeinen Unterweisung in die Handhabung der Lernumgebung auch die Bearbeitung einer exemplarischen Aufgabe, entlang derer die Lernenden die unterschiedlichen Schritte während der Aufgabenbearbeitung kennenlernen. Die Weiterleitung zur Einführung erfolgt automatisch nach der Registrierung und anschließend ersten Anmeldung der/des Nutzer:in.

5.2.2 Übersicht und Aufgabenauswahl

Sobald sich die/der Nutzer:in angemeldet hat, gelangt sie/er zur Übersichtsseite. Von hier aus kann sie/er zum einen die Aufgaben auswählen, um anschließend zur Aufgabenbearbeitung wechseln. Die Aufgaben werden jeweils einer der Kategorien *Variablen & Datentypen*, *Verzweigungen*, *Schleifen* oder *Datenstrukturen* zugeordnet.[18] Diese Unterteilung sowie eine transparente Darstellung der empfohlenen Voraussetzungen für die Bearbeitung von Aufgaben aus der jeweiligen Kategorie adressiert die heterogenen Vorerfahrungen der Nutzer:innen und soll

[17] Anmerkung: Ein näherer Blick in die Einführung erfolgt in Kapitel 5.2.5.
[18] Anmerkung: Die gewählten Kategorien repräsentieren eine Auswahl wesentlicher Programmierkonzepte, die in der Veranstaltung *Grundlagen der Programmierung* der UP thematisiert werden. Der Zugriff auf die Materialien erfolgte am 29.09.2023 über den Moodle-Kurs der Veranstaltung: https://moodle2.uni-potsdam.de/course/view.php?id=38871.

insbesondere auch eine parallele Nutzung der Lernumgebung bereits während des Erlernens des Programmierens ermöglichen; zum Beispiel begleitend zum Besuch der Veranstaltung *Grundlagen der Programmierung* an der UP (siehe Abbildung 6). Die Aufgaben innerhalb einer Kategorie werden zudem nach ansteigender Schwierigkeit sortiert (Collins et al., 1987).

Zum anderen können die Nutzer:innen von der Übersichtsseite aus ihr persönliches Logbuch öffnen und ihren Fortschritt in Form der Anzahl ihrer bereits bearbeiteten Aufgaben sowie kennengelernten Fehlerarten betrachten. Die Darstellung individuellen Fortschritts wirkt sich nach Lischka (2019) positiv auf die Emotionen der Nutzer:innen aus, indem ihnen ein Bewältigungsgefühl vermittelt wird (siehe Abbildung 6).

Der innerhalb der Phase *Reflection* des CAA beschriebene Vergleich des Problemlöseprozesses der Lernenden mit dem einer/eines Expert:in und die hierdurch geförderte Reflexion des eigenen Vorgehens (siehe Kapitel 4.2.2) wird in der Lernumgebung entlang einer Bereitstellung von Lösungen[19] zu den jeweiligen Aufgaben realisiert, auf welche die Lernenden ebenfalls von der Übersichtsseite aus zugreifen können (siehe Abbildung 6).

Außerdem besteht für die Nutzer:innen die Möglichkeit, sich weitergehende Informationen und Übungen zum Debugging anzusehen. Hierfür wird auf die Webseite des Forschungsprojektes *Debugging im Unterricht* an der Technischen Universität München verwiesen.[20] Zuletzt gelangen die Lernenden von der Übersichtsseite aus je nach Bedarf auch erneut zur Einführung (siehe Kapitel 5.2.1) (siehe Abbildung 6).

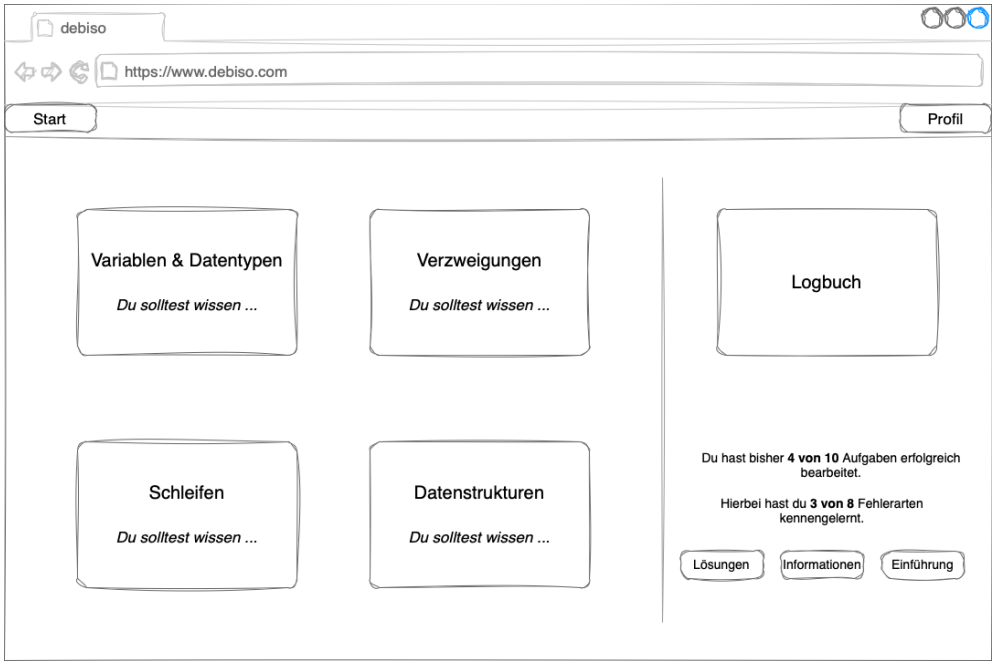


Abbildung 6: Entwurf – Übersicht und Aufgabenauswahl (eigene Darstellung)

[19] Anmerkung: Ein näherer Blick in die Lösungen erfolgt in Kapitel 5.2.5.
[20] Link: https://computingeducation.de/proj-debugging/ (letzter Zugriff am 05.02.2024)

Im Hinblick auf die Gestaltung sowie Anordnung der Auswahlmöglichkeiten finden sowohl das *Spatial Contiguity Principle* als auch das *Temporal Contiguity Principle* nach Mayer (2020) (siehe Kapitel 4.3.3) Anwendung (siehe Abbildung 6).[21]

5.2.3 Aufgabenbearbeitung

Informationsbereich

Der Informationsbereich beinhaltet keine interaktiven Elemente, sondern versorgt die Nutzer:innen während der Aufgabenbearbeitung mit den notwendigen Informationen. Hierzu gehören zum einen die Aufgabeninformationen in Form des Titels der Aufgabe, einer Beschreibung des Quellcodes, der beabsichtigten Ausgabe des Quellcodes (SOLL) sowie der tatsächlichen Ausgabe des Quellcodes (IST). Hierbei wird sich an den Ausführungen in Kapitel 2.4.2 zur Identifikation fehlerhaften Systemverhaltens orientiert. Hierin wird die Bestimmung von Diskrepanzen zwischen dem intendierten (hier: SOLL) und tatsächlichen (hier: IST) Programmverhalten als Ausgangspunkt für die weiteren Schritte des Debuggingprozesses beschrieben (siehe Kapitel 2.4.2). Während das Programm hierzu in der Regel häufig getestet wird, werden diese Informationen aufgrund des Debugging-Schwerpunktes der Lernumgebung, ähnlich wie in Michaeli und Romeike (2019a), zu Beginn bereitgestellt.

Im Anschluss an die Interaktion einer/eines Nutzer:in mit der Lernumgebung sollte stets eine Rückmeldung erfolgen (Semler, 2016). Daher wird den Nutzer:innen mit Hilfe eines Dialogs ein knappes und prägnantes Feedback bezüglich der vorangegangenen Auswahl oder Eingabe zur Verfügung gestellt. Außerdem führt der Dialog die Lernenden entlang von Fragen und Anweisungen durch den weiteren Bearbeitungsprozess. Aktuelles Feedback wird hervorgehoben, während früheres Feedback aber weiterhin verfügbar bleibt, sodass die Nutzer:innen den bisherigen Bearbeitungsverlauf weiterhin nachvollziehen können. Gemäß des *Personalization Principle* erfolgt außerdem sämtliche Kommunikation mit den Lernenden persönlich in der Zweiten Person (siehe Kapitel 4.3.3).

Der Informationsbereich visualisiert den Nutzer:innen außerdem, in welchem Schritt des Debuggingprozesses sie sich derzeit befinden und soll auf diesem Wege den iterativen Charakter des Prozesses veranschaulichen (siehe Kapitel 2.4.3). Zudem sehen die Lernenden den aktuell systemseitig bereitgestellten Unterstützungsgrad in Form einer Reduktion der Aufgabenbearbeitung, welche im späteren Verlauf dieses Kapitels näher erläutert wird. Der Unterstützungsgrad wird in Orientierung an den individuellen Fortschritt der/des Nutzer:in berechnet.[22] Des Weiteren werden die während des Bearbeitungsverlaufs identifizierten Variablenabhängigkeiten dargestellt.

[21] Anmerkung: Eine Berücksichtigung dieser Gestaltungsprinzipien nach Mayer (2020) erfolgt auch in anderen Teilen der Lernumgebung. Zwecks Übersichtlichkeit wird die erneute Berücksichtigung dieser Gestaltungsprinzipien nicht erneut beschrieben. Dies gilt auch für die übrigen Gestaltungsprinzipien.
[22] Anmerkung: Die Reduktion der Aufgabenbearbeitung wird im weiteren Verlauf dieses Kapitels näher betrachtet.

Bearbeitungsbereich

Der Bearbeitungsbereich besteht entgegen dem Informationsbereich vor allem aus interaktiven Elementen. Je nachdem, an welchem Punkt sich die/der Nutzer:in innerhalb des Debugging-prozesses befindet, variiert der Inhalt des Bearbeitungsbereichs; daher erfolgt dessen weitere Beschreibung differenziert nach den unterschiedlichen Bearbeitungsschritten.

Die nachfolgend beschriebenen Bearbeitungsschritte orientieren sich an der Struktur der *Isolation*-Strategie nach Zeller (2009) (siehe Abbildung 7).

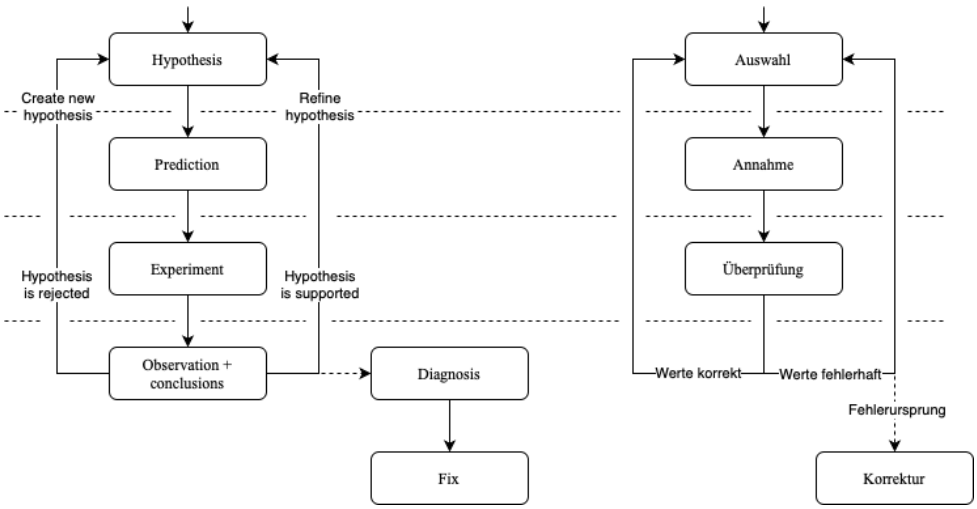


Abbildung 7: Übersetzung der *Isolation*-Strategie nach Zeller (2009) (eigene Darstellung)

Bearbeitungsbereich: Auswahl

Die von Zeller (2009) im ersten Schritt beschriebene Beobachtung eines Fehlers (siehe Kapitel 2.4.3) erfolgt entlang der Aufgabeninformationen und bildet den Ausgangspunkt des weiteren Bearbeitungsprozesses (siehe Kapitel 5.2.3, Informationsbereich). Die innerhalb des zweiten Schrittes dargelegte Formulierung einer Hypothese über die Fehlerursache, welche als besonders herausfordernd gilt (siehe Kapitel 3.1), wird im Rahmen der Lernumgebung unter *Auswahl* gefasst, systemseitig angeleitet und zwecks automatisierter Überprüfbarkeit vereinheitlicht. Hierbei betrachten die Lernenden die Programmausführung rückwärtsgerichtet, verfolgen die Werteänderungen der Variable, dessen tatsächlicher Wert (IST) von dem beabsichtigten Wert (SOLL) nach der Ausführung des Programms abweicht, schrittweise nach und wählen eine Zeile bzw. einen Abschnitt, innerhalb dessen die Variable selbst oder eine Variable, zu der entweder eine direkte oder transitive Abhängigkeit vorliegt, aus (siehe Abbildung 7).

Hierzu wird den Nutzer:innen zum einen der fehlerhafte Quellcode gezeigt, den es zu korrigieren gilt. Eine Syntaxhervorhebung des Quellcodes, wie sie auch in sämtlichen gängigen IDEs Verwendung findet, soll dabei die von Collins et al. (1987) beschriebene Relevanz eines Realitätsbezug der Lernumgebung adressieren (siehe Kapitel 4.2.2) (siehe Abbildung 8). Im Anschluss an die Zeilen- bzw. Abschnittswahl erfolgt die Auswahl der Variablenabhängigkeiten.

Die von Katz und Anderson (1987) sowie Decasse und Emde (1988) beschriebene Verknüpfung des vorliegenden Programmverhaltens mit Beobachtungen früherer Fehler zu Beginn der Fehlerdiagnose (siehe Kapitel 2.4.3) wird von der Lernumgebung explizit eingefordert, indem die Lernenden, insofern sie die jeweilige Fehlerart bereits kennen, ihr Logbuch öffnen und sie auswählen sollen.[23] Unabhängig davon besteht für die Nutzer:innen während der gesamten Aufgabenbearbeitung die Möglichkeit, auf ihr Logbuch und die hierin gespeicherten Informationen bereits abgeschlossener Aufgaben zuzugreifen (siehe Abbildung 8).

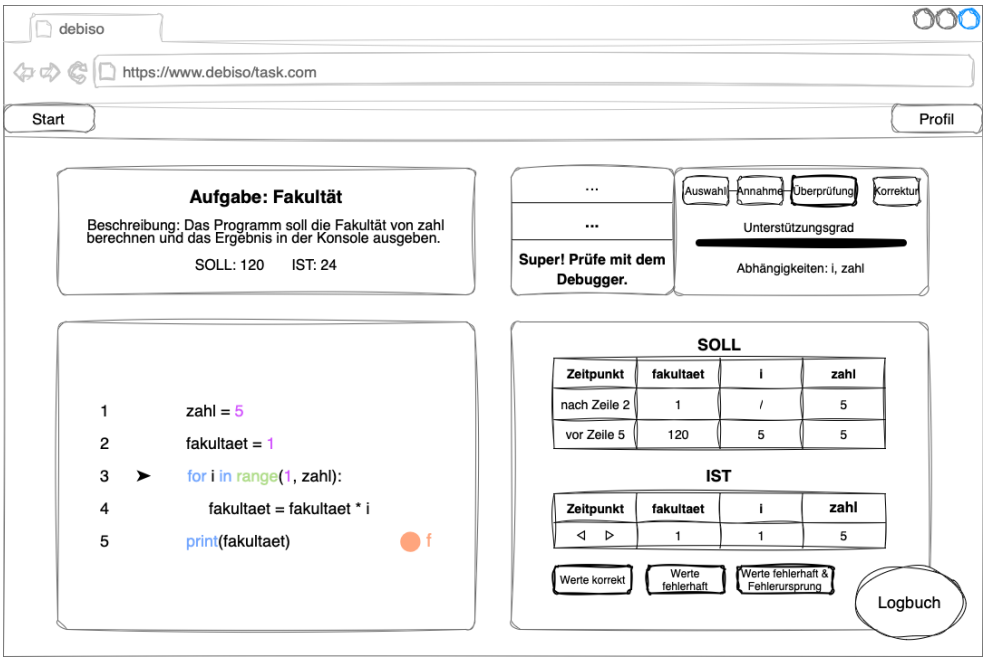


Abbildung 8: Entwurf – Aufgabenbearbeitung, Überprüfung (eigene Darstellung)

Bearbeitungsbereich: Annahme

Die von Zeller (2009) im dritten Schritt der *Isolation*-Strategie aufgeführte Entwicklung von Annahmen, basierend auf der vorangegangenen Hypothesenformulierung (siehe Kapitel 2.4.3), wird in der Lernumgebung als *Annahme* bezeichnet. Sie umfasst die Darlegung des Programmverständnisses in Form einer Formulierung der Werte, welche die Variablen vor und nach Ausführung dieser Zeile bzw. dieses Abschnitts haben sollen (SOLL) (siehe Abbildung 7).

Hierbei tragen die Lernenden die Werte in eine Tabelle ein, dessen Spalten der im Quellcode enthaltenen Variablen entsprechen (siehe Abbildung 8). Orientierend an der Phase *Coaching* des CAA (siehe Kapitel 4.2.2) und der hierin geforderten Betreuung der Lernenden während des Problemlösens wird die Tabelle bereits teilweise vorausgefüllt, indem die Variablenwerte systemseitig eingetragen werden, die sich innerhalb der derzeit betrachteten Zeile bzw. des derzeit betrachteten Abschnitts nicht verändern. Insofern die anschließend manuell eingetragenen Werte fehlerhaft sind, wird innerhalb des Dialogs die Anzahl der fehlerhaften Zellen

[23] Anmerkung: Der Aufbau sowie der Umfang des Logbuchs werden in Kapitel 5.2.4 näher betrachtet.

angegeben. Zudem besteht dann die Möglichkeit, die Tabelle systemseitig vollständig ausfüllen zu lassen, sodass die Aufgabe trotzdem weiterbearbeitet werden kann.

Neben einer Berücksichtigung des *Generative Activity Principle* zeigt sich insbesondere hier die Anwendung des *Segmenting Principle*, indem der Bearbeitungsbereich lediglich die für den aktuellen Bearbeitungsschritt relevanten Elemente umfasst; die übrigen Inhalte werden bewusst ausgeblendet (siehe Kapitel 4.3.3).

Bearbeitungsbereich: Überprüfung

Innerhalb des vierten Schrittes wird nach Zeller (2009) die eingangs formulierte Hypothese getestet und anschließend basierend auf den Beobachtungen bewertet (siehe Kapitel 2.4.3). Im Rahmen der Lernumgebung erfolgt dies in der *Überprüfung* entlang eines Abgleichs der SOLL-Werte mit den tatsächlichen Werten (IST) vor und nach der jeweiligen Zeile bzw. des jeweiligen Abschnitts und einer entsprechenden Bewertung der Zeile bzw. des Abschnitts als *Werte korrekt*, *Werte fehlerhaft* oder *Werte fehlerhaft und Fehlerursprung*. Je nach Auswahl wird dann im nachfolgenden Schritt iterativ und orientierend an den identifizierten Variablenabhängigkeiten die nächste Zeile bzw. der nächste Abschnitt ausgewählt oder zur Korrektur übergeleitet (siehe Abbildung 7).

Hierzu simuliert die Lernumgebung zur weiteren Förderung ihres Realitätsbezugs (siehe Kapitel 4.2.2) orientierend an gängigen IDEs die Setzung eines Haltepunktes vor der ausgewählten Zeile bzw. vor dem ausgewählten Abschnitt und stellt den Lernenden eine vereinfachte Variante eines Debuggers bereit, mit Hilfe dessen sie die tatsächlichen Variablenwerte in nachfolgenden Programmzuständen inspizieren und mit ihren zuvor eingetragenen SOLL-Werten vergleichen können. Dabei wird die aktuell betrachtete Zeile des Quellcodes visuell markiert und die Lernumgebung ermöglicht ihnen, ähnlich wie das in Kapitel 3.2 vorgestellte Tool *ViLLE* (Laakso et al., 2008), zu sowohl nachfolgenden als auch vorangegangenen Zeitpunkten der Programmausführung zu wechseln. Im Falle der Betrachtung einer Schleife wird ihnen außerdem mitgeteilt, in welcher Iteration der Schleife sie sich zurzeit befinden. Damit soll die Handhabung des Debuggers vereinfacht und den Lernenden die Möglichkeit gegeben werden, sich möglichst uneingeschränkt dem Vergleich von SOLL und IST zu widmen.

Im Anschluss erfolgt die Bewertung der Zeile bzw. des Abschnitts. Sowohl der vorangegangene Wertevergleich als auch die explizite Zeilen- bzw. Abschnittsbewertung sollen die Lernenden in der Verwerfung von Überlegungen sowie Aufstellung neuer Ideen über den Fehlerursprung fördern (siehe Kapitel 3.1). Die Bewertungen der bereits bearbeiteten Zeilen bzw. Abschnitte, deren Werte entweder korrekt oder fehlerhaft sind, während der Fehlerursprung jedoch woanders liegt, werden den Nutzer:innen außerdem während des weiteren Bearbeitungsverlaufs angezeigt. Auf diesem Wege visualisiert die Lernumgebung die von Yoon und Garcia (1998) beschriebene, im Rahmen der *Isolation*-Strategie stattfindende, fortlaufende Eingrenzung der Region des Fehlerursprungs. Zudem soll sie das *iterative knowledge* adressieren, indem sie

ähnlich wie das in Kapitel 3.2 beschriebene Tool *PLATO FAULT* (Johnson et al., 1982) den jeweils noch verbleibenden Problemraum abbildet.

Bearbeitungsbereich: Korrektur

Sobald der Fehlerursprung gefunden wurde und es in Anlehnung an den fünften Schritt der *Isolation*-Strategie nach Zeller (2009) keiner weiteren Prozessiteration bedarf, erfolgt die Behebung des Fehlers (siehe Abbildung 7).

Hierfür korrigieren die Nutzer:innen die entsprechende Zeile des Quellcodes. Anschließend wird das Programm erneut ausgeführt und die Eingabe zunächst basierend auf der Ausgabe des Programms validiert. Insofern sich die Ausgabe noch immer von der beabsichtigten Ausgabe unterscheidet oder die Programmausführung aufgrund eines Fehlers fehlschlägt, erhält die/der Nutzer:in eine Rückmeldung über die Ausgabe bzw. die Fehlermeldung. Nachfolgend ist es ihr/ihm möglich, ihre/seine Änderungen erneut und beliebig oft zu überarbeiten. In Anlehnung an Gugerty und Olson (1986) sowie McCauley et al. (2008) wird ihnen dabei stets der ursprüngliche Inhalt der zu korrigierenden Zeile des Quellcodes angezeigt, um das Widerrufen inkorrekter Änderungen zu unterstützen und zu vermeiden, dass sich diese negativ auf spätere Lösungsansätze auswirken (siehe Kapitel 3.1). Falls innerhalb des aktualisierten Quellcodes eine Endlosschleife erkannt wird, wird die Ausführung abgebrochen. Darüber hinaus werden die Änderungen auf einige potenziell schadhaften Inhalte geprüft.[24]

Sobald die tatsächliche Ausgabe mit der beabsichtigten Ausgabe übereinstimmt, gilt die Aufgabe als abgeschlossen. Die innerhalb der Phase *Articulation* des CAA beschriebene Artikulation des Vorgehens während des Problemlösens (siehe Kapitel 4.2.2) wird im Rahmen der Lernumgebung während des Aufgabenabschlusses berücksichtigt, indem die Lernenden eine Übersicht über die Aufgabenbearbeitung erhalten, die neben den Aufgabeninformationen auch den fehlerhaften sowie den korrekten Quellcode umfasst und ihnen ermöglicht, ihre Erfahrungen während der Bearbeitung der Aufgabe auszuformulieren und in ihrem persönlichen Logbuch zu speichern.

