Bachelorarbeit

Codeboard: Verbesserung und Erweiterung der automatisierten Hilfestellungen

Zürcher Hochschule für Angewandte Wissenschaften
School of Management and Law, Winterthur
Studiengang Wirtschaftsinformatik

Verfasser:
Samuel Truniger

Betreuer:
David Grünert

Abgabedatum:
29. Mai 2023

Management Summary

Gerade weil das Erlernen der ersten Programmiersprache herausfordernd sein kann, spielt unmittelbares Feedback für Lernende, welche an Programmierkursen teilnehmen, eine essenzielle Rolle. Novizen sind beim Erkennen und Korrigieren von Fehlern im Code oder beim Erarbeiten eines Lösungsansatzes oft auf Unterstützung angewiesen. Lehrende können diese Art der Unterstützung allerdings nicht rund um die Uhr anbieten. Zusätzlich erschwert die hohe Studierendenanzahl eine individuelle Betreuung im nötigen Umfang. In diesem Kontext spielt das Codeboard, eine webbasierte Entwicklungsumgebung, eine zentrale Rolle. Diese wird am Wirtschaftsdepartement der ZHAW (SML) für diverse Informatikkurse verwendet und ermöglicht Lernenden, Aufgaben in der Programmiersprache Java zu lösen. Um der eingangs genannten Problematik entgegenzuwirken, bietet dieses Tool diverse Hilfestellungen (Helpersysteme). Lernende können Tipps zur Lösung anfragen, sie erhalten Erklärungen für Fehler im Programm oder können den Code testen, um Hinweise zum Testergebnis zu erhalten. Zudem können sie Fragen an Lehrende mit Bezug auf die erarbeitete Lösung stellen. Obschon diese Tools Studierenden unmittelbare Hilfe bieten, können diese weiter optimiert oder gar um neue Systeme und Funktionen ergänzt werden.

Das Verstehen von Programmcode bereitet Lernenden ebenfalls häufig Schwierigkeiten. Aus diesem Grund wurde der Coding-Assistant entwickelt, welcher zum Ziel hat, Java-Code zeilenweise in menschlicher Sprache zu erklären. Eine Evaluation mit Studierenden an der SML hat gezeigt, dass dieses Tool einen hohen Nutzen aufweist. Daraus ergibt sich das grundlegende Ziel dieser Thesis, diese Applikation in das Codeboard zu integrieren. Zusätzlich erfolgt eine Abstimmung und Kombination der integrierten Helpersysteme, um Studierenden eine bestmögliche Lernunterstützung bei Unklarheiten oder Problemen zu bieten.

Die Entwicklung dieser neuen Version des Codeboards basiert auf einem iterativ-inkrementellen Vorgehen. Dabei steht die Identifizierung von möglichen Erweiterungs- sowie Optimierungspotenzialen im Vordergrund. Um Systeme mit ähnlichen Funktionalitäten analysieren zu können, spielt die Aufarbeitung des aktuellen Forschungsstands eine bedeutende Rolle. Zudem wird die neue Version mit der Zielgruppe getestet, um erste Ergebnisse betreffend der Usability sowie der allenfalls optimierten Lernunterstützung zu erzielen.

Zum Ergebnis dieser Arbeit ist festzuhalten, dass die ursprüngliche Version des Codeboards mit Erfolg weiterentwickelt werden konnte. Der Coding-Assistant konnte integriert sowie die einzelnen Helpersysteme optimiert, um weitere Funktionalitäten erweitert und aufeinander abgestimmt werden. Die durchgeführte Evaluation zeigte, dass die neue Version Lernenden eine wertvolle Unterstützung beim Lösen von Aufgaben bietet. Zudem ergab die Auswertung, dass die erhobenen Anforderungen mit Erfolg umgesetzt wurden. Die neue Version kann bereits im nächsten Semester eingesetzt werden, wodurch die Wirksamkeit der Helpersysteme und deren Auswirkungen auf das Lernverhalten von Studierenden weiter untersucht werden können.

Inhaltsverzeichnis

1 Einleitung ... 1

1.1 Problemstellung ... 2

1.2 Zielsetzung ... 3

1.3 Forschungsfragen ... 3

1.4 Abgrenzungen ... 4

2 Methoden und Vorgehen ... 5

2.1 Vorgehen ... 5

2.1.1 Analyse der Ausgangslage ... 5

2.1.2 Requirements Engineering ... 6

2.1.3 Design ... 7

2.1.4 Implementierung ... 8

2.1.5 Testing ... 9

2.1.6 Evaluation ... 9

2.2 Aufbau der Arbeit ... 10

3 Analyse ... 12

3.1 Analyse der Ausgangslage ... 12

3.1.1 Coding-Assistant ... 12

3.1.2 Codeboard ... 14

3.1.3 Entwicklungsumgebung ... 21

3.2 Stand der Technik ... 21

3.2.1 Integrated Development Environments ... 21

3.2.2 Coding-Assistant ... 23

3.2.3 Compiler-Meldungen ... 23

3.2.4 Hinweissystem ... 24

3.3 Vergleich mit vorhandenen Lösungen ... 25

4 Design ... 27

4.1 Zusammenspiel der einzelnen Helpersysteme ... 27

4.1.1 Überlegungen zum Gesamtsystem ... 27

4.1.2 Optimierung des Zusammenspiels ... 28

4.2 Mockups ... 29

4.2.1 Codeboard ... 29

4.2.2 Helpersystem – Coding-Assistant (Erklärungen von Code) ... 31

4.2.3 Helpersystem – Coding-Assistant (Code-Blöcke & Gültigkeitsbereiche) 34

4.2.4 Helpersystem – Compiler-Meldungen ... 35

4.2.5 Helpersystem – Tipps ... 37

5 Codeboard ... 39

5.1 Requirements Engineering ... 39

5.1.1 Funktionale Anforderungen ... 39

5.1.2 Nicht-funktionale Anforderungen ... 40

5.1.3 Priorisierung der Anforderungen ... 40

5.2 Implementierung ... 40

5.2.1 Abweichungen ... 45

5.3 Testing ... 46

6 Helpersystem – Coding-Assistant ... 47

6.1 Sprachelemente und Integrationsvorbereitung ... 47

6.1.1 Requirements Engineering ... 47

6.1.2 Implementierung ... 48

6.1.3 Testing ... 49

6.2 Erklärungen ... 50

6.2.1 Requirements Engineering ... 50

6.2.2 Implementierung ... 51
6.2.3 Testing ... 56
6.3 Gültigkeitsbereiche ... 58
6.3.1 Requirements Engineering ... 58
6.3.2 Implementierung ... 59
6.3.3 Testing ... 62
6.4 Visualisierung von Code-Blöcken ... 64
6.4.1 Requirements Engineering ... 64
6.4.2 Implementierung ... 65
6.5 Code-Review ... 66
7 Helpersystem - Compiler-Meldungen ... 67
7.1 Requirements Engineering ... 67
7.1.1 Funktionale Anforderungen ... 67
7.1.2 Nicht-funktionale Anforderungen ... 67
7.1.3 Priorisierung der Anforderungen ... 68
7.2 Implementierung ... 68
7.2.1 Abweichungen ... 71
7.3 Testing ... 71
8 Helpersystem – Tipps ... 72
8.1 Requirements Engineering ... 72
8.1.1 Funktionale Anforderungen ... 72
8.1.2 Nicht-funktionale Anforderungen ... 72
8.1.3 Priorisierung der Anforderungen ... 72
8.2 Implementierung ... 73
8.2.1 Abweichungen ... 75
8.3 Testing ... 75

9 Resultate 76

9.1 Codeboard 76

9.2 Coding-Assistant 77

9.3 Compiler-Meldungen 78

9.4 Tipps 79

9.5 Manuelle Fragen und Antworten 80

10 Evaluation 81

10.1 Vorgehensweise 81

10.1.1 Aufgaben 81

10.1.2 Beobachtung 82

10.1.3 Befragung 83

10.2 Ergebnisse der Evaluation 83

10.2.1 Aufgaben 84

10.2.2 Beobachtung 84

10.2.3 Befragung 85

10.3 Resümee 88

11 Zusammenfassung und Ausblick 89

11.1 Fazit 89

11.2 Integrationsvorbereitung 92

11.3 Handlungsempfehlungen 93

12 Literaturverzeichnis 95

13 Anhang 105

Tabellenverzeichnis

Tabelle 5-1: Funktionale Anforderungen – Codeboard ... 39

Tabelle 5-2: Nicht-funktionale Anforderungen – Codeboard ... 40

Tabelle 5-3: Priorisierung – Codeboard ... 40

Tabelle 5-4: Sprachelement – Code-beautify ... 42

Tabelle 5-5: Gültigkeitsbereich – Code-beautify ... 43

Tabelle 6-1: Funktionale Anforderungen – Sprachelemente & Integrationsvorbereitung ... 47

Tabelle 6-2: Priorisierung – Sprachelemente & Integrationsvorbereitung ... 48

Tabelle 6-3: Funktionale Anforderungen – Erklärungen ... 50

Tabelle 6-4: Nicht-funktionale Anforderungen – Erklärungen ... 51

Tabelle 6-5: Priorisierung – Erklärungen ... 51

Tabelle 6-6: Speichermethode für Erklärungen (alte Version) ... 52

Tabelle 6-7: Funktionale Anforderungen – Gültigkeitsbereiche ... 58

Tabelle 6-8: Nicht-funktionale Anforderungen – Gültigkeitsbereiche ... 59

Tabelle 6-9: Priorisierung – Gültigkeitsbereiche ... 59

Tabelle 6-10: Funktionale Anforderungen – Visualisierung von Code-Blöcken ... 64

Tabelle 6-11: Nicht-funktionale Anforderungen – Visualisierung von Code-Blöcken ... 64

Tabelle 6-12: Priorisierung – Visualisierung von Code-Blöcken ... 65

Tabelle 7-1: Funktionale Anforderungen – Compiler-Meldungen ... 67

Tabelle 7-2: Priorisierung – Compiler-Meldungen ... 68

Tabelle 8-1: Funktionale Anforderungen – Tipps ... 72

Tabelle 8-2: Priorisierung – Tipps ... 72

Abbildungsverzeichnis

Abbildung 2-1: GitHub Project – KanBan-Board ... 9

Abbildung 3-1: Coding-Assistant – Ausgangslage ... 13

Abbildung 3-2: Codeboard – Ausgangslage ... 17

Abbildung 3-3: Codeboard – Test-Tab ... 18

Abbildung 3-4: Codeboard – Test-Tab – Fehler beim Kompilieren ... 18

Abbildung 3-5: Codeboard – Test-Tab – Test der Ausgabe fehlgeschlagen ... 19

Abbildung 3-6: Codeboard – Hilfe-Tab ... 20

Abbildung 4-1: Navigationsleiste rechts (alte und neue Version) ... 30

Abbildung 4-2: Navigationsleiste oben (alte Version) ... 31

Abbildung 4-3: Navigationsleiste oben (neue Version) ... 31

Abbildung 4-4: Mockup – Coding-Assistant (Erklärungen) ... 32

Abbildung 4-5: Mockup – Coding-Assistant (Code-Blöcke und Gültigkeitsbereiche) . 34

Abbildung 4-6: Mockup – Compiler-Meldungen ... 36

Abbildung 4-7: Mockup – Tipps ... 37

Abbildung 5-1: Fehler-Chatbox (Testen) ... 41

Abbildung 5-2: Fehler-Chatbox (Kompilieren) ... 41

Abbildung 5-3: Korrekte Gültigkeitsbereiche – Code-beautify ... 43

Abbildung 6-1: Speichermethode für Erklärungen (neue Version) ... 53

Abbildung 6-2: Chatbox für Code-Erklärungen ... 53

Abbildung 6-3: Hinweis-Chatbox ... 54

Abbildung 6-4: Chatbox für fehlerhaften Code ... 55

Abbildung 6-5: Warnsymbol für fehlerhaften Code ... 56

Abbildung 6-6: variableMap – Gültigkeitsbereich von Variablen ... 60

Abbildung 6-7: Gültigkeitsbereich von Variablen ... 61

Abbildung 6-8: Gültigkeitsbereich von Variablen – Fehlerchatboxen ... 62

Abbildung 6-9: Visualisierung von Code-Blöcken (erwartete Lösung)........................ 65
Abbildung 6-10: Visualisierung von Code-Blöcken (finale Lösung) 66
Abbildung 6-11: Visualisierung von Code-Blöcken – Konfiguration Ace-Editor......... 66
Abbildung 7-1: Statische Chatbox (1) – Compiler-Meldungen 69
Abbildung 7-2: Statische Chatbox (3) – Compiler-Meldungen 69
Abbildung 7-3: Fehler-Chatbox – Compiler-Meldungen... 69
Abbildung 7-4: Statische Chatbox (2) – Compiler-Meldungen 70
Abbildung 7-5: Fehlerchatbox ausgegraut – Compiler-Meldungen............................. 70
Abbildung 7-6: Statische Chatbox (4) – Compiler-Meldungen 70
Abbildung 8-1: Hinweis-Chatbox – Tipps 74
Abbildung 8-2: Modalfenster – Tipps 75
Abbildung 9-1: Codeboard – finale Version 77
Abbildung 9-2: Coding-Assistant – finale Version 78
Abbildung 9-3: Compiler-Meldungen – finale Version 79
Abbildung 9-4: Tipps – finale Version.. 79
Abbildung 9-5: Manuelle Fragen – finale Version... 80
Abbildung 10-1: Auswertung – Frage Q01 86
Abbildung 10-2: Auswertung – Frage Q02 86
Abbildung 10-3: Auswertung – Frage Q03 87
Abbildung 10-4: Auswertung – Frage Q04 87
Abbildung 10-5: Auswertung – Frage Q05 88

1 Einleitung

Gemäss einer Studie von Kowal et al. (2022, S. 15) sind in der Industrie 4.0 soziale sowie technische Kompetenzen unverzichtbar. Zudem sorgen der Internet-Handel, soziale Netzwerke sowie Online-News dafür, dass die Informationstechnologie den Alltag eines beträchtlichen Teils der Menschheit prägt (Kong & Abelson, 2022, S. 1). Daraus ergibt sich die Notwendigkeit, Informatik bereits in die Grundschule miteinzubeziehen, um Lernende auf diese Welt bestmöglich vorzubereiten (Kong & Abelson, 2022, S. 1).

Auch die Zürcher Hochschule für Angewandte Wissenschaften (ZHAW) kommt dieser Anforderung nach. Das Wirtschaftsdepartment School of Management and Law (SML) ermöglicht durch die Einführung des Wahlpflichtmoduls «Einführung Java Programmierung» Studierenden aller Studiengänge einen Einstieg in das Programmieren. Ein weiteres Angebot ist die «Java Summer School», welche Studierenden und externen Teilnehmenden ebenso eine Einführung in Java ermöglicht. Im Modul Software Engineering 1 des Studiengangs Wirtschaftsinformatik werden ebenfalls die Grundlagen des objektorientierten und systematischen Programmierens vermittelt. All diese Kurse verwenden eine Kombination aus der Kursmanagementplattform Moodle und einem webbasiertem Integrated Development Environment (IDE) Codeboard. Das Codeboard ist eine Entwicklungsumgebung, welche von der ETH Zürich lanciert wurde (Meyer et al., 2017). Diese Version wurde im Anschluss durch die ZHAW weiterentwickelt. Mittels dieser IDE können Studierende Programmiercode zu einer gegebenen Aufgabenstellung über einen Webbrowser erfassen, kompilieren, testen und schlussendlich einreichen.

Studierende, welche an solchen Kursen teilnehmen, müssen individuell unterstützt werden, was jedoch einen hohen Zeitaufwand für Lehrkräfte bedeutet. Hattie (2010) erwähnt die hohe Relevanz von Feedback, denn basierend auf mehr als 800 Meta-Reviews stellt dieses einen der Top 10 von 138 Einflussfaktoren auf die Leistungen von Studierenden dar. Eine Studie von Corbett und Anderson (2001) kam zur Erkenntnis, dass unmittelbares Feedback gerade in der Programmierlehre einen hohen Nutzen für Studierende aufweist. Gemäss einer Studie von Ott et al. (2016) etabliert sich eine gute Feedback-Praxis durch Laborsituationen, in welchen sich Studierende mit Dozierenden zu Programmieraufgaben austauschen können. Ott et al. erwähnen jedoch, dass dieses Verfahren an seine Grenzen stösst, wenn Dozierende abgelenkt, ungeschult oder überlastet sind oder wenn Lernende nicht an solchen Veranstaltungen teilnehmen. Die fehlende Zeit von

Dozierenden, auf die Bedürfnisse aller Studierenden einzugehen, wird auch von Strickroth und Pinkwart (2017, S. 17) als Herausforderung angesehen. In diesem Zusammenhang erwähnt Shute (2008, S. 166), dass sich verzögertes Feedback als frustrierend für Studierende erweisen kann. Um diesen Herausforderungen begegnen und Lernende mit unmittelbarem Feedback bedienen zu können, wurde das Codeboard um Hilfestellungen (Helpersysteme) erweitert. Aktuell wird Studierenden mittels abrufbaren Hinweisen zur Lösung, Erklärungen zu Compiler-Fehlermeldungen sowie der Möglichkeit, das Programm zu testen, Hilfe geboten. Zudem können sie Fragen an Lehrende zur erarbeiteten Lösung direkt über das IDE stellen.

Ein weiteres Tool in diesem Kontext ist der Coding-Assistant. Dabei handelt es sich um eine Webapplikation, welche Code in der Programmiersprache Java bei Bedarf block- und zeilenweise in Textform erklärt. Das Ziel dieser Anwendung ist, Studierenden beim Erlernen der im Modul Software Engineering behandelten Sprachelemente Unterstützung zu leisten. Dieses Hilfsprogramm soll künftig die vorhandenen Helpersysteme im Codeboard ergänzen, was den hohen Stellenwert begründet.

1.1 Problemstellung

Wie bereits im letzten Abschnitt erwähnt, erkennen viele Hochschulen, Schulen und Weiterbildungsanbieter die Relevanz der Informatik und die zentrale Rolle der Programmierlehre (Strickroth & Pinkwart, 2017). Nichtsdestotrotz haben viele Studierende Schwierigkeiten bei der Erlernung einer ersten Programmiersprache, beispielsweise mit dem Verständnis von Exception-Meldungen oder Syntax-Fehlern im Code (Hartmann et al., 2010). Eine Studie von Altadmri und Brown (2015) kam zur Erkenntnis, dass von 265‘000 Anwendern die meisten Fehlermeldungen durch die Nutzung von unausgewogenen geschweiften oder eckigen Klammern und Anführungszeichen entstehen. Daneben führt die menschliche Sprache zu Missverständnissen und Fehlern bei Lernenden (Bonar & Soloway, 1985). Diese entstehen aufgrund der unterschiedlichen Bedeutung von Begriffen in der menschlichen Sprache sowie der Programmiersprache (Miller, 2014, S. 2). Gerade auch die Ermittlung einer ersten Lösung stellt für viele Lernende eine Hürde dar (Antonucci et al., 2015). Die aktuelle Version des Codeboards besitzt bereits Funktionalitäten, um den Studierenden bei diesen Herausforderungen Hilfe zu leisten. Trotzdem ist anzumerken, dass gewisse Eigenschaften erweitert oder gar neue Artefakte hinzugefügt werden können. Beispielsweise kann der Coding-Assistant auf fehlerhaften Code

hinweisen, ohne dass dieser zuerst ausgeführt werden muss. Des Weiteren könnten Erklärungen für Code-Zeilen, die Diskrepanzen zwischen den Wortbedeutungen in der Programmiersprache und der menschlichen Sprache, beseitigen.

1.2 Zielsetzung

Die Integration des Coding-Assistant in das Codeboard stellt das grundlegende Ziel dieser Arbeit dar. Daraus abgeleitet ergibt sich das Ziel, die integrierten Helpersysteme (Coding-Assistant, Tipps, Test, Fragen und Compiler-Meldungen) aufeinander abzustimmen und zu kombinieren. Hieraus ergibt sich das Endprodukt dieser Arbeit, eine neue Version des Codeboards. Diese hat zum Ziel, Studierende bei Unklarheiten oder Problemen bestmöglich zu unterstützen und somit die Anzahl manuell gestellter Fragen von Studierenden an Dozierende in Zukunft zu reduzieren. Zudem sollte durch die Erweiterung des Codebords das Erlernen der Programmiersprache Java weiter vereinfacht werden.

1.3 Forschungsfragen

Aus der vorgängigen Problemstellung und Zielsetzung leitet sich nachfolgende Forschungsfrage ab:

> *Wie ist das bestehende Codeboard zu konfigurieren und zu erweitern, um den Studierenden eine bestmögliche Lernunterstützung und Hilfestellung bei Unklarheiten oder Problemen bieten zu können?*

Zusätzlich sind folgende Arbeitsfragen ebenso in Betracht zu ziehen:

1. Was sind die Anforderungen an die einzelnen Helpersysteme?
2. Wie können die einzelnen Helpersysteme am effektivsten ergänzt und aufeinander abgestimmt werden?
3. Wie kann die Wahl der zu ergänzenden und neu zu implementierenden Helpersysteme im Vergleich zu anderen Lösungen (KI) begründet werden?
4. Wie muss die Benutzeroberfläche des Codeboards gestaltet werden, um den vollen Nutzen aller Helpersysteme ausschöpfen zu können?
5. Welche Möglichkeiten gibt es, die einzelnen Systeme zu testen?
6. Wie kann ein möglicher Beitrag zur verbesserten Lernunterstützung durch die neue Version des Codeboards evaluiert und ausgewertet werden?

1.4 Abgrenzungen

Obschon in dieser Arbeit eine neue funktionsfähige Version des Codeboards entwickelt wird, ist deren tatsächliche Produktivschaltung nicht Teil dieser Arbeit. Aus diesem Grund werden Integrationstests in dieser Arbeit nicht behandelt. Nichtsdestotrotz werden Integrationsvorbereitungen erläutert, um eine mögliche Integration zu vereinfachen.

2 Methoden und Vorgehen

Dieses Kapitel beschreibt die gewählte Vorgehensweise für die Umsetzung der finalen Lösung. Dabei werden die einzelnen Entwicklungsschritte aufgezeigt und erläutert. Ein grober Überblick über den Aufbau dieser Arbeit rundet dieses Kapitel ab.

2.1 Vorgehen

Für die Umsetzung dieser Arbeit wird ein iterativ-inkrementelles Vorgehen gewählt. Bei diesem Vorgehen erhöht sich die Funktionalität der einzelnen Komponenten mit jeder Iteration, wodurch das finale Produkt inkrementell wächst (Höhn & Höppner, 2008, S. 308). Da in jeder Iteration die gleichen Entwicklungsschritte, Analyse-, Design-, Implementierungs-, Test- und Integrationsphase durchgeführt werden, spricht man von einem iterativen Prozess (Höhn & Höppner, 2008, S. 308). Iterationen bedeuten eine Unterteilung der Entwicklung in zeitliche Abschnitte, wohingegen Inkremente die Realisierung des Endproduktes in einzelnen Teilen darstellen (Pflüger & Queins, 2014, S. 53). Zudem ist durch das gewählte Vorgehen die überprüfbare Integration von Änderungen in jeder Entwicklungsphase gewährleistet (Höhn & Höppner, 2008, S. 308).

Diese Vorgehensweise wird gewählt, da dadurch die Erarbeitung einer vollständigen Anforderungsdokumentation im Voraus vermieden werden kann (Robertson & Robertson, 2013, S. 324). Vielmehr werden die Anforderungen parallel zur Produktentwicklung erhoben. Gleichzeitig ist eine gewisse Flexibilität gegeben, um auf neue Anforderungen oder Abweichungen von der ursprünglichen Planung reagieren zu können. Da das Verfassen von Quellcode einen Grossteil dieser Arbeit ausmacht, ist ein kontinuierlicher Austausch mit dem Dozierenden nötig. Dies untermauert gleichzeitig die gewählte Vorgehensweise, da die einzelnen Inkremente sowie die Anforderungen somit gezielter besprochen und umgesetzt werden können.

In den nachfolgenden Kapiteln 2.2.1 bis 2.2.6 werden zusammenfassend die einzelnen Entwicklungsschritte erläutert, welche bis auf die Kapitel 2.2.1 und 2.2.6 als iterativ-inkrementelle Schritte anzusehen sind.

2.1.1 Analyse der Ausgangslage

In einem ersten Schritt erfolgt eine Analyse der Ausgangslage. Zu Beginn erfolgt eine Analyse der bestehenden Komponenten, um einen Überblick über die vorhandenen Funktionalitäten zu gewinnen und mögliche Erweiterungspotenziale zu identifizieren. Die

Aufsetzung einer lokalen Entwicklungsumgebung, um die Implementation durchführen zu können, ist ebenso Bestandteil dieser Phase.

Im Anschluss wird der Forschungsstand durch ein systematisches Literaturreview dargelegt. Der Fokus liegt dabei auf bereits vorhandenen Unterstützungssystemen, welche ähnliche oder identische Funktionalitäten, wie die in dieser Arbeit verwendeten Helpersysteme aufweisen.

Die Aufarbeitung des Standes der Technik und eine Untersuchung der bestehenden Komponenten ermöglichen einen anschliessenden Vergleich der bereits vorhandenen Tools und jenen, welche im Rahmen dieser Arbeit umgesetzt oder erweitert werden. Daraus lassen sich Begründungen für die Wahl der jeweils umgesetzten Helpersysteme ableiten.

2.1.2 Requirements Engineering

Auf Basis der im vorherigen Kapitel beschriebenen Analyse erfolgt eine erste Erhebung von Anforderungen betreffend folgender Komponenten.

- Codeboard
- Helpersystem Coding-Assistant
- Helpersystem Compiler-Meldungen
- Helpersystem Tipps

Indem sich die Ermittlung, Dokumentation, Prüfung, Abstimmung sowie Verwaltung der Anforderungen nach einer ersten Erhebung ständig wiederholt, ist dieser Schritt als iterativ anzusehen (Pflüger & Queins, 2014, S. 69).