Reduktion

Die Bereitstellung von Unterstützungsmechanismen während der Aufgabenbearbeitung orientiert sich neben der Phase *Coaching* vor allem an der Phase *Scaffolding* des CAA, in welcher unter anderem eine Unterstützung der Lernenden entlang einer Reduktion des Problemlöseprozesses denkbar ist (siehe Kapitel 4.2.2) (siehe Abbildung 9).

[24] Anmerkung: Diese Maßnahmen genügen noch nicht, um sicherzustellen, dass über den aktualisierten Quellcode keinerlei schadhafter Inhalt eingeführt werden kann. Aufgrund der im Rahmen dieser Arbeit stattfindenden, lediglich kontrollierten Evaluation der Lernumgebung im Offlinebetrieb ist dies jedoch vertretbar.

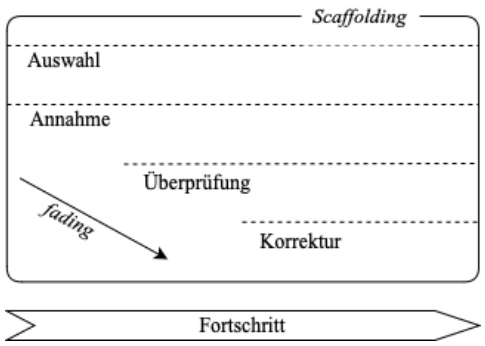


Abbildung 9: Übertrag der Phase *Scaffolding* nach Collins et al. (1987) (eigene Darstellung)

Daher erfolgt während der Bearbeitung der ersten Aufgabe eine Reduktion der Aufgabenbearbeitung um die Schritte *Überprüfung* und *Korrektur*, um die Nutzer:innen sukzessiv und orientierend am individuellen Fortschritt an den vollständigen Umfang eines Bearbeitungsablaufs innerhalb der Lernumgebung heranzuführen. Die Abschnittswahl, die Wahl der Variablenabhängigkeiten (*Auswahl*) sowie die Eintragung der SOLL-Werte (*Annahme*) werden dann in der zweiten Aufgabe um den Schritt des Wertevergleichs von SOLL und IST (*Überprüfung*) und schließlich in der darauffolgenden Aufgabe um die Behebung des Fehlers (*Korrektur*) ergänzt, bis die Lernenden schließlich den gesamten Debuggingprozess durchlaufen (*fading*) (siehe Kapitel 4.2.2) (siehe Abbildung 9).

5.2.4 Logbuch

Die Berücksichtigung des *experience knowledge* erfolgt entlang der von Siegmund et al. (2014) sowie Yoon und Garcia (1998) vorgeschlagenen Dokumentation der Debuggingprozesse in Form eines persönlichen Logbuchs, das für die Lernenden als ständiger Begleiter fungieren soll (siehe Kapitel 3.1). Neben einer Historie der bereits kennengelernten Fehler umfasst dieses Erläuterungen zu den verschiedenen Fehlerarten, die jeweilige Aufgabenbeschreibung und sowohl den fehlerhaften Quellcode und die zugehörige Ausgabe als auch den korrigierten Quellcode und die zugehörige Ausgabe. Darüber hinaus können die Nutzer:innen auf ihre jeweils im Anschluss an die Korrektur formulierten Erfahrungen während der Aufgabenbearbeitung (siehe Kapitel 5.2.3, Bearbeitungsbereich: Korrektur) zugreifen und sie überarbeiten (siehe Abbildung 10).

Im Hinblick auf die Phase *Exploration* des CAA und der hierin beschriebenen, schrittweise zunehmenden Selbständigkeit während des Problemlösens mit dem Ziel, die Lernenden hin zu einem autonomen Handeln zu führen (siehe Kapitel 4.2.2), ermöglicht die Lernumgebung auch den Export des persönlichen Logbuchs im *xlsx*- oder *csv*-Dateiformat, um eine Fortführung der Dokumentation von Debuggingerfahrungen auch über die Nutzung der Lernumgebung hinaus zu erleichtern (siehe Abbildung 10).

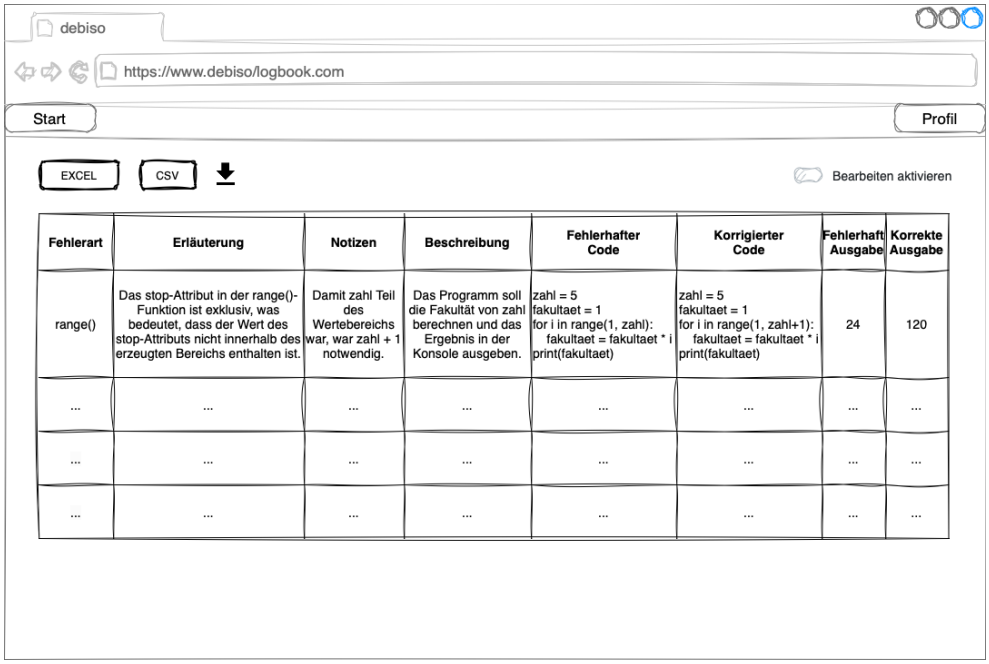


Abbildung 10: Entwurf – Logbuch (eigene Darstellung)

5.2.5 Einführung und Lösungen

Sowohl die im Rahmen der Phase *Modelling* des CAA stattfindende Einführung als auch die Umsetzung der Phase *Articulation* entlang einer Bereitstellung von Lösungen (siehe Kapitel 4.2.2) werden unter Berücksichtigung der innerhalb des *Modality Principle* beschriebenen vorzugsweise auditiven statt visuellen Texteinbindung (siehe Kapitel 4.3.3) mittels Screencasts realisiert.

Die Einführung legt zunächst die Übersichtsseite sowie das Logbuch in ihren Grundzügen dar und erläutert anschließend entlang einer exemplarischen Aufgabe die Bestandteile sowie den Ablauf einer Aufgabenbearbeitung. Neben der Phase *Modelling* des CAA (siehe Kapitel 4.2.2) realisiert die Einführung außerdem einen videobasierten On-Boarding-Prozess, welchem häufig eine wesentliche Bedeutung zugeschrieben wird, um die Nutzer:innen insbesondere bei komplexeren Anwendungen während der erstmaligen Nutzung zu unterstützen (Semler, 2016). Die Lösungen hingegen zeigen jeweils einen möglichen Lösungsweg und widmen sich dabei im Gegensatz zur Einführung eher den konkreten Überlegungen während des Debuggings des spezifischen Quellcodes anstatt der allgemeinen Handhabung der Lernumgebung. Im Hinblick auf das *Voice Principle* nach Mayer (2020) werden die begleitenden Erläuterungen vom Autor selbst eingesprochen (siehe Kapitel 4.3.3).

Die Verwaltung der Screencasts erfolgt über YouTube. Neben dessen einfacher Handhabung ermöglicht dies unter anderem eine einfache Integration und Einbettung der Videos in die Lernumgebung.[25]

5.2.6 Inhaltsumfang

Die Ausgabenausgestaltung bedarf zunächst einmal einer Festlegung auf die in der Lernumgebung zu unterstützende Programmiersprache. Im Hinblick auf die in Kapitel 5.1 definierte Zielgruppe wird Python gewählt.[26] Zudem gilt Python derzeit nach dem PYPL-Index als die sowohl weltweit als auch deutschlandweit populärste Programmiersprache.[27] Damit steigt die Anzahl potenzieller Nutzer:innen. Anschließend können die logischen Fehler bestimmt werden, entlang derer ein systematisches Vorgehen während des Debuggings vermittelt werden soll. Hierbei wird sich zum einen an Literatur über gängige Fehler und Fehlvorstellungen während des Programmierens sowie an den persönlichen Erfahrungen des Autors orientiert. Folgende Fehlerarten werden für eine Integration in die Lernumgebung ausgewählt:

- Fehlerhafte Bedingungsformulierung
- Fehlerhafte Variableninitialisierung (Ebrahimi, 1994)
- Fehlerhafte Verwendung des Modulo-Operators
- Fehlerhafte Verwendung von *range()*
- Inkorrekte Reihenfolge der Operatoren
- Unbeabsichtigter Wert der Laufvariablen
- Verwechslung von Variablen (McCall & Kölling, 2019)
- Überschreibung von Variablen (Žanko et al., 2021)

Anschließend werden die konkreten Quellcodes entwickelt, innerhalb derer die Lernenden während der Aufgabenbearbeitung die zuvor ausgewählten Fehlerarten suchen und beheben sollen. Im Hinblick auf den begrenzten Umfang dieser Arbeit soll die Lernumgebung zunächst zehn Aufgaben umfassen; dies ermöglicht alle ausgewählte Fehlerarten in mindestens eine Aufgabe zu integrieren. Ergänzend hierzu gelten folgende Einschränkungen:

- Die Quellcodes beinhalten immer genau einen Fehler. Dessen Korrektur erfolgt außerdem innerhalb einer festgelegten Zeile. Eine über eine Zeile hinausgehende Korrektur sowie das Hinzufügen oder Löschen von Zeilen sind weder notwendig noch möglich. Dies soll unter anderem das Hartkodieren von Lösungen erschweren.

[25] Link zur Einführung: https://youtu.be/oCTUn_fraUM; Link zu den Lösungen: https://youtu.be/oy8h1x2VIv4. Aufgrund des Evaluationsdesigns, innerhalb dessen die Lösungsvideos nicht zur Anwendung kommen, steht derzeit ein exemplarisches Video zur Verfügung.
[26] Anmerkung: Die gewählte Programmiersprache stimmt mit der in der Veranstaltung *Grundlagen der Programmierung* der UP verwendeten Programmiersprache überein.
[27] Link: https://pypl.github.io/PYPL.html, https://pypl.github.io/PYPL.html?country=DE, letzter Zugriff am 15.02.2024.

- Die letzte Zeile der Quellcodes ist stets eine *print*-Anweisung, welche die Variable mit dem fehlerhaften Wert in der Konsole ausgibt. In Kombination mit der Beschreibung sowie der beabsichtigten Ausgabe wird hierdurch ein aufgabenübergreifend vergleichbarer Ausgangspunkt für die Aufgabenbearbeitung geschaffen.
- Die Quellcodes beschränken sich in ihrem Umfang auf die bereits zuvor genannten Kategorien *Variablen & Datentypen*, *Verzweigungen*, *Schleifen* oder *Datenstrukturen* (siehe Kapitel 5.2.2); hierbei wird auf verschachtelte Schleifen verzichtet.

Der nachfolgende Quellcode demonstriert die von der Lernumgebung unterstützte Struktur der Quellcodes:

```
woerter = ["ist", "Debugging", "toll"]
woerter = sorted(woerter)
verbundeneWoerter = ""
for wort in woerter:
     verbundeneWoerter = wort + " "
verbundeneWoerter = verbundeneWoerter.rstrip()
print(verbundeneWoerter)
```

Code 1: Struktur der Quellcodes

Der Fehler liegt in diesem Beispiel in der fünften Zeile, die eigentlich *verbundeneWoerter += wort + " "* lauten sollte, und ist der Fehlerart *Inkorrekte Reihenfolge der Operatoren* zuzuordnen (siehe Code 1).

5.3 Ästhetik

Semler (2016) schreibt der funktionalen Schönheit einer App eine entscheidende Bedeutung für ein gelungenes Nutzererlebnis zu. Hierbei nimmt unter anderem die grafische Ausgestaltung einen wesentlichen Stellenwert ein (Semler, 2016). Während sich die bisherigen Überlegungen im Wesentlichen an den multimedialen Gestaltungsprinzipien nach Mayer (2020) orientieren, sollen daher im Folgenden auch einige weiterführende Aspekte einer adäquaten ästhetischen Ausgestaltung der Lernumgebung berücksichtigt werden.

Im Hinblick auf eine strukturierte Darstellung der Inhalte werden auf der Übersichtsseite die zu jeweils einer Aufgabenkategorie gehörigen Inhalte in Form des Titels, der Voraussetzungen, eines Bildes und der Verlinkungen zu den Aufgaben gruppiert. Mit Hilfe einer vertikalen Trennlinie werden sie zudem von dem persönlichen Bereich, bestehend sowohl aus dem Logbuch als auch dem individuellen Fortschritt der/des Nutzer:in, separiert. Ausreichend weißer Freiraum zwischen den Elementen soll neben einer beruhigenden Wirkung auch die optische Gliederung unterstützen (Jacobsen, 2014) (siehe Abbildung 11).

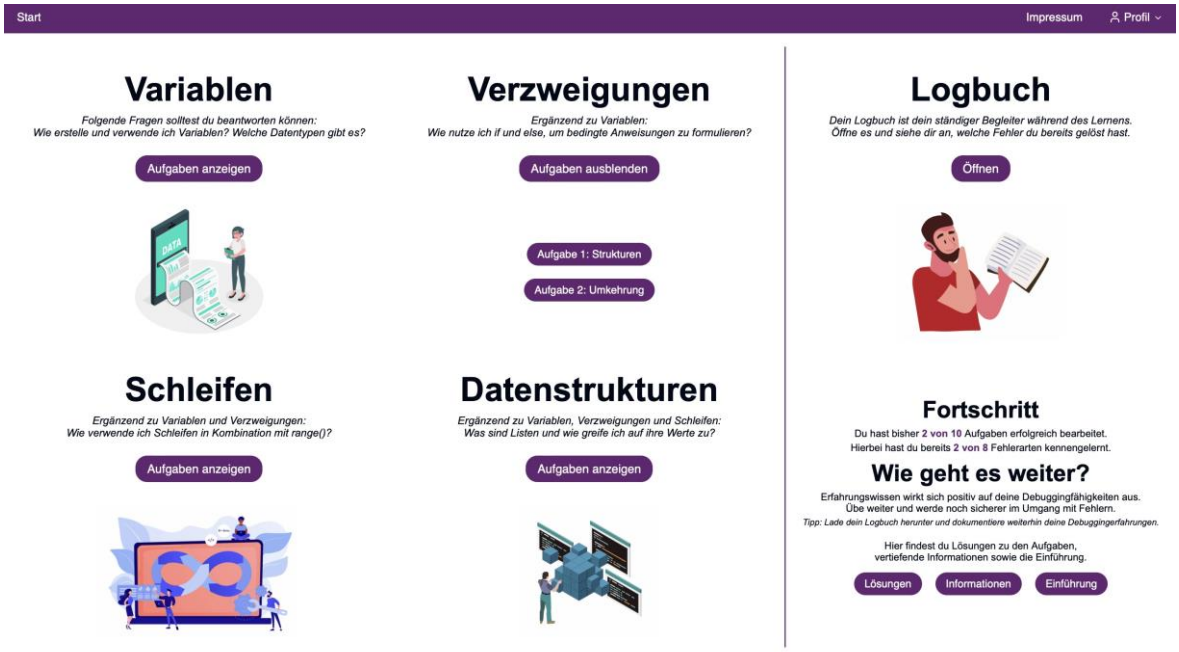


Abbildung 11: Ästhetik – Übersicht und Aufgabenauswahl (eigene Darstellung)

Die Lernumgebung soll eine möglichst flache Navigationsstruktur umfassen (Semler, 2016) (siehe Abbildung 12). Daher ist es den Nutzer:innen möglich, alle Aufgaben bereits von der Übersichtsseite aus zu erreichen, indem sie sich die Aufgaben der verschiedenen Kategorien unabhängig voneinander anzeigen lassen können. Da in diesem Falle das jeweilige Bild der Aufgabenkategorie ausgeblendet wird, bleibt weiterhin genügend Freiraum erhalten (Jacobsen, 2014) (siehe Abbildung 11).

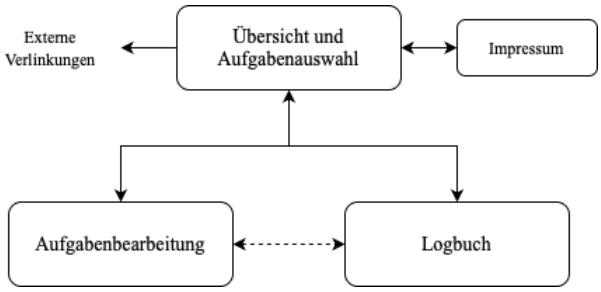


Abbildung 12: Navigationsstruktur (eigene Darstellung)

Die Startseite prägt maßgeblich den ersten Eindruck der Lernumgebung (Jacobsen, 2014). Daher sollen themenorientierte Bilder derselben Größe sowie eines vergleichbaren Animationsstils den Anteil reinen Textes reduzieren und eine entspannte, zwanglose Wirkung auf die Nutzer:innen begünstigen (Jacobsen, 2014; Semler, 2016) (siehe Abbildung 11).

Jacobsen (2014) verdeutlicht die Bedeutung einer konsistenten Gestaltung. Im Rahmen der Lernumgebung wird dies unter anderem entlang eines durchgängigen Stils sowie einer fixierten Positionierung und möglichst einheitlichen Benennung der Buttons sichtbar. Zeitgleich erfolgen an verschiedenen Stellen Schwerpunktsetzungen. So werden beispielsweise der aktuelle Dialogeintrag fett hervorgehoben und der interaktive Bearbeitungsbereich umrahmt. Zudem

wird sich auch hinsichtlich der Typografie der Lernumgebung auf einige wenige *core web fonts* beschränkt (Jacobsen, 2014) (siehe Abbildung 12).

Da Farben stets kommunizieren und die Emotionen sowie das Handeln der Nutzer:innen beeinflussen (Jacobsen, 2014; Semler, 2016), gilt es auch hierbei, die wesentlichen Aspekte im Rahmen der Entwicklung der Lernumgebung zu berücksichtigen. Da eine ruhige Farbgebung sowie starke Kontrastierung eine seriöse und professionelle Wirkung erzielen (Jacobsen, 2014), wird in der Lernumgebung neben den Nichtfarben Schwarz, Weiß und Grau primär ein einziger, eher stärker gesättigter Violettton (#642472) für die farbliche Akzentuierung verwendet. Dessen Repräsentation von Extravaganz und des Unkonventionellen bei gleichzeitig eher seltener Verwendung in anderen Anwendungen (Semler, 2016) soll die Individualität der Lernumgebung stärken. Außerdem wird Blau (#0000FF) für die Hervorhebung der aktuell ausgewählten Zeile oder des derzeit ausgewählten Abschnitts des Quellcodes genutzt. Dies entspricht der von Semler (2016) beschriebenen häufigen Verwendung von Blau für Informationen innerhalb von User Interfaces (UI). Darüber hinaus werden die Signalwirkungen der Komplementärfarben Rot (#ff0000) und Grün (#008000; #006400) in Form von *falsch/fehlerhaft* und *richtig/korrekt* (Semler, 2016) sowie die Farbe Orange (#ffa500) verwendet, um unter anderem die Bewertung einer Zeile bzw. eines Abschnitts zu unterstützen oder die erfolgreiche Aufgabenbearbeitung zu visualisieren. Im Hinblick auf eine mögliche Farbfehlsichtigkeit unter den Nutzer:innen werden zudem relevante Informationen nicht ausschließlich mittels Farbe kommuniziert (Jacobsen, 2014; Semler, 2016). So werden beispielsweise die Bewertungen der bereits bearbeiteten Zeilen bzw. Abschnitte (siehe Kapitel 5.2.3, Bearbeitungsbereich: Überprüfung) entweder um ein *f* (falsch) oder *r* (richtig) ergänzt (siehe Abbildung 13).

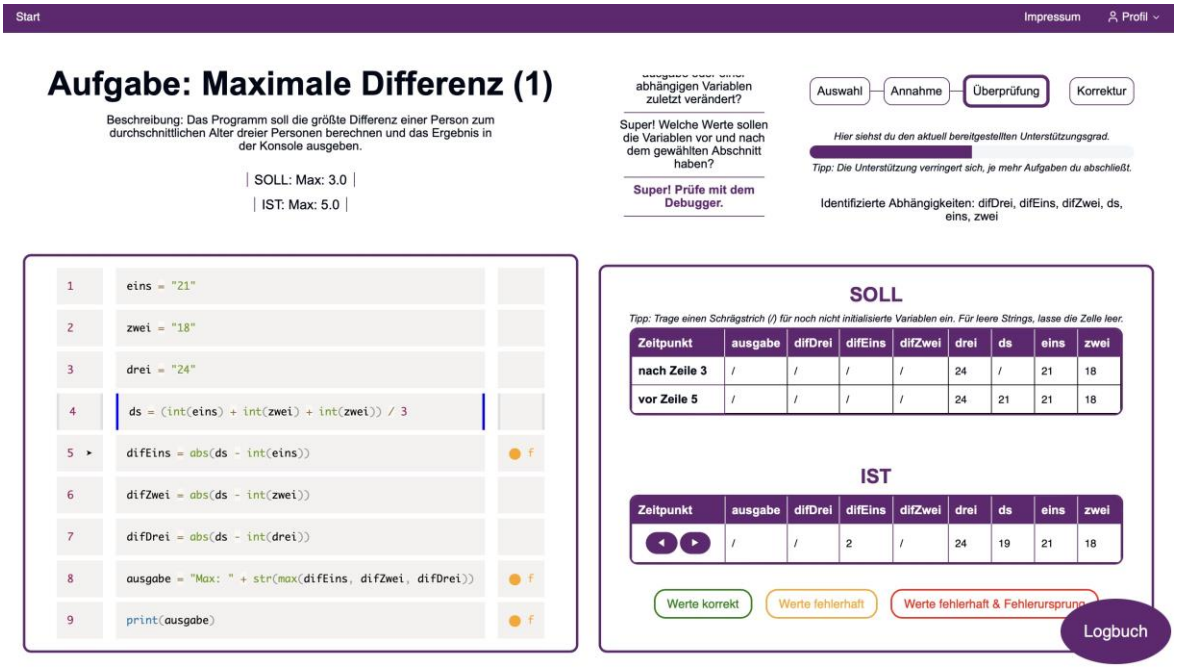


Abbildung 13: Ästhetik – Aufgabenbearbeitung (eigene Darstellung)

Die von Semler (2016) betonte Bedeutung von Feedback für sämtliche Interaktionen zwischen der/dem Nutzenden und der Anwendung wird neben dem in Kapitel 5.2.3 beschriebenen Dialog innerhalb des Informationsbereichs der Aufgabenbearbeitung unter anderem auch entlang verschiedener Zustände der Buttons (Normal, *Mouse-Over*, *Clicked*) und der Verwendung sogenannter *toasts* für abgeschlossene Downloads berücksichtigt (siehe Abbildung 14).

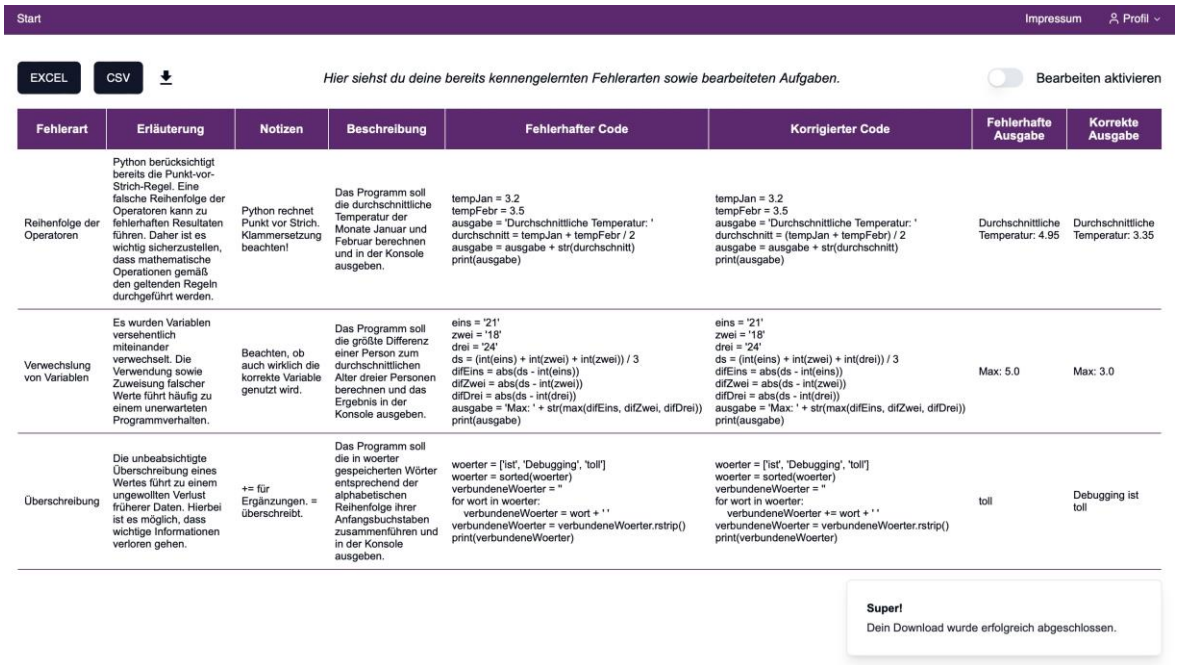


Abbildung 14: Ästhetik – Logbuch (eigene Darstellung)

Zudem werden Ladezeiten, die beispielsweise während eines Seitenwechsels und insbesondere zu Beginn der Aufgabenbearbeitung auftreten, der/dem Nutzenden mit Hilfe eines *spinners* visualisiert. Auf eine Darstellung des Ladefortschritts in Form eines Ladebalkens wird dabei verzichtet, da die Ladezeiten in der Regel von kurzer Dauer sind (Semler, 2016).

Aufgrund der teils hohen Informationsdichte der Lernumgebung, insbesondere während der Aufgabenbearbeitung, und einer gängigen Verwendung von Desktop-Computern, Laptops und Notebooks für das Programmieren in der primären Zielgruppe, werden andere mobile Endgeräte, wie zum Beispiel Smartphones, im Hinblick auf die Portabilität der Lernumgebung vorerst nicht weiter berücksichtigt. Im Rahmen der genannten Gerätegruppen verfolgt die Lernumgebung jedoch das Ziel eines responsiven Designs und soll somit auf verschiedenen Bildschirmgrößen sowohl nutzbar sein als auch möglichst ansprechend dargestellt werden (Jacobsen, 2014; Semler, 2016).

5.4 Technische Aspekte

Die Lernumgebung wurde als Webanwendung implementiert, da sie zum einen aus Sicht der Nutzer:innen aufgrund ihrer Ausführung innerhalb des Webbrowsers weitestgehend plattformunabhängig nutzbar sind und keine vorherige Installation erfordern. Potenzielle Nutzungshürden der Lernumgebung sollen somit verringert werden. Zum anderen ermöglichen sie aus

Entwicklerperspektive unter anderem eine schnelle und weitflächige Verteilung von Updates sowie eine Wiederverwendung der zahlreichen, bereits verfügbaren Ressourcen für die Webentwicklung (Garousi et al., 2013; Semler, 2016) (siehe Kapitel 4.1). Die Umsetzung als Webanwendung erfordert eine ständige Internetverbindung während der Nutzung der Lernumgebung. Im Hinblick auf die primäre Zielgruppe der Lernumgebung und das auf dem Campus der UP nahezu flächendeckend verfügbare WLAN[28], während gleichzeitig etwa 91,4 % der deutschen Haushalte über einen Internetzugang verfügen, wovon wiederum ein Großteil eine minimale Datenübertragungsrate von 16 Mbit/s besitzen (Destatis, 2022), ist diese Internetbindung jedoch tragbar.

Die Entwicklung erfolgte unter Verwendung der Programmiersprachen *JavaScript* und *TypeScript* sowie der JavaScript-Bibliothek *React.js*[29] für die Erstellung des *Frontends* von Webanwendungen im Rahmen des populären *React-Frameworks Next.js*[30]. Dieses erlaubt eine vergleichsweise schnelle Entwicklung von *Full Stack*-Webanwendungen und im Rahmen der Lernumgebung unter anderem eine einfache Verwaltung der Nutzer:innen- und Aufgabeninformationen entlang von *serverless functions*. Für die persistente Speicherung dieser Daten wurde auf das etablierte Datenbankmanagementsystem *PostgreSQL*[31] zurückgegriffen. Die Nutzung der Bibliothek *NextAuth.js*[32] ermöglichte außerdem eine einfache Realisierung des Registrierungs- und Anmeldeprozesses sowie des *session managements*. Die Ver- und Entschlüsselung der Passwörter wurde mit Hilfe der kryptologischen Hashfunktion *bcyrpt* umgesetzt.

Zur Reduktion des Entwicklungsaufwandes bei einem gleichzeitig hochwertigen Erscheinungsbild setzt sich das UI der Lernumgebung in Teilen aus einigen frei verfügbaren Elementen zusammen, die der UI-Komponentensammlung *shadcn/ui*[33] entstammen. Des Weiteren wurde die Icon-Bibliothek *Lucide React*[34] verwendet und die Syntaxhervorhebung des auf der Lernumgebung dargestellten Quellcodes erfolgte mit Hilfe der Bibliothek *Prism*[35]. Die Konvertierung der im Logbuch der/des jeweiligen Nutzer:in gespeicherten Bearbeitungshistorie in das *xlsx- und* csv-Dateiformat sowie der anschließende Download wurden mittels *SheetJS*[36] und *React CSV*[37] realisiert.

Für die Ausführung des Python-Quellcodes innerhalb der Webanwendung wurde der in dem Projekt *Iodide*[38] von Mozilla entwickelte Python-Interpreter *Pyodide*[39] genutzt, welcher eine

[28] Link: https://www.uni-potsdam.de/de/zim/angebote-loesungen/wlan-eduroam, letzter Zugriff am 12.02.2024.
[29] Link: https://react.dev, letzter Zugriff am 13.02.2024.
[30] Link: https://nextjs.org, letzter Zugriff am 13.02.2024.
[31] Link: https://www.postgresql.org, letzter Zugriff am 13.02.2024.
[32] https://next-auth.js.org, letzter Zugriff am 13.02.2024.
[33] Link: https://ui.shadcn.com, letzter Zugriff am 13.02.2024.
[34] Link: https://lucide.dev/guide/packages/lucide-react, letzter Zugriff am 13.02.2024.
[35] Link: https://prismjs.com/index.html, letzter Zugriff am 13.02.2024.
[36] Link: https://sheetjs.com, letzter Zugriff am 13.02.2024.
[37] Link: https://github.com/react-csv, letzter Zugriff am 13.02.2024.
[38] Link: https://github.com/iodide-project, letzter Zugriff am 13.02.2024.
[39] Link: https://pyodide.org/en/stable, letzter Zugriff am 13.02.2024.

Ausführung von Python-Quellcode bei gleichzeitigem Zugriff auf Web-APIs erlaubt. Die hierdurch entstehende geringere Performance der Ausführung stellt aufgrund der geringen Komplexität der in der Lernumgebung enthaltenen Quellcodes kein Problem dar.

Die Versionsverwaltung während der Entwicklung erfolgte mittels *GitLab*[40]. In Verbindung mit *Vercel*[41], welches aus demselben Hause wie *Next.js* stammt und daher auf *Next.js*-Anwendungen zugeschnitten ist, gestattet dies ein einfaches, kostenfreies und stets aktuelles *Deployment*[42] des Projektes. Die *domain* entspricht dem vorläufigen Titel der Lernumgebung: DebIso. Als *Portmanteau* der Begriffe *Debugging* und *Isolation* spiegelt DebIso die Schwerpunkte der Lernumgebung wider, grenzt sich von bereits verfügbarer Software ab und soll aufgrund seiner überschaubaren Länge leicht einzuprägen sein (Semler, 2016). Die Lernumgebung wurde mittels *BrowserStack*[43] auf den drei in Deutschland derzeit meistgenutzten Desktop-Browsern Chrome, Edge und Firefox[44] (unter Windows) sowie auf Safari (unter macOS) getestet.

[40] Link zum *Repository*: https://gitlab.com/felixZie/masterarbeit.
[41] Link: https://vercel.com/home, letzter Zugriff am 13.02.2024.
[42] Link zum *Deployment*: https://debiso.vercel.app.
[43] Link: https://www.browserstack.com, letzter Zugriff am 13.02.2024.
[44] Link: https://gs.statcounter.com/browser-version-market-share/desktop/germany#monthly-202401-202401-bar, letzter Zugriff am 14.02.2024.

6 Evaluation

Im Anschluss an die vorangegangene praktische Umsetzung (siehe Kapitel 5) kann nun die prototypische Lernumgebung von Lernenden genutzt und hinsichtlich der Forschungsfrage dieser Arbeit (siehe Kapitel 1.2) evaluiert werden. Hierzu werden im Anschluss an eine Rekapitulation der Fragestellung (siehe Kapitel 6.1) die Datenerhebung (siehe Kapitel 6.2) entlang der Methodik (siehe Kapitel 6.2.1), der Teilnehmenden (siehe Kapitel 6.2.2) sowie des Ablaufs (siehe Kapitel 6.2.3) und sowohl die Methode (siehe Kapitel 6.3.1) als auch das Vorgehen (siehe Kapitel 6.3.2) während der Datenanalyse (siehe Kapitel 6.3) erläutert. Im weiteren Verlauf werden die Ergebnisse (siehe Kapitel 6.4) hinsichtlich des Verhaltens der Teilnehmenden (siehe Kapitel 6.4.1), ihrer Bearbeitungsschwierigkeiten und weiteren Eindrücke (siehe Kapitel 6.4.2) beschrieben und anschließend hinsichtlich ihrer Implikationen (siehe Kapitel 6.5.1) sowie Limitationen (siehe Kapitel 6.5.2) diskutiert (siehe Kapitel 6.5).

6.1 Fragestellung

Im Rahmen der Evaluation soll die in Kapitel 1.2 formulierte Fragestellung betrachtet werden.

Wie verhalten sich die Lernenden während des kurzzeitigen Gebrauchs einer Lernumgebung nach dem *Cognitive Apprenticeship*-Ansatz mit dem Ziel der expliziten Vermittlung eines systematischen Debuggingvorgehens und welche Eindrücke entstehen während der Bearbeitung?

Hieraus ergeben sich zwei Betrachtungsschwerpunkte: (1) das *Verhalten* der Lernenden während der Nutzung der Lernumgebung sowie (2) ihre artikulierten *Eindrücke* der Anwendung.

6.2 Datenerhebung

6.2.1 Methodik

Die Datenerhebung erfolgte nach der Methode *Lautes Denken*, in der die Proband:innen die Aufnahme und Verarbeitung von Informationen in Form von Eindrücken, Empfindungen und Absichten artikulieren und der/dem Forschenden erlauben, ihre mentalen Prozesse simultan zur entsprechenden Tätigkeit nachzuvollziehen (Buber, 2007; Frommann, 2005).

Nach Frommann (2005) und Konrad (2010) wird die Methode im Rahmen der Softwareentwicklung unter anderem zur Untersuchung der Gebrauchstauglichkeit von (Software-)Systemen verwendet. Hierzu nutzen die Proband:innen zumeist prototypische Versionen dieser Systeme. Damit eignete sich die Methode *Lautes Denken*, um einen Einblick sowohl in das Verhalten als auch in die entstehenden Eindrücke von Lernenden während der Nutzung der Lernumgebung zu erhalten (siehe Kapitel 6.1).

6.2.2 Teilnehmende

Die Akquisition der Teilnehmenden erfolgte zunächst entlang persönlicher Besuche mehrerer Rechnerübungen der Veranstaltung *Grundlagen der Programmierung* an der UP. Hierbei

wurden den Studierenden der Gegenstand der vorliegenden Arbeit sowie das Evaluationsvorhaben skizziert. So sollte eine adäquate Erprobung der Lernumgebung in der primären Zielgruppe sichergestellt werden. Dabei wurden keine monetären Anreize geboten.

Frommann (2005) empfiehlt eine Anzahl von etwa fünf Versuchspersonen für die Durchführung der Methode *Lautes Denken*. An der Evaluation der Lernumgebung nahmen insgesamt sechs Personen teil (n = 6). Hiervon belegten zu dieser Zeit vier Personen die Veranstaltung *Grundlagen der Programmierung* an der UP und waren somit der primären Zielgruppe zuzuordnen (~ 66 %). Alle Teilnehmenden sind Bachelorstudierende der *Informatik/Computational Science* oder der *Wirtschaftsinformatik*. Im Hinblick auf das Vorwissen in der Programmierung gaben zwei der sechs Personen an, entweder im selben oder vorherigen Semester mit dem Erlernen der Programmierung begonnen zu haben (~ 33 %). Die übrigen Personen haben entweder im universitären und/oder privaten Kontext bereits weiterführende Programmiererfahrungen sammeln können.

6.2.3 Ablauf[45]

Einstieg

Der Ablauf der Datenerhebung orientierte sich stark an dem von Frommann (2005) vorgeschlagenen Ablaufschema. So wurde zunächst der Platz für die Versuchsperson vorbereitet. Sobald sie eingetroffen war, erfolgten die Begrüßung und etwas Small Talk, um eine angenehme Atmosphäre zu schaffen. Dies beschreibt Frommann (2005) als den *Grundstein* für den beabsichtigten Erkenntnisgewinn. Hier schlossen dann eine knappe inhaltliche Einführung sowie Einordnung der Evaluation an, bevor die Rolle der Versuchsperson innerhalb dieses Prozesses dargelegt wurde (Frommann, 2005). Im Hinblick auf die Einhaltung des Datenschutzes der Teilnehmenden unterzeichnete die Versuchsperson außerdem eine Einverständniserklärung, in welcher sie unter anderem über die weitere Verarbeitung und mögliche Teilveröffentlichung ihrer Daten informiert wurde.[46]

Daraufhin erfolgten die Vorstellung sowie Demonstration der Methode, indem die Teilnehmenden beispielhafte Formulierungen, wie „jetzt überlege ich gerade wie“ oder „auf dem Bildschirm suche ich“ kennenlernten, die ihnen helfen sollten, zu verstehen, in welcher Weise sie ihre Gedanken artikulieren können (Frommann, 2005). Zudem schlagen Buber (2007) und Frommann (2005) eine Aufwärmaufgabe vor, entlang derer die Teilnehmenden an das laute Denken herangeführt werden und die im Anschluss gemeinsam reflektiert wird. Hierzu wurde in Anlehnung an Buber (2007) die Bearbeitung einer Divisionsaufgabe gewählt.

[45] Anmerkung: Das Evaluationsvorhaben wurde zuvor im Rahmen einer Pre-Studie mit einer Kommilitonin simuliert. Anschließend erfolgten vereinzelte Verbesserungen und Anpassungen.
[46] Link zu der Vorlage der Einverständniserklärung: https://zenodo.org/records/10790985.

Schließlich wurde sichergestellt, dass die Handhabung des Gerätes verständlich ist sowie später keine Irritationen oder Verzögerungen verursacht (Buber, 2007) und der Versuchsperson gemäß des CAA die Einführung in die Lernumgebung (siehe Kapitel 5.2.5) vorgespielt.

Beobachtung

Innerhalb der Beobachtungsphase bearbeiteten die Versuchspersonen die Aufgaben *Fakultät* und *Summierung*. Hierbei handelt es sich um die jeweils erste Aufgabe aus den Kategorien *Schleifen* und *Datenstrukturen*. Da die Bearbeitungsdauer der ersten Aufgabe bei Versuchsperson 5 deutlich länger als erwartet ausfiel, wurde dort auf die Bearbeitung der zweiten Aufgabe verzichtet, da die zur Verfügung stehende Zeit dafür nicht ausreichte. Die Integration der Phase *Scaffolding* nach dem CAA sieht normalerweise vor, dass die Bearbeitung der ersten beiden Aufgaben in reduzierter Weise erfolgt und alle Bearbeitungsschritte erst während der dritten Aufgabe absolviert werden (siehe Kapitel 5.2.3, Reduktion). Um die Teilnehmenden im Rahmen der Evaluation alle Bearbeitungsschritte zumindest einmal durchlaufen zu lassen, wurde die Reduktion der Aufgabenbearbeitung so angepasst, dass bereits während der Bearbeitung der ersten Aufgabe der Schritt *Überprüfung* gefordert wurde und dann in der Bearbeitung der zweiten Aufgabe auch die *Korrektur* hinzukam. Sämtliche Aussagen, die während der Beobachtungs- und Interviewphase geäußert wurden, wurden aufgezeichnet. Um die Aussagen später besser rekonstruieren zu können, wurden ebenfalls die Interaktionen der Teilnehmenden mit der Lernumgebung aufgezeichnet. Hierfür wurde die in *MacOS* integrierte Funktion der Bildschirmaufnahme verwendet.[47] Orientierend an Frommann (2005) wurden den Versuchspersonen außerdem der Beginn und das Ende der Beobachtungsphase mitgeteilt sowie, auch hinsichtlich der unterschiedlichen Vorerfahrungen, eine variable Beobachtungszeit von 15-25 Minuten angestrebt. Die von Frommann (2005) vorgeschlagene Unterteilung in Einheiten von unter zehn Minuten war aus organisatorischen Gründen nicht möglich. Die Evaluationen fanden teilweise zwischen dem Besuch zweier Veranstaltungen der Teilnehmenden statt. Daher sollte die Evaluationsdauer möglichst nicht mehr als 30 Minuten betragen.

Darüber hinaus sollten die Teilnehmenden durch positive Rückmeldungen in der Artikulation ihrer Gedanken bestätigt werden. Insofern sie über einige Zeit hinweg wenige oder keine Äußerungen tätigen, wurden die Versuchspersonen gelegentlich zum lauten Denken animiert (Frommann, 2005). Zudem wurde darauf geachtet, dass die Teilnehmenden möglichst eigenständig mit der Lernumgebung arbeiten (Frommann, 2005). Hilfestellungen wurden daher lediglich in Ausnahmen bereitgestellt, etwa wenn sich eine Versuchsperson in einer Sackgasse befand und explizit Hilfe einforderte. Dabei wurde sich jedoch auch an den in der Fragestellung genannten Beobachtungsschwerpunkten (siehe Kapitel 6.1) orientiert. So erfolgte beispielsweise im Falle von Schwierigkeiten, die auf ein fehlerhaftes Programmverständnis zurückzuführen waren, eine eher frühere Intervention als bei Bearbeitungshürden, die in der Lernumgebung selbst begründet lagen.

[47] Link zu den Bildschirmaufnahmen: https://zenodo.org/records/10790985.

Interview

Innerhalb des anschließenden Interviews wurden die Teilnehmenden zunächst zu ihren Vorerfahrungen in der Programmierung befragt. Aufgrund der begrenzten zur Verfügung stehenden Zeit folgten anschließend lediglich zwei Fragen, entlang derer die Versuchspersonen nach einem negativen und einem positiven Aspekt befragt wurden, die ihnen während der Aufgabenbearbeitung aufgefallen sind. Buber (2007) berichtet von einer möglichen Negativtendenz der im Rahmen der Methode *Lautes Denken* gewonnenen Berichte, da auftretende Schwierigkeiten das Verhalten der jeweiligen Versuchsperson eher stärker beeinflussen als wenn dessen Erwartungen erfüllt werden und die jeweilige Handlung wie geplant fortgesetzt werden kann. Neben einer Identifikation weiterer Schwierigkeiten, die während der Bearbeitung der Aufgaben eventuell noch nicht geäußert wurden, sollte die Frage nach einem positiven Aspekt daher insbesondere dieser möglichen Negativtendenz entgegenwirken, um auch positive Eindrücke der Lernumgebung zu erheben.

Zwecks Objektivität wurde die gesamte Datenerhebung, insbesondere der Einstieg und das Interview, unter allen Teilnehmenden möglichst identisch durchgeführt. Abweichungen waren vor allem im Hinblick auf die Wahl der Methodik zwar nicht immer zu vermeiden, sollten aber so weit wie möglich reduziert werden. Daher wurde sich in weiten Teilen an einem vorbereiteten Ablaufplan orientiert, der unter anderem die Erklärungen und Fragen an die Teilnehmenden umfasste.

6.3 Datenanalyse

6.3.1 Methodik

Die Analyse der erhobenen Daten erfolgte unter Anwendung der qualitativen Inhaltsanalyse nach Mayring. Diese ist eine häufig genutzte, qualitativ orientierte textanalytische Methode, die sich unter anderem auch für große Materialmengen eignet und aufgrund ihres streng regelgeleiteten Ablaufs eine starke intersubjektive Überprüfbarkeit ermöglicht (Mayring & Fenzl, 2019).

Mayring (2022) differenziert dabei zwischen den folgenden Analysetechniken:

- Die *Zusammenfassung* soll das Material zunächst reduzieren, wobei die wesentlichen Inhalte erhalten bleiben, und anschließend eine Abstraktion des Grundmaterials bilden.
- Die *Explikation* soll das Verständnis spezifischer Materialteile entlang einer Berücksichtigung ergänzenden Materials erweitern.
- Die *Strukturierung* soll das Material im Hinblick auf vorher definierte Kriterien erfassen (Mayring, 2022).