2.1.2.1 Arten von Anforderungen

Die erhobenen Anforderungen werden in funktionale und nicht-funktionale Anforderungen unterteilt. Funktionale Anforderungen beschreiben die Funktionalitäten, die das zu entwickelnde System bieten muss (Pohl & Rupp, 2015, S. 8). Die Grundlegenden Eigenschaften eines Systems werden hingegen durch nicht-funktionale Anforderungen definiert (Balzert, 2009, S. 489). Typische Anforderungen dieser Art umfassen die Zuverlässigkeit, Leistung, Verfügbarkeit oder Skalierbarkeit eines Systems (Pohl & Rupp, 2015, S. 8).

2.1.2.2 Priorisierung von Anforderungen

Eine weitere Überlegung im Zusammenhang mit der Erhebung von Anforderungen ist deren Priorisierung. Dadurch, dass viele einzelne Komponenten zur finalen Lösung

beitragen, wird dementsprechend eine grössere Anzahl von Anforderungen erhoben. Die Priorisierung wird durchgeführt, da oft nicht alle Anforderungen in der verfügbaren Zeit umgesetzt werden können (Berander & Andrews, 2005; Karlsson et al., 1998; Lehtola et al., 2004; McZara et al., 2015). Weil nicht alle Anforderungen dieselbe Relevanz aufweisen, erfolgt eine Priorisierung nach einem einzelnen Kriterium, um die Umsetzung planen zu können. Als Kriterium wird die Notwendigkeit *(necessity)* gewählt, welches in IEEE830 (1998) vorgeschlagen wird.

Für die Priorisierung nach diesem Kriterium wird die MoSCoW-Methode verwendet. Dabei werden die Anforderungen in folgende Kategorien eingeteilt:

- ***Must***: Diese Art von Anforderungen müssen im finalen Produkt umgesetzt sein, damit dieses als Erfolg angesehen werden kann (Brennan, 2009, S. 102).
- ***Should***: Es handelt sich um Anforderungen, welche wichtig sind und bei Bedarf umgesetzt werden sollen (Wiegers & Beatty, 2013, S. 320).
- ***Could***: Alle Anforderungen, welche wünschenswert, aber nicht notwendig sind, werden in diese Kategorie eingeteilt (Brennan, 2009, S. 102).
- ***Won't***: Anforderungen dieser Art sind nicht unwichtig, werden aber während des laufenden Projekt nicht umgesetzt (Hatton, 2008, S. 518). Dabei gilt anzumerken, dass dieses Kriterium in dieser Arbeit nicht verwendet wird. Vielmehr liegt der Fokus dieser Arbeit auf Anforderungen, welche in der vorhandenen Zeit umgesetzt werden können.

Svensson et al. (2011, S. 73) kamen in ihrer Studie zur Erkenntnis, dass die Zeit zur Produkteinführung von höherer Priorität ist als die Umsetzung einer komplexen Priorisierungstechnik. Weil die Zeit für die Umsetzung dieses Projekts begrenzt ist, wird die MoSCoW-Methode angewendet, welche schnell durchgeführt werden kann und einfach zu verstehen ist (Hatton, 2008, S. 523).

2.1.3 Design

In dieser Phase geht es darum, das Zusammenspiel zwischen den einzelnen Komponenten und Überlegungen bezüglich dem Gesamtsystem zu beschreiben. Dabei soll aufgezeigt werden, welche Abhängigkeiten zwischen den verschiedenen Helpersystemen in der neuen Version des Codeboards bestehen.

Zusätzlich werden für eine erste visuelle Darstellung der finalen Lösung Mockups für die einzelnen Helper-Systeme erstellt. Gemäss einer Studie von Ricca et al. (2010)

verbessern Mockups das Verständnis von Anforderungen. Weiter erwähnen Rivero et al. (2014, S. 671), dass die Modellierung von Anforderungen für eine Aufwandsminderung der Übersetzung von Anforderungen in Software-Artefakte sorgt. Die Mockups ergeben sich aus einer vorhergehenden Erhebung der Anforderungen, eigenen Überlegungen zum Gesamtsystem sowie aus dem iterativen Feedback zu den mit dem Auftraggeber besprochenen, bereits erstellten Mockups.

Dieser Schritt hängt eng mit dem *Requirements Engineering* zusammen. Einerseits ermöglicht die Evaluation der Mockups mit dem Auftraggeber das Identifizieren von neuen Anforderungen an das Endprodukt. Andererseits können dadurch bestehende Anforderungen hinterfragt und mögliche Folgen von geänderten oder neuen Anforderungen identifiziert werden (Pohl & Rupp, 2015, S. 30f.).

2.1.4 Implementierung

Im Schritt *Implementierung* wird basierend auf der vorhergehenden Analyse Quellcode geschrieben, um die jeweilige Komponente zu entwickeln oder zu erweitern. Dabei wird wo möglich auf bestehenden Code zurückgegriffen, um diesen zu ergänzen und somit die gewünschte Funktionalität zu erreichen. Wo nötig, wird zusätzlich in Zusammenarbeit mit dem Betreuer neuer Quellcode erarbeitet. Die gesamte Implementierung wird zur Veranschaulichung dokumentiert. Dabei sollen Entscheidungen, welche zur Umsetzung der finalen Lösung beitragen, aufgezeigt werden. Mögliche Variationen sollen erläutert und beschrieben und schließlich jene ausgewählt werden, welche sich am besten für die Umsetzung eignet.

Für die Organisation des Projekts wird ein GitHub Repository verwendet. Ein Repository umfasst alle zu einem Projekt dazugehörigen Dateien sowie deren Versionsverlauf (*GitHub - About Repositories*, o. J.). Alle erhobenen Anforderungen werden darin als Issues erfasst und in einem KanBan-Board dargestellt. Issues dienen der Protokollierung von Aktivitäten während der Entwicklung (*GitHub - About Issues*, o. J.). Diese Organisationsmethode wird gewählt, um einen Überblick über die umzusetzenden Anforderungen zu gewährleisten. Zusätzlich fällt es aufgrund der Versionierung aussenstehenden Personen leichter die Entwicklung nachzuvollziehen. Die nachfolgende Abbildung veranschaulicht das KanBan-Board, welches für diese Arbeit genutzt wird.

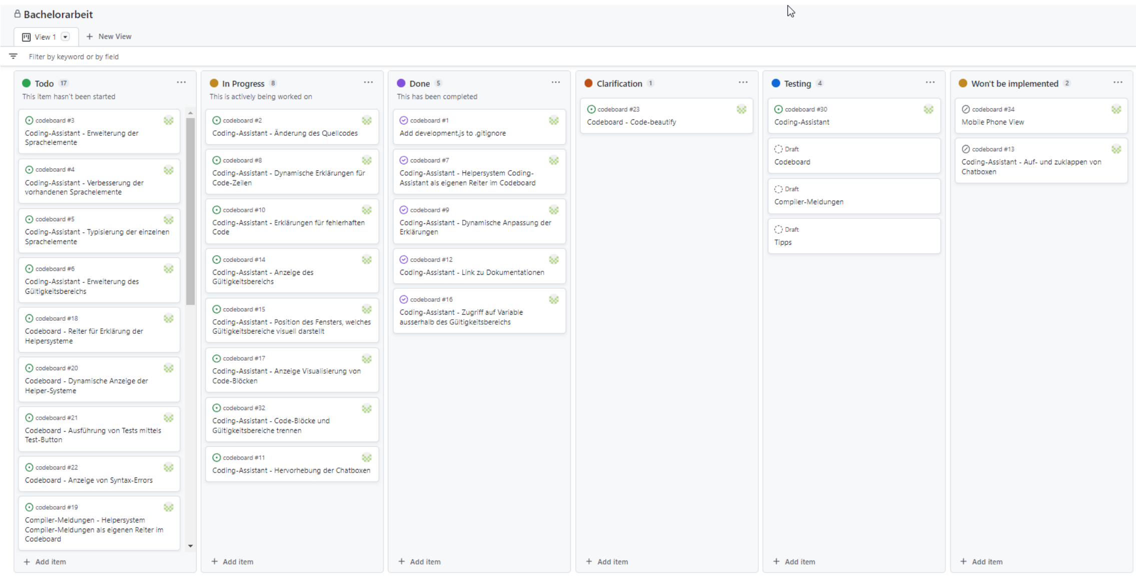


Abbildung 2-1: GitHub Project – KanBan-Board

2.1.5 Testing

Beim *Testing* werden die wichtigsten Funktionalitäten der einzelnen Komponenten getestet, um sicherzustellen, dass diese den Anforderungen entsprechen. Dabei werden Blackbox-Tests durchgeführt. Bei diesem Testverfahren handelt es sich um eine funktionale Testmethode (Naik & Tripathy, 2008, S. 20). Das Ziel dieses Verfahrens ist, die Korrektheit des Verhaltens des Systems zu überprüfen, indem die erwarteten mit den tatsächlichen Ergebnissen verglichen werden (Everett & McLeod, 2007, S. 68ff.). Dabei werden Eingaben auf das System angewendet, um zu validieren, ob das sichtbare Ergebnis des Programms mit dem erwarteten Resultat übereinstimmt (Naik & Tripathy, 2008, S. 21). Zusätzlich werden gegen Ende der Entwicklungsphase Usability-Tests durchgeführt, welche im darauffolgenden Kapitel erläutert werden.

2.1.6 Evaluation

Um die Funktionalität und den Mehrwert der gesamten Anwendung in einer realen Umgebung auswerten zu können, werden Usability-Tests durchgeführt. Unter Usability versteht man „das Ausmaß, in dem ein System, ein Produkt oder eine Dienstleistung von bestimmten Nutzern verwendet werden kann, um bestimmte Ziele mit Effektivität, Effizienz und Zufriedenheit in einem bestimmten Nutzungskontext zu erreichen“ (International Organization for Standardization, 2019, S. 3). Solche Tests werden durchgeführt, um Benutzer bei der Ausführung einer Aufgabe zu beobachten (Barnum, 2011, S. 13). Dabei handelt es sich um ein aus klassischen experimentellen Methoden abgeleitetes

Forschungsinstrument (Rubin & Chisnell, 2008). Das Ziel ist eine Gruppe von fünf Studierenden bezüglich ihrer Interaktion mit der finalen Lösung zu beobachten und im Anschluss zu befragen.

Eine Evaluation wird als wichtig erachtet, da somit eine erste Analyse des Mehrwertes der neuen Version des Codeboards durchgeführt werden kann. Zusätzlich ermöglicht das Testing eine zusätzliche Validierung, ob die Applikation die Anforderungen erfüllt (Barnum, 2011, S. 14).

2.2 Aufbau der Arbeit

Die Ausgangslage dieser Arbeit ergibt sich aus einer Analyse der technischen Ausgangslage sowie dem Stand der Technik. Die Erläuterung der technischen Ausgangslage hat zum Ziel, die Lesenden mit den grundlegenden Artefakten (Codeboard und Coding-Assistant) dieser Arbeit vertraut zu machen. Ein Vergleich der zu implementierenden Systeme mit möglichen Lösungen, welche im Stand der Technik erläutert werden, schliesst das Kapitel 3 ab. Gleichzeitig wird in diesem Kapitel die Beantwortung der dritten Arbeitsfrage behandelt.

Das Kapitel 4 umfasst die Beantwortung der zweiten Arbeitsfrage. Darin werden Zusammenhänge sowie mögliche Kombinationen der einzelnen Systeme erläutert. Ein weiterer Bestandteil dieses Kapitels ist die Beschreibung von Mockups, welche die erhobenen Anforderungen visuell darstellen. In den Kapiteln 5, 6, 7 und 8 wird die Optimierung der einzelnen Helpersysteme sowie die Integration des Coding-Assistant in das Codeboard erläutert. Dabei ist dem Codeboard sowie den drei Helpersystemen je ein eigenes Kapitel gewidmet, um deren Entwicklung getrennt voneinander beschreiben zu können. All diese Kapitel beinhalten die finale Anforderungsdokumentation, wobei zu beachten ist, dass sich der Changelog der Anforderungen im Anhang 1 befindet. Weitere Bestandteile sind die Dokumentation der Umsetzung sowie eine Beschreibung des Testing. Somit ist die Beantwortung der ersten, vierten und fünften Arbeitsfrage ebenfalls Thematik dieser Kapitel. Eine detaillierte Implementationsdokumentation kann zusätzlich dem Anhang 2 entnommen werden. Die finale Lösung wird im Anschluss in Kapitel 9 erläutert. Inhalt dieses Kapitels sind eine Darstellung der finalen Version sowie eine Zusammenfassung der wichtigsten Funktionalitäten.

Im vorletzten Kapitel erfolgt eine Beschreibung und Auswertung der Evaluation der neuen Version des Codeboards. Zusätzlich wird dadurch die letzte Arbeitsfrage

beantwortet. Eine Zusammenfassung der Antworten auf die Forschungsfrage und die dazugehörigen Arbeitsfragen sowie ein Ausblick obliegen dem Kapitel 11 und runden die Arbeit ab.

3 Analyse

Da diese Arbeit auf der aktuell genutzten Version des Codeboards sowie den dazugehörigen Helpersystemen aufbaut, erfolgt deren Beschreibung in der Analyse der technischen Ausgangslage. Zugleich werden die Funktionalitäten des Coding-Assistant beschrieben. Im Anschluss erfolgt eine Erläuterung des aktuellen Standes der Technik. Dabei werden dem Codeboard ähnliche Systeme sowie dazugehörige Helpersysteme beschrieben. Das Kapitel wird schlussendlich mit der Begründung zur Wahl der jeweiligen Systeme abgerundet.

3.1 Analyse der Ausgangslage

In den nachfolgenden Kapiteln werden das Codeboard und der Coding-Assistant sowie deren aktuellen Funktionalitäten näher beschrieben. Für jedes System werden zusätzlich mögliche Verbesserungspotenziale erläutert, welche bei Bedarf in Anforderungen umgesetzt werden. Die Beschreibung der für dieses Projekt genutzten Entwicklungsumgebung ist ebenso Teil dieses Kapitels.

3.1.1 Coding-Assistant

Der Coding-Assistant ist eine Webapplikation, welcher Studierenden Unterstützung beim Erlernen der Programmiersprache Java bietet. Eine erste Version wurde im Rahmen einer Bachelorarbeit entwickelt. Diese wurde anschliessend im Rahmen eines Integrationsseminars erweitert und verbessert. Das grundlegende Ziel ist, den von Studierenden erfassten Code zeilen- und blockweise in Textform zu erklären und den Gültigkeitsbereich von Variablen visuell darzustellen.

Der Assistant setzt sich aus folgenden drei Komponenten zusammen:

- *Code-Editor*: Der Editor namens Ace ist eine Java-Applikation, welche in Anwendungen oder Webseiten eingebettet werden kann (*Ace*, o. J.). Dieser bietet Studierenden die Möglichkeit, Code zu erfassen, welcher anschliessend in einem separaten Fenster erklärt wird.
- *Fenster für Erklärungen*: Der von Lernenden im Code-Editor eingegebene Code wird darin in Textform zeilenweise erklärt. Zudem werden Code-Blöcke visuell anhand von Farben dargestellt. Die Erklärungen sind zusätzlich mit einem Link versehen, welcher auf weiterführende Dokumentationen im Internet verweist.

Falls eine Code-Zeile einen Fehler aufweist, wird im Erklärungsfenster der dazugehörige Code angezeigt, welcher zusätzlich rot markiert wird.

- *Fenster für Gültigkeitsbereiche*: Dieses Fenster befindet sich auf der linken Seite des Code-Editors und zeigt Studierenden mittels farbiger Balken die Gültigkeitsbereiche von neu deklarierten Variablen an. Die Farben der Blöcke stimmen dabei mit den Markern für Variablennamen, welche direkt im Code-Editor markiert werden, überein. Somit ist klar ersichtlich, welcher Balken zu welcher Variable gehört.

Die nachfolgende Abbildung zeigt den Coding-Assistant und dessen Funktionalitäten.

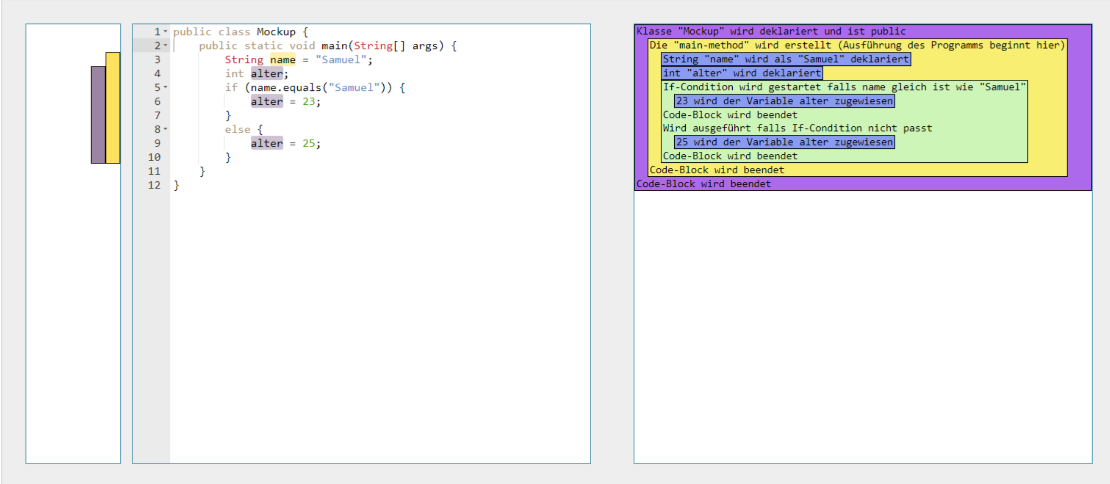


Abbildung 3-1: Coding-Assistant – Ausgangslage

Nachfolgend erfolgt eine zusammenfassende Erläuterung des Aufbaus und der Funktionalität dieser Applikation. Der von Lernenden erfasste Code wird jede Sekunde aus dem Code-Editor ausgelesen und zeilenweise getrennt. Anschliessend wird der gesamte Code über vorgängig definierte reguläre Ausdrücke (Regex) iteriert, welche sich zusammen mit den Erklärungen in einem JSON-File befinden. Regex sind Zeichenketten, welche verwendet werden, um "Suchen-und-Ersetzen"-Funktionen auf Text anzuwenden (*W3Schools - RegExp*, o. J.). Falls der Code mit einem Regex übereinstimmt, wird die dazugehörige Erklärung in einer Variable gespeichert und anschliessend auf derselben Höhe wie die dazugehörige Code-Zeile, im Fenster für Erklärungen angezeigt. Im Falle einer Deklaration einer Variable wird vorab geprüft, ob diese bereits deklariert wurde. Falls dies der Fall ist, wird die dazugehörige Erklärung im Erklärungsfenster rot markiert, um Lernende auf einen Fehler hinzuweisen. Dasselbe Prinzip gilt, falls auf eine Variable

zugegriffen wird, welche noch nicht deklariert wurde oder sich ausserhalb des Gültigkeitsbereichs befindet. In allen anderen Fällen wird der Variablenname und die Position des Balkens in einer Hashmap gespeichert und ein neuer Balken beginnend ab der Zeile, wo die Variable deklariert wurde, angezeigt. Durch die eckigen Klammern "{}" im Code-Editor wird die Position des Balkens berechnet, damit dieser korrekt angezeigt werden kann. Zusätzlich werden alle Variablennamen farbig markiert, welche mit dem dazugehörigen Balken, welcher den Gültigkeitsbereich visualisiert, übereinstimmt.

3.1.1.1 Verbesserungspotenzial

Obschon der Coding-Assistant sein grundlegendes Ziel erfüllt, gibt es dennoch Verbesserungspotenzial. Aktuell umfasst die Konfiguration des Coding-Assistant alle Sprachelemente der ersten fünf Semesterwochen des Moduls «Software Engineering 1». Um den vollen Nutzen dieses Tools ausschöpfen zu können, sollte es um die Sprachelemente des gesamten Moduls erweitert werden. Somit wären alle Sprachelemente, welche in diesem Modul behandelt werden, abgedeckt. Eine weitere Verbesserungsmassnahme betrifft das Überarbeiten der Erklärungen der bereits konfigurierten Sprachelemente. Diese könnten bezüglich Prägnanz, Detaillierungsgrad und Länge optimiert werden.

3.1.2 Codeboard

Wie in der Einleitung erwähnt, bietet die Zürcher Hochschule für Angewandte Wissenschaften (ZHAW) für diverse Programmierkurse ein E-Learning-Angebot bestehend aus Moodle und dem Codeboard an. Das Open-Source-Lernmanagementsystem Moodle ermöglicht ein ansprechendes und flexibles Online-Lernerlebnis (Rice, 2015, S. 1). Es dient als Verzeichnis, um Dateien hochzuladen, als Forum oder auch als Umfrage-Tool (Rice, 2015, S. 1). Nichtsdestotrotz beinhaltet Moodle keine eigene Entwicklungsumgebung, weshalb das Codeboard hinzugezogen wurde. Das Codeboard, eine webbasierte Open-Source IDE, wurde 2014 im Rahmen eines Forschungsprojekts an der ETH Zürich entwickelt (*codeboard.io*, o. J.). Es ermöglicht Nutzenden Programmiercode über einen Webbrowser einzugeben, diesen auszuführen und zu kompilieren (Meyer et al., 2017, S. 41). Registrierte Nutzerinnen und Nutzer haben zudem die Möglichkeit ihre Fortschritte zu speichern, sodass die Aufgabe auch zu einem späteren Zeitpunkt fertiggestellt werden kann (Gallego-Romero et al., 2020, S. 2507). Aktuell bietet das Codeboard Schnittstellen zu den Massive Open Online Courses (MOOCs) Coursera, Moodle und edX (Meyer et al., 2017, S. 41).

Das Codeboard wurde im Anschluss durch die ZHAW weiterentwickelt. Ein wichtiger Bestandteil der neuen Version sind Helpersysteme, welche Lernenden bei Unklarheiten oder Problemen zur Verfügung stehen. Aktuell umfasst das Codeboard folgende Systeme, welche in den nachfolgenden Kapiteln detaillierter erläutert, werden:

- *Tipps*: Für jede Aufgabe lassen sich unabhängig vom Stand der Lösung eine bestimmte Anzahl Hinweise abrufen. Diese sollen Lernende beim Erarbeiten der finalen Lösung unterstützen. Die Tipps müssen im Vorfeld für jede Aufgabe manuell in einer Konfigurationsdatei erfasst werden. Falls Studierende diese Hinweise anfordern, werden diese gemäss vorherig definierter Reihenfolge inkrementell angezeigt.
- *Compiler-Meldungen*: Falls Studierende den erarbeiteten Code ausführen und dieser syntaktische oder semantische Fehler aufweist, erscheint in der Konsole eine Fehlermeldung, welche zugleich in Textform erläutert wird. Falls im Programm mehrere Fehler vorhanden sind, wird jeweils der erste Fehler erklärt. Im folgenden Kapitel 3.2.2.2 erfolgt eine detailliertere Erläuterung dieses Verhaltens. Dieses System wurde im Rahmen eines Integrationsseminars entwickelt, wobei die gesamte Funktionalität in der dazugehörigen Dokumentation beschrieben ist.
- *Tests*: Studierende haben die Möglichkeit, den Code vor der Abgabe testen zu lassen. Dabei wird der Code zuerst kompiliert und bei einer Fehlermeldung eine dazugehörige Erklärung, analog zu den Compiler-Meldungen, angezeigt. Falls die Kompilierung erfolgreich war, wird in einem zweiten Schritt geprüft, ob das effektive Verhalten mit dem erwarteten Verhalten des Programms übereinstimmt. Dabei muss das erwartete Verhalten vorab in derselben Datei definiert werden, in welchem sich die Tipps befinden. Falls dieser Input-/Outputtest fehlschlägt, erfolgen entsprechende Hinweise mit Verbesserungsvorschlägen an die Lernenden.
- *Fragen*: Studierende haben die Möglichkeit Fragen mit Bezug auf die Aufgabe an die Dozierenden zu stellen, welche anschliessend durch diese manuell beantwortet werden können. Dabei wird der aktuelle Stand der Lösung zusammen mit der Frage an den/die Dozierende/n gesendet, damit diese/r Kenntnis über die vom Lernenden erarbeitete Lösung hat.

Nachfolgende Abbildung 3-2 veranschaulicht das Codeboard und dessen Grundfunktionalitäten. Es ist anzumerken, dass Tipps, Erklärungen für Compiler-Meldungen sowie die manuellen Fragen und Antworten über den Hilfe-Tab in der rechten Navigationsleiste

aufgerufen werden können. Die Resultate des Tests sind im Test-Tab ersichtlich. Im anschliessenden Abschnitt erfolgt eine Beschreibung der Buttons, welche sich in der oberen Navigationsleiste befinden.

Mittels diesem Button gelangen Studierende zurück zur Kursseite (Moodle).

Dieser Button bietet die Möglichkeit, den Code zu speichern, um die Aufgabe zu einem späteren Zeitpunkt fertigstellen zu können.

Durch diesen Button können Änderungen im Code widerrufen werden.

Durch diesen Button können Änderungen im Code wiederhergestellt werden.

Run Mittels dieses Buttons kann der Code kompiliert werden. Falls der Code bei der Ausführung syntaktische oder semantische Fehler aufweist, wird automatisch in den Hilfe-Tab gewechselt, um die Compiler-Meldung-Chatbox anzuzeigen.

Test Anhand dieses Buttons kann der Code getestet werden. Dabei wird automatisch in den Test-Tab gewechselt, in welchem die entsprechenden Tests durchgeführt werden.

Submit Mittels diesem Button können die Studierenden die Aufgabe einreichen, um diese als gelöst zu markieren. Dabei wird das Programm zuerst getestet und bei erfolgreicher Überprüfung an die/den Lehrende/n übermittelt.

Dieser Button ermöglicht, Anpassungen betreffend des Code-Editors vorzunehmen, wie beispielsweise die Schriftgrösse anzupassen.

Falls Studierende das Original (Ausgangslage) wiederherstellen möchten, können sie diesen Button betätigen.

Dieser Button bietet die Möglichkeit, in den Vollbildmodus zu wechseln.

Ich brauche Hilfe Bei einem Klick auf diesen Button wird der Hilfe-Tab angezeigt.

Abbildung 3-2: Codeboard – Ausgangslage

Die nachfolgenden zwei Kapitel befassen sich mit dem Test- und Hilfe-Tab, welche sich auf der rechten Seite der vorherigen Abbildung befinden.