Die Wahl der Analysetechnik beeinflusst den Ablauf der qualitativen Inhaltsanalyse (Mayring, 2022). Im Hinblick auf die Fragestellung, ihren eher explorativen Charakter sowie die hiermit

verbundene Schwerpunktsetzung, erfolgte die Datenanalyse entlang einer zusammenfassenden Inhaltsanalyse in Form einer induktiven Kategorienbildung.

6.3.2 Vorgehen

Zunächst wurden die Audioaufnahmen, die während der Beobachtung entstanden sind, transkribiert (Mayring, 2022). Hierfür wurden die Aufnahmen zunächst mit Hilfe von *maxqda*[48]automatisch transkribiert. Die Wahl von *maxqda* lag vor allem in dessen hohen Datenschutzstandards begründet. Anschließend wurden die Transkriptionen orientierend an den Transkriptionsregeln von Dresing und Pehl (2018) manuell überarbeitet. Hierbei wurden die Besonderheiten des lauten Denkens berücksichtigt und beispielsweise Halbsätze, aufgrund des häufigen Auftretens, nicht gesondert gekennzeichnet.[49]

Außerdem wurden die *Analyseeinheiten* festgelegt. Hierunter zählen sowohl der minimale (*Kodiereinheit*) als auch der maximale Textbestandteil, der einer Kategorie zugeordnet werden kann (Kontexteinheit) sowie die Reihenfolge der Auswertung der Textteile (*Auswertungseinheit*) (Mayring, 2022). Als Kodiereinheit wurde ein einzelnes Wort gewählt.[50] Ein Sprecherbeitrag in Form eines Absatzes bildete die Kontexteinheit. Die Auswertungseinheit entsprach dem gesamten Material.

Im Rahmen der Analyse erfolgte dann im Anschluss an eine Paraphrasierung bedeutungstragender Textstellen die induktive Bildung eines Kategoriensystems entlang einer iterativen Anwendung der Z-Regeln. Dabei wurden die Paraphrasen unter Berücksichtigung einer zuvor definierten Abstraktionsebene generalisiert, reduziert und zusammengefasst (Mayring, 2022) (siehe Anhang A). Hierbei wurde unterstützend *QCAmap*[51] verwendet, welches unter anderem von Mayring für eine interaktive und schrittweise Durchführung der von ihm beschriebenen qualitativen Inhaltsanalyse entwickelt wurde.[52]

6.4 Ergebnisse

6.4.1 Verhalten

Das Verhalten der Teilnehmenden während der Aufgabenbearbeitung wurde in 19 Kategorien gruppiert, die innerhalb der übergeordneten Kategorien *Bearbeitungsbeginn (A)*, *Auswahl (B)*, *Annahme (C)*, *Überprüfung (D)*, *Korrektur (E)* zusammengefasst wurden, welche wiederum die Schritte einer Aufgabenbearbeitung in der Lernumgebung (siehe Kapitel 5.2.3, Bearbeitungsbereich) repräsentieren. Hinzu kommen die Kategorien *Logbuch (F)* und *Fehlerfindung*

[48] Link: https://www.maxqda.com/de, letzter Zugriff am 18.02.2024.
[49] Link zu den Transkriptionen: https://zenodo.org/records/10790985.
[50] Anmerkung: Im Hinblick auf die Untersuchung des Verhaltens der Teilnehmenden ergibt sich die Zuordnung mancher Textbestandteile, insbesondere einzelner Wörter, erst aus der ergänzenden Betrachtung der jeweiligen Bildschirmaufnahme.
[51] Link: https://www.qcamap.org/ui/de/home, letzter Zugriff am 18.02.2024.
[52] Link zu den Zuordnungen zu den Kategoriensystemen: https://zenodo.org/records/10790985.

(G). Im Folgenden soll das hieraus resultierende Kategoriensystem zunächst entlang prägnanter Definitionen sowie zugehörigen Beispielen beschrieben werden.

Die Kategorie *Bearbeitungsbeginn (A)* beschreibt das Verhalten der Teilnehmenden unmittelbar nach der Auswahl einer Aufgabe. Sie unterteilt sich in vier Subkategorien.

(1) *Orientierung an Dialogeintrag*: Die/Der Teilnehmende erkennt und nutzt den Dialog als Orientierungshilfe für die weitere Bearbeitung der Aufgabe. Beispiel: „Das heißt, jetzt finde ich auch das, wo erstmal gefragt wird, wo wurde der Wert zuletzt verändert (P6)."
(2) *Lesen der Aufgabenbeschreibung*: Die/Der Teilnehmende liest die zur Verfügung gestellten Aufgabeinformationen in Form der Beschreibung des Programms sowie dessen SOLL- und IST-Werten. Beispiel: „[...] ich lese es mir erstmal durch. Beschreibung, das Programm soll die Fakultät von zahl berechnen und das Ergebnis in der Konsole ausgeben. Soll, 120, Ist 24 [...] (P3)."
(3) *Lesen des Quellcodes*: Die/Der Teilnehmende betrachtet den Programmablauf schrittweise. Beispiel: „Okay, schau ich mir mal den Code an. Werte gleich 10, 15, 5. Die sollen alle aufsummiert werden. Genau. werteLaenge ist die Länge des Wertes, x gleich eins, ausgabe Summe, werteSumme gleich null (P3)."
(4) *Suche nach Hilfe*: Die/Der Teilnehmende versucht aufgrund eines Missverständnisses der Erläuterungen in der Einführung einen *Helfer* aufzurufen. Beispiel: „Dann werde ich mal jetzt die Hilfe suchen. [...] Oder zumindest, dass ich verstanden habe von der Einführung, dass ich jetzt einen Helfer so quasi aufrufen kann (P2)."

Die Kategorie *Auswahl (B)* beschreibt wiederum das Verhalten der Teilnehmenden während der Zeilen- bzw. Abschnittswahl sowie der anschließenden Identifikation von Abhängigkeiten. Sie gliedert sich ebenfalls in vier Subkategorien.

(1) *Systematische Abschnittswahl*: Die/Der Teilnehmende wählt einen Abschnitt systematisch, unter Berücksichtigung des aktuellen Dialogeintrags aus. Beispiel: „Wo wurde der Wert von fakultaet zuletzt verändert? Das würde, ich glaube, so hier sein. Dann klicke ich hier [...] (P1)."
(2) *Unsystematische Abschnittswahl*: Die/Der Teilnehmende wählt einen Abschnitt unsystematisch, entweder nach mehrmaligem Versuchen oder während eines mehrfachen, unklaren Klickens aus. Beispiel: „Das heißt, ich gehe jetzt gerade. Guck mal, was ich alles anklicken kann. Okay (P5)."
(3) *Systematische Wahl der Abhängigkeiten*: Die/Der Teilnehmende wählt die Abhängigkeiten systematisch, unter Berücksichtigung des aktuellen Dialogeintrags aus. Beispiel: „Dementsprechend ist das von i abhängig. I ist aber an der Stelle von zahl abhängig. Muss ich dann zahl auch auswählen, dass. Ich versuche es einfach mal ohne zahl. Super. Okay (P4)."
(4) *Unsystematische Wahl der Abhängigkeiten*: Die/Der Teilnehmende wählt die Abhängigkeiten unsystematisch, nach mehrmaligem Versuchen aus. Beispiel: „Hm. Okay. Ich meine, gut, dann habe ich jetzt durchgetestet. Super (P5)."

Das Verhalten der Teilnehmenden während der Eintragung von Werten in die SOLL-Tabelle wird in der Kategorie *Annahme (C)* beschrieben. Die Kategorie besteht aus drei Subkategorien.

(1) *Manuelle Eintragung der SOLL-Werte*: Die/Der Teilnehmende trägt die SOLL-Werte in die Tabelle ein. Beispiel: „Fakultät nach Zeile zwei. (...) Sollte den Wert eins haben. (...) Vor zwei fünf. Er sollte den Wert. (...) Fünf. (..) Fünf Fakultät. Das ist. (...) 20, 60, 120. (...) i sollte, eigentlich existiert nicht [...] (P1).“
(2) *Manuelle Eintragung der IST-Werte*: Die/Der Teilnehmende trägt entweder vollständig oder in Teilen die IST-Werte in die Tabelle ein. Beispiel: „Dann ist fakultaet 24 und i sollte vier sein. (...) Leider falsch. [...] Ach, soll (P4).“
(3) *Systemseitige Eintragung*: Die/Der Teilnehmende wählt entweder nachdem die eigenständige Eintragung fehlschlug oder ohne es selbst zu versuchen aus, dass die Werte systemseitig ausgefüllt werden sollen. Beispiel: „Ist leider falsch. Dann muss ich die Tabelle ausfüllen [...] (P2).“

Die Kategorie *Überprüfung (D)* beschreibt das Verhalten der Teilnehmenden während des Vergleichs der SOLL- und IST-Werte sowie der anschließenden Bewertung des Abschnitts. Sie umfasst sechs Subkategorien.

(1) *Dynamischer Wertevergleich*: Die/Der Teilnehmende vergleicht die SOLL- und IST-Werte dynamisch, unter Verwendung einer mehrmaligen Änderung der Zeitpunkte. Beispiel: „Es gibt jetzt hier das Fehler, weil es sollte eigentlich bis zu fünf gehen, aber es gibt keine fünfte Iteration, weil so in range geht nur von eins bis vier (P1).“
(2) *Statischer Wertevergleich*: Die/Der Teilnehmende vergleicht die SOLL- und IST-Werte statisch, ohne den Zeitpunkt zu verändern. Beispiel: „Wenn man Zeitpunkt, da kann man den Code nachprüfen. Ich sollte es mehrmals drücken. Das war ein bisschen unübersichtlich (P2).“
(3) *Systematische, vollständige Abschnittsbewertung*: Die/Der Teilnehmende bewertet den Abschnitt begründet und als Ganzes als fehlerfrei, fehlerhaft oder ursächlich fehlerhaft. Beispiel: „Also in Zeile neun am Anfang hast du erstmal noch diesen Wert als ausgabe und dann nach Zeile neun, dann die 20, die aus der while Schleife entstehen. Und dieser Wert ist fehlerhaft, aber nicht der Fehlerursprung [...] (P3).“
(4) *Systematische, zeilenweise Abschnittsbewertung*: Die/Der Teilnehmende bewertet den Abschnitt begründet, jedoch zeilenweise als fehlerfrei, fehlerhaft oder ursächlich fehlerhaft. Beispiel: „Und jetzt meinst du, ich soll das, diese Eingabe jetzt kann ich auf diese Zeile prüfen? Okay, fakultaet ist eins. Ist richtig. i ist eins. zahl ist immer noch. Fünf mal eins. Werte korrekt. [...] also weil für die erste Iteration ist es dann korrekt (P5).“
(5) *Unsystematische Abschnittsbewertung*: Die/Der Teilnehmende bewertet den Abschnitt nach mehrmaligem Versuchen von fehlerfrei, fehlerhaft und/oder ursächlich fehlerhaft. Beispiel: „Ich gebe einfach, die Werte korrekt ist hier falsch. Ich will mal auch Fehler eingeben. Zeigt mir immer noch Fehler (P2).“

(6) *Unbeabsichtigte Abschnittsbewertung*: Die/Der Teilnehmende bewertet den Abschnitt anders als beabsichtigt. Der Unterschied zwischen den Auswahlmöglichkeiten ist unklar. Beispiel: „Das heißt in dem Fall x auf null möchte ich einsetzen. (...) Und. (...) Genau wieder weiß ich. (...) Nicht wie (P2)."

Das Verhalten der Teilnehmenden während der und im Anschluss an die Fehlerbehebung wird innerhalb der Kategorie *Korrektur (E)* zusammengefasst. Die Kategorie unterteilt sich in zwei Subkategorien.

(1) *Fehlerkorrektur*: Die/Der Teilnehmende wählt je nach Unterstützungsgrad, dass der Fehler systemseitig korrigiert werden soll, oder korrigiert den Fehler selbständig. Beispiel: „Korrigiere die Zeile. Hm. (...) Deine Änderung. x gleich Null. Alles klar (P4)."
(2) *Aufgabenabschluss*: Die/Der Teilnehmende schließt die Aufgabenbearbeitung ab. Beispiel: „Und dann. Aufgabe aufschließen, deine Notizen (P1)."

Außerdem spiegelt die Kategorie *Logbuch (F)* die Öffnung des Logbuchs seitens der/des Teilnehmenden wider. Beispiel: „Deswegen, ich gucke einfach mal kurz ins Logbuch. Okay, es ist noch nichts drin. Richtig (P4)." Darüber hinaus ist auch die Identifikation des Fehlerursprungs als Kategorie *Fehlerfindung (G)* Teil des Kategoriensystems. Beispiel: „Ich denke, dass das Problem ist, dass x gleich eins gesetzt wurde (P3)."

Im Anschluss an die Entwicklung des Kategoriensystems wurden die Verhaltensabläufe aller Teilnehmenden basierend auf den jeweiligen Zuordnungen ihrer Aussagen modelliert, um ihr Verhalten als Ganzes sowie im Vergleich zu den anderen Teilnehmenden betrachten zu können. Die Modellierungsweise orientiert sich dabei in Teilen an den aus der Softwareentwicklung bekannten Sequenzdiagrammen. Die horizontale Struktur der Modelle resultiert aus den Schritten der Aufgabenbearbeitung in Form der übergeordneten Kategorien. Die Kategorien F und G werden dabei jeweils unter dem Bearbeitungsschritt aufgeführt, innerhalb dessen das jeweilige Verhalten beobachtet wurde.[53]

Einige Verhaltensphasen werden zudem farblich markiert. Eine orangene Markierung zeigt dabei an, dass dieses Verhalten im Anschluss an eine leichte Hilfestellung seitens des Autors, beispielsweise als Antwort auf eine Rückfrage der/des Teilnehmenden, erfolgte. Eine rote Markierung hingegen stellt eine Abweichung des Verhaltens der/des Teilnehmenden von dem vorgesehenen Verhalten während der Nutzung der Lernumgebung dar. Insofern der Fehler von der/dem Teilnehmenden selbständig erkannt und korrigiert wurde, wird dies mittels eines K hinter der Bezeichnung der jeweiligen Kategorie dargestellt (siehe Anhang B).

[53] Anmerkung: Da die in Kategorie G beschriebene Identifikation des Fehlerursprungs in der Regel nebenher geschah, erfolgt dessen Darstellung mit Hilfe eines F neben der jeweiligen Verhaltensphase, in der sie geäußert wurde. Die Identifikation des Fehlerursprungs wurde nicht immer von den Teilnehmenden geäußert; dennoch war es allen Teilnehmenden stets möglich, den Fehlerursprung zu lokalisieren.

6.4.2 Eindrücke

Die von den Teilnehmenden während der Bearbeitung artikulierten Eindrücke wurden in 18 Kategorien zusammengefasst, die innerhalb der übergeordneten Kategorien *Unverständliche Formulierungen (H)*, *Geringe Sichtbarkeit (I)*, *Geringe Informationsdichte (J)*, *Unklare Handhabung (K)* und *Unerwartetes Verhalten (L)* gruppiert wurden. Hinzu kommt die Kategorie *Restriktive Benutzerführung (M)*.

Die im Anschluss an die Aufgabenbearbeitung geäußerten positiven sowie negativen Eindrücke der Lernumgebung, welche noch nicht zuvor während der Bearbeitung genannt wurden, wurden unter Verwendung der Audiomitschnitte analysiert und in den übergeordneten Kategorien *Integration der Isolation-Strategie (N)*, *Gestaltung (O)* sowie *Informationsgehalt der Einführung (P)* gesammelt. Aufgrund des nicht-simultanen Erhebungszeitpunktes dieser Daten werden sie separiert aufgeführt.

Im Folgenden soll das hieraus resultierende Kategoriensystem zunächst entlang prägnanter Definitionen sowie zugehöriger Beispiele beschrieben werden.

Die Kategorie *Unverständliche Formulierungen (H)* umfasst Verständnisschwierigkeiten, die auf spezifische Formulierungen innerhalb der Lernumgebung zurückzuführen sind. Sie gliedert sich in drei Subkategorien.

(1) *Frage nach Abhängigkeiten*: Die Fragen nach den Variablenabhängigkeiten entweder in der Zeile der Werteänderung oder in der Bedingung werden von der/dem Teilnehmenden nicht oder anders verstanden als beabsichtigt. Beispiel: „Dann ist das Abhängigkeit von i und zahl, ja okay. Also in Abhängigkeit von i und zahl, wobei i in dieser Zeile ja eigentlich festgelegt wird, dementsprechend ist es ja eigentlich nicht abhängig, oder? Ich weiß nicht. Okay, auf jeden Fall. Hm (P4).“
(2) *Beschreibung der Zeitpunkte*: Die Formulierung der Zeitpunkte in Form von *vor Zeile X* und *nach Zeile X* wird nicht oder anders verstanden als beabsichtigt. Beispiel: „Nach Zeile acht und vor Zeile zehn? Ist das nicht dasselbe? (P4)“
(3) *Unterschied zwischen Bewertungsmöglichkeiten*: Die Formulierungen der Bewertungsmöglichkeiten in Form von *Werte korrekt*, *Werte fehlerhaft* und *Werte fehlerhaft & Fehlerursprung* werden nicht oder anders verstanden als beabsichtigt. Beispiel: „Das heißt in dem Fall x auf null möchte ich einsetzen. (...) Und. (...) Genau wieder weiß ich. (...) Nicht wie (P2).“

Die Kategorie *Geringe Sichtbarkeit (I)* repräsentiert Schwierigkeiten während der Bearbeitung, die in einer geringen Sichtbarkeit einzelner Teile der UI der Lernumgebung begründet liegen. Die Kategorie unterteilt sich in vier Subkategorien.

(1) *Dialog*: Der Dialog wird als zu unscheinbar beschrieben und/oder übersehen. Beispiel: „Was mir glaube ich persönlich bisschen helfen würde beschrieben und/oder übersehen. ist, wenn das hier noch mal in einem Kasten oder so farblich hervorgehoben wird (P4).“

(2) *Ähnlichkeit der Fragen nach Abhängigkeiten*: Die Formulierung der Fragen nach den Variablenabhängigkeiten (siehe G1) ähneln sich sehr. Das aktualisierte Feedback wird übersehen. Beispiel: „Weil ich gerade nicht so den Unterschied zur ersten Frage davor. [...] Hier wird mir nicht so richtig der Unterschied klar (P5).“
(3) *Eintragungshinweise für nicht-initialisierte Variablen*: Der Hinweis, dass für nicht-initialisierte Variablen ein Schrägstrich eingetragen werden soll, wird übersehen. Beispiel: „i sollte, eigentlich existiert nicht, aber dann würde ich so eins sagen oder null oder.“ (P1)
(4) *Visuelle Darstellung des aktuellen Überprüfungszeitpunktes*: Die visuelle Darstellung während des Überprüfungszeitpunktes in Form eines Pfeils wird übersehen. Beispiel: „Ach so, jetzt habe ich auch erst gesehen, dass sich da dieser Pfeil bewegt (P5).“

Unter der Kategorie *Geringe Informationsdichte (J)* werden Schwierigkeiten aufgrund fehlender oder nicht ausreichender systemseitiger Informationen während der Bearbeitung gefasst. Sie besteht aus drei Subkategorien.

(1) *Unzureichendes Feedback*: Das erhaltende Feedback wird als zu unspezifisch beschrieben. Beispiel: „Was mich auch auf Moodle ein bisschen nervt, wenn einem nicht genau gesagt wird, was falsch ist (P6).“
(2) *Fehlendes Feedback*: Es irritiert, dass kein Feedback für fehlerhafte Eingaben während der Auswahl von Abhängigkeiten angezeigt wird. Beispiel: „Okay, der Button bestätigt nicht, wenn ich drauf klicke (P4).“
(3) *Eintragungshinweise für verschiedene Datentypen*: Die Eintragung verschiedener Datentypen ist unklar, z. B. dass Strings ohne Anführungszeichen eingetragen werden sollen. Beispiel: „ein ganz bisschen verwirrt, wie genau ich den Typen angeben würde […] (P6).“

In der Kategorie *Unklare Handhabung (K)* werden Probleme resultierend aus einer uneindeutigen Verwendung der Lernumgebung zusammengefasst. Die Kategorie umfasst fünf Subkategorien.

(1) *Betrachtung aller Zeilen bis zum Fehlerursprung*: Es besteht die Auffassung, dass alle Zeilen bis zum Fehlerursprung betrachtet werden müssen. Beispiel: „Okay, also wir gehen jetzt wirklich komplett zurück (P6).“
(2) *Vorgriff auf Fehlerfindung während Eintragung von SOLL*: Die Eintragung der SOLL-Werte wird als Vorwegnahme der Fehlerfindung verstanden. Beispiel: „Ja, okay, das ist ja eigentlich schon Fehlerfindung dann, oder nicht? (P3)“
(3) *Verwechslung von SOLL und IST*: Während der geforderten Eintragung der Werte werden SOLL und IST verwechselt. Beispiel: „Und ich habe schon wieder verwirrt, dass es Soll und Ist ist (P6).“
(4) *Wechsel des aktuellen Überprüfungszeitpunktes*: Es ist unklar, dass bzw. wie der aktuelle Überprüfungszeitpunkt verändert werden kann. Beispiel: „Wenn man Zeitpunkt, da kann man den Code nachprüfen. Ich sollte es mehrmals drücken. Das war ein bisschen unübersichtlich (P2).“

(5) *Zeilen-/ statt abschnittsweiser Bewertung*: Es ist unklar, ob der Abschnitt zeilenweise oder als Ganzes bewertet werden soll. Beispiel: „[…] unsicher, ob ich in dem Falle auswählen soll, den Zeitpunkt und dann drauf drücken soll oder ob es halt generell über alle diese Zeilen geht (P6).“

Die Kategorie *Unerwartetes Verhalten (L)* fasst das Verhalten der Lernumgebung zusammen, welches von den Erwartungen der/des Teilnehmenden abweicht. Sie gliedert sich in drei Subkategorien.