3.1.2.1 Test-Tab

Das Ziel dieses Tabs ist, das Programm auf syntaktische oder semantische Fehler zu überprüfen sowie die Ausgabe des Programms zu testen. Dieser Tab, welcher in Abbildung 3-3 dargestellt ist, setzt sich aus folgenden Elementen zusammen:

- Statische Chatbox mit kurzer Beschreibung der Funktionalität des Tabs.
- Zum Bestehen erforderliche Tests (Überprüfung der Kompilierung sowie Input-/Outputtest).
- Button (grün), um die Lösung zu testen.
- Button (türkis), um in den Hilfe-Tab zu gelangen.

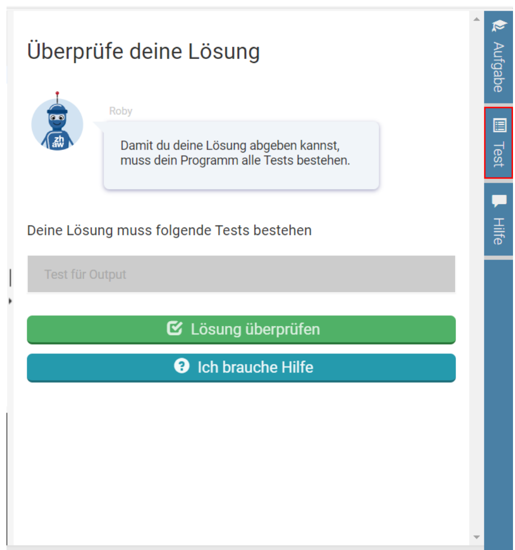


Abbildung 3-3: Codeboard – Test-Tab

Falls das Programm bei der Ausführung des Tests Fehler aufweist, wird der Test abgebrochen und das Helpersystem Compiler-Meldungen kommt zum Zug. Dabei wird eine Chatbox mit der entsprechenden Erklärung zum Fehler angezeigt. Diese ist zusätzlich mit einer Rating-Funktion versehen, um die angezeigte Erklärung bewerten zu können. Des Weiteren wird die ursprüngliche Fehlermeldung aus der Konsole unterhalb der Chatbox angezeigt. Die untenstehende Grafik veranschaulicht dieses Verhalten.

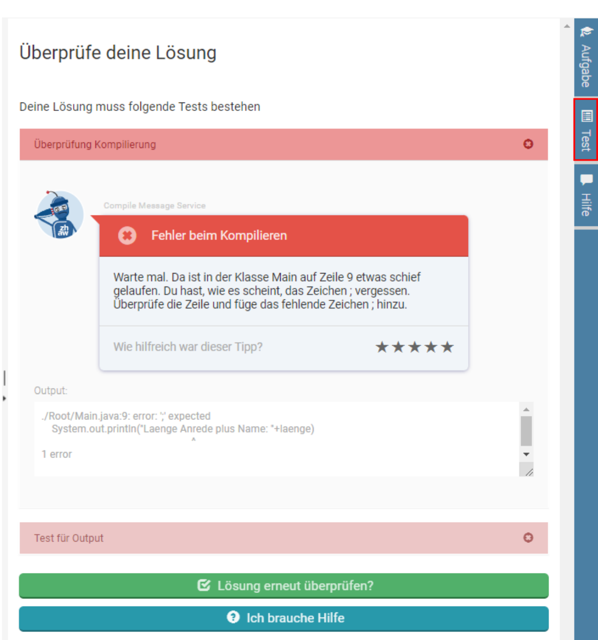


Abbildung 3-4: Codeboard – Test-Tab – Fehler beim Kompilieren

Falls der Code keine Fehler aufweist, gelangt der/die Nutzende zum nächsten Test, wobei das effektive Verhalten des Programms überprüft wird. Wenn dieser Test fehlschlägt, wird der Test abgebrochen und eine Chatbox mit einem Hinweis, abhängig vom effektiven Verhalten, angezeigt. Insofern das Programm während des Tests Eingaben über die Konsole erwartet, werden diese unterhalb der Chatbox (Input-Feld) dargestellt. Diese müssen ebenso vorgängig in der Konfigurationsdatei definiert werden. In Abbildung 3-5 ist das Ergebnis des Tests dargestellt, wenn das effektive Verhalten nicht mit dem erwarteten Verhalten übereinstimmt.

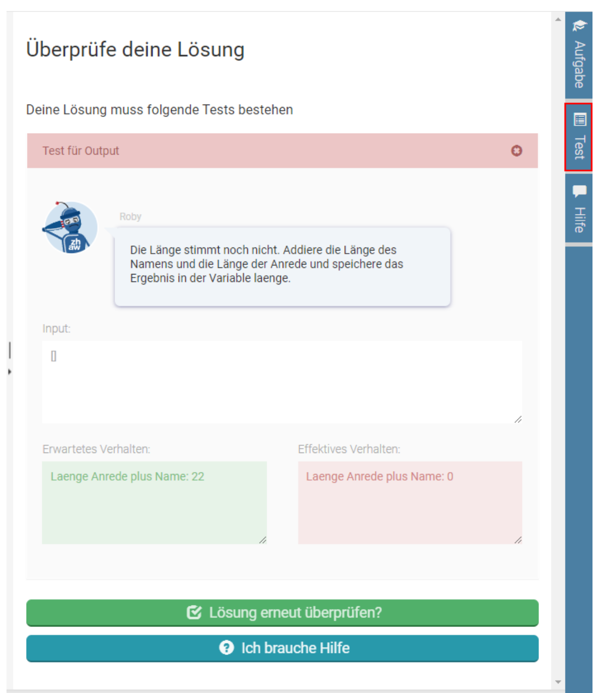


Abbildung 3-5: Codeboard – Test-Tab – Test der Ausgabe fehlgeschlagen

Falls auch dieser Test korrekt durchlaufen wird, erscheint eine entsprechende Meldung und die erarbeitete Lösung kann mittels Submit-Button abgegeben werden.

3.1.2.2 Hilfe-Tab

Neben dem Test-Tab gibt es zusätzlich einen Hilfe-Tab, welcher in der nachfolgenden Abbildung 3-6 dargestellt ist. Dieser umfasst die Helpersysteme Tipps, Compiler-Meldungen sowie manuelle Fragen und Antworten und besteht aus folgenden Komponenten:

- Statische Chatbox mit kurzer Beschreibung des Tabs.
- Chatboxen für *Tipps* (rot). Diese Chatboxen werden angezeigt, sobald der «Tipp anfordern» Button (grün) gedrückt wird.

- Chatbox für *Compiler-Meldungen* (orange). Diese werden angezeigt, sobald das Programm mittels Run-Button ausgeführt wird und der Code fehlerhaft ist. Falls sich Lernende bei der Ausführung des Programms nicht in diesem Tab befinden, wird dieser automatisch aufgerufen und die entsprechende Chatbox angezeigt.
- Eingabefeld für *manuelle Fragen*. Falls Studierende die Aufgabe trotz Tipps und Compiler-Meldungen nicht lösen können, haben sie die Möglichkeit, individuelle Fragen an die Dozierenden zu stellen. Nach erfolgreicher Übermittlung wird eine neue Chatbox angezeigt, welche die Frage beinhaltet. Wenn Lehrende die Frage beantworten, ist die Antwort ebenso in Form einer Chatbox in diesem Tab ersichtlich.

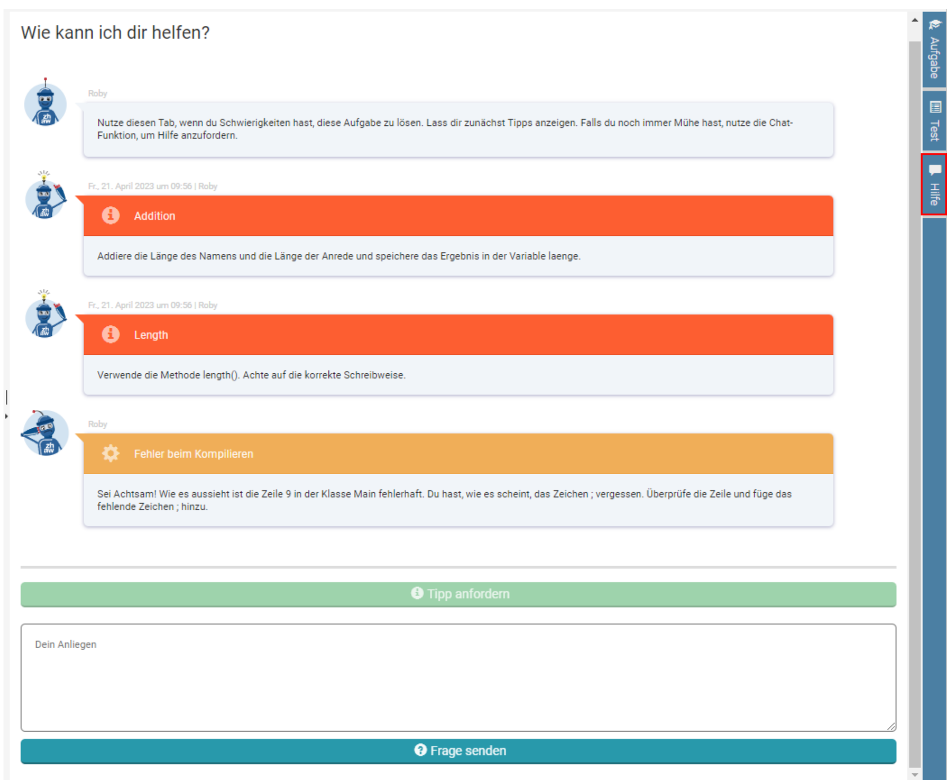


Abbildung 3-6: Codeboard – Hilfe-Tab

3.1.2.3 *Verbesserungspotenzial*

Eine wichtige Anpassung ist das Trennen des Zugriffs auf die einzelnen Helper-Systeme. In der aktuellen Version können die einzelnen Hilfestellungen Tipps, Compiler-Meldungen sowie manuelle Fragen und Antworten alle über den Hilfe-Tab aufgerufen werden. Dieser kann bei vielen Chatboxen schnell unübersichtlich werden. Aus diesem Grund ist zukünftig eine Trennung dieser einzelnen Tools geplant. Eine weitere Verbesserung betrifft die Ausgabe der Tipps, welche aktuell unabhängig vom aktuellen Stand der Lösung

angezeigt werden. Das bedeutet, dass Studierende Hinweise erhalten können, welche nicht relevant sind, da sie diese in ihrer Lösung bereits umgesetzt haben. Um dieser Problematik entgegenzuwirken, sollen Tipps in Zukunft abhängig vom Stand der aktuellen Lösung der Studierenden priorisiert und entsprechend ausgegeben werden. Des Weiteren werden Compiler-Meldungen nach der Korrektur des Codes nicht entfernt. Das hat zur Folge, dass bei mehrfacher Ausführung von fehlerhaftem Code dementsprechend viele Chatboxen angezeigt werden, was die Usability beeinträchtigt. Somit soll in Zukunft jeweils nur eine Meldung angezeigt werden, welche verschwindet, falls der Fehler behoben oder das Programm erneut ausgeführt wird.

3.1.3 Entwicklungsumgebung

Zur Weiterentwicklung der ursprünglichen Version musste das Codeboard lokal installiert werden. Die Zürcher Hochschule für Angewandte Wissenschaften (ZHAW) besitzt ein GitHub-Repository, auf welchem sich der Sourcecode der aktuellen Version des Codeboards befindet. Dieser Code diente als Grundlage für die Weiterentwicklung der Applikation in dieser Arbeit. Das Codeboard konnte lokal mittels eines Proxys über einen Test-Server gestartet werden. Dieser Test-Server wird an der ZHAW genutzt, um zusätzlich Anpassungen am Codeboard vor der Produktivschaltung zu testen.

3.2 Stand der Technik

In diesem Kapitel erfolgt eine Analyse und Beschreibung von vorhandenen Tools, welche ähnliche Funktionalitäten wie die umzusetzenden Artefakte aufweisen.

3.2.1 Integrated Development Environments

Mittels webbasierter IDEs können Studierende Programmieraufgaben direkt in einem Browser lösen, ohne eine Software auf ihrem Desktop installieren zu müssen (Gallego-Romero et al., 2020). Syntax-Hervorhebung, visuelle Debugger, Code-Vervollständigung oder Test-Tools sind weitere Funktionalitäten, welche IDEs aufweisen (Pears & Seidman, 2007, S. 210). Gemäss einer Studie von Gallego-Romero et al. (2020) verbringen Lernende, welche eine solche IDE nutzen, mehr Zeit mit dem Programmieren als Studierende, welche Desktop-Software nutzen. Des Weiteren erwähnen die Autoren, dass sich das Engagement der Studierenden bei der Nutzung einer solchen IDE erhöht. Eine mögliche Ursache für diese Erkenntnisse könnte sein, dass in Einführungskursen zur Programmierlehre das Einarbeiten in eine herkömmliche IDE für Studierende zu aufwendig ist (Pears & Seidman, 2007, S. 210). Des Weiteren erwähnen die Autoren, dass die

Komplexität einer professionellen IDE deren Vorteile für Novizen zunichtemacht. In den anschliessenden Abschnitten erfolgt einer Erläuterung von dem Codeboard ähnlichen webbasierten IDEs.

Replit: Ein sogenanntes Repl dient dabei als interaktive Programmierumgebung, um Code in Echtzeit zu erfassen und auszuführen (*Replit*, o. J.). Weil sich diese IDE in der Cloud befindet, ist der Zugriff auf diese Entwicklungsumgebung, welche über 50 Programmiersprachen unterstützt, mit jedem internetfähigen Device gewährleistet (*Replit*, o. J.). Ein weiteres Feature ist die sogenannte Multiplayer-Funktion. Dabei können mehrere Personen simultan am gleichen Projekt arbeiten (*Replit*, o. J.). Mit der Einführung des Features „Team for Education“ schaffe Replit einen kollaborativen Arbeitsbereich für Lehrende und Lernende (*Replit*, o. J.). Dabei können Dozierende Aufgaben und Projekte einrichten, welche mit Fälligkeitsdatum, Startcode, Materialien sowie Anweisungen versehen sind (*Replit*, o. J.). Die von Studierenden gelösten Aufgaben lassen sich im Anschluss einerseits mittels Input-/Outputtests überprüfen, wobei die effektive mit der erwartenden Ausgabe verglichen wird (*Replit*, o. J.). Andererseits können Lehrende selber Unit-Tests in Java (JUnits), JavaScript (Jest) oder Python (unittest) schreiben, um die Aufgaben automatisiert bewerten zu können (*Replit*, o. J.).

Codeanywhere: Dabei handelt es sich um eine plattformübergreifende Cloud-IDE, die dieselben Funktionen wie eine herkömmliche Desktop-IDE aufweist (*Codeanywhere*, o. J.). Es werden über 120 Programmiersprachen und alle internetfähigen Geräte sowie Browser unterstützt (*Codeanywhere*, o. J.). Des Weiteren lässt sich eine Vorschau der entwickelten Applikation direkt in der Entwicklungsumgebung generieren (*Codeanywhere*, o. J.). Codeanywhere ermöglicht Lehrenden die Erstellung von Bibliotheken mit verschiedenen Entwicklungsumgebungen, welche auf bestimmte Vorlesungen oder Klassen zugeschnitten sind (*Codeanywhere*, o. J.).

Coding Rooms: Diese browserbasierte Entwicklungsumgebung ermöglicht Unterricht in Echtzeit, integrierte Videokonferenzen sowie eine automatische Bewertung von Aufgaben (*Coding Rooms*, o. J.). Zusätzlich lässt sich das Wissen der Lernenden mittels Drag & Drop, Multiple Choice oder Code-Auswahl Aufgaben überprüfen (*Coding Rooms*, o. J.). Es können Kurse erstellt werden, welchen eine beliebige Anzahl Studierende hinzugefügt werden kann. Diese Kurse können um Lektionen ergänzt werden, welche diverse interaktive und nicht-interaktive Elemente (wie beispielsweise Multiple-Choice-Fragen) beinhalten (*Coding Rooms - Help*, o. J.). Zusätzlich besteht die Möglichkeit, das

Wissen der Lernenden mittels Leistungsnachweisen zu überprüfen, welche im Anschluss mittels Input-/Output-Vergleichen, Bash-Skript-Tests oder Unit-Tests überprüft und automatisch ausgewertet werden können (*Coding Rooms - Help*, o. J.).

3.2.2 Coding-Assistant

Seit Jahren rücken Aufgaben, wie Fehlervorhersagen, Codegenerierung oder -zusammenfassung in den Vordergrund der Forschung (Zeng et al., 2022, S. 39). Da das Erklären von Programmcode das grundlegende Ziel des Coding-Assistant ist, liegt der Fokus in diesem Kapitel auf Technologien, welche ebenso darauf ausgelegt sind. Eine Vielzahl der vorhandenen Tools verfolgt ein ähnliches Ziel, indem sie eine zusammenfassende Beschreibung der Funktionalität des Codes erzeugen (*Code GPT*, o. J.; *GitHub Copilot*, o. J.; Feng et al., 2020; Iyer et al., 2016; Kuang et al., 2022; Zhang et al., 2023). In den folgenden Abschnitten werden zwei Modelle näher erläutert.

CodeGPT: Bei CodeGPT handelt es sich um eine kostenpflichtige Erweiterung für die Desktop-IDE Visual Studio Code (*Code GPT*, o. J.). Eine Funktionalität dieser Erweiterung ist die Generierung von Erklärungen zu ausgewähltem Code in Textform (*Code GPT*, o. J.). Es ist anzumerken, dass der Code dabei nicht zeilenweise, sondern vielmehr in Form einer Zusammenfassung der Funktionalität erläutert wird (*Code GPT*, o. J.). Zusätzlich kann Code ausgewählt werden, um diesen auf Fehler oder anderweitige Probleme zu überprüfen (*Code GPT*, o. J.).

CodeBERT: Dieses Modell ermöglicht die Generierung von Code-Dokumentationen in natürlicher Sprache (Feng et al., 2020, S. 1536). Die Erklärungen erfolgen analog *CodeGPT* ebenfalls nicht zeilenweise, sondern in Form einer Zusammenfassung (Feng et al., 2020). Bei diesem Tool handelt es sich um das erste natürlich- und programmiersprachlich vortrainierte Modell, welches Programmiersprachen wie Java, JavaScript, oder Python unterstützt (Feng et al., 2020).

3.2.3 Compiler-Meldungen

Das Verstehen von Compiler-Fehlermeldungen stellt für viele Programmieranfängerinnen und -anfänger eine Herausforderung dar (Becker et al., 2019, 2021; Denny et al., 2021; Ettles et al., 2018). Eine solche Fehlermeldung wird als Reaktion auf einen anormalen Zustand während der Ausführung des Codes geworfen oder ausgelöst (Assiri & Elazhary, 2020, S. 628). Dabei wird zwischen semantischen (wie beispielsweise unbekannten Variablen) oder syntaktischen Fehlern (wie fehlerhafte Schreibweisen)

unterschieden (Keuning et al., 2019, S. 8). Tools, welche solche Fehlermeldungen in natürlicher Sprache darstellen, haben eine positive Auswirkung auf die Lernerfahrung von Lernenden (Assiri & Elazhary, 2020; Becker et al., 2016). In den kommenden Abschnitten werden Systeme erläutert, welche Compiler-Meldungen in natürlicher Sprache erklären können.

JENL: Assiri und Elazhary (2020) erläutern ein Tool, welches Exception-Meldungen in der Programmiersprache Java in verständliche Anweisungen in natürlicher Sprache konvertiert. Die Information an den Nutzenden setzt sich aus vordefinierten Texten und aus den in den Compiler-Meldungen eingebetteten Informationen zusammen (Assiri & Elazhary, 2020, S. 632). Der Ort des Fehlers im Code, der Typ der Exception oder der Grund für den Fehler sind einige dieser Informationen, welche der Compiler-Meldung entnommen werden können (Assiri & Elazhary, 2020, S. 632f.).

Decaf: Dieser Java-Editor nutzt vorhandene Informationen wie die erzeugte Compiler-Fehlermeldung sowie die fehlerhafte Codezeile, um Lernenden verbesserte Fehlermeldungen anzeigen zu können (Becker et al., 2016, S. 149). In ihrer Studie haben Becker et al. 30 Fehlermeldungen verbessert. Dabei wurden Fehlermeldungen, welche in der Praxis häufig vorkommen, sowie Empfehlungen aus der Literatur berücksichtigt (Becker et al., 2016, S. 154f.).

3.2.4 Hinweissystem

In letzter Zeit wurden vermehrt Technologien vorgestellt, welche Studierende beim Lösen von Programmieraufgaben durch die Generierung von Hinweisen unterstützen (McBroom et al., 2021). Nachfolgend werden zwei dieser Tools näher erläutert.

AskElle: Dieser Tutor bietet Unterstützung bei der schrittweisen Entwicklung von Programmieraufgaben in der Programmiersprache Haskell (Gerdes et al., 2017, S. 2). Die Grundlage für die Generierung von Feedback stellen Musterlösungen dar, welche von Dozierenden erstellt werden. Aus diesen lassen sich Strategien ableiten, welche für die Generierung von Hinweisen und die Erfassung des Standes der Lösung der Studierenden verwendet werden (Gerdes et al., 2017). Lernende können das Tool verwenden, um einerseits zu prüfen, ob die Anpassungen im Code sie näher zur finalen Lösung bringen (Gerdes et al., 2017, S. 5). Andererseits können Hinweise zum aktuellen Stand der Lösung abgefragt werden, falls Studierende Schwierigkeiten mit der Umsetzung der Aufgabe haben (Gerdes et al., 2017, S. 5).

AutoTeach: Dieses Tool ermöglicht die Erzeugung von zunehmend detaillierteren Hinweisen (Meyer et al., 2017, S. 42). Dabei werden Studierenden bei Bedarf nach und nach Lösungsschritte zur finalen Lösung angezeigt (McBroom et al., 2021). Es kommen textuelle Hinweise zum Einsatz, welche von den Dozierenden erfasst werden müssen, oder reduzierte Versionen der finalen Lösung, in denen Teile des Codes ausgeblendet sind (Antonucci, 2014, S. 11). Die Hinweise werden dabei unabhängig vom Stand der Lösung der Studierenden ausgegeben (Keuning et al., 2019, S. 19). Es ist anzumerken, dass AutoTeach auf die Programmiersprache Eiffel ausgelegt ist (Antonucci, 2014, S. 11).

3.3 Vergleich mit vorhandenen Lösungen

Grundsätzlich darf gerade die Verwendung von künstlicher Intelligenz im Zusammenhang mit dem Codeboard nicht ausgeblendet werden. Wie die letzten Abschnitte aufzeigten, existiert bereits eine Vielzahl von Tools, welche ähnliche Ziele wie die aktuell genutzten Helpersysteme verfolgen. Viele Studien (Feng et al., 2020; Iyer et al., 2016; Keuning et al., 2019; Lin et al., 2022; Lu et al., 2021; Zeng et al., 2022) zeigen auf, dass solche Modelle bereits gute Ergebnisse betreffend ihres Anwendungszweckes erzielen. Nichtsdestotrotz zeigen diese Studien allerdings auch, dass solche Modelle noch weiter verbessert werden müssen, um optimale Ergebnisse betreffend ihres Anwendungszwecks erzielen zu können. Beispielsweise erwähnen Keuning et al. (2019), dass solche Tools nicht optimal auf die Bedürfnisse der Lehrenden abgestimmt werden können. Des Weiteren erwähnen sie, dass die Erstellung von neuen Übungen durch die Erfordernis von grossen Datensätzen verkompliziert wird. Price et al. (2019) erwähnen, dass beispielsweise die Qualität von datenbasierten Hinweisen schwindet, wenn der Code der Studierenden zu fest von den üblich erzielten Lösungen abweicht. Ebenso erwähnen sie, dass den Algorithmen die Priorisierung der wichtigsten Hinweise teilweise schwerfällt. Eine Studie von Zeng et al. (2022), welche die Performance von diversen Modellen evaluierte, kam zum Schluss, dass die Wahl eines idealen Modells herausfordernd ist. Dies wird durch die beobachteten Leistungsschwankungen von verschiedenen vortrainierten Modellen bei unterschiedlichen Datensätzen und Aufgaben begründet. Diese Gründe tragen massgebend zur Wahl der genutzten Helpersysteme im Codeboard bei. Alle bereits genutzten Tools wie auch der Coding-Assistant können nach Belieben angepasst und erweitert werden. Daraus ergibt sich die Möglichkeit, das genaue Verhalten dieser Systeme zu kontrollieren. Dies stellt sicherlich einer der Hauptgründe dar, weshalb diese Systeme auch in Zukunft genutzt werden.

Zusammenfassend hat diese Analyse ergeben, dass aktuell kein System die derzeit genutzten Helpersysteme ablösen könnte. Beispielsweise folgen die Tools *JENL* und *Decaf* grundsätzlich demselben Prinzip betreffend der Erzeugung von Erklärungen für Fehlermeldungen wie das im Codeboard verwendete Helpersystem *Compiler-Meldungen*. Zusätzlich wurde kein System gefunden, welches wie der *Coding-Assistant* Code-Zeilen zeilenweise erklären kann. *CodeGPT* und *CodeBERT* sowie andere identifizierte Tools geben, wie bereits erwähnt, Erklärungen in Form einer Zusammenfassung zurück. Dies verfehlt das grundlegende Ziel, Studierenden die genaue Bedeutung einer Code-Zeile zu veranschaulichen. Zusätzlich sind die bereits implementierten Helpersysteme bereits seit einer geraumen Zeit im Einsatz und erfüllen alle ihren Anwendungszweck. Dennoch sei an dieser Stelle angemerkt, dass eine allfällige Verwendung von künstlicher Intelligenz nicht unberücksichtigt werden darf und in Zukunft erneut evaluiert werden müsste.

4 Design

Dieses Kapitel hat zum Ziel, das Zusammenspiel der einzelnen Helpersysteme und die Überlegungen zur finalen Lösung zu erläutern. Dafür werden Abhängigkeiten zwischen den Helpersystemen aufgezeigt und Möglichkeiten erarbeitet, wie diese am besten aufeinander abgestimmt oder miteinander kombiniert werden können. Des Weiteren werden Mockups erstellt, welche wiederum Teil dieses Kapitels sind.

4.1 Zusammenspiel der einzelnen Helpersysteme

Obschon die einzelnen Helpersysteme unterschiedliche Funktionalitäten aufweisen, gibt es dennoch gewisse Abhängigkeiten, welche zusammen mit den Überlegungen zur finalen Lösung im folgenden Kapitel erläutert werden. Die Optimierung des Zusammenspiels dieser einzelnen Komponente wird in Kapitel 4.1.2 behandelt. Es ist anzumerken, dass diese Überlegungen zusammen mit den in Kapitel 3.1 erläuterten Verbesserungsmöglichkeiten in die Anforderungserhebung einfliessen.

4.1.1 Überlegungen zum Gesamtsystem

Grundsätzlich können die in Kapitel 3.1.2 beschriebenen Funktionalitäten der einzelnen Systeme beibehalten werden. Da die bereits vorhandenen Helpersysteme (Compiler-Meldungen, Tipps sowie manuelle Fragen und Antworten) unabhängig voneinander genutzt werden können und grundsätzlich keine gegenseitigen Abhängigkeiten aufweisen, lassen sich deren Funktionalitäten trennen. Von hoher Relevanz ist jedoch das Helpersystem Test, welches Abhängigkeiten zum System Compiler-Meldungen aufweist. Wie bereits in Abbildung 3-4 visualisiert, nutzt es dieses Tool, falls der Code zum Zeitpunkt des Tests Fehler aufweist, um Erklärungen für Kompilierfehler im Test-Tab anzeigen zu lassen. Die Handhabung für einen solchen Fall sowie die Trennung der einzelnen Systeme wird im folgenden Kapitel geschildert.

Eine weitere wichtige Überlegung betrifft die Integration des Coding-Assistant. Da viele Elemente im Codeboard bereits vorgegeben sind, besteht beispielsweise aufgrund des vorgegebenen Layouts keine Möglichkeit, ein neues Fenster, welches die Code-Erklärungen beinhaltet, einzufügen. Vielmehr steht die Verwendung von Tabs im Vordergrund, um auf die einzelnen Systeme zugreifen zu können. Aus diesem Grund muss die ursprüngliche Version des Coding-Assistant grundlegend angepasst werden, um die Konsistenz des Codeboards nicht zu verletzen. Die drei Grundfunktionalitäten Code-

Erklärungen, Gültigkeitsbereiche von Variablen sowie die Visualisierung von Code-Blöcken sollen voneinander losgelöst genutzt werden können. Das bedeutet, dass diese Features bei Bedarf beispielsweise aktiviert oder deaktiviert werden können. Ein Weiteres Feature, welches sich betreffend des Coding-Assistant ergibt, ist das Formatieren von Code im Editor. Diese Funktionalität soll implementiert werden, da die Visualisierungen der Code-Blöcke für unformatierten Code allenfalls nicht korrekt dargestellt werden können. Aus diesem Grund ist geplant, bei der Anzeige der Code-Blöcke den Code vorab zu formatieren. Des Weiteren muss ein optimales Zusammenspiel zwischen den Code-Erklärungen sowie dem Helpersystem Compiler-Meldungen gewährleistet sein. Obschon die beiden Tools getrennt voneinander agieren können, verfolgen diese teilweise denselben Anwendungszweck. Beispielsweise kann der Coding-Assistant Studierende auf Fehler im Code aufmerksam machen, ohne dass das Programm zuerst kompiliert werden muss.

4.1.2 Optimierung des Zusammenspiels

Um die Effektivität dieser teils miteinander verbunden Systeme zu maximieren, muss eine optimale Benutzerfreundlichkeit gewährleistet sein. Daraus ergibt sich die Notwendigkeit die Benutzeroberfläche grundlegend anzupassen und den Zugriff auf die einzelnen Helpersysteme in Zukunft zu trennen. Dementsprechend soll jedes System über einen eigenen Tab aufgerufen werden können. In diesem Zusammenhang sei noch einmal auf das Zusammenspiel zwischen dem Test-System und den Compiler-Meldungen aufmerksam gemacht. Falls der Code bei der Ausführung des Tests fehlerhaft ist, soll automatisch in den Tab für Compiler-Meldungen gewechselt werden, in dem die entsprechende Erklärung angezeigt wird. Somit ist sichergestellt, dass jedes Helpersystem seinen eigenen Anwendungszweck aufweist. Zusätzlich ermöglicht eine solche Trennung eine simple Deaktivierung der einzelnen Systeme. Falls für bestimmte Aufgaben gewisse Helpersysteme deaktiviert werden sollen, können deren Tabs deaktiviert werden, um den Zugriff zu unterbinden. Zur Unterstützung der Studierenden soll zusätzlich ein Info-Tab implementiert werden, welcher die einzelnen Helpersysteme erläutert.

Betreffend des Coding-Assistant sollen im dazugehörigen Tab die Code-Erklärungen angezeigt werden. Dieser Tab soll nur die Erklärungen beinhalten, da diese die wichtigste Funktion dieses Tools darstellen. Der Zugriff auf die zusätzlichen Funktionalitäten, den Gültigkeitsbereich von Variablen sowie die Visualisierung von Code-Blöcken, soll mittels Buttons möglich sein. Beim Zusammenspiel zwischen dem Coding-Assistant und den

Compiler-Meldungen erschliesst sich das Erfordernis, Studierende während dem Lösen der Aufgabe dynamisch auf Fehler im Code in Form von Erklärungen hinzuweisen. Somit kann sichergestellt werden, dass Nutzende den Code bereits vor der Ausführung des Programms verbessern können. Dies könnte zu einer effizienteren Aufgabenbearbeitung führen, da das Kompilieren der Aufgaben durchaus ein paar Sekunden in Anspruch nimmt. Dennoch gilt anzumerken, dass das grundlegende Ziel des Coding-Assistant das Erklären von fehlerfreiem Code ist. Somit sind die Compiler-Meldungen weiterhin von hoher Relevanz.

4.2 Mockups

In den nächsten fünf Kapiteln erfolgt eine Beschreibung der Mockups, welche im Verlaufe des Projekts für die einzelnen Helpersysteme erstellt wurden. Diese ergeben sich aus der Erhebung der Anforderungen in den Kapiteln 5, 6, 7 und 8, dem iterativen Feedback zu bereits erstellten Mockups sowie den im Kapitel 4.1 beschriebenen Überlegungen zum Gesamtsystem. Des Weiteren werden für jedes Mockup die aus der Besprechung mit dem Betreuer getroffenen Verbesserungen erläutert.

4.2.1 Codeboard

Die Abbildungen in diesem Kapitel sollen die grundlegenden geplanten Anpassungen im Codeboard visualisieren. Diese Visualisierungen stellen den Ausgangspunkt für die nachfolgenden Mockups dar, da dargestellt wird, wie die Helpersysteme und deren zusätzliche Funktionalitäten aufgerufen werden können.

4.2.1.1 Tabs für den Zugriff auf die einzelnen Helpersysteme

Die einzelnen Helpersysteme sollen neu alle über einen eigenen Tab in der rechten Navigationsleiste des Codeboards aufgerufen werden können. In der nachfolgenden Abbildung 4-1 ist die neu geplante Navigationsleiste dargestellt, welche sich auf der rechten Seite in der Grafik befindet. Dabei sind die neu hinzugefügten Tabs darin zur Verdeutlichung rot markiert.

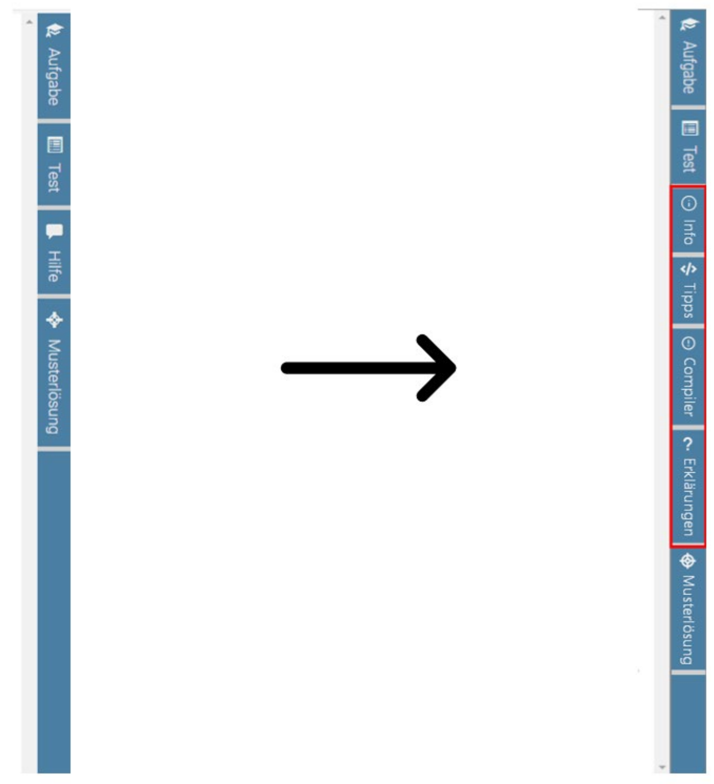


Abbildung 4-1: Navigationsleiste rechts (alte und neue Version)

Diesen Tabs liegen folgende Funktionalitäten zugrunde.

- *Info-Tab*: Dieser Tab dient der Beschreibung der einzelnen Hilfesysteme. Dabei wird der Anwendungszweck der einzelnen Systeme erläutert, um damit die Wahl des richtigen Tabs zu unterstützen.
- *Tipps*: In diesem Tab werden die einzelnen Tipps sowie das Feld, um manuelle Fragen an Dozierende zu stellen, dargestellt. Des Weiteren sind die Antworten der Dozierenden auf die Fragen Bestandteil dieses Tabs.
- *Compiler*: Über diesen Tab kann auf das Helpersystem Compiler-Meldungen zugegriffen werden, welches Erklärungen zu vorhandenem syntaktischem oder semantischem Fehlern im Code erzeugt.
- *Erklärungen*: Dieser Tab beinhaltet den Coding-Assistant und somit die Erklärungen zu jeder geschriebenen Code-Zeile in Textform. Eine entsprechende Visualisierung ist in Abbildung 4-1 zu finden. Der Zugriff auf die zusätzlichen Funktionalitäten dieses Tools wird im anschliessenden Kapitel 4.2.1.2 erläutert.

4.2.1.2 Buttons für den Aufruf von weiteren Funktionalitäten

Nachfolgende Abbildungen zeigen die Anpassungen in der oberen Navigationsleiste. Die erste Grafik zeigt die Ausgangslage und die zweite die angepasste, neue Version. Die neu hinzugefügten Elemente sind darin rot markiert.

Abbildung 4-2: Navigationsleiste oben (alte Version)

Abbildung 4-3: Navigationsleiste oben (neue Version)

Den neuen, blauen Buttons in der oberen Navigationsleiste liegen folgende Funktionalitäten zugrunde.

- *Gültigkeitsbereich*: Über diesen Button soll der Gültigkeitsbereich von Variablen (vgl. Kapitel 4.2.3) ein- und ausgeblendet werden können.
- *Code-Blöcke*: Dieser Button hat zum Zweck, die Visualisierung von Code-Blöcken (vgl. Kapitel 4.2.3) im Code-Editor ein- und auszublenden.
- *beautify*: Der Code im Editor soll mittels dieses Buttons formatiert werden können.

Die Buttons werden in der oberen Navigationsleiste positioniert, um Studierenden einen schnellen Zugriff auf diese Funktionen zu ermöglichen.

4.2.2 Helpersystem – Coding-Assistant (Erklärungen von Code)

Die nachfolgende Abbildung stellt das Mockup für den Coding-Assistant dar. Es ist anzumerken, dass einige aus der Besprechung mit dem Betreuer gewonnenen Verbesserungsmöglichkeiten, welche in Kapitel 4.2.2.2 erläutert werden, bereits in den anschliessend beschriebenen Mockups umgesetzt wurden.

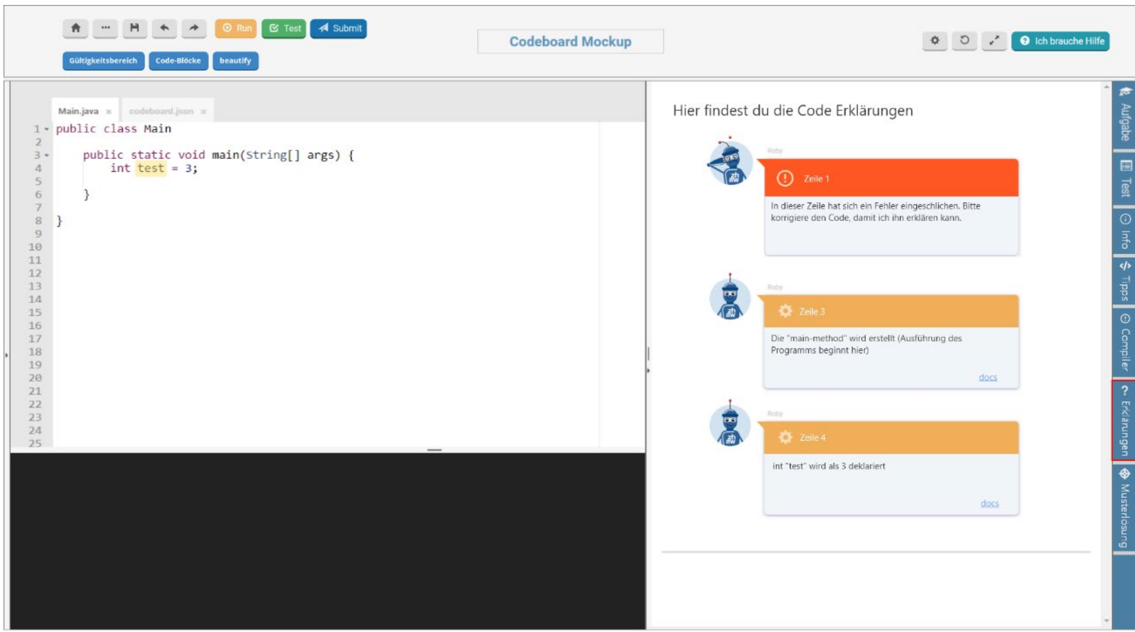


Abbildung 4-4: Mockup – Coding-Assistant (Erklärungen)

In den nachfolgenden Kapiteln erfolgt eine umfassende Beschreibung des Mockups sowie der aus der Besprechung gewonnenen Verbesserungsmöglichkeiten.

4.2.2.1 Tab für Code-Erklärungen

In diesem Tab soll dynamisch für jede geschriebene Code-Zeile eine neue Chatbox mit der entsprechenden Code-Erklärung erzeugt werden. Für die Darstellung der Erklärungen werden Chatboxen verwendet, da diese das grundlegende Element für alle Helpersysteme darstellen. Falls der dazugehörige Code gelöscht, oder angepasst wird, soll die Chatbox entfernt werden oder sich an die Änderung adaptieren. Dabei wird zwischen zwei Arten von Chatboxen unterschieden:

- *Erklärungen für fehlerhaften Code*: Falls der Code fehlerhaft ist, soll in der dazugehörigen Chatbox darauf aufmerksam gemacht werden. Diese Chatbox soll sich von den anderen visuell unterscheiden, um zu verdeutlichen, dass die dazugehörige Code-Zeile Fehler aufweist. Einerseits wird dabei ein anderes Bild des Avatars (Roby) gewählt, andererseits das Icon in der Kopfzeile angepasst und die Chatbox mit einer roten Kopfzeile versehen.
- *Erklärungen für korrekten Code*: Für alle Code-Zeilen ohne Fehler soll der Programmcode in der dazugehörigen Chatbox in Textform erklärt werden. Zusätzlich wird die Chatbox mit einem Link „Docs“ ergänzt, welcher auf weiterführende Dokumentationen im Internet verweist. Für diese Art von Erklärungen wird

wiederum ein anderes Bild von Roby, ein anderes Icon sowie eine andere Farbe für die Kopfzeile gewählt.

Die Chatboxen setzten sich aus den folgenden grundlegenden Komponenten zusammen:

- *Avatar (Roby)*: Links von jeder Chatbox findet sich ein Bild von Roby, welches von der Art der Chatbox abhängig ist.
- *Titel*: Als Titel wird aktuell das Wort "Roby" oberhalb der Chatbox angezeigt.
- *Kopfzeile*: Das Icon sowie die Farbe der Kopfzeile hängt von der Art der Chatbox ab. Zusätzlich soll die dazugehörige Zeilennummer des Codes im Editor angegeben werden, welche sich dynamisch ändert, sobald der Code angepasst wird.
- *Erklärung*: Unter der Kopfzeile wird der dazugehörige Code in Textform erklärt. Falls der Code fehlerhaft ist, hat die Erklärung zum Zweck, Lernende auf den Fehler aufmerksam zu machen.
- *Docs*: Über diesen Link kann auf weiterführende Dokumentationen zur entsprechenden Code-Zeile zugegriffen werden.

4.2.2.2 Verbesserungen

In diesem Kapitel werden die aus der Besprechung mit dem Betreuer gewonnenen Verbesserungsmöglichkeiten in Form einer Aufzählung erläutert. Diese Erkenntnisse werden bei Bedarf wiederum in neue Anforderungen umgesetzt, welche sich im Kapitel 6.2.1 finden.

- *Anpassung des Titels der Chatbox*: Die zur Erklärung dazugehörige Zeilennummer soll als Titel der Chatbox anstelle von "Roby" gewählt werden. Dabei wäre ein Satz wie: "Roby erklärt Zeile x" sinnvoll.
- *Grösse der Chatboxen*: Die Chatboxen sollen kleiner dargestellt werden. Die Kopfzeile mit der Zeilennummer und dem Icon kann weggelassen werden. Dies gilt auch für Fehler-Chatboxen.
- *Chatboxen für* Fehler: Die Chatboxen für Fehler sollen durch eine andere Farbe der gesamten Chatbox und einem roten Schatten von der anderen Chatbox unterscheidbar sein.
- *Anzeige von Syntax-Fehler*: Falls eine Code-Zeile syntaktische oder semantische Fehler aufweist, soll die zur Erklärung dazugehörige Code-Zeile im Editor mit einem roten Marker markiert werden.

- *Darstellung der Buttons für erweiterte Funktionalitäten*: Die drei blauen Buttons in der oberen Navigationsleiste sollen mit Symbolen anstelle von Text versehen werden. Zusätzlich soll im Info-Tab unter der Beschreibung der einzelnen Helpersysteme auf die dazugehörigen Buttons verwiesen werden.

4.2.3 Helpersystem – Coding-Assistant (Code-Blöcke & Gültigkeitsbereiche)

Die Visualisierung der Code-Blöcke und der Gültigkeitsbereiche von Variablen stellt eine weitere Anforderung an die Integration des Coding-Assistant in das Codeboard dar. Das dazugehörige Mockup ist in Abbildung 4-5 zu sehen, worin die Buttons, um diese Funktionalitäten aufzurufen, rot markiert sind.

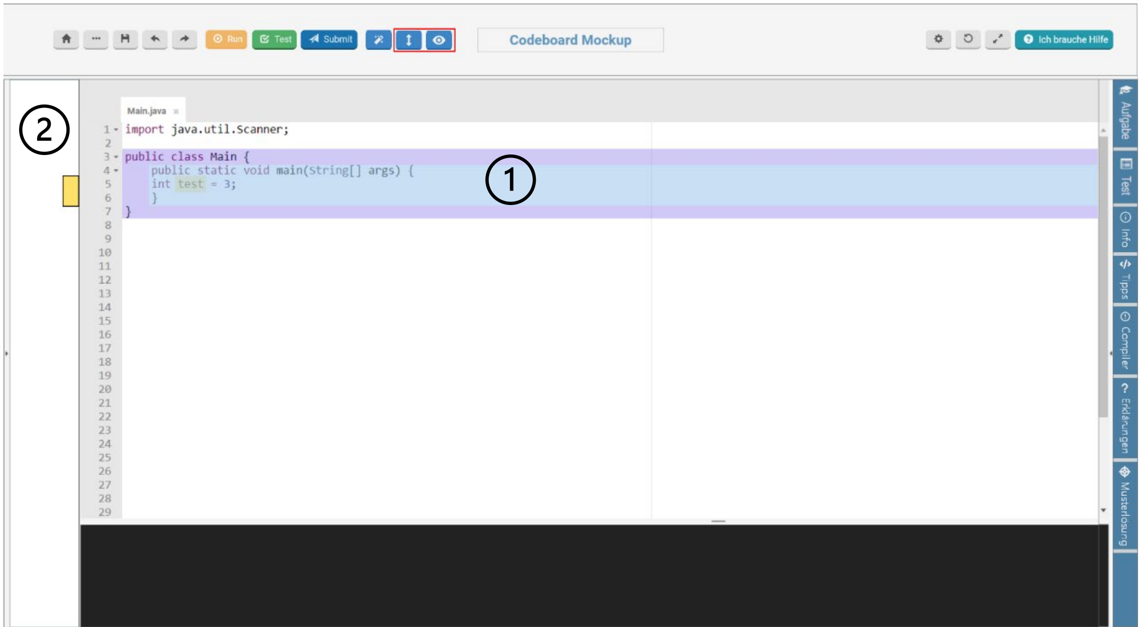


Abbildung 4-5: Mockup – Coding-Assistant (Code-Blöcke und Gültigkeitsbereiche)

Die folgenden Kapitel beinhalten eine Beschreibung des Mockups sowie die aus dem Austausch mit dem Auftraggeber gewonnenen Verbesserungspotenziale.

4.2.3.1 Darstellung der Buttons für erweiterte Funktionalitäten

Die Buttons in der oberen Navigationsleiste betreffend beautify (links), Visualisierung von Gültigkeitsbereichen (mittig) und Code-Blöcken (rechts) bestehen nur noch aus einem Symbol ohne dazugehörigen Text.

4.2.3.2 Visualisierung des Gültigkeitsbereichs von Variablen

Das Fenster (Nummer 2 in Abbildung 4-5), welches die Visualisierungen der Gültigkeitsbereiche beinhaltet, soll sich links vom Code-Editor befinden. Dabei soll für jede neu

deklarierte Variable ein Balken angezeigt werden, der dieselbe Farbe hat wie die Markierung für den Variablennamen. Dieser Balken visualisiert den Gültigkeitsbereich der dazugehörigen Variable. Dieses Fenster kann bei Bedarf mittels des Buttons (mittig) in der oberen Navigationsleiste ein- und ausgeblendet werden. Diese Positionierung wird gewählt, da der Code im Editor links ausgerichtet und somit besser ersichtlich ist, welcher Balken zu welcher Variable gehört.

4.2.3.3 Visualisierung der Code-Blöcke

Wie in Abbildung 3-1 ersichtlich, werden Code-Blöcke in der ursprünglichen Version des Coding-Assistant direkt im Erklärungsfenster angezeigt. Da in der neuen Version des Codeboards ein solches Fenster nicht mehr vorhanden ist, sollen Code-Blöcke direkt im Code-Editor visualisiert werden. Dabei werden analog zu den Markierungen von Variablennamen Marker verwendet, welche diese Blöcke farblich hervorheben. Diese Funktionalität kann ebenso mittels eines Buttons (rechts) in der oberen Navigationsleiste ein- und ausgeblendet werden.

4.2.3.4 Verbesserungen

Es folgt eine Auflistung der Verbesserungen, wobei diese bei Bedarf in neue Anforderungen umgesetzt werden, welche sich in den Kapitel 6.3.1 respektive 6.4.1 finden.

- *Gestaltung „beautify“ Button*: Der Button soll neu analog zum Run- und Test-Button aus einem Symbol und dazugehörigem Text bestehen.
- *Trennung der Funktionalitäten*: Die Visualisierungen von Code-Blöcken oder Gültigkeitsbereichen sollen nicht gleichzeitig angezeigt werden können. Falls ein Button gedrückt wird, soll der andere Button deaktiviert werden. Durch eine solche Trennung wird gewährleistet, dass es keine Überschneidungen der Visualisierungen gibt und diese die Usability nicht negativ beeinflussen.
- *Markierung von Variablennamen*: Variablennamen sollen nur bei Einblendung des Gültigkeitsbereichs mit einer Farbe markiert werden.

4.2.4 Helpersystem – Compiler-Meldungen

Das Helpersystem Compiler-Meldungen soll ebenfalls über einen eigenen Tab aufgerufen werden können. Die folgende Darstellung veranschaulicht, wie dieser Tab aussehen könnte.

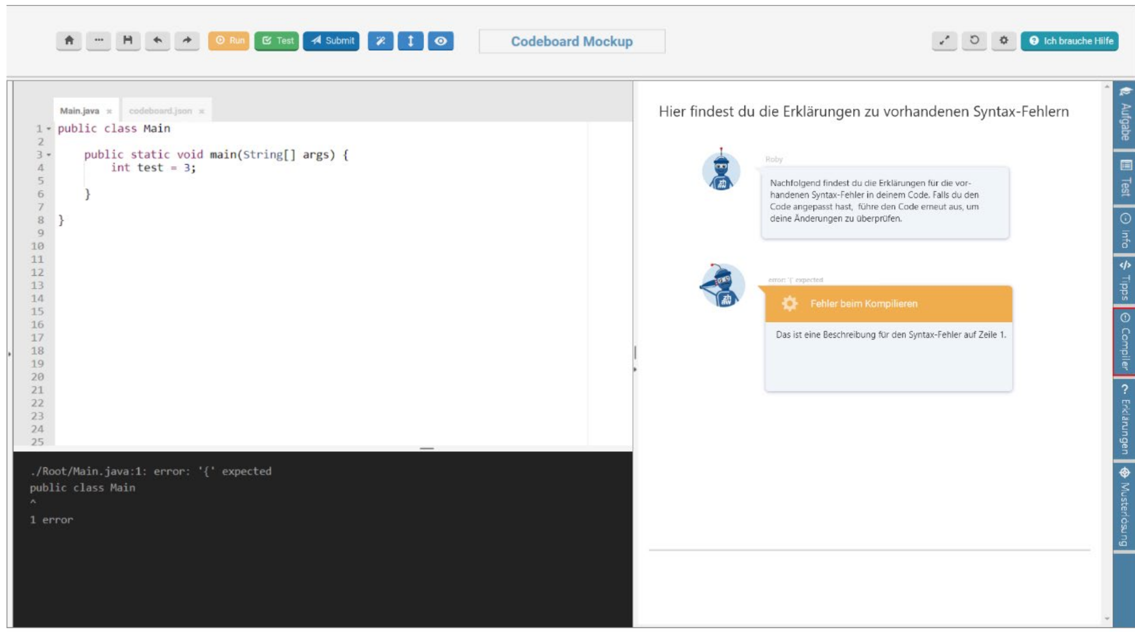


Abbildung 4-6: Mockup – Compiler-Meldungen

Die nachfolgenden Kapitel befassen sich mit der Erklärung des Mockups sowie der aus der Besprechung gewonnenen Erkenntnisse.