(1) *Darstellung identischen Feedbacks*: Es wird erwartet, dass identisches Feedback erneut im Dialog angezeigt wird. Beispiel: „Aber irgendwie wundert es mich gerade, dass ich gar kein Feedback kriege, wenn ich was klicke (P5).“
(2) *Simulation eines Haltepunktes*: Das simulierte Verhalten eines Haltepunktes während der Überprüfung wird nicht als solches erkannt. Beispiel: „Ich bin kurz verwirrt, warum das weiter geht. Ich dachte, das bleibt einfach bei dieser Zeile hängen (P3).“
(3) *Systemseitige Korrektur*: Anstatt einer systemseitigen Korrektur wird eine manuelle Korrektur des Fehlerursprungs erwartet. Beispiel: „[…] ich hätte jetzt erwartet, dass ich den richtigen Code eingeben soll. Okay (P6).“

Zudem wird in der Kategorie *Restriktive Benutzerführung (M)* die Anmerkung, dass lediglich die Korrektur einer spezifischen Zeile unterstützt wird, obgleich auch andere, abweichende Korrekturmöglichkeiten existieren, aufgeführt. Beispiel: „[…] das ist irgendwie ein bisschen komisch, weil es gibt ja an der Stelle mehrere unterschiedliche Wege an so ein Debugging ran zu gehen (P4).“

Darüber hinaus wurde im Anschluss an die Aufgabenbearbeitung zum einen die Systematisierung des Debuggingprozesses nach der *Isolation*-Strategie als positiv bewertet; dargestellt in der Kategorie *Integration der Isolation-Strategie (N)*. Beispiel: „Es gab einen klaren, eindeutigen Weg, um eine Lösung zu finden. Das fand ich richtig gut (P1).“ Zum anderen wurde die *Gestaltung (O)* in Form der Ästhetik und/oder Struktur der Lernumgebung sowohl positiv (*O1*), als auch negativ (*O2*) bewertet. Beispiele: „Ich finde es grafisch sehr, sehr schön. […] und an sich auch gut strukturiert (P5)“ und „Ich finde das System ein ganz klein wenig clunky (P6).“ Außerdem wurde der *Informationsgehalt der Einführung (P)* kritisiert und die Einführung als überladen beschrieben; weswegen ein Erinnern an die gezeigten Abläufe schwerfiel. Beispiel: „Nach der Einführung war mir nicht so ganz klar, was hier jetzt von mir gewollt ist, also was die Intention ist (P5).“

6.5 Diskussion

6.5.1 Implikationen

Im Folgenden sollen die bisweilen rein deskriptiv dargestellten Ergebnisse hinsichtlich ihrer Implikationen diskutiert werden.

Im Hinblick auf den Bearbeitungsbeginn (A) innerhalb der ersten Aufgabe, die im Rahmen der Evaluation bearbeitet wurde, zeigten sich unter den Teilnehmenden variierende Herangehensweisen. So wurde der initiale Dialogeintrag nicht von allen Teilnehmenden zu Beginn gelesen (A1); dies führte im nächsten Schritt zu einer unsystematischen Wahl des Abschnitts (B2) oder der Variablenabhängigkeiten (B4) (siehe P2, P5). Auffällig ist, dass der Bearbeitungsbeginn (A) der zweiten Aufgabe durchweg strukturierter erscheint. So lasen hier alle Teilnehmenden die Aufgabenbeschreibung (A2) und orientierten sich im Anschluss auch am Dialogeintrag (A1). Dies legt nahe, dass das Vorgehen zu Beginn während der Bearbeitung der ersten Aufgabe stellenweise noch unklar war; dies schien dann aber in der zweiten Aufgabe bereits weniger problematisch. Mögliche Ursachen könnten die beschriebene geringe Sichtbarkeit des Dialogs (I1) sowie die von den Teilnehmenden als überladen wahrgenommene Einführung in die Lernumgebung (P) gewesen sein, wodurch sie sich nur schwer an die dort dargestellten Abläufe erinnern konnten. Eine striktere Benutzerführung und ausgeprägtere Schwerpunktsetzung (Jacobsen, 2014) unmittelbar zu Beginn der Aufgabenbearbeitung, beispielsweise entlang einer markanteren Hervorhebung des Dialogs, wäre denkbar, um den Blick der Teilnehmenden bereits während der ersten Aufgabe auf die für sie derzeit relevanten Bereiche der UI zu lenken.

Häufig betrachteten die Lernenden außerdem bereits während des Bearbeitungsbeginns den Quellcode und versuchten, den Programmablauf nachzuvollziehen (A3), ohne dass sie hierzu explizit aufgefordert wurden. Ihr Verhalten ähnelte damit, scheinbar intuitiv, dem gängigen, ersten Schritt des Debuggingprozesses in Form der Entwicklung eines Programmverständnisses (siehe Kapitel 2.4.1). Hierbei bewegten die Teilnehmenden teilweise den Cursor über den Quellcode und wählten unbeabsichtigt die korrekte Zeile bzw. den korrekten Abschnitt aus (B2), woraufhin sie bereits in der Bearbeitung voranschritten, ohne die entsprechende Zeile bzw. den entsprechenden Abschnitt bewusst oder begründet ausgewählt zu haben. Eine erneute Bestätigung der Auswahl einer Zeile bzw. eines Abschnitts seitens der Nutzer:innen wäre als Maßnahme denkbar, um eine solche unbeabsichtigte Auswahl zu vermeiden. Hierbei könnte auch eine stärkere Berücksichtigung nicht-funktionaler Anforderungen an Software (siehe Kapitel 5.1) helfen, wovon einige, unter anderem auch die *Fehlertoleranz*, innerhalb der ISO-Norm 9241-110 (2020) zusammengefasst werden.

Außerdem erfolgte stellenweise auch die Wahl der Abhängigkeiten im Rahmen der Auswahl (B) unsystematisch (B4). Gründe hierfür könnten unter Berücksichtigung der Äußerungen der Teilnehmenden sowohl in den Formulierungen der Fragen selbst (H1) als auch in der starken Ähnlichkeit der Fragen liegen, wodurch die zweite Frage nach den Variablenabhängigkeiten in der Bedingung innerhalb des Dialogs nicht wahrgenommen wurde (I2). Daher sollten die Formulierungen entsprechend angepasst und stärker voneinander differenziert werden.

Hinsichtlich des Bearbeitungsschrittes Annahme (C), in dem die Lernenden die SOLL-Werte tabellarisch eintragen sollten, fällt zunächst auf, dass einige Teilnehmende und insbesondere auch diejenigen, deren Aufgabenbearbeitung größtenteils unproblematisch ablief, das bedeutet, ohne viele Abweichungen ihres Verhaltens von dem vorgesehenen Verhalten während der

Verwendung der Lernumgebung (siehe P3, P4, P6), anstatt der SOLL-Werte die IST-Werte des fehlerhaften Quellcodes zum jeweiligen Ausführungszeitpunkt eintrugen (C2, K3). Es ist zum einen denkbar, dass den Teilnehmenden der Transfer von IST zu SOLL schwerfiel, indem sie nicht die Variablenwerte des sichtbaren Quellcodes eintragen, sondern die beabsichtigten Werte einer korrigierten Variante dieses Quellcodes bestimmen und eintragen sollten. Zum anderen wurde auch geäußert, dass dieser Schritt bereits der Fehleridentifikation vorgreift (K2); tatsächlich zeigen auch die Verhaltensabläufe der Teilnehmenden, dass die Identifikation des Fehlerursprungs (G) häufig innerhalb dieser Verhaltensphase erfolgte. Daher ist es möglich, dass die derzeitige Gestaltung dieses Schrittes eine noch zu ausführliche Vorstellung des korrigierten Quellcodes seitens der Nutzer:innen voraussetzt, um die SOLL-Werte ermitteln zu können und in Anlehnung an das *Segmenting Principle* mehr, kleinschrittigerer Unterstützung bedarf (siehe Kapitel 4.3.3).

Die Lernumgebung ermöglicht zu diesem Zeitpunkt außerdem eine systemseitige Eintragung der korrekten Werte, sobald einmalig falsche Werte eingetragen und bestätigt wurden (C3). Der durchgängige Gebrauch dieses Unterstützungsmechanismus, der in den Verhaltensabläufen von P2 zu erkennen ist, könnte darauf hindeuten, dass die Voraussetzungen zur Bereitstellung dieser Unterstützung eventuell angehoben werden sollten. Orientierend an den Äußerungen der Teilnehmenden wäre es außerdem hilfreich, die Eintragungshinweise sowohl zu erweitern (J3) als auch sichtbarer zu gestalten (I3) und die aktuelle Benennung der Zeitpunkte in Form von *vor Zeile X* und *nach Zeile X* zu konkretisieren (H2).

Der im Rahmen der Überprüfung vorgesehene Vergleich von SOLL und IST (D) erfolgte stellenweise statisch, ohne dass der Überprüfungszeitpunkt verändert wurde (D2). Darüber hinaus geschah die anschließende Bewertung eines Abschnitts teils zeilenweise statt als Ganzes (D4). Da sowohl die Möglichkeit des Wechsels des Zeitpunkts (K4) als auch das Vorgehen während der Bewertung in Teilen unklar war (K5), wäre es möglicherweise hilfreich, sie den Nutzerinnen und Nutzern künftig deutlicher zu kommunizieren. Außerdem wurde die visuelle Darstellung des jeweils aktuellen Überprüfungszeitpunktes in Form eines Pfeils teilweise übersehen (I4) und das simulierte Verhalten eines Haltepunktes während der Überprüfung nicht immer als solches erkannt (L2). Daher sollte zum einen die derzeitige Visualisierung des Zeitpunktes markanter hervorgehoben werden. Zum anderen wäre auch eine Ergänzung um eine realitätsnahe (Collins et al., 1987) visuelle Darstellung des Haltepunktes, beispielsweise in Form eines rot ausgefüllten Kreises, ähnlich wie in gängigen IDEs, denkbar.

Während der Aufgabenbearbeitung wurde mitunter außerdem häufigeres (J2) sowie ausführlicheres (J1) Feedback erwartet. Zudem führte die Entscheidung, identisches Feedback übersichthalber nicht erneut mit in den Dialog aufzunehmen, in Teilen dazu, dass Teilnehmende dies so interpretierten, als hätten sie für die jeweilige Interaktion kein Feedback erhalten (L1). Eine Anpassung des zur Verfügung gestellten Feedbacks könnte hier schnell Abhilfe schaffen, sollte jedoch berücksichtigen, dass die Formulierungen weiterhin prägnant und somit schnell sowie leicht zu lesen sind (Jacobsen, 2014).

Neben ihrer Ästhetik und Struktur (O1) wurde auch die Systematisierung des Debuggingprozesses nach der *Isolation*-Strategie innerhalb der Lernumgebung im Anschluss an die Aufgabenbearbeitung von einigen Teilnehmenden gelobt (N). Gleichzeitig zeigen die Verhaltensabläufe der zweiten Aufgabe von P1, P3 und P6, dass teils die Auffassung bestand, alle Zeilen bis zum Fehlerursprung betrachten zu müssen (K1). Dabei können die Teilnehmenden zwischen den Zeilen bzw. Abschnitten wählen, innerhalb derer der Wert von entweder der fehlerhaften Variable selbst oder einer direkt oder transitiv abhängigen Variablen zuletzt verändert wurde. Auch hier scheint eine spezifischere Formulierung innerhalb des Dialogs sinnvoll, um dies zu verdeutlichen.

Darüber hinaus ist es denkbar, die Variablenabhängigkeiten anstatt innerhalb des Schrittes *Auswahl* nach der Abschnittsbewertung und damit unmittelbar vor der Auswahl der nächsten Zeile bzw. des nächsten Abschnitts wählen zu lassen. Eventuell orientieren sich die Teilnehmenden dann aufgrund der zeitlichen Nähe dieser Schritte eher an den identifizierten Abhängigkeiten, sobald sie die nächste Zeile bzw. den nächsten Abschnitt auswählen.

Schließlich wurde auch darauf hingewiesen, dass neben der vorgesehenen Korrekturmöglichkeit weitere Wege existierten, das beabsichtigte Verhalten des Quellcodes zu erreichen (M). Hierbei sei jedoch anzumerken, dass bereits während der Aufgabenerstellung darauf geachtet wurde, dass es so weit wie möglich lediglich eine *naheliegende* Möglichkeit gibt, um den Quellcode entsprechend dessen Beschreibung zu korrigieren. Da die Lernumgebung außerdem das systematische Debugging entlang spezifischer logischer Fehler vermittelt, die gezielt innerhalb der Quellcodes platziert wurden, entspräche die Unterstützung mehrerer Korrekturmöglichkeiten weder der Idee der Lernumgebung noch ihrer derzeitigen Struktur.

6.5.2 Limitationen

Die vorgestellten Ergebnisse und ihre Diskussion unterliegen verschiedenen Einschränkungen, die nachfolgend näher dargelegt werden sollen.

Im Hinblick auf die Wahl der Methodik im Rahmen der Datenerhebung sei zunächst darauf hingewiesen, dass obgleich das *laute Denken* eine Betrachtung kognitiver Prozesse während einer Handlung erlaubt und insbesondere dessen Eignung für eine Untersuchung der Gebrauchstauglichkeit von (Software-)Systemen von verschiedenen Autor:innen bestätigt wird, diese Methode auch Herausforderungen und Schwierigkeiten mit sich bringt, die sich auf die Validität der Ergebnisse auswirken können (Frommann, 2005; Konrad, 2010).

Zum einen erbaten bzw. benötigten manche Teilnehmende zu verschiedenen Zeiten und in unterschiedlichen Bearbeitungsschritten Hilfestellungen. Infolge dieses variierenden äußeren Einflusses wichen die ihnen während der Aufgabenbearbeitung zur Verfügung stehenden Informationen, beispielsweise hinsichtlich der Handhabung der Lernumgebung, teils mehr, teils minder voneinander ab. Zum anderen wurde während der Aufgabenbearbeitung folgendes geäußert: „In so einer Situation möchte ich jetzt auch nicht komplett doof rüberkommen (P6).“ Dies

deutet darauf hin, dass die von den Teilnehmenden artikulierten Gedanken zuvor unter anderem hinsichtlich ihrer Wirkung auf andere Personen zum Teil vorgefiltert werden. Außerdem berichten Buber (2007) und Konrad (2010) von einem möglichen Einfluss der Verbalisierung mentaler Prozesse auf die kognitive Leistung der Versuchspersonen; dies kann sich beispielsweise in einer langsameren Aufgabenbearbeitung widerspiegeln. Darüber hinaus ist auch umstritten, inwieweit Versuchspersonen ihre kognitiven Prozesse überhaupt adäquat artikulieren können (Buber, 2007; Konrad, 2010); laut Konrad (2010) erfordert dies unter anderem ein geeignetes Vokabular. Diese Einschränkung ist insbesondere im Hinblick auf P1 und P2 relevant, die als Nicht-Deutsch-Muttersprachler:innen in der Verwendung der deutschen Sprache noch einige Unsicherheiten aufwiesen.

Weiterhin limitiert auch die Auswahl der Teilnehmenden die Validität der Ergebnisse. Da keinerlei externen Anreize für eine Teilnahme an der Evaluation in Aussicht gestellt wurden, liegt es nahe, dass die Personen, die sich dazu bereit erklärten, an der Evaluation teilzunehmen, Motivation sowie ein gewisses Interesse für den Gegenstand der Untersuchung mitbrachten. Damit spiegeln sie nicht unbedingt die breite Masse der Studierenden aus der primären Zielgruppe wider.

Eine wesentliche Einschränkung stellen die Untersuchungsdauer sowie der hiermit zusammenhängende Untersuchungsumfang dar. Um eine für die Teilnehmenden leistbare Untersuchungsdauer zu erreichen, umfasste die Evaluation die Bearbeitung lediglich zweier vorausgewählter Aufgaben; infolgedessen wurde auch der Übertrag der Phase *Scaffolding* nach Collins et al. (1987) im Rahmen der Evaluation komprimiert (siehe Kapitel 6.2.3). Somit wich die Bereitstellung von Unterstützungsmechanismen während der Evaluation von den ursprünglichen Überlegungen, wie sie in Kapitel 5.2.3 beschrieben werden, ab. Es ist möglich, dass die dort dargestellte, sukzessivere Komplettierung der Aufgabenbearbeitung zu einem abweichenden Verhalten und unterschiedlichen Eindrücken der Teilnehmenden geführt hätte als die im Rahmen der Evaluation verwendete, angepasste Reduktion der Aufgabenbearbeitung. Da es sich bei den ausgewählten Aufgaben um gängige Übungsaufgaben für Programmieranfänger:innen handelt (Berechnung der Fakultät, Summierung aller Werte in einer Liste), ist es außerdem denkbar, dass einige Teilnehmende zuvor schon selbst eine vergleichbare Aufgabenstellung bearbeitet haben. Daher besaßen sie möglicherweise bereits konkrete Vorstellungen, wie der korrekte Quellcode aussehen musste.

Zudem wächst der Nutzen der Logbuch-Komponente per se erst durch eine häufigere Nutzung der Lernumgebung und einer wachsenden Anzahl bearbeiteter Aufgaben sowie kennengelernter Fehlerarten. Daher bedarf es eines längerfristig ausgelegten Studiendesigns, um Aussagen über dessen derzeitige Ausgestaltung treffen zu können. Bisweilen wurde das Verhalten der Teilnehmenden ausschließlich während der Verwendung der Lernumgebung untersucht. Gemäß ihrer Zielsetzung, ein systematisches Vorgehen während des Debuggings zu vermitteln, führt sie die Nutzer:innen durch die Aufgabenbearbeitung und beschränkt dabei ihre Handlungsoptionen mal mehr, mal weniger. Daher erfordert eine Beurteilung des tatsächlichen

Einflusses der Lernumgebung auf das Vorgehen der Nutzer:innen während des Debuggings auch eine vergleichende Betrachtung ihres Vorgehens während der Fehlersuche und -behebung außerhalb der Lernumgebung. Hierbei wäre dann zu prüfen, inwieweit die Nutzer:innen die von der Lernumgebung vermittelte Systematik während des Debuggings eigenständig anwenden.

7 Fazit und Ausblick

Die vorliegende Arbeit verfolgte das Ziel, sowohl ein systematisches Vorgehen während des Debuggings zu vermitteln als auch Gestaltungsfolgerungen für ebensolche Lernumgebungen zu identifizieren (siehe Kapitel 1). Hierzu wurde unter anderem in Orientierung an dem CAA nach Collins et al. (1987) und den multimedialen Gestaltungskriterien nach Mayer (2020) eine prototypische Lernumgebung entwickelt, die ein systematisches Vorgehen entlang der *Isolation*-Strategie nach Zeller (2009) vermitteln und dabei insbesondere auch das *iterative knowledge* sowie das *experience knowledge* der Lernenden adressieren soll (siehe Kapitel 5).

Im Rahmen der anschließenden empirischen Untersuchung wurde die Lernumgebung von einigen Proband:innen für die Bearbeitung einer bzw. zweier Aufgaben genutzt, innerhalb derer sie systematisch einen spezifischen Fehler in einem bereitgestellten Quellcode finden und beheben sollten. Orientierend an der Forschungsfrage wurden dabei die Teilnehmenden gebeten, unter Berücksichtigung der Methode *Lautes Denken* ihre Eindrücke und Absichten während der Aufgabenbearbeitung zu verbalisieren, um ihr Verhalten und mögliche Schwierigkeiten während der Nutzung der Lernumgebung erheben zu können. Im Anschluss wurden die Ergebnisse entlang einer *zusammenfassenden Inhaltsanalyse* nach Mayring (2022) in Kategoriensysteme und Verhaltensabläufe überführt (siehe Kapitel 6).

Hierbei zeigte sich unter anderem, dass eine in Teilen striktere Benutzerführung entlang einer prägnanteren Schwerpunktsetzung, eine stärkere Sichtbarkeit ausgewählter Komponenten der Lernumgebung, verständlichere Formulierungen, häufigeres und präziseres Feedback sowie eine leicht veränderte Reihenfolge der Bearbeitungsschritte die vorgesehene Aufgabenbearbeitung weiter fördern würden. Darüber hinaus schien den Teilnehmenden insbesondere die Ermittlung der SOLL-Werte schwer zu fallen und bedarf daher zusätzlicher Unterstützung (siehe Kapitel 6).

Die Teilnehmenden lobten jedoch vor allem die Systematisierung des Debuggingprozesses nach der *Isolation*-Strategie (siehe Kapitel 6), welche in Kapitel 3.2.1 als übergeordnetes Ziel der Lernumgebung definiert wurde. Eine Betrachtung der Frage, inwieweit sich die Nutzung der Lernumgebung langfristig auf das Verhalten von Personen und insbesondere ihre Vorgehensweisen während des Debuggings auswirkt, könnte Gegenstand kommender Arbeiten sein. Hierbei wäre es beispielsweise denkbar, dass eine größere Gruppe von Proband:innen die Lernumgebung über eine längere Zeit hinweg nutzt und entlang von Untersuchungen ihres Verhaltens sowohl vor als auch im Anschluss an den Nutzungszeitraum geprüft wird, inwiefern ein möglicher Einfluss der Lernumgebung erkennbar ist. Hierbei könnte zudem auch betrachtet werden, ob die Nutzung der Lernumgebung zu einer effizienteren und sichereren Fehlersuche und -behebung unter den Teilnehmenden führt. Möglicherweise ließe sich daran feststellen, ob sich eine solche Lernumgebung eignet, um auch längerfristig das Vorgehen während des Debuggings unter Programmieranfänger:innen zu systematisieren.

Literaturverzeichnis

Ahmadzadeh, M., Elliman, D. & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. In J. Cunha, W. Fleischman, V. K. Proulx & J. Lourenço (Hrsg.), *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (S. 84–88). ACM. https://doi.org/10.1145/1067445.1067472

Altadmri, A. & Brown, N. C. (2015). 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In A. Decker, K. Eiselt, C. Alphonce & J. Tims (Hrsg.), *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (S. 522–527). ACM. https://doi.org/10.1145/2676723.2677258

Anderson, J. R., Reder, L. M. & Simon, H. A. (1996). Situated Learning and Education. *Educational Researcher*, *25*(4), 5–11. https://doi.org/10.3102/0013189X025004005

Ardimento, P., Bernardi, M. L., Cimitile, M. & Ruvo, G. de (2020). Reusing Bugged Source Code to Support Novice Programmers in Debugging Tasks. *ACM Transactions on Computing Education*, *20*(1), 1–24. https://doi.org/10.1145/3355616

Arnold, P., Kilian, L., Thillosen, A. & Zimmer, G. M. (2018). *Handbuch E-Learning: Lehren und Lernen mit digitalen Medien* (5. Aufl.). UTB; wbv. https://doi.org/10.36198/9783838549651

Brandt-Pook, H. & Kollmeier, R. (2020). *Softwareentwicklung kompakt und verständlich*. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-30631-1

Broy, M. & Kuhrmann, M. (2021). *Einführung in die Softwaretechnik*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-50263-1

Buber, R. (2007). Denke-Laut-Protokolle. In R. Buber (Hrsg.), *Lehrbuch. Qualitative Marktforschung: Konzepte, Methoden, Analysen* (S. 555–568). Gabler. https://doi.org/10.1007/978-3-8349-9258-1_35

Buber, R. (Hrsg.). (2007). *Lehrbuch. Qualitative Marktforschung: Konzepte, Methoden, Analysen*. Gabler. https://doi.org/10.1007/978-3-8349-9258-1

Carter, E. E. (2014). *An intelligent debugging tutor for novice computer science students*. Lehigh University. https://api.semanticscholar.org/CorpusID:60764322

Carver, M. S. & Risinger, S. C. (1987). Improving children's debugging skills. In G. M. Olson, S. Sheppard, E. Soloway (Hrsg.), *Empirical Studies of Programmers: Second Workshop* (S. 147–171). Ablex Publishing Corp.

Chandler, P. & Sweller, J. (1991). Cognitive Load Theory and the Format of Instruction. *Cognition and Instruction*, *8*(4), 293–332. https://doi.org/10.1207/s1532690xci0804_2

Claus, V. & Schwill, A. (Hrsg.). (2004). *Duden Informatik: Ein Fachlexikon für Studium und Praxis* (3. Aufl.). Duden Verlag.