4.2.4.1 Tab für Compiler-Meldungen

Der Tab für Compiler-Meldungen setzt sich aus zwei Elementen zusammen: einer statischen Chatbox, welche eine kurze Beschreibung des Helpersystems beinhaltet, und einer dynamischen Chatbox (orange), welche angezeigt wird, falls der Code ausgeführt wird und Fehler aufweist. Es ist wichtig anzumerken, dass automatisch in diesen Tab gewechselt wird, sobald der Code mittels Run-Button ausgeführt wird und Fehler aufweist. Die dynamisch erzeugte Chatbox setzt sich aus den folgenden Komponenten zusammen:

- *Avatar (Roby)*: Links von jeder Chatbox findet sich ein Bild von Roby. Dieser Avatar ist für alle Erklärungen in diesem Tab identisch.
- *Titel*: Als Titel oberhalb der Chatbox, soll die Fehlermeldung aus der Konsole angezeigt werden.
- *Kopfzeile*: Die Kopfzeile setzt sich aus einem Icon und dem Text „Fehler beim Kompilieren“ zusammen.
- *Erklärung des Syntax-Fehlers*: Unter der Kopfzeile wird der Fehler im Code in Textform erklärt.

4.2.4.2 Verbesserungen

Für dieses Helpersystem wurden folgende Ergänzungen erarbeitet:

- *Einblendung der statischen Chatbox*: Die statische Chatbox soll erst eingeblendet werden, falls sich der Code ändert. Diese soll mit einem Hinweis versehen sein, dass der Code erneut ausgeführt werden muss, um die Änderungen im Code erneut zu überprüfen.
- *Änderung der Chatbox*: Die Chatbox mit der Erklärung soll ausgegraut dargestellt werden, sobald der Code geändert wird. Dies soll zusammen mit der vorab genannten Verbesserung verdeutlichen, dass diese Chatbox nicht mehr aktuell ist und der Code erneut ausgeführt werden muss, um die Änderungen abermals zu überprüfen.
- *Entfernen der Chatbox*: Die Chatbox, welche die Erklärung für den Fehler beinhaltet, soll nach jeder Ausführung des Codes automatisch entfernt werden. Falls der Code weiterhin fehlerhaft ist, wird eine neue Chatbox angezeigt. Ansonsten verschwindet diese Chatbox und es bleibt nur die statische Chatbox ersichtlich.

4.2.5 Helpersystem – Tipps

Das letzte Mockup befasst sich mit dem Helpersystem Tipps. Abbildung 4-7 veranschaulicht die dazugehörigen Überlegungen.

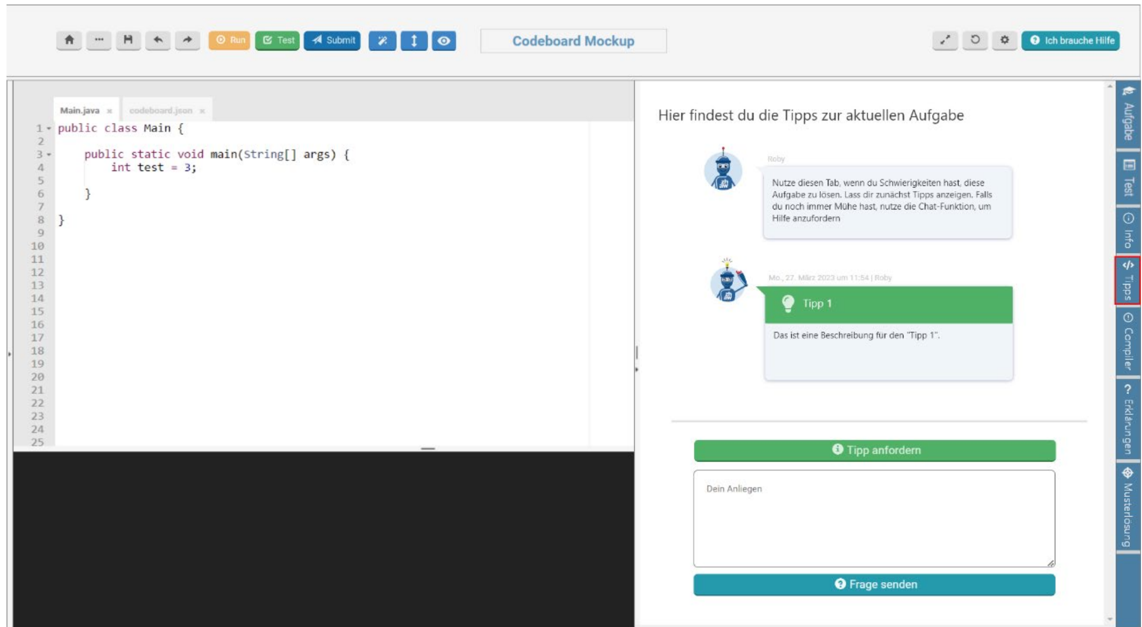


Abbildung 4-7: Mockup – Tipps

Im nachfolgenden Kapitel wird das Mockup beschrieben und Möglichkeiten zur Verbesserung dieses Tabs erläutert.

4.2.5.1 Tab für Tipps

Der dazugehörige Tab setzt sich aus Chatboxen für Tipps und einem Eingabefeld für individuelle Fragen zusammen. Zusätzlich werden die Fragen sowie die dazugehörigen Antworten in Form von Chatboxen in diesem Tab angezeigt. In der nachfolgenden Auflistung erfolgt eine detaillierte Beschreibung der dazugehörigen Elemente:

- *Statische Chatbox*: Diese beinhaltet eine kurze Beschreibung der Funktionalitäten dieses Helpersystems.
- *Chatboxen für Tipps*: Für jeden angeforderten Tipp erscheint eine neue Chatbox (grüne Kopfzeile), welche den entsprechenden Hinweis beinhaltet.
- *Button «Tipp anfordern»*: Sobald dieser Button gedrückt wird, wird eine neue Chatbox mit dem dazugehörigen Tipp angezeigt. Falls alle hinterlegten Tipps angefordert wurden, verschwindet dieser Button.
- *Eingabefeld für Fragen*: Studierende können in diesem Feld ihre individuellen Fragen an die Dozierenden eingeben.
- *Button «Frage senden»*: Durch die Betätigung dieses Buttons wird die Frage an die Dozierenden übermittelt. Nach der Übermittlung erscheint eine Chatbox mit der Frage oberhalb des Eingabefelds. Dasselbe gilt für Antworten auf Fragen durch Lehrende. Diese erscheinen ebenso oberhalb des Eingabefelds, sobald die Dozierenden die Frage beantwortet haben.

4.2.5.2 Verbesserungen

Folgende Erweiterungen wurden für dieses Helper-System erarbeitet:

- *Trennung von Tipps und manuellen Fragen*: Die beiden Komponenten sollen getrennt dargestellt werden. Dabei soll nach der statischen Chatbox und nach dem Button «Tipp anfordern» eine horizontale Linie eingefügt werden, um eine bessere Trennung dieser Helpersysteme zu gewährleisten.

5 Codeboard

Dieses Kapitel dient der Dokumentation der einzelnen Entwicklungsschritte der neuen Version des Codeboards. Zunächst werden die erhobenen Anforderungen aufgezeigt und beschrieben. Im Anschluss folgt eine Dokumentation der Implementierung, welche die wichtigsten Überlegungen zur Umsetzung beinhaltet. Abschliessend werden im Kapitel *Testing* die durchgeführten Tests erläutert.

5.1 Requirements Engineering

Die Dokumentation sowie die entsprechende Priorisierung der Anforderungen, welche an die neue Version des Codeboards (CB) gestellt werden, finden sich in den folgenden drei Kapiteln.

5.1.1 Funktionale Anforderungen

Die nachfolgende Tabelle beinhaltet die funktionalen Anforderungen.

Req. #	Beschreibung
CB-A-1	Die Funktionalität der einzelnen Helpersysteme soll in einem zusätzlichen Tab erläutert werden. Zusätzlich sollen die Helpersysteme direkt aus diesem Tab aufrufbar sein.
CB-A-2	Die Helpersysteme (Test und Compiler-Meldungen) sollen dynamisch angezeigt werden. Zum Beispiel sollen nach der Ausführung von fehlerhaftem Code automatisch die Erklärungen zu den Compiler-Meldungen angezeigt werden.
CB-A-3	Falls ein Test durchgeführt wird und der Code fehlerhaft ist, soll der Tab des Helpersystems Compiler-Meldungen angezeigt werden. Falls der Code fehlerfrei ist, wird das IO-Testing (im Test-Tab) durchgeführt.
CB-A-5	Der Code soll mittels eines Buttons formatiert werden können.
CB-A-6	Der Zugriff auf die manuellen Fragen sowie Antworten auf die Fragen von Dozierenden soll über einen eigenen Tab in der rechten Navigationsleiste gewährleistet sein.

Tabelle 5-1: Funktionale Anforderungen – Codeboard

5.1.2 Nicht-funktionale Anforderungen

Es wurde lediglich eine nicht-funktionale Anforderung erhoben, welche in der folgenden Tabelle dargestellt ist.

Req. #	Beschreibung
CB-NFA-1	Das GUI soll im Allgemeinen einfach zu bedienen, übersichtlich und intuitiv sein.

Tabelle 5-2: Nicht-funktionale Anforderungen – Codeboard

5.1.3 Priorisierung der Anforderungen

Die Priorisierung der einzelnen Anforderungen nach der MoSCoW-Methode ist in der nachfolgenden Tabelle ersichtlich.

Req. #	Must have	Should have	Could have
CB-A-1		X	
CB-A-2	X		
CB-A-3	X		
CB-A-5	X		
CB-A-6		X	
CB-NFA-1	X		

Tabelle 5-3: Priorisierung – Codeboard

5.2 Implementierung

In den nachfolgenden Abschnitten wird die Umsetzung von vier Anforderungen näher erläutert. Im Kapitel Abweichungen erfolgt eine Darlegung von Anforderungen, welche nicht umgesetzt wurden. Diese Anforderungen finden sich im Changelog unter dem Anhang 1.1.

Die Umsetzung der Anforderung *CB-A-1* wurde als wichtig erachtet, da die neue Version des Codeboards doch um einige Funktionalitäten erweitert wurde. Beispielsweise sind neue Buttons in der oberen Navigationsleiste vorhanden und die einzelnen Helpersysteme sind alle über einen eigenen Tab aufrufbar. Die vielen Anpassungen in der Benutzeroberfläche könnten zu Unsicherheiten oder Unklarheiten bei Studierenden führen, wenn diese das erste Mal mit der angepassten Version arbeiten. Aus diesem Grund wurde ein Info-

Tab implementiert, welcher die einzelnen Helpersysteme sowie die dazugehörigen Buttons erläutert. Dieser Tab wird zusätzlich angezeigt, falls Studierende auf den Button "Ich brauche Hilfe" in der oberen Navigationsleiste klicken.

Eine wichtige Überlegung in der Umsetzung der Anforderung *CB-A-3* war, ob die Fehler-Chatbox, welche während des Testens von fehlerhaftem Code generiert wird, weiterhin im Test-Tab angezeigt werden solle. Durch die Anforderung, dass jedes Helpersystem seinen eigenen Anwendungszweck aufweisen soll, wurde dieses Verhalten angepasst. Falls der Code getestet wird und dabei syntaktische oder semantische Fehler aufweist, wird die dazugehörige Chatbox neu im Compiler-Tab angezeigt. Somit ist die Erfüllung der vorab genannten Anforderung sichergestellt. Diese Chatbox kann der Abbildung 5-1 entnommen werden.

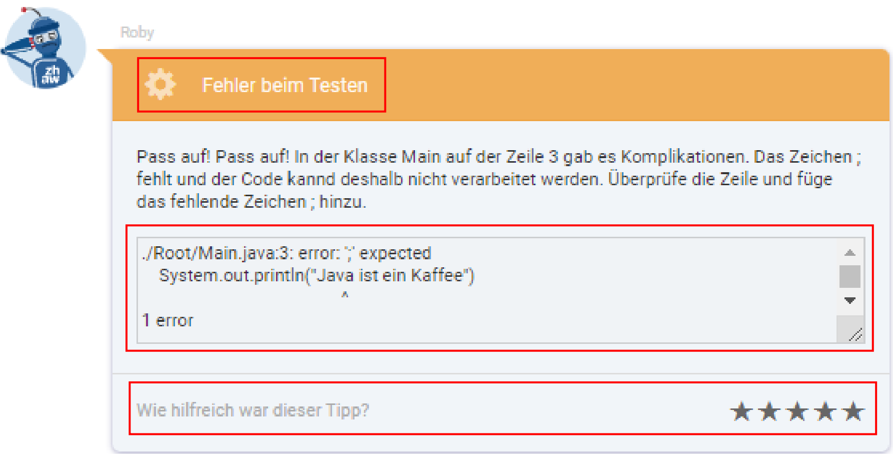


Abbildung 5-1: Fehler-Chatbox (Testen)

Zur Veranschaulichung der Unterschiede wird nachfolgend eine Compiler-Chatbox dargestellt, welche angezeigt wird, falls der Code mittels Run-Button ausgeführt wird.

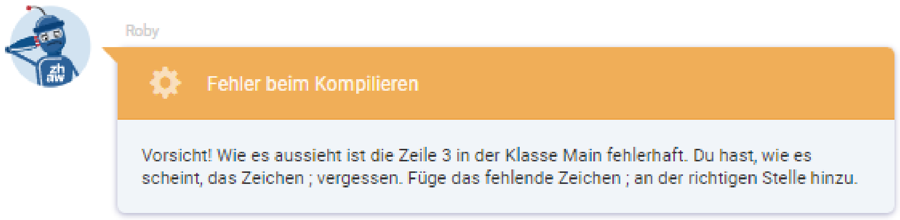


Abbildung 5-2: Fehler-Chatbox (Kompilieren)

Damit klar ist, welche Chatbox zu welchem Szenario gehört, wurde die Chatbox, welche beim Testen generiert wird, angepasst. Die Unterschiede sind in Abbildung 5-1 rot

markiert. Einerseits weist die Chatbox einen anderen Titel ("Fehler beim Testen") auf, damit klar ist, dass diese Chatbox durch das Helpersystem Test erzeugt wurde. Andererseits wird die Fehlermeldung unterhalb der Erklärung in einem Textfeld angezeigt. Diese wird eingeblendet, da keine Fehlermeldung in der Konsole ersichtlich ist. Zusätzlich ist die Chatbox mit einer Rating-Funktion versehen, wobei die Studierenden die angezeigte Erklärung bewerten können.

Die Anforderung *CB-A-5* wurde umgesetzt, da die korrekte Formatierung von Code im Editor eine Herausforderung für einige Studierende darstellt. Für die Umsetzung dieser Funktionalität wurde die externe Library *js-beautify* hinzugezogen (*Js-Beautify*, 2022). Eine Schwierigkeit bei der Umsetzung war, dass einige Sprachelemente nach der Formatierung nicht mehr vom Coding-Assistant erkannt werden konnten. Nachfolgende Tabelle veranschaulicht ein betroffenes Sprachelement.

Vor der Formatierung	Nach der Formatierung

`int[] g = {18, 32, 16, 24};`	`int[] g = {`
	`  18,`
	`  32,`
	`  16,`
	`  24`
	`};`

Tabelle 5-4: Sprachelement – Code-beautify

Weil der Coding-Assistant den Code zeilenweise erklärt, war es nicht mehr möglich, eine Chatbox mit dazugehöriger Erklärung für dieses Sprachelement zu generieren. Eine weitere Herausforderung ergab sich in der Darstellung des Gültigkeitsbereichs von Variablen. In der ursprünglichen Version des Coding-Assistant wurde, falls eine Variable deklariert wurde, geprüft, ob in den Zeilen vor oder nach der Deklaration eine öffnende beziehungsweise schliessende eckige Klammer ({}) vorhanden ist. Anhand dieser Klammern konnte anschliessend die Höhe des Balkens, welcher den Gültigkeitsbereich darstellt, berechnet werden. Die schliessenden Klammern (}) konnten dabei nur erkannt werden, falls diese auf einer neuen Code-Zeile ohne nachfolgenden Code vorhanden waren. Bei der Formatierung von Code, welcher if-else-Anweisungen enthält, wurden die jeweils nachfolgenden Anweisungen um eine Zeile nach oben verschoben. Die nachfolgende Tabelle 5-5 veranschaulicht dieses Verhalten. Dadurch konnten die schliessenden

Klammern nicht mehr bestimmt werden und der Balken des Gültigkeitsbereichs wurde inkorrekt dargestellt.

Vor der Formatierung (korrekt)	Nach der Formatierung (inkorrekt)

1 if (Bedingung) {
2   int a;
3
4 }
5 else if (Bedingung) {
6 }
7 else {
8 }
9
	1 if (Bedingung) {
	2 int a;
	3 } else if (Bedingung) {
	4
	5 } else {
	6
	7 }
	8
	9

Tabelle 5-5: Gültigkeitsbereich – Code-beautify

Für erstere Problematik wurde vorerst geprüft, welche Sprachelemente durch die Formatierung des Codes nicht mehr erklärt werden konnten. Eine Analyse ergab, dass lediglich ein paar Sprachelemente von dieser Herausforderung betroffen waren. Deshalb fiel der Entschied, diese Anforderung dennoch umzusetzen und den bestehenden Code anzupassen. Um der Herausforderung betreffend den falsch formulierten Sprachelementen entgegenzuwirken, konnte die Methode, welche den Code formatiert, um folgende Option ergänzt werden:

```
brace_style: 'collapse, preserve-inline'
```

Dadurch wird sichergestellt, dass sich bspw. das Sprachelement aus Tabelle 5-4 nach der Formatierung nicht über mehrere Code-Zeilen erstreckt, wie in der rechten Seite der Tabelle ersichtlich ist. Für die Darstellung des Gültigkeitsbereichs wurde der Code so angepasst, dass neu auch schliessende eckige Klammern (}) mit nachfolgender else-if- oder else-Anweisung erkannt werden. Dadurch konnte dieses Problem behoben werden und die Gültigkeitsbereiche werden, wie in Abbildung 5-3 ersichtlich, korrekt dargestellt.

```
1 if (Bedingung) {
2     int a;
3
4 } else if (Bedingung) {
5
6 } else {
7
8 }
9
```

Abbildung 5-3: Korrekte Gültigkeitsbereiche – Code-beautify

Betreffend der Anforderung CB-A-6 galt es zu prüfen, wie die individuelle Fragen sowie Antworten in einem eigenen Tab dargestellt werden können. Wie im Kapitel 3.1.2.2

erwähnt, beinhaltete der Hilfe-Tab in der ursprünglichen Version des Codeboards die Chatboxen für Compiler-Meldungen, Tipps, manuelle Fragen sowie Antworten. All diese Chatboxen wurden bei Bedarf jeweils in einem Array `$scope.chatLines` gespeichert und anschliessend im Tab angezeigt. Da die Trennung der einzelnen Helpersysteme eine Anforderung von hoher Priorität war, musste diese Konfiguration angepasst werden. Es ist anzumerken, dass für jeden neuen Tab jeweils ein neues HTML-File (Template) erstellt werden musste. Damit die jeweiligen Chatboxen im korrekten Tab angezeigt werden können, wurde folgende Anpassung vorgenommen. Mittels nachfolgender Funktion ist es möglich, dieses `$scope.chatLines` Array auf Änderungen zu überprüfen.

```
$scope.$watch('chatLines', function() {
  filterCompilerChatLines();
  filterTipChatLines();
  filterHelpChatLines();
}, true);
```

Falls dem Array eine neue Chatbox hinzugefügt wird, wird diese Funktion ausgeführt, welche wiederrum drei Filterfunktionen aufruft. Zur Veranschaulichung wird nachfolgend die `filterHelpChatLines()` Funktion erläutert, welche Chatboxen für manuelle Fragen sowie Antworten erkennt.

```
function filterHelpChatLines() {
  $scope.filteredHelpRequestChatLines = $scope.chatLines.filter(func-
tion(chatLine) {
    return chatLine.type === 'helpRequest' || chatLine.type === 'helpRe-
questAnswer';
  });
}
```

Nach dem Aufruf dieser Funktion wird geprüft, ob das `$scope.chatLines` Array Chatboxen vom Typ "helpRequest" (Fragen von Studierenden) oder "helpRequestAnswer" (Antworten von Dozierenden) beinhaltet. Falls dem so ist, werden diese Chatboxen in einem neuen Array `$scope.filteredHelpRequestChatLines` gespeichert. Mit dem folgenden Code im Template können diese Chatboxen anschliessend in dem Fragen-Tab angezeigt werden.

```
<chat chat-lines="filteredHelpRequestChatLines" class="px-4">
```

Es ist anzumerken, dass diese Implementierung zusätzlich die Anzeige von Compiler-Meldungen im Compiler-Tab (*CM-A-1*) sowie Tipps im Tipps-Tab (*T-A-1*) abdeckt. Die `filterCompilerChatLines()` Funktion überprüft das `$scope.chatLines` Array auf Chatboxen des Typs "`compiler`" (Ausführung des Programms) oder "`compilerTest`"

(Test des Programms). Die Funktion `filterTipChatLines()` überprüft dieses Array auf Chatboxen des Typs "`hint`". Durch diese Filterfunktionen musste die Konfiguration zur Erzeugung und Anzeige von Chatboxen nur bedingt angepasst werden. Eine Anpassung war, dass jeder Chatbox bei der Erzeugung einer der vorgängig erwähnten Typen übergeben wird.

5.2.1 Abweichungen

Im Zusammenhang mit der Anforderung *CB-A-4* wurde vorab evaluiert, ob der Ace-Editor eine Option bietet, welche Fehler im Code zur Laufzeit markiert, ohne dass der Code ausgeführt werden muss. Zusätzlich zur Markierung sollte der Fehler beschrieben werden. Die Evaluation ergab, dass der Ace-Editor zwar eine solche Funktionalität bietet, diese jedoch nicht für die Programmiersprache Java anwendbar ist. Ein weiterer Grund, welcher gegen die Umsetzung dieser Anforderung sprach, betraf die Didaktik. Lernende sollten lernen, Fehler im Code selbstständig zu identifizieren und zu beheben. Folglich wurde zusammen mit dem Auftraggeber der Entscheid getroffen, diese Anforderung nicht umzusetzen. Nichtsdestotrotz wurde eine andere Möglichkeit erarbeitet, Fehler im Code-Editor zu markieren, welche in Kapitel 6.2.2 unter der Anforderung *CA-E-A-4* erläutert wird. Es ist anzumerken, dass diese Lösung den Fehler nicht erläutert, sondern lediglich die Code-Zeile markiert, welche den Fehler beinhaltet. Daher müssen Studierende den Fehler in der Zeile selbstständig ausfindig machen, was den vorab erwähnten Punkt betreffend Didaktik berücksichtigt.

Eine weitere Anforderung, welche sich während der Umsetzung ergab, war das Aktualisieren des Ace-Editors auf die neueste Version (*CB-A-7*). In der aktuellen Version des Codeboards wird die Version 1.4.6 verwendet, wohingegen bereits die Version 1.19.0 verfügbar ist (*Ace*, 2023). Um sicherzustellen, dass ein Update des Editors keine negativen Auswirkungen auf den vorhandenen Code hat, wurde eine separate Branch (updated-ace) erstellt. Eine Branch ermöglicht die Isolation von Entwicklungsarbeiten, ohne eine andere Branch im selben Repository zu beeinflussen (*GitHub - About Branches*, o. J.). Nach der Aktualisierung wurde festgestellt, dass die neue Version negative Auswirkungen auf die Benutzeroberfläche des Codeboards hat. Beispielsweise wurden Chatboxen nicht mehr korrekt dargestellt, weswegen der Entscheid fiel, diese Anforderungen nicht umzusetzen. Es wäre dennoch möglich zu prüfen, ob das Aktualisieren von weiteren Dependencies diese Problematik beseitigen würde, was allerdings den Rahmen dieser Arbeit sprengen würde.

5.3 Testing

In den nachfolgenden Abschnitten werden die durchgeführten Tests betreffenden den umgesetzten Anforderungen zusammenfassend erläutert. Eine ausführliche Dokumentation der durchgeführten Tests ist dem Anhang 3.1 zu entnehmen.

Zusammenfassend lässt sich sagen, dass alle an das Codeboard erhobenen Anforderungen erfolgreich umgesetzt wurden. Ein wichtiger Punkt betraf die dynamische Anzeige der einzelnen Helpersysteme (*CB-A-2*). Beispielsweise wurde geprüft, ob bei der Ausführung von fehlerhaftem Code in den Compiler-Tab gewechselt und in diesem die korrekte Chatbox für die Erklärung der Compiler-Meldung angezeigt wird. Zusätzlich erfolgte eine ganzheitliche Prüfung des Test-Tabs (*CB-A-3*). Einerseits wurde überprüft, ob beim Testen von fehlerhaftem Code automatisch in den Compiler-Tab gewechselt wird und dort die entsprechende Chatbox angezeigt wird. Andererseits wurde geprüft, ob beim Testen von fehlerfreiem Code das Input-/Output-Testing im Test-Tab durchgeführt wird. Bei der Anforderung *CB-A-5* wurde evaluiert, ob sich der Code im Editor mittels beautify-Button formatieren lässt. Dabei wurde mehrfach unformatierter Code im Code-Editor erfasst, um diesen im Anschluss zu formatieren. In allen Fällen wurde der Code korrekt formatiert. Der Fragen-Tab (*CB-A-6*) funktioniert ebenso wie erwartet. Studierende können Fragen erfolgreich erfassen und Dozierende können diese im Anschluss beantworten. In beiden Fällen werden die entsprechenden Chatboxen korrekt angezeigt.