Collins, A. & And Others. (1987). *Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. Technical Report No. 403*. https://eric.ed.gov/?id=ed284181

Decasse, M. & Emde, A.-M. (1988). A review of automated debugging systems: knowledge, strategies and techniques. In T. N. Nam (Hrsg.), *Proceedings of the 10th International Conference on Software Engineering* (S. 162–171). IEEE. https://doi.org/10.1109/ICSE.1988.93698

Dowson, M. (1997). The Ariane 5 Software Failure. *Software Engineering Notes, 22*(2), 84.

Dresing, T. & Pehl, T. (2018). *Praxisbuch Interview, Transkription & Analyse: Anleitungen und Regelsysteme für qualitativ Forschende* (8. Aufl.). Eigenverlag.

Ebrahimi, A. (1994). Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies*, *41*(4), 457–480. https://doi.org/10.1006/ijhc.1994.1069

Elting, A. (1996). *Das Lernprogramm „AVL": Konzeption, Entwicklung und empirische Untersuchung eines auf der Grundlage des Cognitive-Apprenticeship-Ansatzes erstellten Lernprogramms*. Peter Lang Gmbh, Internationaler Verlag der Wissenschaften.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L. & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Fix, V., Wiedenbeck, S. & Scholtz, J. (1993). Mental representations of programs by novices and experts. In B. Arnold (Hrsg.), *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (S. 74–79). ACM. https://doi.org/10.1145/169059.169088

Flindt, N. (2006). *e-learning - Theoriekonzepte und Praxiswirklichkeit*. Universitätsbibliothek Heidelberg. https://doi.org/10.11588/heidok.00006907

Frommann, U. (2005). *Die Methode „Lautes Denken"*. https://www.e-teaching.org/didaktik/qualitaet/usability/Lautes%20Denken_e-teaching_org.pdf

Garousi, V., Mesbah, A., Betin-Can, A. & Mirshokraie, S. (2013). A systematic mapping study of web application testing. *Information and Software Technology, 55*(8), 1374–1396. https://doi.org/10.1016/j.infsof.2013.02.006

Gerstenmaier, J., & Mandl, H. (1995). Wissenserwerb unter konstruktivistischer Perspektive. *Zeitschrift für Pädagogik*, *41*(6), 867-888. https://doi.org/10.1080/00461520.1986.9653026

Gick, M. L. (1986). Problem-Solving Strategies. *Educational Psychologist*, *21*(1-2), 99–120. https://doi.org/10.1080/00461520.1986.9653026

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, *7*(2), 151–182. https://doi.org/10.1016/S0020-7373(75)80005-8

Gräsel, C. & Gniewosz, B. (2011). Überblick Lehr-Lernforschung. In H. Reinders, H. Ditton, C. Gräsel & B. Gniewosz (Hrsg.), *Empirische Bildungsforschung* (S. 15–20). VS Verlag für Sozialwissenschaften. https://doi.org/10.1007/978-3-531-93021-3_1

Gräsel, C., & Mandl, H. (1993). Förderung des Erwerbs diagnostischer Strategien in fallbasierten Lernumgebungen. *Unterrichtswissenschaft*, *21*(4), 355-369. https://doi.org/10.25656/01:8195

Gruber, T. (2018). *Gedächtnis*. *Basiswissen Psychologie* (2. Aufl.). VS Verlag für Sozialwissenschaften.

Gugerty, L. & Olson, G. (1986). Debugging by skilled and novice programmers. In M. Mantei (Hrsg.), *ACM Digital Library, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (S. 171–174). ACM. https://doi.org/10.1145/22627.22367

Gumm, H.-P. & Sommer, M. (2011). *Einführung in die Informatik* (9. Aufl.). De Gruyter. https://doi.org/10.1524/9783486704587

Hailpern, B. & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, *41*(1), 4–12. https://doi.org/10.1147/sj.411.0004

Hicking, J. & Völkel, A. (2022). Anforderungsmanagement. In G. Schuh, V. Zeller & V. Stich (Hrsg.), *Digitalisierungs- und Informationsmanagement. Handbuch Produktion und Management 9* (S. 249–296). Springer Vieweg. https://doi.org/10.1007/978-3-662-63758-6_11

Hristova, M., Misra, A., Rutter, M. & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, *35*(1), 153–156. https://doi.org/10.1145/792548.611956

International Organization for Standardization (ISO). (2010). *24765:2010, Systems and software engineering — Vocabulary.*

International Organization for Standardization (ISO). (2011). *25010:2011, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE).*

International Organization for Standardization (ISO). (2020). *9241-110:2020, Ergonomics of human-system interaction – Interaction principles.*

Jacobsen, J. (2013). *Website-Konzeption* (7. Aufl.). Dpunkt Verlag.

Johnson, W. B., Entwistle, W. C. & Gaddis, K. S. (1982). *Development and demonstration of a laboratory tool for research in the design of games for training of troubleshooting skills*. https://apps.dtic.mil/sti/citations/tr/ada127128

Jonassen, D. H. & Hung, W. (2006). Learning to Troubleshoot: A New Theory-Based Design Architecture. *Educational Psychology Review*, *18*(1), 77–114. https://doi.org/10.1007/s10648-006-9001-8

Katz, I. & Anderson, J. (1987). Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction*, *3*(4), 351–399. https://doi.org/10.1207/s15327051hci0304_2

Kessler, C., & Anderson, J. R. (1986). A model of novice debugging in LISP. In E. Soloway & S. Iyengar (Hrsg.), *Empirical studies in programmers*, Norwood, NJ: Ablex.

Kieras, D. E. (1988). What mental model should be taught: Choosing instructional content for complex engineered systems. In J. Psotka, L. D. Massey, & S. A. Mutter (Eds.), *Intelligent tutoring systems: Lessons learned* (pp. 85–111). Lawrence Erlbaum Associates, Inc.

Klahr, D. & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, *20*(3), 362–404. https://doi.org/10.1016/0010-0285(88)90004-7

Konrad, K. (2010). Lautes Denken. In G. Mey & K. Mruck (Hrsg.), *Handbuch Qualitative Forschung in der Psychologie* (S. 476–490). doi:10.1007/978-3-531-92052-8_34

Kumar, A. N. (2002). Model-Based Reasoning for Domain Modeling in a Web-Based Intelligent Tutoring System to Help Students Learn to Debug C++ Programs. In S. A. Cerri (Hrsg.), *Lecture Notes in Computer Science:* (Bd. 2363, S. 792-801). Springer. https://doi.org/10.1007/3-540-47987-2_79

Kumar Basak, S., Wotto, M. & Bélanger, P. (2018). E-learning, M-learning and D-learning: Conceptual definition and comparative analysis. *E-Learning and Digital Media*, *15*(4), 191–216. https://doi.org/10.1177/2042753018785180

Laakso, M.-J., Malmi, L., Korhonen, A., Rajala, T., Kaila, E. & Salakoski, T. (2008). Using Roles of Variables to Enhance Novice's Debugging Work. In *Proceedings of the 2008 InSITE Conference.* Informing Science Institute. https://doi.org/10.28945/3229

Lee, G. C. & Wu, J. C. (1999). Debug It: A debugging practicing system. *Computers & Education*, *32*(2), 165–179. https://doi.org/10.1016/S0360-1315(98)00063-3

Li, C., Chan, E., Denny, P., Luxton-Reilly, A. & Tempero, E. (2019). Towards a Framework for Teaching Debugging. In A. Luxton-Reilly (Hrsg.), *Proceedings of the Twenty-First Australasian Computing Education Conference* (S. 79–86). ACM. https://doi.org/10.1145/3286960.3286970

Lischka, K. (2019). Entwicklung eines Lernsystems – Überblick, Aufbereitung und Integration von Lernmodulen. *Chancen und Herausforderungen des digitalen Lernens*, 167–181. https://doi.org/10.1007/978-3-662-59390-5_9

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119–150. https://doi.org/10.1145/1041624.1041673

Liu, T.-C. (2005). Web-based cognitive apprenticeship model for improving pre-service teachers' performances and attitudes towards instructional planning: Design and field experiment. *Educational Technology & Society*, *8*(2), 136–149.

Mandl, H., & Winkler, K. (2004). Von E-Learning-Trends und zukünftige Entwicklungen. In *GIL Jahrestagung* (S. 21-24).

Manochehr, N.-N. (2006). The Influence of Learning Styles on Learners in E-Learning Environments: An Empirical Study. *Computers in Higher Education Economics Review*, *18*(1), 10–14.

Martinez, M. E. (1998). What Is Problem Solving? *The Phi Delta Kappan*, *79*(8), 605–609. http://www.jstor.org/stable/20439287

Mayer, R. E. (2009). *Multimedia learning* (2 Aufl.). Cambridge University Press. https://doi.org/10.1017/CBO9780511811678

Mayer, R. E. (2020). *Multimedia learning* (3. Aufl.). Cambridge University Press. https://doi.org/10.1017/9781316941355

Mayring, P. (2022). *Qualitative Inhaltsanalyse: Grundlagen und Techniken*. Beltz Verlagsgruppe.

Mayring, P. & Fenzl, T. (2019). Qualitative Inhaltsanalyse. In N. Baur & J. Blasius (Hrsg.), *Handbuch Methoden der empirischen Sozialforschung* (S. 633–648). Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-21308-4_42

McCall, D. & Kölling, M. (2019). A New Look at Novice Programmer Errors. *ACM Transactions on Computing Education*, *19*(4), 1–30. https://doi.org/10.1145/3335814

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L. & Zander, C. (2008). Debugging: a review of the literature from an educational perspective.

Computer Science Education, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

Metzger, R. C. (2004). *Debugging by thinking: A multidisciplinary approach*. Elsevier Digital Press. https://www.sciencedirect.com/science/book/9781555583071

Michaeli, T. & Romeike, R. (2019a). Debuggen im Unterricht – Ein systematisches Vorgehen macht den Unterschied. *Informatik für alle.* https://doi.org/10.18420/INFOS2019-B9

Michaeli, T., & Romeike, R. (2019b). 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* (S. 1–7). ACM. https://doi.org/10.1145/3361721.3361724

Michaeli, T. & Romeike, R. (2019c). Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study. In *2019 IEEE Global Engineering Education Conference (educon)* (S. 1030–1038). IEEE. https://doi.org/10.1109/EDUCON.2019.8725282

Michaeli, T. & Romeike, R. (2021). Developing a Real World Escape Room for Assessing Preexisting Debugging Experience of K12 Students. In *2021 IEEE Global Engineering Education Conference (educon)* (S. 521–529). IEEE. https://doi.org/10.1109/EDUCON46332.2021.9453972

Mietzel, G. (2017). *Pädagogische Psychologie des Lernens und Lehrens* (9. Aufl.). Hogrefe Verlag.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *63*(2), 81–97. https://doi.org/10.1037/h0043158

Moore, J. L., Dickson-Deane, C. & Galyen, K. (2011). e-Learning, online learning, and distance learning environments: Are they the same? *The Internet and Higher Education*, *14*(2), 129–135. https://doi.org/10.1016/j.iheduc.2010.10.001

Mousavinasab, E., Zarifsanaiey, N., R. Niakan Kalhori, S., Rakhshan, M., Keikha, L., & Ghazi Saeedi, M. (2021). Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods. *Interactive Learning Environments*, *29*(1), 142-163.

Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: the good, the bad, and the quirky--a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, *40*(1), 163-167. https://doi.org/10.1145/1352322.1352191

Paas, F., Renkl, A. & Sweller, J. (2003). Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist*, *38*(1), 1–4. https://doi.org/10.1207/S15326985EP3801_1

Ruf, A., Berges, M. & Hubwieser, P. (2013). Types of assignments for novice programmers. In M. E. Caspersen, M. Knobelsdorf & R. Romeike (Hrsg.), *Proceedings of the 8th Workshop in Primary and Secondary Computing Education* (S. 43–44). ACM. https://doi.org/10.1145/2532748.2532777

Saadati, F., Ahmad Tarmizi, R., Mohd Ayub, A. F. & Abu Bakar, K. (2015). Effect of Internet-Based Cognitive Apprenticeship Model (i-CAM) on Statistics Learning among Postgraduate Students. *PloS one*, *10*(7). https://doi.org/10.1371/journal.pone.0129938

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (S. 37–39). IEEE. 10.1109/HCC.2002.1046340

Ministerium für Bildung, Jugend und Sport (MBJS). (2015). *Rahmenlehrplan Informatik für die Jahrgangsstufen 7 – 10*. https://bildungsserver.berlin-brandenburg.de/fileadmin/bbb/unterricht/rahmenlehrplaene/Rahmenlehrplanprojekt/amtliche_Fassung/Teil_C_Informatik_2015_11_10_WEB.pdf

Ministerium für Bildung, Jugend und Sport (MBJS). (2022). *Rahmenlehrplan Informatik für die gymnasiale Oberstufe*. https://bildungsserver.berlin-brandenburg.de/fileadmin/bbb/unterricht/rahmenlehrplaene/gymnasiale_oberstufe/curricula/2022/Teil_C_RLP_GOST_2022_Informatik.pdf

Schaafstal, A., Schraagen, J. M. & van Berlo, M. (2000). Cognitive task analysis and innovation of training: the case of structured troubleshooting. *Human factors*, *42*(1), 75–86. https://doi.org/10.1518/001872000779656570

Schneider, S., Beege, M., Nebel, S., Schnaubert, L. & Rey, G. D. (2022). The Cognitive-Affective-Social Theory of Learning in digital Environments (CASTLE). *Educational Psychology Review*, *34*(1), 1–38. https://doi.org/10.1007/s10648-021-09626-5

Schulmeister, R. (2005). *Lernplattformen für das virtuelle Lernen: Evaluation und Didaktik* (2. Aufl.). Oldenbourg Verlag.

Sedlmeyer, R. L., Thompson, W. B. & Johnson, P. E. (1983). Knowledge-based fault localization in debugging. In M. S. Johnson (Hrsg.), *Proceedings of the symposium on High-level debugging* (S. 25–31). ACM. https://doi.org/10.1145/1006147.1006154

Semler, J. (2016). *App-Design: Alles zu Gestaltung, Usability und User Experience* (1. Aufl.). Rheinwerk Verlag.

Siegmund, B., Perscheid, M., Taeumel, M. & Hirschfeld, R. (2014). Studying the Advancement in Debugging Practice of Professional Software Developers. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops* (S. 269–274). IEEE. https://doi.org/10.1109/ISSREW.2014.36

Siemens, G. (2005). Connectivism: A learning theory for the digital age. *International Journal of Instructional Technology and Distance Learning*. http://www.itdl.org/Journal/Jan_05/article01.htm

Sommerville, I. (2018). *Software Engineering* (9. Aufl.). Pearson Deutschland. https://elibrary.pearson.de/book/99.150005/9783863265120

Sonntag, K., & Stegmaier, R. (2007). *Arbeitsorientiertes Lernen: Zur Psychologie der Integration von Lernen und Arbeit* (1. Aufl.). W. Kohlhammer Verlag.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. https://doi.org/10.1145/2483710.2483713

Universität Potsdam (UP). (2019). *Modulkatalog des Studiengangs Informatik/Computational Science*. https://puls.uni-potsdam.de/qisserver/rds?state=verpublish&status=transform&vmfile=no&moduleCall=ModulkatalogAnzeigen&publishConfFile=modulkatalog&publishSubDir=up/modulkatalog&modulkatalog.mk_id=262&xslobject=pdf1

Verband zur Förderung des MINT-Unterrichts (MNU) & Gesellschaft für Informatik (GI). (2020). Gemeinsamer Referenzrahmen Informatik (GeRRI). https://www.mnu.de/images/publikationen/Informatik/GeRRI_komplett_WEB.pdf

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459–494. https://doi.org/10.1016/S0020-7373(85)80054-7

Yoon, B.-D. & Garcia, O. N. (1998). Cognitive activities and support in debugging. In *Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems* (S. 160–169). IEEE. https://doi.org/10.1109/HUICS.1998.659974

Žanko, Ž., Mladenović, M. & Krpan, D. (2022). Analysis of school students' misconceptions about basic programming concepts. *Journal of Computer Assisted Learning*, *38*(3), 719–730. https://doi.org/10.1111/jcal.12643

Zeller, A. (2009). *Why programs fail: a guide to systematic debugging*. Elsevier.

Anhang

A Kategoriensysteme

A.1 Verhalten

Kategorie	Subkategorie	Definition	Ankerbeispiel
Bearbeitungsbeginn (A)	Orientierung an Dialogeintrag (A1)	Der Dialog wird als Orientierungshilfe für die weitere Bearbeitung genutzt.	„Das heißt, jetzt finde ich auch das, wo erstmal gefragt wird, wo wurde der Wert zuletzt verändert.“ (P6)
	Lesen der Aufgabeninformationen (A2)	Die Beschreibung des Programms, sowie dessen SOLL- und IST-Werte werden gelesen.	„[…] ich lese es mir erstmal durch. Beschreibung, das Programm soll die Fakultät von zahl berechnen und das Ergebnis in der Konsole ausgeben. Soll, 120, Ist 24 […].“ (P3)
	Lesen des Quellcodes (A3)	Der Programmablauf wird schrittweise betrachtet.	„Okay, schau ich mir mal den Code an. Werte gleich 10, 15, 5. Die sollen alle aufsummiert werden. Genau. werteLaenge ist die Länge des Wertes, x gleich eins, ausgabe Summe, werteSumme gleich null.“ (P3)
	Suche nach Hilfe (A4)	Es wird versucht, einen „Helfer“ aufzurufen. Es liegt ein Missverständnis hinsichtlich der Erläuterungen in der Einführung vor.	„Dann werde ich mal jetzt die Hilfe suchen. […] Oder zumindest, dass ich verstanden habe von der Einführung, dass ich jetzt einen Helfer so quasi aufrufen kann.“ (P2)
Auswahl (B)	Systematische Abschnittswahl (B1)	Der Abschnitt wird basierend auf dem aktuellen Dialogeintrag ausgewählt.	„Wo wurde der Wert von fakultaet zuletzt verändert? Das würde, ich glaube, so hier sein. Dann klicke ich hier […].“ (P1)
	Unsystematische Abschnittswahl (B2)	Der Abschnitt wird ausgewählt… • nach mehrmaligem Versuchen *oder* • während eines mehrfachen, unklaren Klickens.	„Das heißt, ich gehe jetzt gerade. Guck mal, was ich alles anklicken kann. Okay.“ (P5)
	Systematische Wahl der Abhängigkeiten (B3)	Die Abhängigkeiten werden basierend auf dem aktuellen Dialogeintrag ausgewählt.	„Dementsprechend ist das von i abhängig. i ist aber an der Stelle von zahl abhängig. Muss ich dann zahl auch auswählen, dass. Ich versuche es einfach mal ohne zahl. Super. Okay.“ (P4)
	Unsystematische Wahl der Abhängigkeiten (B4)	Die Abhängigkeiten werden nach mehrmaligem Versuchen ausgewählt.	„Hm. Okay. Ich meine, gut, dann habe ich jetzt durchgetestet. Super.“ (P5)
Annahme (C)	Manuelle Eintragung der SOLL-Werte (C1)	Die SOLL-Werte werden in die Tabelle eingetragen.	„Fakultät nach Zeile zwei. (...) Sollte den Wert eins haben. (...) Vor zwei fünf. Er sollte den Wert. (...) Fünf. (..) Fünf Fakultät. Das ist. (...) 20, 60, 120. (...) i sollte, eigentlich existiert nicht […].“ (P1)
	Manuelle Eintragung der IST-Werte (C2)	Die IST-Werte werden in Teilen oder vollständig anstatt der SOLL-Werte in die Tabelle eingetragen.	„Dann ist fakultaet 24 und i sollte vier sein. (…) Leider falsch. […] Ach, soll.“ (P4)
	Systemseitige Eintragung (C3)	Es wird gewählt, dass die Werte systemseitig eingetragen werden	„Ist leider falsch. Dann muss ich die Tabelle ausfüllen […].“ (P2)

		• nachdem die eigenständige Eintragung fehlschlug *oder* • ohne es selbst zu versuchen.	
Überprüfung (D)	Dynamischer Wertevergleich (D1)	Die SOLL- und IST-Werte werden unter Verwendung einer mehrmaligen Änderung der Zeitpunkte miteinander verglichen.	„Es gibt jetzt hier das Fehler, weil es sollte eigentlich bis zu fünf gehen, aber es gibt keine fünfte Iteration, weil so in range geht nur von eins bis vier.“ (P1)
	Statischer Wertevergleich (D2)	Die SOLL- und IST-Werte werden statisch verglichen. Der Zeitpunkt wird nicht verändert.	„Wenn man Zeitpunkt, da kann man den Code nachprüfen. Ich sollte es mehrmals drücken. Das war ein bisschen unübersichtlich.“ (P2)
	Systematische, vollständige Abschnittsbewertung (D3)	Der Abschnitt wird begründet und als Ganzes als fehlerfrei, fehlerhaft oder ursächlich fehlerhaft bewertet.	„Also in Zeile neun am Anfang hast du erstmal noch diesen Wert als ausgabe und dann nach Zeile neun, dann die 20, die aus der while Schleife entstehen. Und dieser Wert ist fehlerhaft, aber nicht der Fehlerursprung […].“ (P3)
	Systematische, zeilenweise Abschnittsbewertung (D4)	Der Abschnitt wird begründet, jedoch zeilenweise als fehlerfrei, fehlerhaft oder ursächlich fehlerhaft bewertet.	„Und jetzt meinst du, ich soll das, diese Eingabe jetzt kann ich auf diese Zeile prüfen? Okay, fakultaet ist eins. Ist richtig. i ist eins. zahl ist immer noch. Fünf mal eins. Werte korrekt. […] also weil für die erste Iteration ist es dann korrekt.“ (P5)
	Unsystematische Abschnittsbewertung (D5)	Der Abschnitt wird nach mehrmaligem Versuchen von fehlerfrei, fehlerhaft und/ oder ursächlich fehlerhaft bewertet.	„Ich gebe einfach, die Werte korrekt ist hier falsch. Ich will mal auch Fehler eingeben. Zeigt mir immer noch Fehler.“ (P2)
	Unbeabsichtigte Abschnittsbewertung (D6)	Die Bewertung des Abschnitts stimmt nicht mit der intendierten Bewertung überein. Der Unterschied zwischen den Auswahlmöglichkeiten ist unklar.	„Das heißt in dem Fall x auf null möchte ich einsetzen. (...) Und. (...) Genau wieder weiß ich. (...) Nicht wie.“ (P2)
Korrektur (E)	Fehlerkorrektur (E1)	Der Fehler wird je nach Unterstützungsgrad systemseitig oder manuell korrigiert.	„Korrigiere die Zeile. Hm. (...) Deine Änderung. x gleich Null. Alles klar.“ (P4)
	Aufgabenabschluss (E2)	Die Aufgabenbearbeitung wird abgeschlossen.	„Und dann. Aufgabe aufschließen, deine Notizen.“ (P1)
Logbuch (F)		Das Logbuch wird geöffnet.	„Deswegen, ich gucke einfach mal kurz ins Logbuch. Okay, es ist noch nichts drin. Richtig.“ (P4)
Fehlerfindung (G)		Der Fehlerursprung wird identifiziert.	„Ich denke, dass das Problem ist, dass x gleich eins gesetzt wurde.“ (P3)