6 Helpersystem – Coding-Assistant

Dieses Kapitel dient der Entwicklungsdokumentation des Coding-Assistant (CA). Es gilt zu beachten, dass den wichtigsten Features jeweils ein Kapitel gewidmet ist. Dabei wird zwischen folgenden Funktionalitäten unterschieden.

1. Sprachelemente und Integrationsvorbereitung
2. Erklärungen von Code-Zeilen
3. Gültigkeitsbereiche von Variablen
4. Visualisierung von Code-Blöcken

Alle vier Kapitel beinhalten eine Beschreibung des Requirements Engineering, der Implementation sowie des Testings. Zusätzlich wird im Kapitel 6.5 ein kurzer Austausch mit dem Entwickler, welcher die ursprüngliche Version des Codeboards auf die Bedürfnisse der Zürcher Hochschule für Angewandte Wissenschaften zugeschnitten hat, erläutert.

6.1 Sprachelemente und Integrationsvorbereitung

Inhalt dieses Kapitel ist die Beschreibung der Entwicklungsschritte des Coding-Assistant betreffend *Sprachelemente und Integrationsvorbereitung* (SI).

6.1.1 Requirements Engineering

Die nachfolgenden drei Kapitel veranschaulichen die erhobenen Anforderungen sowie deren Priorisierung.

6.1.1.1 Funktionale Anforderungen

Folgende funktionale Anforderungen wurden an den Coding-Assistant erhoben.

Req. #	Beschreibung
CA-SI-A-1	Der Quellcode soll auf das Framework AngularJS abgestimmt werden.
CA-SI-A-4	Den einzelnen Sprachelementen soll in der Konfigurationsdatei eine neue Eigenschaft hinzugefügt werden, welche die Sprachelemente kategorisiert.

Tabelle 6-1: Funktionale Anforderungen – Sprachelemente & Integrationsvorbereitung

6.1.1.2 Nicht-funktionale Anforderungen

Es wurden keine nicht-funktionalen Anforderungen erhoben.

6.1.1.3 *Priorisierung der Anforderungen*

Die nachfolgende Tabelle veranschaulicht die Priorisierung der Anforderungen.

Req. #	Must have	Should have	Could have
CA-SI-A-1	X		
CA-SI-A-4			X

Tabelle 6-2: Priorisierung – Sprachelemente & Integrationsvorbereitung

6.1.2 Implementierung

In den nachfolgenden zwei Abschnitten wird die Umsetzung der erhobenen Anforderungen zusammenfassend erläutert. Im Anschluss erfolgt eine kurze Darlegung der Anforderungen, welche nicht umgesetzt wurden.

Die Umsetzung der Anforderung CA-SI-A-1 war von hoher Relevanz für das gesamte Kapitel 6. Die ursprüngliche Version des Coding-Assistant wurde in der Programmiersprache JavaScript programmiert. Das Codeboard hingegen basiert auf dem JavaScript basierten Framework AngularJS. Dieses Framework bietet verschiedene Konzepte wie beispielsweise Services, welche die Business Logic beinhalten, oder Controller, welche mit der View (also dem, was Nutzende sehen) interagieren (*AngularJS - docs*, o. J.). Diese Konzepte werden im Rahmen dieser Arbeit nicht genauer erläutert. Aus Wartbarkeits-, Verständlichkeits-, Konfigurierbarkeits- und Überprüfbarkeitsgründen musste der Code der ursprünglichen Version grundlegend auf dieses Framework abgestimmt werden. Im Allgemeinen haben folgende Überlegungen die gesamte Integration des Coding-Assistant geprägt.

- Was kann beibehalten werden?
- Was muss gelöscht werden?
- Was muss angepasst werden?

Viele dieser Überlegungen ergeben sich aus der Implementationsdokumentation dieses Kapitels. Dennoch sei in diesem Abschnitt auf die drei grundlegenden Dateien, welche einen Grossteil der Funktionalität des Coding-Assistant beinhalten, eingegangen werden.

- `codingAssistantMainCtrl.js`: Dieser Controller sorgt beispielsweise dafür, dass die Daten aus den Dateien `explanations.json` sowie `colors.json` zur Verfügung stehen oder die Chatboxen korrekt erzeugt werden. Zusätzlich beinhaltet

diese Datei das Auslesen des Codes der Studierenden aus dem Code-Editor und das Aufrufen des Services zur Generierung des Inhaltes der Chatboxen bei jeder Änderung im Code-Editor.

- `codingAssistantCodeMatchSrv.js`: Dieser Service beinhaltet die gesamte Logik zur Generierung des Inhaltes der Chatboxen. Zusammenfassend wird der Code der Studierenden mit den regulären Ausdrücken aus der Datei `explanations.json` verglichen, um bei Bedarf die entsprechenden Erklärungen zu generieren.
- `navBarRightExplanation.html`: Dieses Template sorgt dafür, dass die Chatboxen im Erklärungs-Tab angezeigt werden.

Gewisse Funktionalitäten, welche Bestandteil der vorab erwähnten Dateien sind, werden in den nachfolgenden Kapiteln erneut aufgegriffen und genauer erläutert.

Betreffend der Anforderung *CA-SI-A-4* wurden die einzelnen Elemente in der Konfigurationsdatei um Typen ergänzt. Aktuell befinden sich alle regulären Ausdrücke sowie die dazugehörigen Erklärungen in einem JSON-File `explanations.json`. Der Aufbau dieser Datei wird im Anhang 2.2.1 detaillierter beschrieben. Falls in Zukunft eine andere Speichermethode gewählt wird, können die Typisierungen hilfreich sein. Die verwendeten Typen können ebenso dem Anhang 2.2.1 entnommen werden. Die Umsetzung dieser Anforderung hat zum Zweck, einzelne Erklärungen filtern zu können, wenn diese zukünftig beispielsweise in einer Datenbank gespeichert werden. Dies würde allenfalls die Wartbarkeit verbessern.

6.1.2.1 Abweichungen

Aus Priorisierungsgründen wurden die Anforderungen *CA-SI-A-2*, *CA-SI-A-3* sowie *CA-SI-A-5* nicht umgesetzt. Diese wiesen eine geringe Priorität auf, weswegen dieser Entscheid zusammen mit dem Auftraggeber getroffen wurde.

6.1.3 Testing

Beim Entwicklungsschritt *Sprachelemente und Integrationsvorbereitung* wurden keine Tests durchgeführt. Da die Anforderung *CA-SI-A-1* grundsätzlich die gesamte Integration des Coding-Assistant betrifft, finden sich entsprechende Tests in den nachfolgenden Kapiteln. Die Umsetzung der Anforderung *CA-SI-A-4* zu testen, erwies sich unnötig, da die einzelnen Elemente in der Konfigurationsdatei lediglich um einen Typ ergänzt wurden. Dieser hat dabei keinerlei Einfluss auf die Funktionalität des Coding-Assistant.

6.2 Erklärungen

Unter *Erklärungen* wird die Ausgabe von Erklärungen in Textform für korrekte oder nicht-korrekte Code-Zeilen verstanden. Die Kapitel 6.2.1 bis 6.2.3 befassen sich mit der Dokumentation der Umsetzung dieser Funktionalität.

6.2.1 Requirements Engineering

Folgende drei Kapitel veranschaulichen die Anforderungen sowie deren Priorisierung.

6.2.1.1 Funktionale Anforderungen

Es wurden folgende funktionale Anforderungen im Rahmen dieser Arbeit an die Funktionalität *Erklärungen* (E) erhoben.

Req. #	Beschreibung
CA-E-A-1	Der Zugriff auf die Erklärungen soll über einen eigenen Reiter in der rechten Navigationsliste des Codeboards gewährleistet sein.
CA-E-A-2	Für jede geschriebene Code-Zeile soll Roby die Erklärungen in Form von Chatboxen ausgeben. Dabei soll dynamisch für jede vollständige Zeile eine neue Chatbox angezeigt werden.
CA-E-A-3	Sobald eine Chatbox erzeugt wird, soll sich deren Inhalt automatisch an Codeänderungen auf der entsprechenden Zeile anpassen. Falls die entsprechende Code-Zeile gelöscht wird, soll auch die dazugehörige Chatbox entfernt werden.
CA-E-A-4	Falls die Code-Zeile Fehler aufweist, soll in der Chatbox auf diese aufmerksam gemacht werden. Zum Beispiel: «In dieser Zeile hat sich ein Fehler eingeschlichen. Bitte korrigiere den Code, damit ich ihn erklären kann.» Zusätzlich zur Erklärung soll die dazugehörige Code-Zeile rot markiert werden. Des Weiteren soll die Chatbox erst angezeigt werden, wenn man auf die nächste Zeile wechselt.
CA-E-A-5	Bei der Auswahl einer Code-Zeile soll die dazugehörige Chatbox hervorgehoben werden. Ebenso soll die Chatbox mittig zur entsprechenden Code-Zeile angezeigt werden.
CA-E-A-6	Die Chatboxen für die Erklärungen sollen mit einem Link zur dazugehörigen Dokumentation (z. B. W3Schools) ergänzt werden.
CA-E-A-8	Falls kein Code im Editor vorhanden ist, soll eine statische Chatbox mit folgendem Inhalt angezeigt werden: "Bitte öffne eine Java-Datei oder schreibe Code, damit ich den Coder erklären kann".

Tabelle 6-3: Funktionale Anforderungen – Erklärungen

6.2.1.2 Nicht-funktionale Anforderungen

Betreffend Code-Erklärungen wurde lediglich eine nicht-funktionale Anforderung erhoben, welche in der folgenden Tabelle ersichtlich ist.

Req. #	Beschreibung
CA-E-NFA-1	Die Erklärungen sollen mit einer möglichst geringen Zeitverzögerung angezeigt werden.

Tabelle 6-4: Nicht-funktionale Anforderungen – Erklärungen

6.2.1.3 Priorisierung der Anforderungen

Die Priorisierung der einzelnen Anforderungen ist in der untenstehenden Tabelle dargestellt.

Req. #	Must have	Should have	Could have
CA-E-A-1	X		
CA-E-A-2	X		
CA-E-A-3	X		
CA-E-A-4	X		
CA-E-A-5		X	
CA-E-A-6		X	
CA-E-A-8			X
CA-E-NFA-1	X		

Tabelle 6-5: Priorisierung – Erklärungen

6.2.2 Implementierung

In den nachfolgenden Abschnitten werden einige der umgesetzten Anforderungen näher erläutert.

Eine grundlegende Überlegung bei der Umsetzung der Anforderung *CA-E-A-2* war, wie für jede Code-Zeile eine neue Chatbox generiert werden kann. In der ursprünglichen Version des Coding-Assistant wurde für jede geschriebene Code-Zeile ein neues `<div>` Element mit dazugehöriger Erklärung (orange), Link zu weiterführenden Dokumentationen (blau) sowie CSS-Klasse (grün) erzeugt. Diese Daten wurden alle aus dem JSON-File `explanations.json` geladen, welches im Anhang 2.2.1 erläutert wird. Im Anschluss

wurden diese Elemente gemäss der Reihenfolge der Code-Zeilen im Code-Editor in einer Variable `outputText` gespeichert. Untenstehende Tabelle veranschaulicht den Inhalt dieser Variable für den Code, welcher in der oberen Hälfte der Tabelle zu finden ist.

Code-Editor	public class Hallo {
	  public static void main(String[] args) {
	    System.out.println("Hallo, Welt!");
	  }
	}

outputText	<div onclick="window.open('https://studyflix.de/informatik/java-klassen-1902', '_blank'); event.stopPropagation();" style="cursor: pointer;" class="anyDiv class level0"><div class="looksNiceStart">Klasse "Hallo" wird deklariert und ist public</div> <div onclick="window.open('https://www.geeksforgeeks.org/java-main-method-public-static-void-main-string-args/', '_blank'); event.stopPropagation();" style="cursor: pointer;" class="anyDiv method level"><div class="looksNiceStart">Die "main-method" wird erstellt (Ausführung des Programms beginnt hier)</div> <div onclick="window.open('https://www.w3schools.com/java/java_variables_print.asp', '_blank'); event.stopPropagation();" style="cursor: pointer;" class="anyDiv print level">""Hallo, Welt!"" wird auf der Konsole ausgegeben (mit Zeilenumbruch) </div> <div class="looksNiceEnd">Code-Block wird beendet</div></div> <div class="looksNiceEnd">Code-Block wird beendet</div></div>

Tabelle 6-6: Speichermethode für Erklärungen (alte Version)

Im Anschluss wurde der Inhalt dieser Variable mit folgendem Code ins Erklärungsfenster geladen. Mittels `innerHTML` ist es möglich, den Inhalt eines HTML-Elements, in diesem Fall `editorExp`, festzulegen (*W3Schools - HTML*, o. J.).

```
document.getElementById("editorExp").innerHTML = outputText;
```

Daraus ergab sich die Notwendigkeit, eine neue Speichermethode für Erklärungen zu implementieren. Der Grund dafür war, dass mit der vorhandenen Methode die Erzeugung von separaten Chatboxen für jede Code-Zeile nicht möglich war. Die Lösung für dieses Problem ist ein Array, welches mehrere Werte speichern kann (*W3Schools - Arrays*, o. J.). Der Code wurde so angepasst, dass für jede Code-Zeile ein neues Objekt bestehend aus folgenden Eigenschaften in einem Array `explanations` gespeichert wird.

- `answer`: Die Erklärung des dazugehörigen Codes in Textform.
- `isError`: true, falls der Code nicht erklärt werden kann, ansonsten false.

- `lineLevel`: Die Zeilennummer des erklärten Codes im Code-Editor.
- `link`: Der zur Erklärung dazugehörige Link, welcher auf weiterführende Dokumentationen verweist.

```
▾ Array(3) ℹ
  ▾ 0:
      answer: "Klasse \"Hallo\" wird deklariert und ist public" ①
      isError: false
      lineLevel: 1 ③
      link: "https://studyflix.de/informatik/java-klassen-1902" ②
    ▸ [[Prototype]]: Object
  ▾ 1:
      answer: "Die \"main-method\" wird erstellt (Ausführung des Programms beginnt hier)" ①
      isError: false
      lineLevel: 2 ③
      link: "https://www.geeksforgeeks.org/java-main-method-public-static-void-main-string-args/" ②
    ▸ [[Prototype]]: Object
  ▾ 2:
      answer: "\"\"Hallo, Welt!\"\"  wird auf der Konsole ausgegeben (mit Zeilenumbruch) " ①
      isError: false
      lineLevel: 3 ③
      link: "https://www.w3schools.com/java/java_variables_print.asp" ②
    ▸ [[Prototype]]: Object
    length: 3
  ▸ [[Prototype]]: Array(0)
```

Abbildung 6-1: Speichermethode für Erklärungen (neue Version)

Durch diese Anpassung kann dynamisch für jede Code-Zeile eine neue Chatbox generiert werden. Dabei wird mittels einer `forEach()`-Methode eine Funktion (`explanation`) für jedes Element im Array aufgerufen, um auf deren Eigenschaften zugreifen zu können. Die Chatbox für korrekten Code hat den Typ `explanation` und wird folgendermassen erzeugt. Eine beispielhafte Chatbox ist in Abbildung 6-2 ersichtlich.

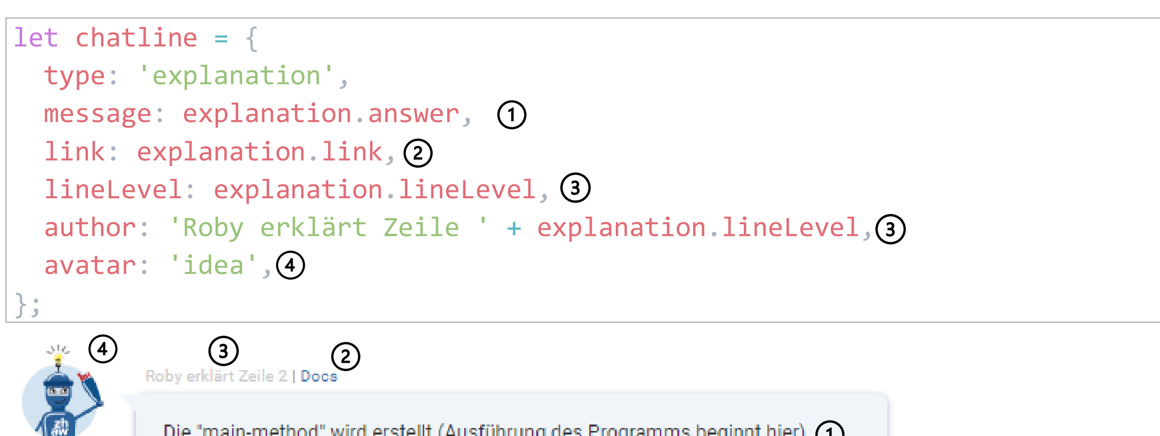


Abbildung 6-2: Chatbox für Code-Erklärungen

Die Umsetzung der Anforderung *CA-E-A-6* konnte mittels dieser Implementation ebenfalls realisiert werden. Der Zugriff auf weiterführende Dokumentationen ist mit einem Klick auf das Wort "Docs" (2), welchem der dazugehörige Link hinterlegt ist, möglich.

Zusätzlich wurde durch die gewählte Speichermethode die Implementierung der Anforderung *CA-E-A-8* vereinfacht. Falls das Array keine Objekte beinhaltet, weil im Code-Editor kein Code vorhanden ist, wird eine Chatbox angezeigt, welche die Studierenden darauf aufmerksam macht. In der darauffolgenden Abbildung 6-3 ist diese Chatbox ersichtlich.

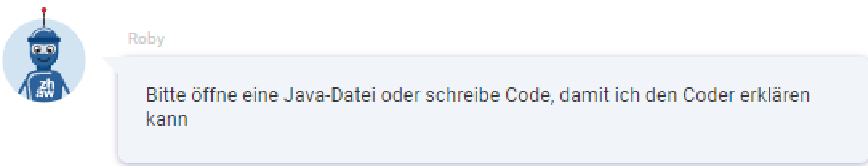


Abbildung 6-3: Hinweis-Chatbox

Das Vorgehen, wie sich die Chatboxen dynamisch an Code-Änderungen anpassen, wird unter dem Requirement *CA-E-A-3* erklärt.

Das Anpassen der Chatboxen an Änderungen im Code-Editor (*CA-E-A-3*) war eine weitere Herausforderung, welche sich während der Implementation ergab. In der ursprünglichen Version wurde, wie bereits in Kapitel 3.1.1 erwähnt, der Code im Sekundentakt aus dem Editor ausgelesen. Falls der Code geändert wurde, aktualisierte sich der Inhalt der Variable `outputText` und die neuen Erklärungen wurden im Erklärungsfenster angezeigt. Dieses Vorgehen konnte in der neuen Version aus Performance- und Usability-Gründen nicht verwendet werden. Einerseits wurde die Performance negativ beeinflusst, sobald der Editor mehrere Code-Zeilen aufwies. Andererseits war es dem Nutzenden nicht mehr möglich, im Tab, welches die Chatboxen mit den Erklärungen beinhaltet, zu scrollen. Aus diesem Grund wurde folgende Funktion erarbeitet.

```
service.aceChangeListener = function (aceEditor, callback) {
  aceEditor.on('change', function () {
    callback();
  });
};
```

Mittels dieser Funktion wird bei jeder Änderung im Code-Editor eine `callback()`-Funktion aufgerufen, welche schliesslich den Inhalt des Arrays `explanations` an die Änderungen anpasst. Dadurch wird sichergestellt, dass, falls eine Code-Zeile gelöscht wird, die dazugehörige Chatbox verschwindet und dass bei Anpassungen des Codes die dazugehörige Chatbox ebenfalls angepasst wird. Zusätzlich wurden dadurch die Performance- und Scrolling-Probleme behoben. Mit dieser Anpassung wurde zugleich die nicht-funktionale Anforderung *CA-E-NFA-1* umgesetzt. Durch die Anpassung des Inhalts des

Arrays `explanations` bei jeder Änderung im Code ist eine praktisch unmittelbare Generierung und Anzeige der Chatboxen möglich.

Die Generierung von Chatboxen für Fehler im Code (*CA-E-A-4*) folgt dem gleichen Prinzip wie der Erstellung von Chatboxen für korrekten Code. Die Eigenschaft `isError`, welche jedem Objekt im Array `explanations` hinzugefügt wird, ermöglicht eine Differenzierung zwischen Erklärungen für korrekten und fehlerhaften Code. Falls eine Code-Zeile nicht erklärt werden kann, wird diese Eigenschaft auf `true` gesetzt, damit klar ist, dass diese Zeile einen Fehler aufweist. Die Chatbox für nicht-korrekten Code hat den Typ `error` und setzt sich aus nachfolgenden Eigenschaften zusammen. Es ist anzumerken, dass diese Chatbox erst angezeigt wird, wenn Studierende auf eine neue Code-Zeile wechseln. Dadurch wird sichergestellt, dass Studierende nicht auf einen Fehler hingewiesen werden, falls diese die Code-Zeile noch nicht vervollständigt haben.

```
let chatline = {
  type: 'error',
  message: explanation.answer, ①
  link: explanation.link,
  lineLevel: explanation.lineLevel,
  author: 'Roby erklärt Zeile ' + explanation.lineLevel, ②
  avatar: 'worried',④
};
```

Die nachfolgende Abbildung 6-4 visualisiert die Chatbox für fehlerhaften Code. Diese unterschiedet sich von Chatboxen des Typs `explanation` durch ein anderes Bild des Avatars, den Text "Achtung Fehler" anstelle des "Docs"-Links und einer roten Schattierung.

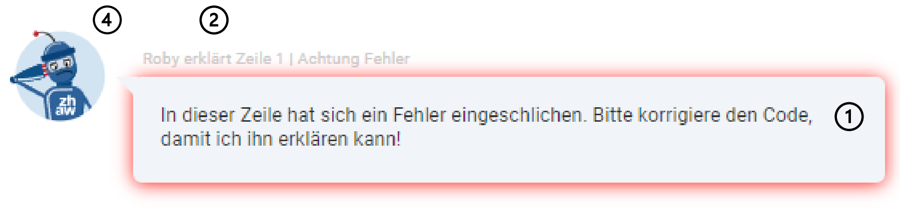


Abbildung 6-4: Chatbox für fehlerhaften Code

Zusätzlich zur Fehler-Chatbox wird die entsprechende Code-Zeile direkt im Editor mit einem Warn-Symbol versehen, was in Abbildung 6-5 ersichtlich ist. Falls man mit dem Mauszeiger über das Symbol fährt, wird zusätzlich der Text aus der dazugehörigen Chatbox angezeigt.

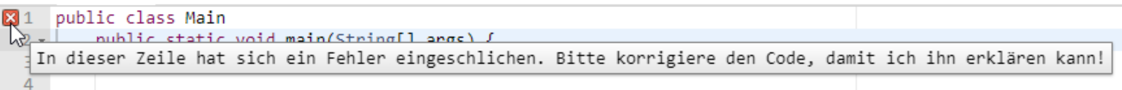


Abbildung 6-5: Warnsymbol für fehlerhaften Code

All diese Elemente sollen dazu beitragen, dass sofort ersichtlich ist, dass sich in der entsprechenden Code-Zeile ein Fehler eingeschlichen hat.

6.2.3 Testing

Alle an den Coding-Assistant betreffend der Funktionalität *Erklärungen* erhobenen Anforderungen konnten erfolgreich implementiert werden. Folgend werden wichtige Erkenntnisse zusammengefasst erläutert. Die gesamte Dokumentation des Tests dieser Funktionalität kann dem Anhang 3.2.1 entnommen werden.

Grundsätzlich kann der Coding-Assistant für jede geschriebene Code-Zeile eine neue Chatbox mit entsprechender Erklärung generieren (*CA-E-A-2*). Zusätzlich passen sich diese Chatboxen dynamisch an Änderungen im Code an (*CA-E-A-3*). Für diesen Test wurden vereinzelte Sprachelemente im Code-Editor erfasst, um zu prüfen, ob die korrekten Erklärungen in Form von Chatboxen ausgegeben werden. Zusätzlich wurde geprüft, ob die Zeilennummern in den Chatboxen mit den dazugehörigen Code-Zeilen übereinstimmen und ob der Link unter "Docs" auf die korrekten Dokumentationen verweist (*CA-E-A-6*). Es gilt anzumerken, dass der Coding-Assistant ausschliesslich Sprachelemente erklären kann, welche in der Konfigurationsdatei `explanations.json` erfasst sind. Aktuell umfasst diese Konfiguration alle Sprachelemente, welche Teil der ersten fünf Semesterwochen des Moduls Software Engineering 1 sind. Da die zu erkennenden Sprachelemente nicht im Rahmen dieser Arbeit erweitert wurden, wird die Umsetzung dieser Anforderungen daher als erfolgreich angesehen.

Betreffend der Anforderung *CA-E-A-4* wurde geprüft, ob der Coding-Assistant fehlerhaften Code erkennt und entsprechende Chatboxen anzeigt. Fehler, welche die aktuell erfassten Sprachelemente betreffen, werden allesamt erkannt. An diesem Punkt sei noch einmal die Notwendigkeit erwähnt, die Sprachelemente weiter zu ergänzen. Falls der Coding-Assistant ein solches Sprachelement nicht erkennt, wird der dazugehörige Code als fehlerhaft gekennzeichnet. Um dieser Problematik begegnen zu können, sollte der Coding-Assistant in der Lage sein alle Sprachelemente, welche in den an der Zürcher

Hochschule für Angewandte Wissenschaften angebotenen Kursen behandelt werden, zu erkennen.

Das Hervorheben von Chatboxen bei einem Klick in die dazugehörige Code-Zeile funktioniert ebenfalls einwandfrei. Zusätzlich wird bei einem Programm mit vielen Code-Zeilen automatisch zur entsprechenden Chatbox gescrollt, falls der Nutzende in eine Zeile im Code-Editor drückt (*CA-E-A-6*).