A.2 Eindrücke

Kategorie	Subkategorie	Definition	Ankerbeispiel
Unverständliche Formulierungen (H)	Fragen nach Abhängigkeiten (H1)	Die Fragen nach den Variablenabhängigkeiten in • der Zeile selbst *oder* • in der Bedingung werden nicht oder anders verstanden als beabsichtigt.	„Dann ist das Abhängigkeit von i und zahl, ja okay. Also in Abhängigkeit von i und zahl, wobei i in dieser Zeile ja eigentlich festgelegt wird, dementsprechend ist es ja eigentlich nicht abhängig, oder? Ich weiß nicht. Okay, auf jeden Fall. Hm.“ (P4)
	Beschreibung der Zeitpunkte (H2)	Die Formulierung der Zeitpunkte in Form von „vor Zeile X“ und „nach Zeile X“ wird nicht oder anders verstanden als beabsichtigt.	„Nach Zeile acht und vor Zeile zehn? Ist das nicht dasselbe?“ (P4)
	Unterschied zwischen Bewertungsmöglichkeiten (H3)	Die Formulierungen der Bewertungsmöglichkeiten in Form von „Werte korrekt“, „Werte fehlerhaft“ und „Werte fehlerhaft & Fehlerursprung“ werden nicht oder anders verstanden als beabsichtigt.	„Das heißt in dem Fall x auf null möchte ich einsetzen. (...) Und. (...) Genau wieder weiß ich. (...) Nicht wie.“ (P2)
Geringe Sichtbarkeit (I)	Dialog (I1)	Der Dialog wird als zu unscheinbar beschrieben und/oder übersehen.	„Was mir glaube ich persönlich bisschen helfen würde ist, wenn das hier noch mal in einem Kasten oder so farblich hervorgehoben wird.“ (P4)
	Ähnlichkeit der Fragen nach Abhängigkeiten (I2)	Die Formulierung der Fragen nach den Variablenabhängigkeiten (vgl. G1) ähneln sich sehr. Das aktualisierte Feedback wird übersehen.	„Weil ich gerade nicht so den Unterschied zur ersten Frage davor. […] Hier wird mir nicht so richtig der Unterschied klar.“ (P5)
	Eintragungshinweise für nicht-initialisierte Variablen (I3)	Der Hinweis, dass für nicht-initialisierte Variablen ein Schrägstrich eingetragen werden soll, wird übersehen.	„i sollte, eigentlich existiert nicht, aber dann würde ich so eins sagen oder null oder.“ (P1)
	Visuelle Darstellung des aktuellen Überprüfungszeitpunktes (I4)	Die visuelle Darstellung während des Überprüfungszeitpunktes in Form eines Pfeils wird übersehen.	„Ach so, jetzt habe ich auch erst gesehen, dass sich da dieser Pfeil bewegt.“ (P5)
Geringe Informationsdichte (J)	Unzureichendes Feedback (J1)	Das erhaltende Feedback wird als zu unspezifisch beschrieben.	„Was mich auch auf Moodle ein bisschen nervt, wenn einem nicht genau gesagt wird, was falsch ist.“ (P6)
	Fehlendes Feedback (J2)	Es irritiert, dass kein Feedback für fehlerhafte Eingaben während der	„Okay, der Button bestätigt nicht, wenn ich drauf klicke.“ (P4)

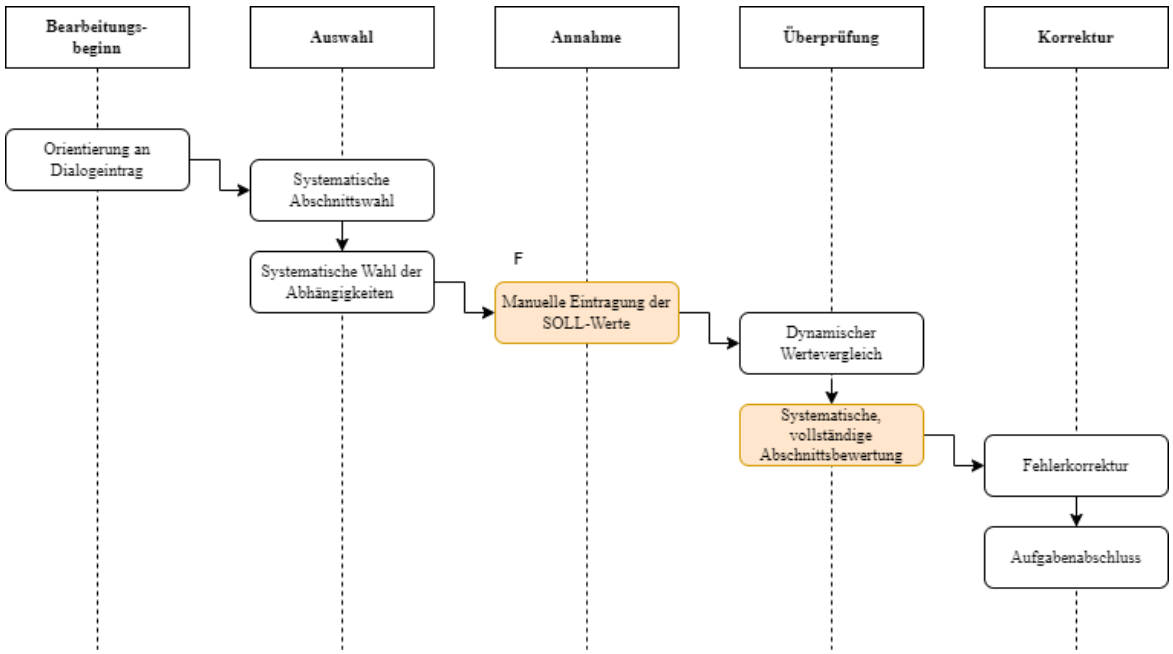
		Auswahl von Abhängigkeiten angezeigt wird.	
	Eintragungshinweise für verschiedene Datentypen (J3)	Die Eintragung verschiedener Datentypen ist unklar, z. B. dass Strings ohne Anführungszeichen eingetragen werden sollen.	„ein ganz bisschen verwirrt, wie genau ich den Typen angeben würde […].“ (P6)
Unklare Handhabung (K)	Betrachtung aller Zeilen bis zum Fehlerursprung (K1)	Es besteht die Auffassung, dass alle Zeilen bis zum Fehlerursprung betrachtet werden müssen.	„Okay, also wir gehen jetzt wirklich komplett zurück.“ (P6)
	Vorgriff auf Fehlerfindung während Eintragung von SOLL (K2)	Die Eintragung der SOLL-Werte wird als Vorwegnahme der Fehlerfindung verstanden.	„Ja, okay, das ist ja eigentlich schon Fehlerfindung dann, oder nicht?“ (P3)
	Verwechslung von SOLL und IST (K3)	Während der geforderten Eintragung der Werte werden SOLL und IST verwechselt.	„Und ich habe schon wieder verwirrt, dass es Soll und Ist ist.“ (P6)
	Wechsel des aktuellen Überprüfungszeitpunktes (K4)	Es ist unklar, dass bzw. wie der aktuelle Überprüfungszeitpunkt verändert werden kann.	„Wenn man Zeitpunkt, da kann man den Code nachprüfen. Ich sollte es mehrmals drücken. Das war ein bisschen unübersichtlich.“ (P2)
	Zeilen-/ statt abschnittsweiser Bewertung (K5)	Es ist unklar, ob der Abschnitt zeilenweise oder als Ganzes bewertet werden soll.	„[…] unsicher, ob ich in dem Falle auswählen soll, den Zeitpunkt und dann drauf drücken soll oder ob es halt generell über alle diese Zeilen geht.“ (P6)
Unerwartetes Verhalten (L)	Darstellung identischen Feedbacks (L1)	Es wird erwartet, dass identisches Feedback erneut im Dialog angezeigt wird.	„Aber irgendwie wundert es mich gerade, dass ich gar kein Feedback kriege, wenn ich was klicke.“ (P5)
	Simulation eines Haltepunktes (L2)	Das simulierte Verhalten eines Haltepunktes während der Überprüfung wird nicht als solches erkannt.	„Ich bin kurz verwirrt, warum das weiter geht. Ich dachte, das bleibt einfach bei dieser Zeile hängen.“ (P3)
	Systemseitige Korrektur (L3)	Anstatt einer systemseitigen Korrektur wird eine manuelle Korrektur des Fehlerursprungs erwartet.	„[…] ich hätte jetzt erwartet, dass ich den richtigen Code eingeben soll. Okay.“ (P6)
Restriktive Benutzerführung (M)		Es wird angemerkt, dass lediglich die Korrektur einer spezifischen Zeile unterstützt wird, obgleich auch andere, abweichende Korrekturmöglichkeiten existieren.	„[…] das ist irgendwie ein bisschen komisch, weil es gibt ja an der Stelle mehrere unterschiedliche Wege an so ein Debugging ran zu gehen.“ (P4)
Integration der *Isolation*-Strategie (N)		Die Systematisierung des Debuggingprozesses nach der *Isolation*-Strategie wird positiv bewertet.	„Es gab einen klaren, eindeutigen Weg, um eine Lösung zu finden. Das fand ich richtig gut.“ (P1) „Ich finde es positiv, dass man wirklich den Code durchgeht und die Abhängigkeiten zeigt, weil ich

			glaube, das fehlt am Anfang ein bisschen, wenn man noch kein algorithmisches Denken hat.“ (P3) „Ich fand das Interaktive super. Ich fand das richtig genial. Das hat einen ziemlich gut durch diesen ganzen Prozess vom Backtracking geführt.“ (P4)
Gestaltung (O)	Erfreulich (O1)	Die Software wird hinsichtlich ihrer Ästhetik und/oder Struktur positiv bewertet.	„Ich finde es grafisch sehr, sehr schön. […] und an sich auch gut strukturiert.“ (P5)
	Unerfreulich (O2)	Die Software wird hinsichtlich ihrer Ästhetik und/oder Struktur negativ bewertet.	„Ich finde das System ein ganz klein wenig clunky.“ (P6)
Informationsgehalt der Einführung (P)		Die Einführung wird als überladen beschrieben. Daher fällt ein Erinnern an die gezeigten Abläufe schwer.	„Nach der Einführung war mir nicht so ganz klar, was hier jetzt von mir gewollt ist, also was die Intention ist.“ (P5)