Zum Abschluss dieses Kapitels sei eine Problematik erläutert, welche die nicht-funktionale Anforderung (*CA-E-NFA-1*) betrifft. Grundsätzlich werden alle Chatboxen praktisch ohne Zeitverzögerung angezeigt. Der Test dieser Anforderung hat jedoch ergeben, dass viele Chatboxen (mehr als 150) geringe negative Auswirkungen auf die Performance der Applikation haben. Ein Grund könnte sein, dass der Generierung von Chatboxen ein komplizierter Prozess zugrunde liegt, welcher bei jeder Änderung im Code erneut durchlaufen wird. Im Allgemeinen stehen die gesamten Funktionalitäten des Codeboards den Nutzenden allerdings noch immer zur Verfügung. Jedoch erfolgt die Eingabe von neuem Code in den Editor mit einer geringen Zeitverzögerung. Das hat zur Folge, dass einzelne Zeichen erst nach einer minimalen Verzögerung angezeigt werden, was einen negativen Einfluss auf die Usability der Applikation haben könnte. Es gilt jedoch anzumerken, dass die im Codeboard zu lösenden Aufgaben meist eine geringere Anzahl Zeilen aufweisen und somit ohne Performance-Probleme gelöst werden können. Nichtsdestotrotz ergibt sich daraus die Notwendigkeit, zukünftig eine Lösung zu finden, um diese Problematik beheben zu können. Eine Möglichkeit wäre beispielsweise, den Coding-Assistant für Aufgaben mit vielen Code-Zeilen zu deaktivieren - oder nur Erklärungen für Code anzuzeigen, welcher von Studierenden ausgewählt wird.

6.3 Gültigkeitsbereiche

Die Dokumentation der einzelnen Entwicklungsschritte betreffend der Funktionalität *Gültigkeitsbereiche von Variablen* (GB) ist Teil dieses Kapitels.

6.3.1 Requirements Engineering

In den nachfolgenden Kapiteln 6.3.1.1 sowie 6.3.1.2 erfolgt eine Darstellung der erhobenen Anforderungen.

6.3.1.1 Funktionale Anforderungen

Folgende funktionale Anforderungen wurden an diese Funktionalität erhoben.

Req. #	Beschreibung
CA-GB-A-1	Das Fenster, welches die Gültigkeitsbereiche beinhaltet, soll dynamisch mittels eines Buttons ein- und ausgeblendet werden können.
CA-GB-A-2	Das Fenster, welches die Gültigkeitsbereiche von Variablen visuell darstellt, sollte sich auf der linken Seite des Code-Editors befinden. Darin sollten die Gültigkeitsbereiche in Form von Balken dargestellt werden.
CA-GB-A-3	Falls Variablen ausserhalb des Gültigkeitsbereichs verwendet werden, soll in der Chatbox (Fenster für Code-Erklärungen) darauf aufmerksam gemacht werden. Dasselbe gilt für Variablen, welche mit gleichem Namen doppelt deklariert werden.
CA-GB-A-4	Marker für Variablennamen sollen nur bei Einblendung des Gültigkeitsbereichs angezeigt werden.
CA-GB-A-6	Falls der Gültigkeitsbereich eingeblendet wird, sollen für alle Variablennamen im Code-Editor Marker eingeblendet werden, damit klar ist, welche Variable zu welcher Visualisierung des Gültigkeitsbereichs gehört.
CA-GB-A-8	Synchrones Scrollen im Code-Editor und im Fenster für Gültigkeitsbereiche von Variablen muss gewährleistet sein, damit Variablen und dazugehörige Balken auf einer Linie bleiben.

Tabelle 6-7: Funktionale Anforderungen – Gültigkeitsbereiche

6.3.1.2 Nicht-funktionale Anforderungen

Die nachfolgende Tabelle zeigt eine nicht-funktionale Anforderung.

Req. #	Beschreibung
CA-GB-NFA-1	Für die Visualisierung der Gültigkeitsbereiche sollen Farben gewählt werden, welche mit den Markern der Variablennamen übereinstimmen.

Tabelle 6-8: Nicht-funktionale Anforderungen – Gültigkeitsbereiche

6.3.1.3 Priorisierung der Anforderungen

Wie die einzelnen funktionalen und nicht-funktionalen Anforderungen priorisiert wurden, kann der folgenden Tabelle entnommen werden.

Req. #	Must have	Should have	Could have
CA-GB-A-1	X		
CA-GB-A-2	X		
CA-GB-A-3		X	
CA-GB-A-4	X		
CA-GB-A-6	X		
CA-GB-A-8	X		
CA-GB-NFA-1	X		

Tabelle 6-9: Priorisierung – Gültigkeitsbereiche

6.3.2 Implementierung

In den nachfolgenden Abschnitten werden einige der erhobenen Anforderungen betreffend dieser Funktionalität genauer erläutert.

Die Integration des Gültigkeitsbereichs von Variablen (*CA-GB-A-2*) stellte eine weitere Anforderung von hoher Relevanz dar. In diesem Abschnitt wird kurz auf die Implementation dieser Funktionalität eingegangen. Grundsätzlich konnte der Code von der ursprünglichen Version des Coding-Assistant übernommen werden. Für die Darstellung der Balken, welche die Gültigkeitsbereiche visualisieren, wurde ein Map-Objekt verwendet, welches key-value (Schlüssel-Werte-Paare) speichern kann (*MDN - Map*, 2023). Für jede

neu deklarierte Variable (bspw. `int a;`) wird ein neues Objekt in der Map `variableMap` gespeichert, welches der folgenden Abbildung entnommen werden kann.

```
▼{a: {…}}
  ▼a:
     blockLevel: 2
     color: "#FFE15D"
     height: "16.65625px"
     lineLevelEnd: 5
     lineLevelStart: 4
     margin: "49.96875px"
    ▶[[Prototype]]: Object
  ▶[[Prototype]]: Object
```

Abbildung 6-6: variableMap – Gültigkeitsbereich von Variablen

In Abbildung 6-6 stellt "a" den Schlüssel und die nachfolgenden Elemente die dazugehörigen Werte dar. In der nachfolgenden Aufzählung werden die einzelnen Werte erläutert.

- `blockLevel`: Enthält das Blocklevel, in welchem die Variable deklariert wurde. Wird die Variable beispielsweise in einer Main-Methode deklariert, welche sich wiederrum in einer Klasse befindet, wird dieser Wert auf 2 gesetzt.
- `color`: Beinhaltet die Farbe des Balkens sowie des dazugehörigen Markers als Hexacode.
- `height`: Enthält die Höhe des Balkens, welcher den Gültigkeitsbereich von Variablen darstellt. Dieser Wert wird mittels folgender Formel berechnet: `(lineLevelEnd - lineLevelStart + 1) * editorLineHeigth`. Die Variable `editorLineHeight` beinhaltet die Höhe einer Code-Zeile im Editor in Pixel.
- `lineLevelEnd`: Enthält die Zeilennummer, bei welcher der Gültigkeitsbereich der Variable endet.
- `lineLevelStart`: Enthält die Zeilennummer einer Variablendeklaration.
- `margin`: Dieser Wert beinhaltet den Abstand als "margin-top" des Balkens. Der Wert wird wie folgt berechnet: `(linelevel - 1) * editorLineHeight`. Die Variable `linelevel` beinhaltet ebenso die Zeilennummer, in welcher eine Variable deklariert wurde.

Das Fenster, welches die Balken beinhaltet setzt sich aus einem simplen <div> Element zusammen. Falls eine neue Variable deklariert wird, wird der dazugehörige Balken mit folgendem Code in diesem Fenster angezeigt. Dieses Fenster, welches sich auf der linken

Seite des Code-Editors befindet, lässt sich bei Bedarf dynamisch mit einem Button ein- und ausblenden (*CA-GB-A-1*).

```
<div ng-repeat="(key, value) in variableMap" id="{{key}}" class="var-
Scope" ng-style="{'height': value.height, 'marginTop': value.margin,
'backgroundColor': value.color}">
```

Mittels `ng-repeat` kann über alle Elemente in der `variableMap` iteriert werden (*AngularJS - ngRepeat*, o. J.). Zusätzlich ermöglicht `ng-style` das dynamische Festlegen von CSS-Designs für HTML-Elemente (*AngularJS - ngStyle*, o. J.). Dabei wird für jedes Element in der `variableMap` ein neuer Balken mit den vorher erläuterten Werten erzeugt und angezeigt.

Parallel zur Generierung der Balken werden jeweils die dazugehörigen Variablennamen mit einem Marker farblich hervorgehoben (*CA-GB-A-6*). Dies stellt sicher, dass klar ist, welcher Marker zu welcher Variable gehört (*CA-GB-NFA-1*). Es ist anzumerken, dass die Implementation dieser Funktionalität nicht näher erläutert wird. Die Abbildung 6-7 zeigt das finale Ergebnis der Implementation dieses Features.

```
1  public class Main {
2      public static void main(String[] args) {
3          System.out.println("Java ist ein Kaffee");
4          int test = 3;
5          if (test == 3) {
6              String name = "Ben";
7          }
8      }
9  }
10
```

Abbildung 6-7: Gültigkeitsbereich von Variablen

Die Umsetzung der Anforderung *CA-GB-A-3* erwies sich als unproblematisch, da mittels vorgängig beschriebener `variableMap` geprüft werden kann, ob Variablen mehrfach mit einem identischen Namen deklariert werden. Zusätzlich ermöglicht die ursprüngliche Konfiguration des Coding-Assistant eine Prüfung, ob Variablen ausserhalb ihres Gültigkeitsbereichs verwendet werden. Dadurch können für diese zwei Szenarien spezielle Chatboxen angezeigt werden, welche in Abbildung 6-8 dargestellt werden. Diese Chatboxen werden mittels unter Anforderung *CA-E-A-4* beschriebenen Vorgehens erzeugt.

Abbildung 6-8: Gültigkeitsbereich von Variablen – Fehlerchatboxen

6.3.2.1 Abweichungen

Was die Anforderung *CA-GB-A-7* anbelangt, musste festgestellt werden, dass sich deren Umsetzung als herausfordernd erweist. Das dynamische Hinzufügen von Markern für Variablennamen, wenn das Fenster, welches die Gültigkeitsbereiche beinhaltet, aktiviert ist, war nicht möglich. Es wurden diverse Möglichkeiten evaluiert, von welchen sich allerdings keine als passend erwies. Das grundlegende Problem war, dass die bereits vorhandenen Marker nicht mehr korrekt dargestellt wurden. Im Falle einer neuen Variablendeklaration wurden alle nachfolgenden, bereits vorhandenen Marker nicht an die neue Position angepasst. Dies hatte zur Folge, dass nicht mehr die Variablennamen, sondern zufällige Elemente im Code markiert wurden. Um dieses Problem beheben zu können, wird in der aktuellen Version das Fenster, welches die Gültigkeitsbereiche beinhaltet, ausgeblendet, sobald Anpassungen im Code vorgenommen werden. Nutzende müssen das Fenster dementsprechend erneut einblenden, um die Gültigkeitsbereiche sowie die dazugehörigen Marker einsehen zu können. Dies widerspricht zwar der ursprünglich erwarteten Funktionalität, ermöglicht allerdings eine durchgängig korrekte Anzeige der Marker.

6.3.3 Testing

Das Testing hat ergeben, dass sich diese Funktionalität grundsätzlich wie erwartet verhält. Die dazugehörige Dokumentation kann dem Anhang 3.2.2 entnommen werden. Die nachfolgenden Abschnitte dienen einer zusammenfassenden Beschreibung der durchgeführten Tests.

Das Fenster, welches die Gültigkeitsbereiche visualisiert, kann dynamisch mit einem Button ein- und ausgeblendet werden (*CA-GB-A-1*). Zusätzlich wird für jede neu deklarierte Variable ein neuer Balken in diesem Fenster angezeigt, welcher den Gültigkeitsbereich visualisiert (*CA-GB-A-2*). Um diese Funktionalität zu prüfen, wurden diverse Fälle von Variablendeklarationen durchgespielt. Zugleich konnte somit geprüft werden, ob die Variablennamen korrekt markiert werden (*CA-GB-A-6*). Dabei mussten die Farben der Balken mit dem dazugehörigen Marker übereinstimmen (*CA-GB-NFA-1*).

Betreffend der Anforderung *CA-GB-A-3* wurde evaluiert, ob der Coding-Assistant Chatboxen für folgende zwei Fälle anzeigt.

- Deklaration von mehr als einer Variablen, welche den identischen Namen aufweisen.
- Zugriff auf Variable ausserhalb ihres Gültigkeitsbereichs.

Das Testing hat ergeben, dass die Chatboxen grundsätzlich korrekt angezeigt werden. Jedoch hat sich ein Problem offenbart, welches den ersten Fall betrifft. Falls eine Java-Klasse eine Main-Methode sowie eine zusätzliche Methode aufweist und in beiden Methoden eine Variable mit demselben Namen deklariert wird, kennzeichnet der Coding-Assistant die zweite Deklaration als Fehler. Dieser Fehler tritt auf, da der Coding-Assistant jeweils den gesamten Code auf identische Variablennamen überprüft und nicht zwischen verschiedenen Methoden unterscheiden kann. Hieraus ergibt sich die Notwendigkeit, die Konfiguration zu dieser Anforderung anzupassen. Diese Anpassung wird nicht im Rahmen dieser Arbeit vorgenommen.

Eine weitere Problematik betrifft die Umsetzung der Anforderung *CA-GB-A-8*. Prinzipiell ist synchrones Scrollen zwischen dem Code-Editor und dem Fenster, welches die Gültigkeitsbereiche beinhaltet möglich. Falls Nutzende allerdings im Display zoomen, werden die Balken der Gültigkeitsbereiche teilweise nicht mehr korrekt dargestellt. Diese Problematik tritt beim Zoomen (> 100%, < 100%) auf.

Falls das Fenster nach dem Zoomen aktualisiert wird, werden die Gültigkeitsbereiche erneut passend dargestellt. Die Ursache für dieses Problem konnte nicht ausfindig gemacht werden. Es gilt daher, vor einer Produktivschaltung zu prüfen, wie diese Herausforderungen behoben werden können.

6.4 Visualisierung von Code-Blöcken

In diesem Kapitel wird erläutert, aus welchem Grund die Funktionalität *Visualisierung von Code-Blöcken* (CB) nicht umgesetzt wurde. Der Entscheid, diese Funktionalität nicht zu implementieren, wurde erst spät in der Entwicklungsphase getroffen. Aus diesem Grund erweist es sich als nötig, diesen Entscheid zu begründen.

6.4.1 Requirements Engineering

In den nachfolgenden drei Kapitel werden die erhobenen Anforderungen sowie deren Priorisierung erläutert.

6.4.1.1 Funktionale Anforderungen

Folgende funktionale Anforderungen wurden an den Coding-Assistant zu dieser Funktionalität erhoben.

Req. #	Beschreibung
CA-CB-A-1	Die Visualisierung soll mittels eines Buttons ein- und ausblendbar sein. Dabei werden die Code-Blöcke direkt im Code-Editor farblich hervorgehoben
CA-CB-A-2	Falls die Visualisierung von Code-Blöcken eingeblendet wird, soll der Button, um den Gültigkeitsbereich von Variablen anzuzeigen, deaktiviert werden (Gültigkeitsbereiche A-1).

Tabelle 6-10: Funktionale Anforderungen – Visualisierung von Code-Blöcken

6.4.1.2 Nicht-funktionale Anforderungen

Die folgende Tabelle veranschaulicht die nicht-funktionalen Anforderungen, welche erhoben wurden.

Req. #	Beschreibung
CA-CB-NFA-1	Für die Visualisierung von Code-Blöcken sollen Farben gewählt werden, welche die Lesbarkeit des Codes nicht beeinflussen.

Tabelle 6-11: Nicht-funktionale Anforderungen – Visualisierung von Code-Blöcken

6.4.1.3 Priorisierung der Anforderungen

Die Einteilung in die drei Kategorien der MoSCoW-Methode kann der folgenden Tabelle entnommen werden.

Req. #	Must have	Should have	Could have
CA-CB-A-1	X		
CA-CB-A-2		X	
CA-CB-NFA-1		X	

Tabelle 6-12: Priorisierung – Visualisierung von Code-Blöcken

6.4.2 Implementierung

Wie der Abbildung 3-1 in Kapitel 3.1.1 entnommen werden kann, wurden die Code-Blöcke in der ursprünglichen Version des Coding-Assistant direkt im Erklärungsfenster angezeigt. Da in der neuen Version des Codeboards der Code mittels Chatboxen erklärt wird, musste eine Alternative für die Umsetzung dieser Anforderung gefunden werden. In Zusammenarbeit mit dem Auftraggeber wurde der Entscheid getroffen, die Code-Blöcke direkt im Code-Editor hervorzuheben. Eine Analyse hat ergeben, dass die Visualisierung der Code-Blöcken anhand von Markern (analog Marker für Variablennamen) die einzige Option darstellt. Die Umsetzung erwies sich jedoch als herausfordernd, da die Marker nicht korrekt dargestellt werden konnten. Das Ziel war, die Marker wie in Abbildung 6-9 darzustellen.

```
Main.java ×
1  import java.util.Scanner;
2
3  public class Main {
4      public static void main(String[] args) {
5      int test = 3;
6      }
7  }
8
```

Abbildung 6-9: Visualisierung von Code-Blöcken (erwartete Lösung)

Das Darstellen von eingerückten Markern über mehrere Code-Zeilen war jedoch nicht möglich. Aus diesem Grund wurde der Entscheid getroffen, diese Funktionalität nicht im Rahmen dieser Arbeit zu implementieren. Eine abweichende Möglichkeit, die Code-Blöcke dennoch anzuzeigen, ist in der nachfolgenden Abbildung 6-10 illustriert.

```
1  public class Main {
2      public static void main(String[] args) {
3          System.out.println("Java ist ein Kaffee");
4          int a = 3;
5          if (a == 3) {
6              System.out.println(a);
7          } else {
8              System.out.println("a ist nicht 3");
9          }
10     }
11 }
12
```

Abbildung 6-10: Visualisierung von Code-Blöcken (finale Lösung)

Die standardmässige Konfiguration des Ace-Editors visualisiert Code-Blöcke einerseits anhand von Linien (schwarze Pfeile). Andererseits haben Nutzende die Möglichkeit, gesamte Blöcke zuzuklappen (blaue Pfeile). Die nachfolgende Abbildung visualisiert den Code, wenn einzelne Blöcke zugeklappt werden.

```
1  public class Main {
2      public static void main(String[] args) {↔}
11 }
12
```

Abbildung 6-11: Visualisierung von Code-Blöcken – Konfiguration Ace-Editor

Dies entspricht zwar nicht dem ursprünglich geplanten Vorhaben, bietet den Studierenden aber dennoch die Möglichkeit, bei Bedarf die einzelnen Code-Blöcke zu visualisieren.

6.5 Code-Review

Das Ziel des Code-Review war, gegen Ende der Entwicklungsphase, die vorgenommenen Anpassungen einem wissenschaftlichen Mitarbeiter an der Zürcher Hochschule für Angewandte Wissenschaften (ZHAW) zu präsentieren. Dabei handelt es sich um Janik Michot, welcher die an der ZHAW genutzte Version des Codeboards erweitert hat. Das Code-Review wurde nach der Integration des Coding-Assistant durchgeführt. Während circa eineinhalb Stunden wurden die vorgenommenen Anpassungen im Code diskutiert. Dieses Review wurde als wichtig erachtet, um zusätzliche Inputs zu erhalten sowie Unklarheiten besprechen zu können. Das Review verlief erfolgreich, da die neuen Funktionalitäten ohne Komplikationen präsentiert und Unklarheiten geklärt werden konnten.

7 Helpersystem - Compiler-Meldungen

In diesem Kapitel werden die einzelnen Entwicklungsschritte betreffend des Helpersystems *Compiler-Meldungen* (CM) genauer erläutert.

7.1 Requirements Engineering

Die im Anschluss folgenden drei Kapitel beinhalten die an das Helpersystem *Compiler-Meldungen* erhobenen Anforderungen sowie deren Priorisierung.

7.1.1 Funktionale Anforderungen

Es wurden folgende Anforderungen an das Helpersystem erhoben.

Req. #	Beschreibung
CM-A-1	Der Zugriff auf die Erklärungen soll über einen eigenen Reiter in der rechten Navigationsliste des Codeboards gewährleistet sein.
CM-A-3	Nach der Ausführung des Programms soll nur der erste Fehler in der Konsole erklärt werden.
CM-A-4	Es soll eine statische Chatbox eingeblendet werden, wenn Änderungen im Code vorhanden sind. Diese soll die Studierenden auf die erneute Ausführung des Programms, zur Überprüfung der Änderungen, aufmerksam machen.
CM-A-5	Die Chatbox mit der Erklärung zum Fehler soll nach erfolgreicher Ausführung des Programms entfernt werden.
CM-A-6	Falls der Code nach der Einblendung der Erklärungs-Chatbox geändert wird, soll diese ausgegraut werden.

Tabelle 7-1: Funktionale Anforderungen – Compiler-Meldungen

7.1.2 Nicht-funktionale Anforderungen

Es wurden keine nicht-funktionalen Anforderungen an dieses Helpersystem erhoben.

7.1.3 Priorisierung der Anforderungen

Die Priorisierung der funktionalen Anforderungen kann der folgenden Tabelle entnommen werden.

Req. #	Must have	Should have	Could have
CM-A-1	X		
CM-A-3	X		
CM-A-4		X	
CM-A-5	X		
CM-A-6		X	

Tabelle 7-2: Priorisierung – Compiler-Meldungen

7.2 Implementierung

In den nachfolgenden Abschnitten wird die Optimierung dieses Helpersystems zusammenfassend erläutert.

Betreffend der Anforderung *CM-A-3*, wurde ursprünglich festgehalten, alle Exceptions in der Konsole zu erklären, wie dem Anhang 1.3 entnommen werden kann. Es kann allerdings durchaus vorkommen, dass bei mehreren Fehlern im Code Exceptions angezeigt werden, welche durch Folgefehler verursacht werden. Aus diesem Grund wurde beschlossen, zukünftig, wie in der ursprünglichen Version des Codeboards, jeweils nur die erste Fehlermeldung genauer zu erklären.

Weil die ursprüngliche Konfiguration beibehalten wurde, musste die Konfiguration zur Generierung einer Chatbox für die erste Exception in der Konsole nicht angepasst werden. Änderungen, welche nachfolgend erläutert werden, mussten hingegen zu den Anforderungen *CM-A-4, CM-A-5* sowie *CM-A-6* vorgenommen werden. Nachfolgende Abbildungen (7-1, 7-2, 7-5 und 7-6) veranschaulichen die verschiedenen statischen Chatboxen (*CM-A-4*), welche bei der Nutzung dieses Helpersystems angezeigt werden.

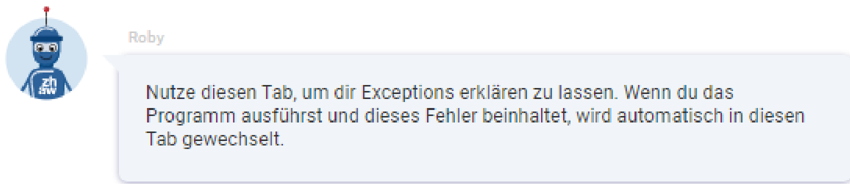


Abbildung 7-1: Statische Chatbox (1) – Compiler-Meldungen

Die Chatbox in Abbildung 7-1 wird angezeigt, falls Studierende den Compiler-Tab öffnen, ohne dass diese den Code zuvor ausgeführt oder Anpassungen vorgenommen haben.

Wird der Code hingegen mittels Run-Button ausgeführt und weist er einen Fehler auf, wird eine neue statische Chatbox angezeigt, welche zusätzlich rot schattiert ist. Zusätzlich wird die Chatbox angezeigt, welche den Fehler in Textform erläutert. Diese kann der Abbildung 7-3 entnommen werden. Es ist anzumerken, dass pro Ausführung jeweils eine Erklärungschatbox angezeigt wird. Falls der Code beispielsweise nach der ersten Ausführung angepasst und erneut ausgeführt wird und noch immer einen Fehler aufweist, verschwindet die vorherige Chatbox und es wird eine aktualisierte Chatbox angezeigt. Dasselbe gilt für den Fall, dass der Fehler behoben und der Code erneut ausgeführt wird (*CM-A-5*). In diesem Fall verschwindet die Fehler-Chatbox und die statische Chatbox in Abbildung 7-6 wird angezeigt.

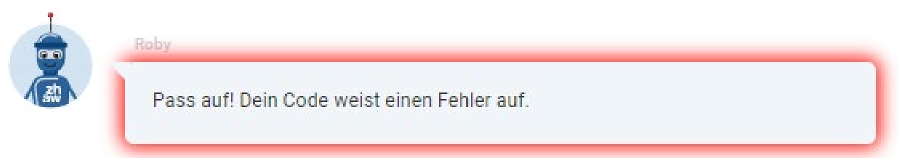


Abbildung 7-2: Statische Chatbox (3) – Compiler-Meldungen

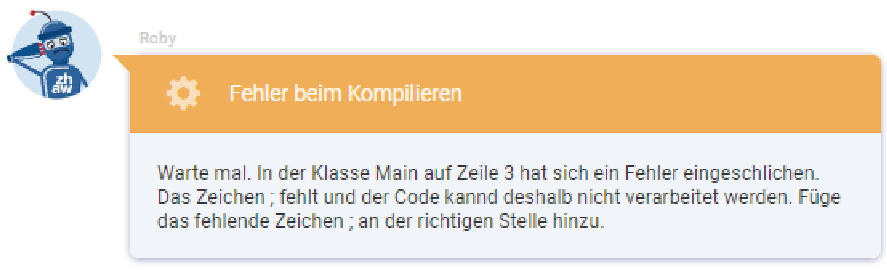


Abbildung 7-3: Fehler-Chatbox – Compiler-Meldungen

Falls Studierende Anpassungen im Code vornehmen, ändert sich der Inhalt der statischen Chatbox erneut. Die Chatbox in Abbildung 7-4 wird einerseits angezeigt, falls keine

Chatbox vorhanden ist, welche die Compiler-Meldung erläutert. Andererseits wird diese zusätzlich angezeigt, falls zuvor eine solche Erklärungschatbox generiert wurde. Im zweiten Fall wird diese Erklärungschatbox zusätzlich ausgegraut (*CM-A-6*). Dieses Verhalten ist in Abbildung 7-5 dargestellt. Dabei wurde für beide Fälle wiederrum die in Kapitel 6.2.2 unter der Anforderung *CA-E-A-3* beschriebene Funktion verwendet, welche prüft, ob Studierende Änderungen im Code-Editor vorgenommen haben.