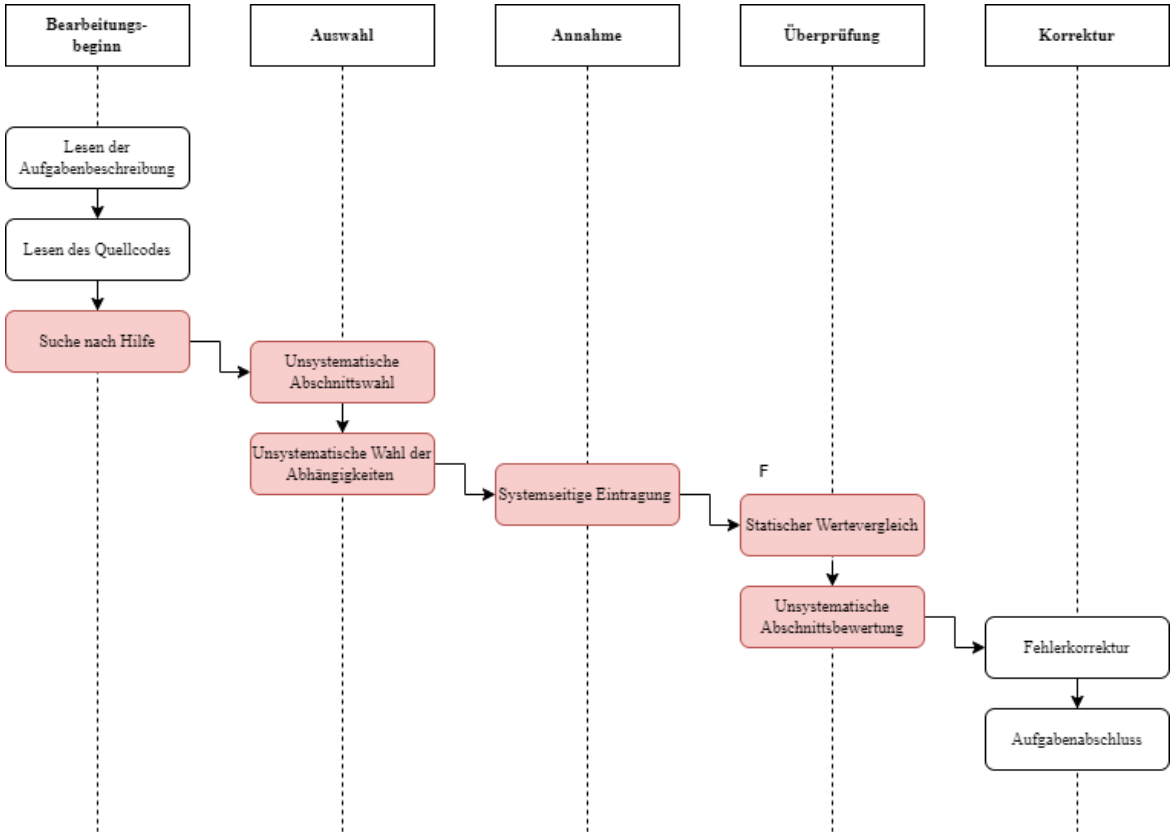
B Verhaltensabläufe

B.1 Aufgabe *Fakultät*

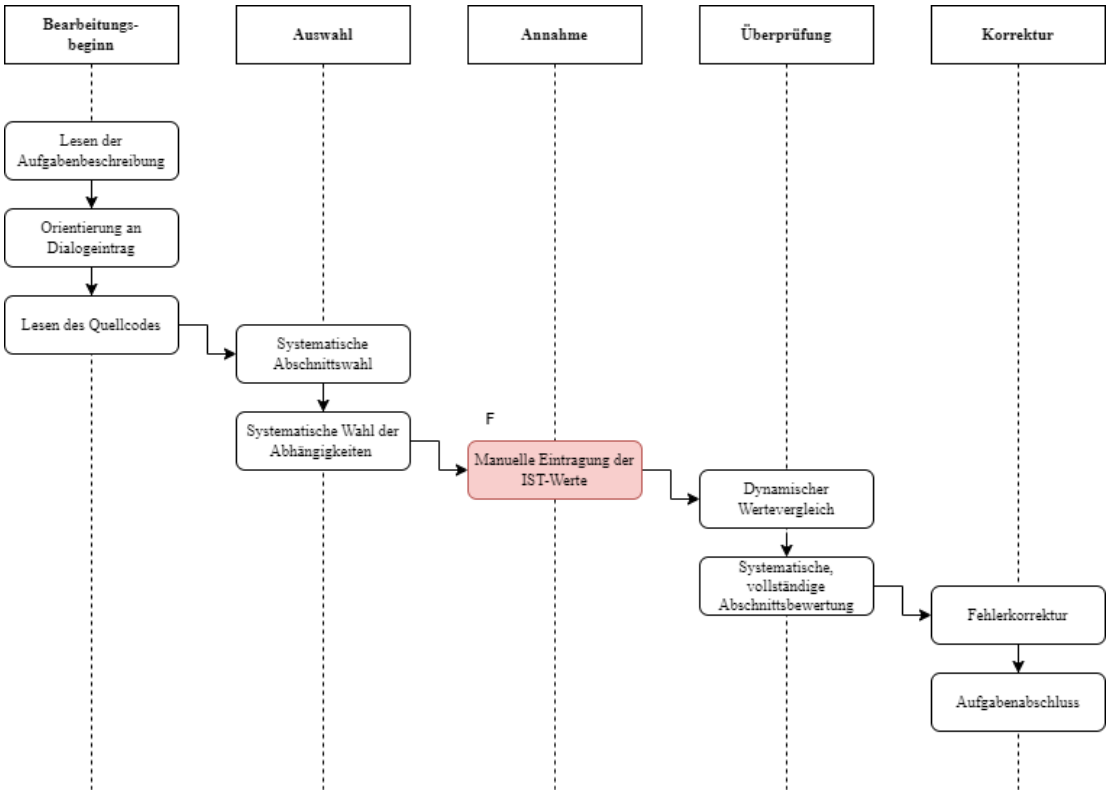
B.1.1 Person 1



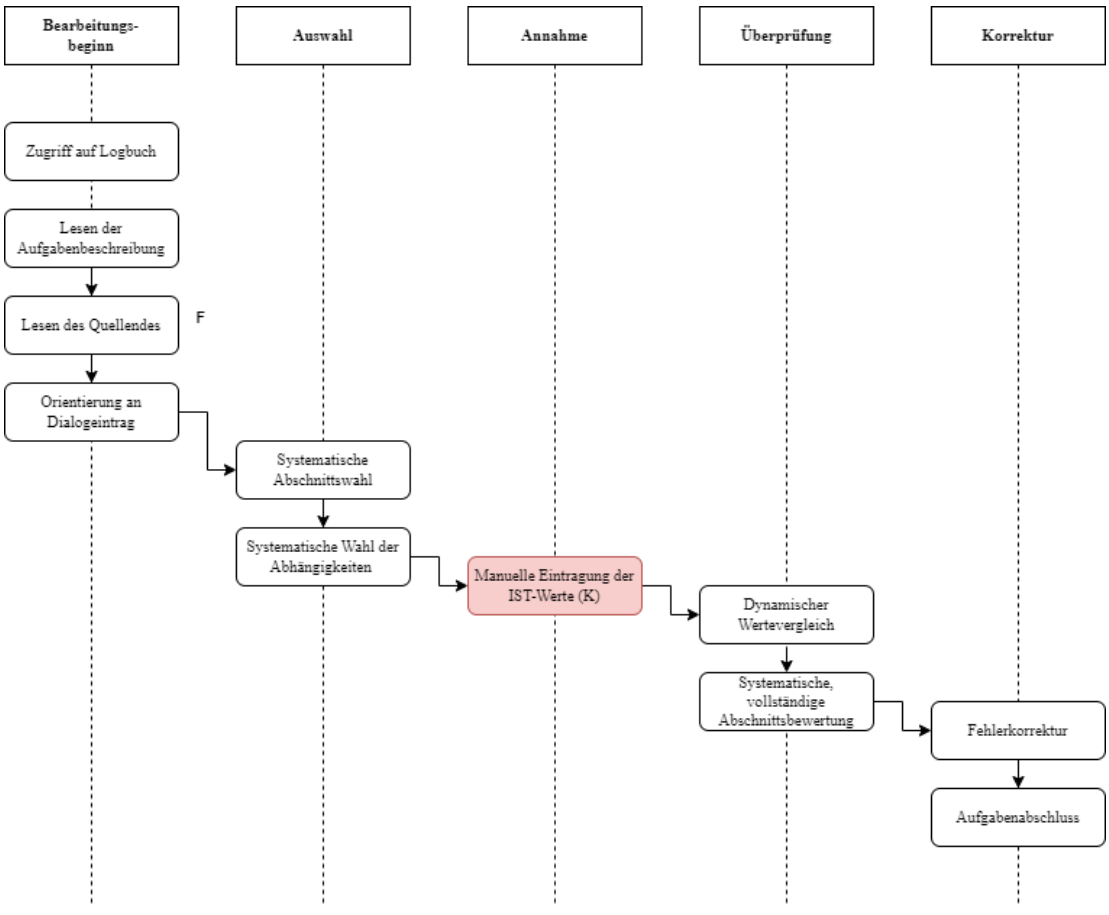
B.1.2 Person 2



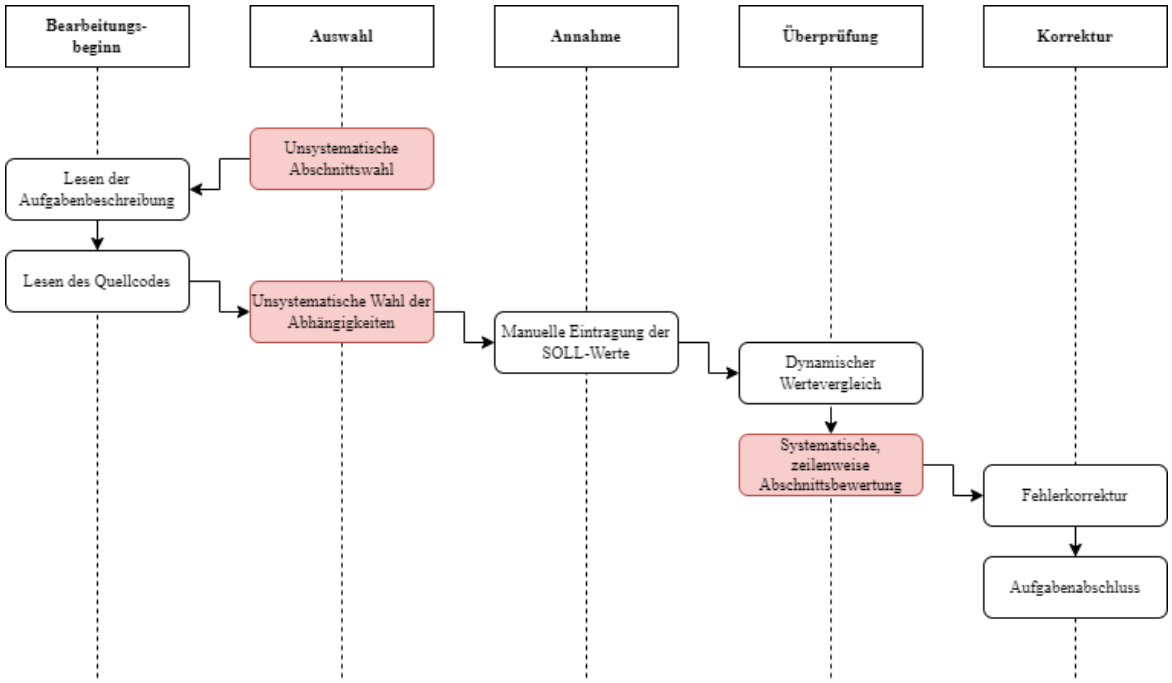
B.1.3 Person 3



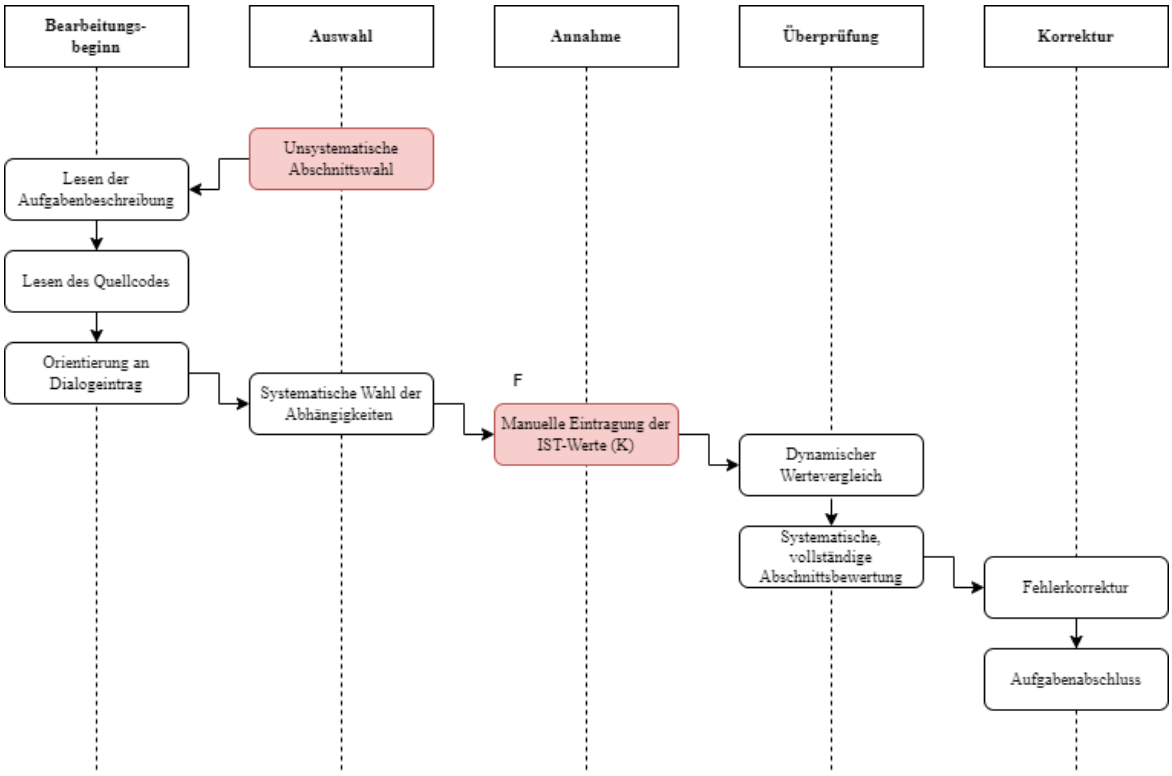
B.1.4 Person 4



B.1.5 Person 5

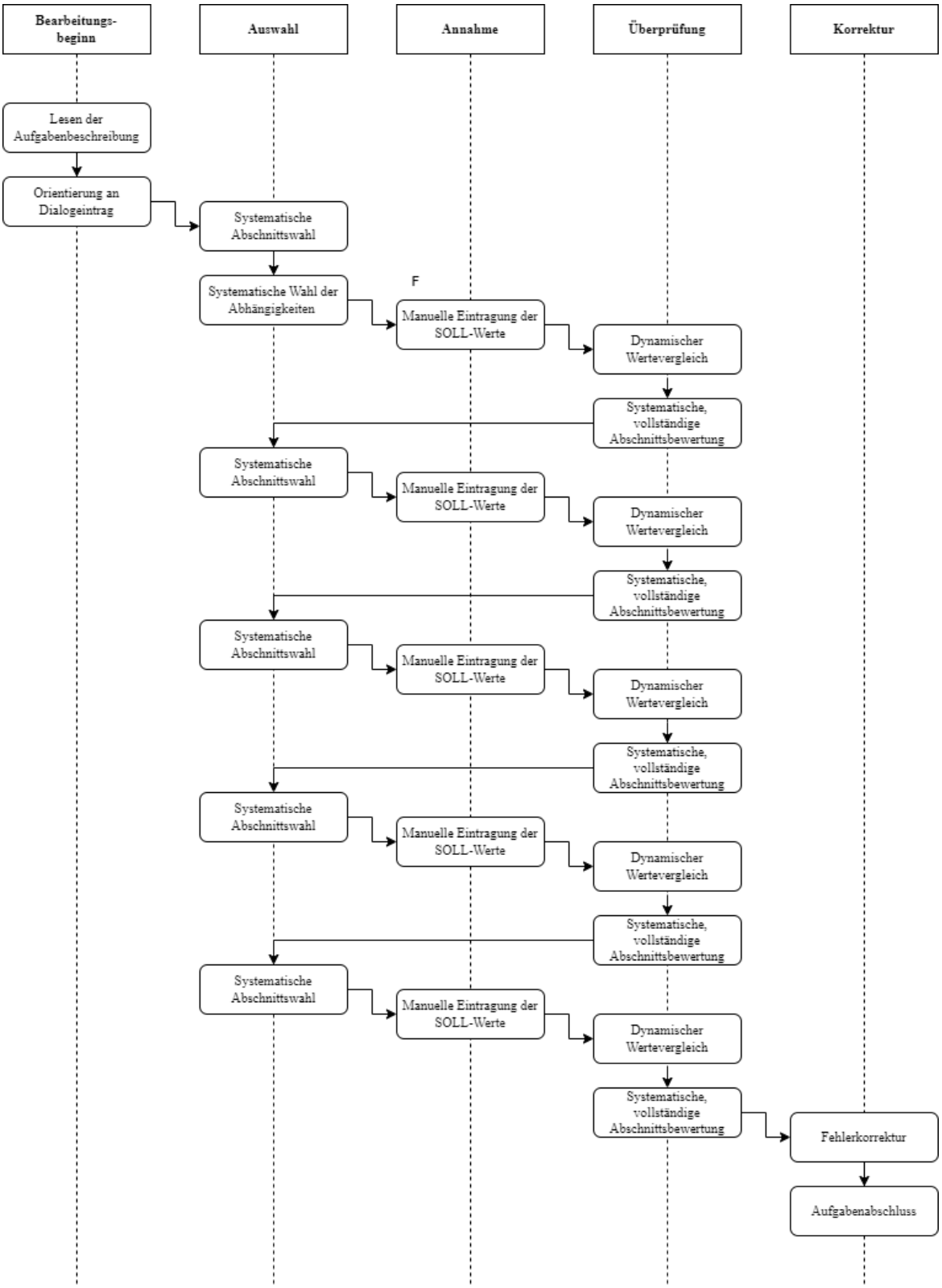


B.1.6 Person 6

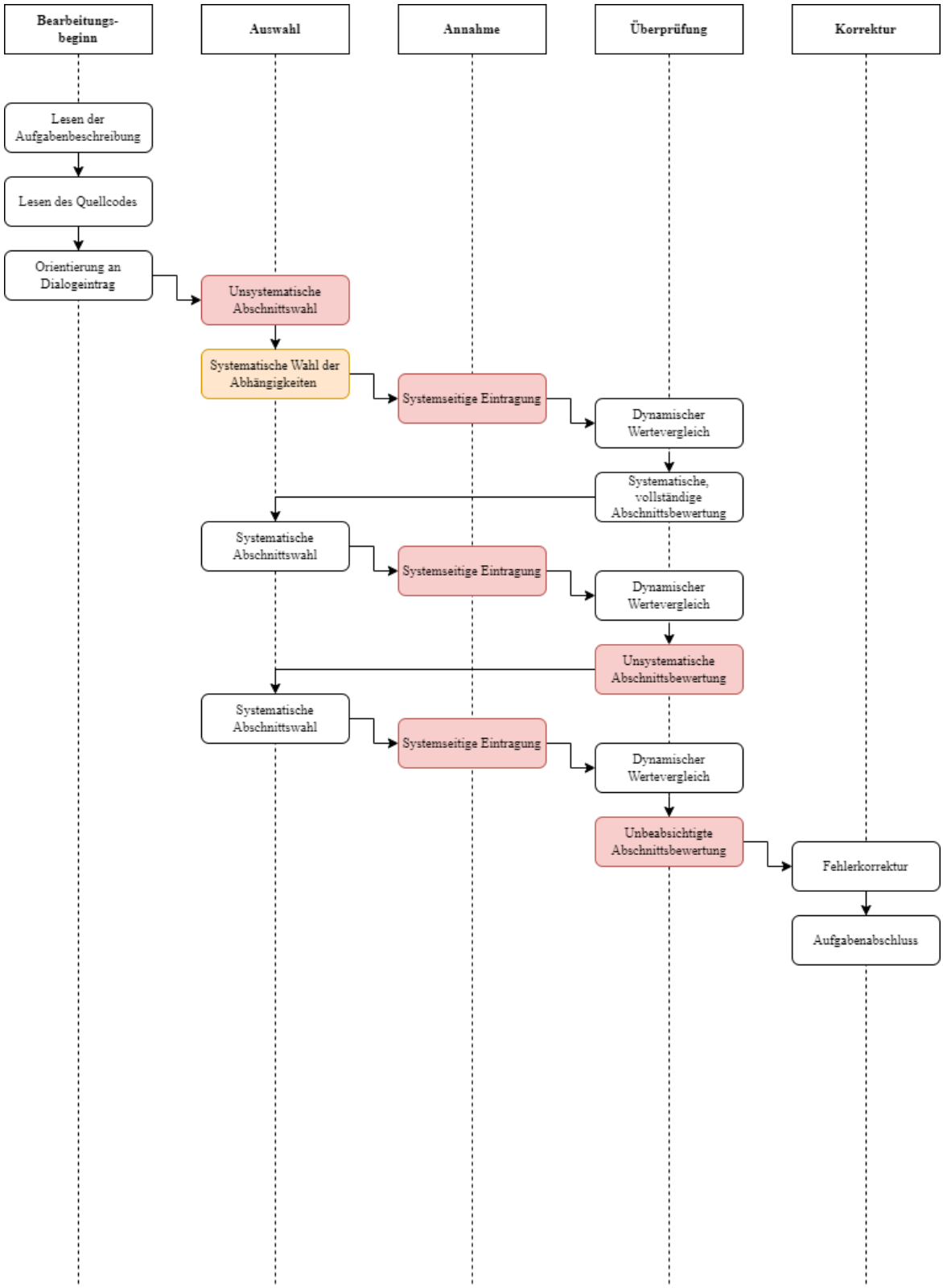


B.2 Aufgabe *Summierung*

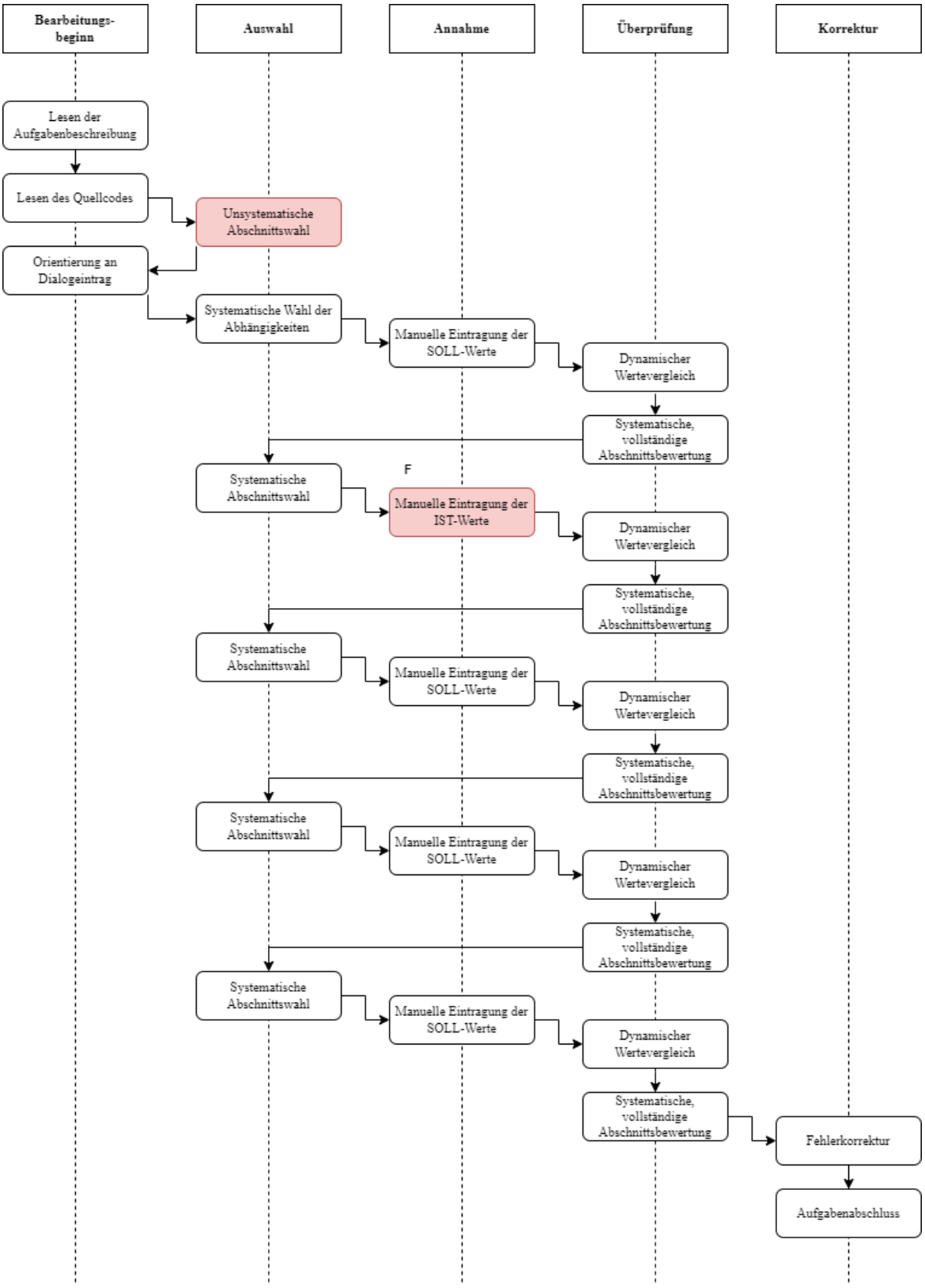
B.2.1 Person 1



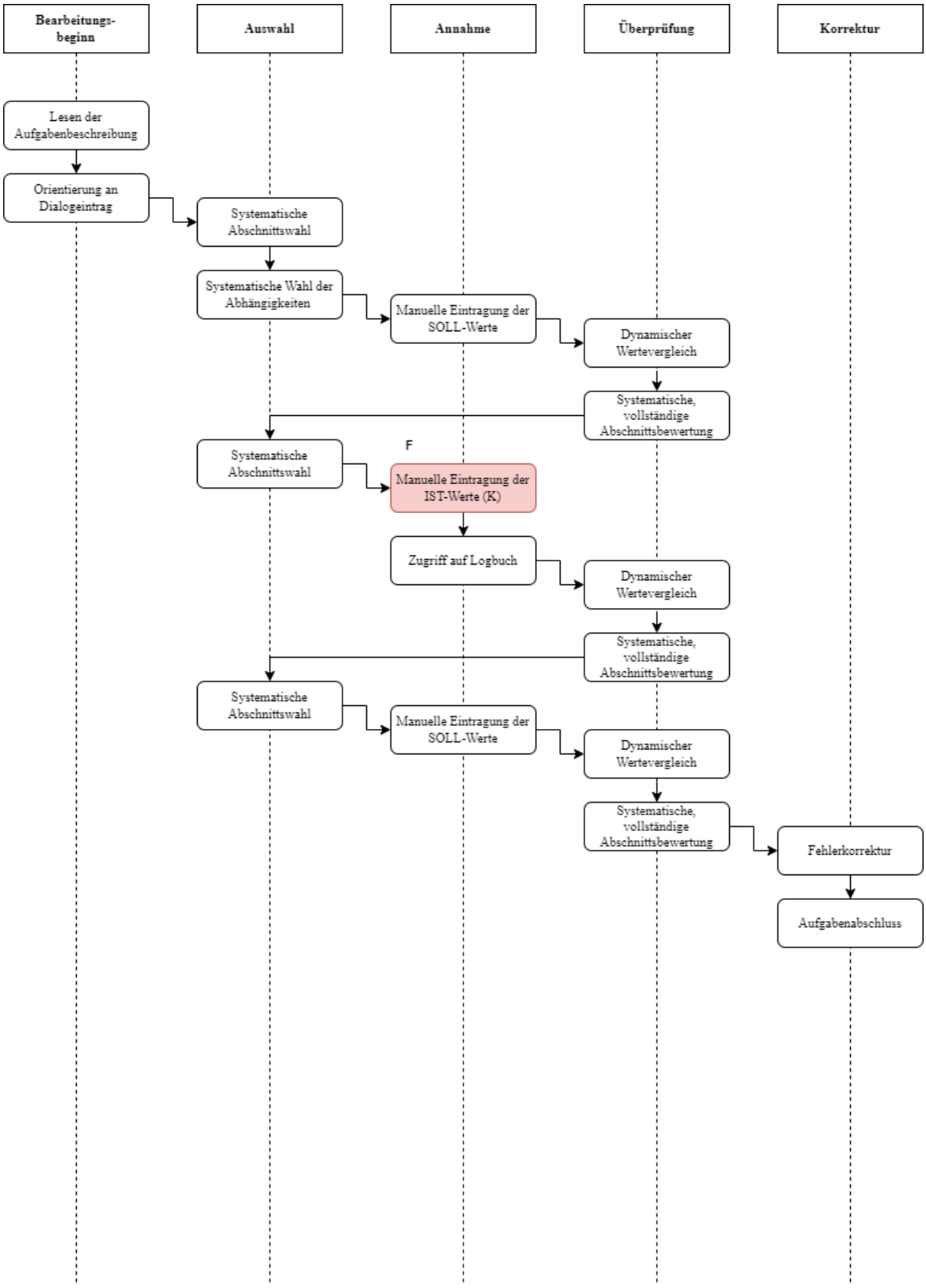
B.2.2 Person 2


Bearbeitungs-beginn
Auswahl
Annahme
Überprüfung
Korrektur
Lesen der Aufgabenbeschreibung
Lesen des Quellcodes
Orientierung an Dialogeintrag
Unsystematische Abschnittswahl
Systematische Wahl der Abhängigkeiten
Systemseitige Eintragung
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Systemseitige Eintragung
Dynamischer Wertevergleich
Unsystematische Abschnittsbewertung
Systematische Abschnittswahl
Systemseitige Eintragung
Dynamischer Wertevergleich
Unbeabsichtigte Abschnittsbewertung
Fehlerkorrektur
Aufgabenabschluss

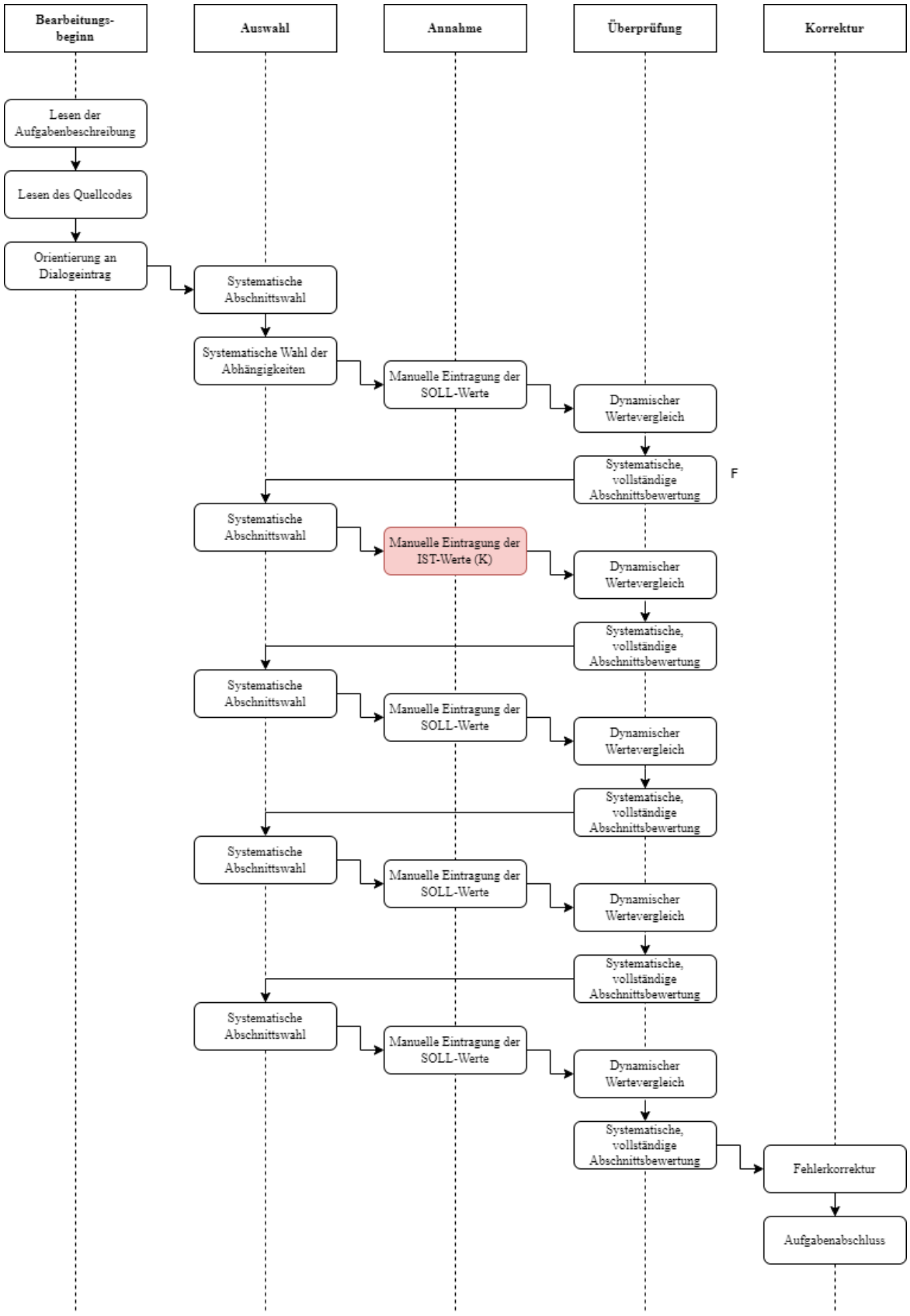
B.2.3 Person 3


Bearbeitungs-beginn
Auswahl
Annahme
Überprüfung
Korrektur
Lesen der Aufgabenbeschreibung
Lesen des Quellcodes
Unsystematische Abschnittswahl
Orientierung an Dialogeintrag
Systematische Wahl der Abhängigkeiten
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
F
Manuelle Eintragung der IST-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Fehlerkorrektur
Aufgabenabschluss

B.2.4 Person 4


Bearbeitungs-beginn
Auswahl
Annahme
Überprüfung
Korrektur
Lesen der Aufgabenbeschreibung
Orientierung an Dialogeintrag
Systematische Abschnittswahl
Systematische Wahl der Abhängigkeiten
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
F
Manuelle Eintragung der IST-Werte (K)
Zugriff auf Logbuch
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Fehlerkorrektur
Aufgabenabschluss

B.2.5 Person 6


Bearbeitungs-beginn
Auswahl
Annahme
Überprüfung
Korrektur
Lesen der Aufgabenbeschreibung
Lesen des Quellcodes
Orientierung an Dialogeintrag
Systematische Abschnittswahl
Systematische Wahl der Abhängigkeiten
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
F
Systematische Abschnittswahl
Manuelle Eintragung der IST-Werte (K)
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Systematische Abschnittswahl
Manuelle Eintragung der SOLL-Werte
Dynamischer Wertevergleich
Systematische, vollständige Abschnittsbewertung
Fehlerkorrektur
Aufgabenabschluss

Selbständigkeitserklärung

Ich, Felix Ziemann, versichere, dass ich die vorliegende Masterarbeit mit dem Titel „Entwicklung und Evaluation einer prototypischen Lernumgebung für das systematische Debugging logischer Fehler in Quellcode“ selbständig verfasst habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen und Formulierungen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem einzelnen Fall unter genauer Angabe der Quelle als Entlehnung kenntlich gemacht.

_____ _____

(Ort, Datum) (Unterschrift)