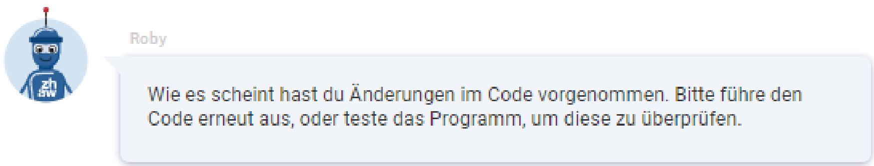


Abbildung 7-4: Statische Chatbox (2) – Compiler-Meldungen

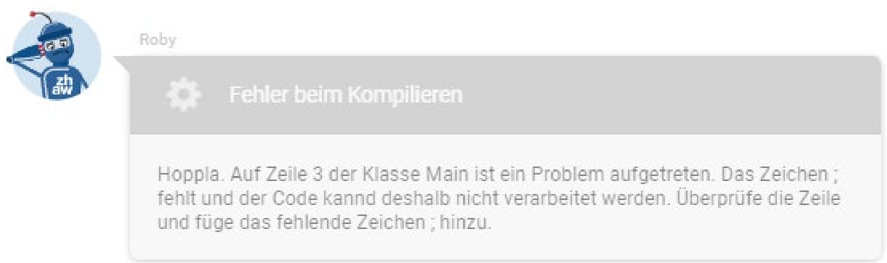


Abbildung 7-5: Fehlerchatbox ausgegraut – Compiler-Meldungen

Die Chatbox mit einer grünen Schattierung in Abbildung 7-5 wird schliesslich dargestellt, wenn die Studierenden den Code ausführen und dieser keine Fehler aufweist.

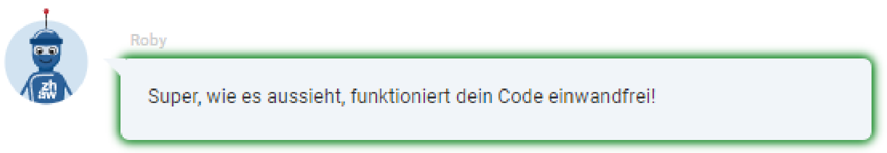


Abbildung 7-6: Statische Chatbox (4) – Compiler-Meldungen

All diese Anpassungen sollen dazu führen, dass Studierende optimal mit Fehler im Code umgehen können. Einerseits wird durch die Anpassung der statischen Chatbox sowie dem Ausgrauen der Erklärungschatbox klar, dass diese Erklärung nicht mehr aktuell ist. Andererseits sorgt das Entfernen der Chatboxen nach jeder Ausführung für eine bessere Usability, da dieser Tab somit übersichtlich bleibt.

7.2.1 Abweichungen

Aus Priorisierungsgründen wurde der Entscheid getroffen, die Anforderung *CM-A-2* nicht im Rahmen dieser Arbeit umzusetzen.

7.3 Testing

Beim Testen dieser Funktionalität wurden keinerlei Probleme identifiziert. Eine Beschreibung der durchgeführten Tests kann dem Anhang 3.3 entnommen werden. Im folgenden Abschnitt wird zusammenfassend die Vorgehensweise erläutert.

Massgebend für eine erfolgreiche Durchführung dieses Tests waren folgende Punkte.

- Wird jeweils nur der erste Fehler aus der Konsole erklärt?
- Wird nach jeder Ausführung von fehlerhaftem Code die zuvor angezeigte Chatbox entfernt und durch eine neue ersetzt?
- Wird die statische Chatbox entsprechend den verschiedenen Szenarien angezeigt?
- Wird die Chatbox, welche den Fehler aus der Konsole erklärt, bei Anpassungen im Code ausgegraut?
- Wird die Chatbox, welche den Fehler aus der Konsole erklärt, nach erfolgreicher Ausführung des Programms entfernt?

Um diese Fragen beantworten zu können, wurden diverse Test-Cases durchgespielt. Diese können dem Anhang 3.3 entnommen werden. Das Testing hat ergeben, dass alle Punkte mit "Ja" beantwortet werden können. Aus diesem Grund gelten die durchgeführten Tests als bestanden.

8 Helpersystem – Tipps

In Kapitel 8 erfolgt eine Beschreibung der Entwicklungsschritte zur Komponente *Tipps* (T). Die Schritte umfassen ein Requirements Engineering, die Dokumentation der Umsetzung sowie ein Testing.

8.1 Requirements Engineering

In den nachfolgenden drei Kapiteln befinden sich die Anforderungen sowie deren Priorisierung betreffend des Helpersystems *Tipps*.

8.1.1 Funktionale Anforderungen

Nachfolgende funktionalen Anforderungen wurden an dieses Helpersystem erhoben.

Req. #	Beschreibung
T-A-1	Der Zugriff auf die Tipps soll über einen eigenen Reiter in der rechten Navigationsliste des Codeboards gewährleistet sein.
T-A-2	Hinweise sollen abhängig vom Stand der Lösung angezeigt werden. Das bedeutet, dass Tipps gemäss aktuellem Stand der Lösung priorisiert und ausgegeben werden.
T-A-3	Falls kein Tipp für die aktuelle Lösung des/der Studierenden relevant ist, soll ein entsprechendes Pop-up-Fenster angezeigt werden, welches den/die Studierende/n darauf aufmerksam macht.

Tabelle 8-1: Funktionale Anforderungen – Tipps

8.1.2 Nicht-funktionale Anforderungen

Es wurde eine nicht-funktionale Anforderung erhoben, welche sich durch die Umsetzung der Anforderung *CB-A-6* erübrigt hat.

8.1.3 Priorisierung der Anforderungen

Die Priorisierung der Anforderungen ist der nachfolgenden Tabelle zu entnehmen.

Requirement #	Must have	Should have	Could have
T-A-1	X		
T-A-2	X		
T-A-3		X	

Tabelle 8-2: Priorisierung – Tipps

8.2 Implementierung

In den nächsten Abschnitten erfolgt eine Beschreibung der umgesetzten Funktionalitäten zur Optimierung dieses Helpersystems.

Wie bereits erwähnt, wurden Tipps in der ursprünglichen Version des Codeboards gemäss einer vorgängig definierten Reihenfolge angezeigt. Die Umsetzung der Anforderung *T-A-2* hatte zum Ziel, diese Funktionalität anzupassen und Hinweise abhängig vom Stand der Lösung der Studierenden auszugeben. Zur Erreichung dieses Ziels müssen die vorgängig definierten Tipps in der Konfigurationsdatei `codeboard.json` um zwei Eigenschaften erweitert werden. Der nachfolgende Code stellt einen Ausschnitt aus dieser Konfigurationsdatei dar, wobei die neu hinzugefügten Eigenschaften gelb markiert sind.

```
{
    "Help": {
        "tips": [
            {
                "name": "Test-Tipp",
                "note": "Verwende die Methode <tt>length()</tt>. Achte
auf die korrekte Schreibweise.",
                "mustMatch": false,
                "matching": "\\w*\\.length\\(\\)"
            }
        ]
    }
}
```

Der Eigenschaft "`mustMatch`" können dabei folgende Werte zugewiesen werden:

- `true`: Der entsprechende Tipp wird angezeigt, falls ein Teil der aktuellen Lösung des Studierenden mit dem regulären Ausdruck unter der Eigenschaft "`matching`" übereinstimmt.
- `false`: Der entsprechende Tipp wird angezeigt, falls ein Teil der aktuellen Lösung des Studierenden nicht mit dem regulären Ausdruck unter der Eigenschaft "`matching`" übereinstimmt.

Wie bereits in der vorgängigen Aufzählung erwähnt, werden unter der Eigenschaft "`matching`" die regulären Ausdrücke definiert, welche mit dem Code der Studierenden abgeglichen werden. Es ist anzumerken, dass, sobald ein Tipp angefordert wird, der Code aus dem Code-Editor ausgelesen und anschliessend mit dem regulären Ausdruck verglichen wird. Zur Veranschaulichung wird die Ausgabe des "Test-Tipps" erläutert.

Dieser Tipp wird angezeigt, falls der Code die folgende Methode nicht beinhaltet.

```
something.length()
```

Falls in diesem Fall ein Tipp angefordert wird, erscheint folgende Chatbox im Tipp-Tab.

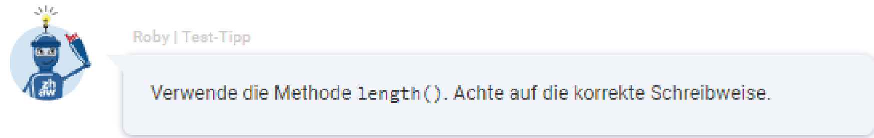


Abbildung 8-1: Hinweis-Chatbox – Tipps

Tipps werden zusätzlich zur Anzeige in einer Datenbank persistiert. Diese Funktionalität war bereits in der ursprünglichen Version des Codeboards gewährleistet. Um sicherzustellen, dass derselbe Hinweis weder zur Laufzeit noch nach einem Neustart der Applikation zweimal angezeigt wird, wurde dieser Datenbank-Eintrag um folgende Eigenschaften erweitert.

- `"tipSent"`: Diese Eigenschaft wird auf true gesetzt, sobald der Hinweis im Tipp-Tab angezeigt wird.
- `"tipIndex"`: Da die Tipps nicht mehr gemäss der vorgängig definierten Reihenfolge ausgegeben werden, muss nach der Anzeige des Tipps dessen Index gespeichert werden.

Damit nach der Produktivschaltung der neuen Version des Codeboards nicht alle Tipps in den Konfigurationsdateien mit den zwei neuen Eigenschaften ergänzt werden müssen, besteht die Möglichkeit, Tipps noch immer unabhängig vom Stand der Lösung anzeigen zu lassen. In diesem Fall sollten diese Hinweise am Ende der Konfigurationsdatei eingefügt werden. Dadurch werden vorerst die relevanten Tipps ausgegeben und im Anschluss jene Tipps, welche unabhängig vom Stand der Lösung angezeigt werden können.

Im Falle, dass Studierende einen Tipp anfordern und kein relevanter Tipp gefunden werden kann, wird ein Modal-Fenster angezeigt (*T-A-3*), welches der Abbildung 8-2 entnommen werden kann. Dieses soll verdeutlichen, dass die Studierenden die relevanten Hinweise bereits in ihrer Lösung umgesetzt haben.

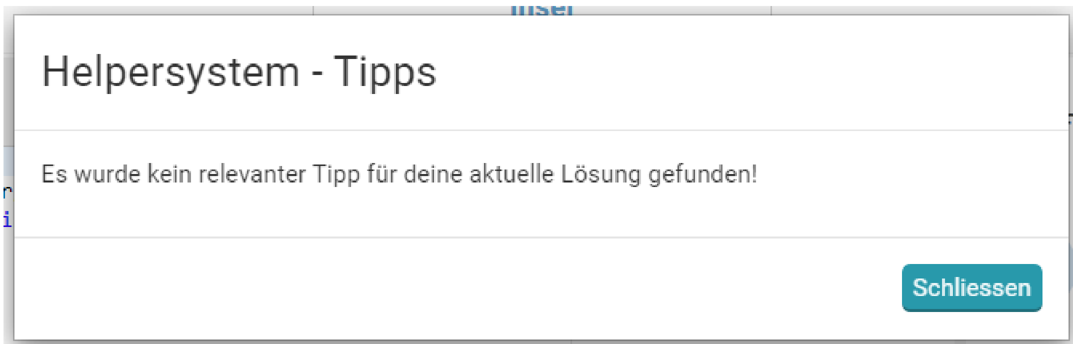


Abbildung 8-2: Modalfenster – Tipps

8.2.1 Abweichungen

Beim Helpersystem Tipps wurde die Anforderung *T-NFA-1* nicht umgesetzt und deshalb aus dem Anforderungskatalog entfernt. Ursprünglich war geplant, Tipps sowie die manuellen Fragen und Antworten im gleichen Tab anzuzeigen. Da manuelle Fragen und Antworten in der neuen Version jedoch in einem eigenen Tab angezeigt werden, hat sich die Umsetzung dieser Anforderung erübrigt. Diese Anforderung *CB-A-6* kann der Tabelle 5-1 im Kapitel 5.1.1 entnommen werden.

8.3 Testing

Das Testing dieses Helpersystems verlief erfolgreich. Die dazugehörige Dokumentation ist dem Anhang 3.4 zu entnehmen. Im folgenden Abschnitt werden die durchgeführten Tests resümierend dargestellt.

Das Ziel der Tests war es zu prüfen, ob Hinweise an Studierende abhängig vom Stand der Lösung angezeigt werden. Zusätzlich wurde kontrolliert, ob ein Modalfenster angezeigt wird, falls keine Tipps angezeigt werden können. Dafür wurde die Konfigurationsdatei `codeboard.json` vorab um Tipps ergänzt, welche die beiden möglichen Fälle ("mustMatch": true sowie "mustMatch": false) abdecken. Im Anschluss wurde der Code im Code-Editor jeweils so angepasst, dass die entsprechenden Tipps angezeigt werden. Die Überprüfung hat ergeben, dass jeweils die korrekten Tipps angezeigt und in allen anderen Fällen das Modalfenster eingeblendet wurde.

9 Resultate

Dieses Kapitel hat zum Zweck, die neue Version des Codeboards ganzheitlich zu präsentieren. Da die Beschreibung der Umsetzung der einzelnen Anforderungen bereits Inhalt der vorherigen Kapitel ist, werden in diesem Kapitel ausschliesslich die wichtigsten Funktionalitäten der einzelnen Systeme erläutert. Vorerst wird die Benutzeroberfläche des Codeboards sowie der Zugriff auf die einzelnen Helpersysteme visualisiert. Im Anschluss erfolgt eine Darstellung und Beschreibung der wichtigsten Funktionalitäten der einzelnen Helpersysteme. Zur Veranschaulichung der Unterschiede zur ursprünglichen Version des Codeboards und der Mockups wird auf die Abbildungen in Kapitel 3.1.2 sowie 4.2 verwiesen.

9.1 Codeboard

In der nachfolgenden Abbildung ist die Benutzeroberfläche der neuen Version des Codeboards abgebildet. Die einzelnen Helpersysteme können über die rot markierten Tabs, welche sich in der rechten Navigationsleiste befinden, aufgerufen werden. Zusätzlich wurde die obere Navigationsleiste grundlegend angepasst. Auf der rechten Seite (rote Markierung) befinden sich der Button, um die Gültigkeitsbereiche von Variablen anzuzeigen, sowie der Button, um den Code im Editor zu formatieren. Die Beschreibung der Buttons auf der linken Seite können dem Kapitel 3.1.2 entnommen werden, wobei anzumerken ist, dass deren Funktionalität nicht angepasst wurde. Beim aktuell angezeigten Tab handelt es sich um den Info-Tab, welcher eine Erläuterung der Funktionalitäten der einzelnen Helpersysteme sowie der zwei dazugehörigen Buttons betreffend Code-beautify und Gültigkeitsbereiche von Variablen beinhaltet. Dieser Tab kann zusätzlich mit dem "Ich brauche Hilfe"-Button in der oberen Navigationsleiste aufgerufen werden.

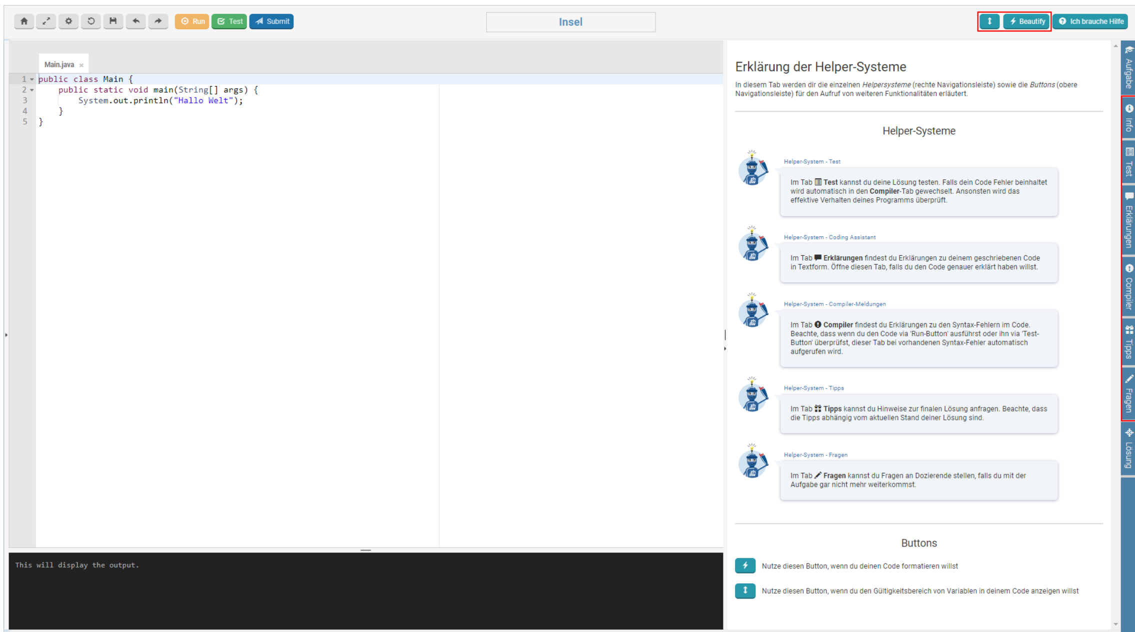


Abbildung 9-1: Codeboard – finale Version

9.2 Coding-Assistant

In der nachfolgenden Abbildung sind die grundlegenden Funktionalitäten des Coding-Assistant abgebildet. Einerseits befinden sich auf der rechten Seite unter dem "Erklärungen"-Tab die Erklärungen für den Code im Code-Editor. Die Hervorhebung einer Chatbox, falls in eine Zeile geklickt wird, ist in dieser Darstellung ebenfalls ersichtlich. Da sich der Zeiger der Maus aktuell in Zeile 5 befindet, wird die dazugehörige Chatbox grün hervorgehoben. Zusätzlich wird auf den Fehler in Zeile 9 mit einem roten Kreuz im Editor und einer dazugehörigen Chatbox aufmerksam gemacht. Das Fenster, welches die Gültigkeitsbereiche darstellt, befindet sich auf der linken Seite des Editors. Dabei stimmen die Farben der Balken mit den Farben der Marker für die Variablennamen überein. Dieses Fenster kann bei Bedarf mit in Kapitel 9.1 erwähntem Button ein- und ausgeblendet werden.

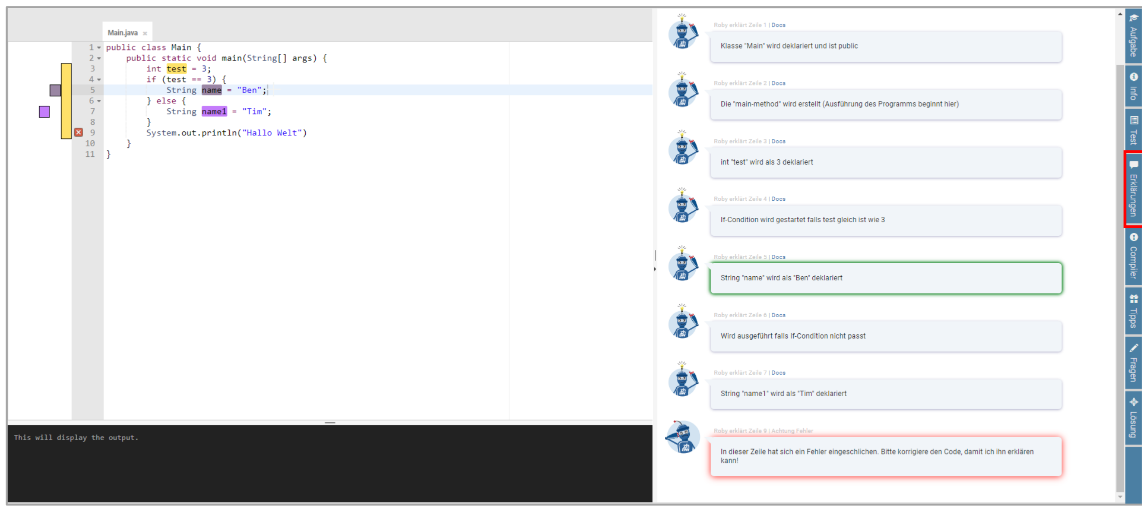


Abbildung 9-2: Coding-Assistant – finale Version

9.3 Compiler-Meldungen

Die nachfolgende Abbildung visualisiert das Helpersystem *Compiler-Meldungen*. In diesem Beispiel wurde der Code, welcher einen Fehler in Zeile 3 aufweist, ausgeführt. Folgend wird im Compiler-Tab eine Chatbox angezeigt, welche diesen Fehler, welcher zusätzlich in der Konsole (untere Seite) ersichtlich ist, erklärt. Die weiteren Funktionalitäten dieses Systems - wie das Entfernen von Chatboxen nach jeder Ausführung, oder das Ausgrauen von Chatboxen nach Änderungen im Code - können dem Kapitel 7.2 entnommen werden. Zusätzlich werden Chatboxen für Fehler, welche durch das Helpersystem *Test* erzeugt werden, in diesem Tab angezeigt. Eine beispielhafte Chatbox ist dem Kapitel 5.2 zu entnehmen.

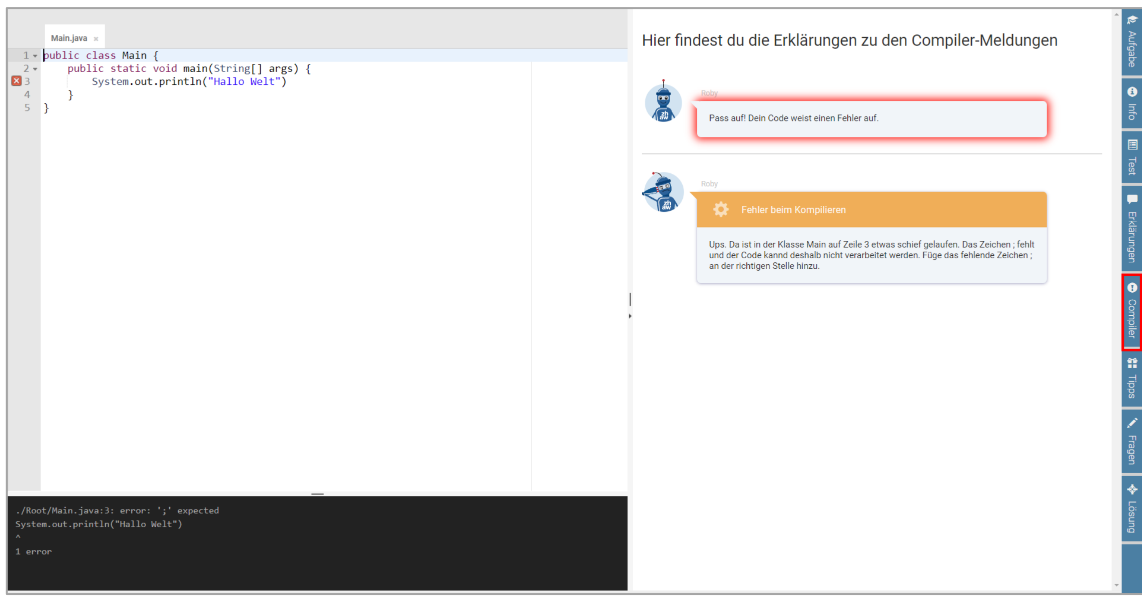


Abbildung 9-3: Compiler-Meldungen – finale Version

9.4 Tipps

Der Tipps-Tab wird in der nachfolgenden Abbildung dargestellt. In diesem können Hinweise abhängig vom aktuellen Stand der Lösung angefordert werden. Falls kein relevanter Tipp ausgegeben werden kann, wird das Modalfenster, welches in Kapitel 8.2 erläutert wird, angezeigt.

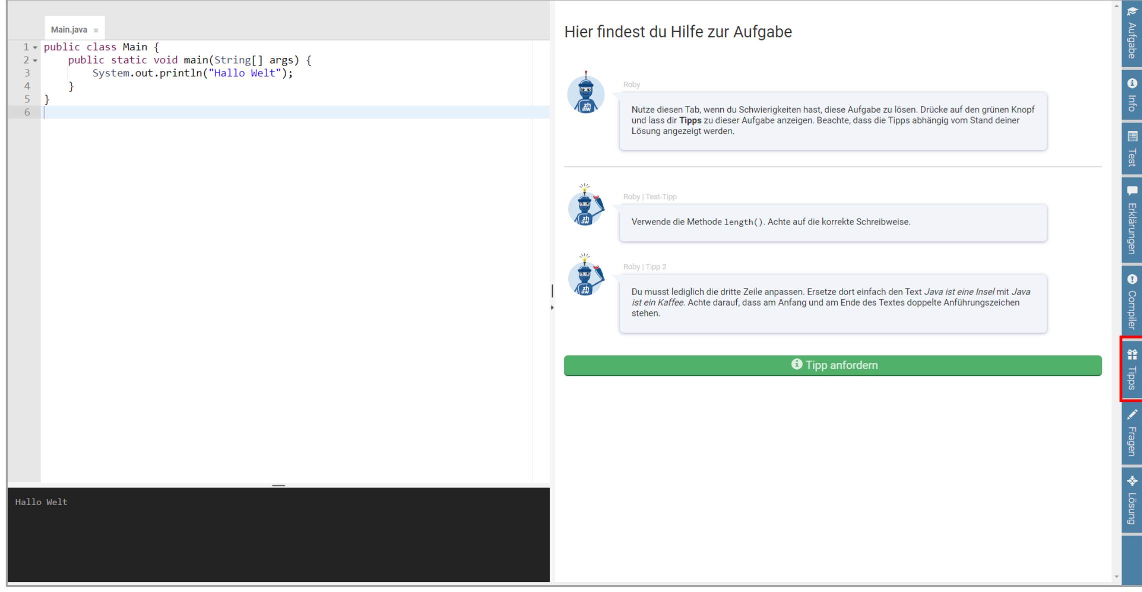


Abbildung 9-4: Tipps – finale Version

9.5 Manuelle Fragen und Antworten

Der Tab, welcher die manuellen Fragen von Studierenden sowie Antworten von Dozierenden beinhaltet, ist in der nachfolgenden Abbildung dargestellt. Fragen werden dabei mit einer Chatbox, welche eine grüne Kopfzeile aufweist, visualisiert. Die Antworten der Dozierenden befinden sich jeweils unterhalb dieser Chatbox.

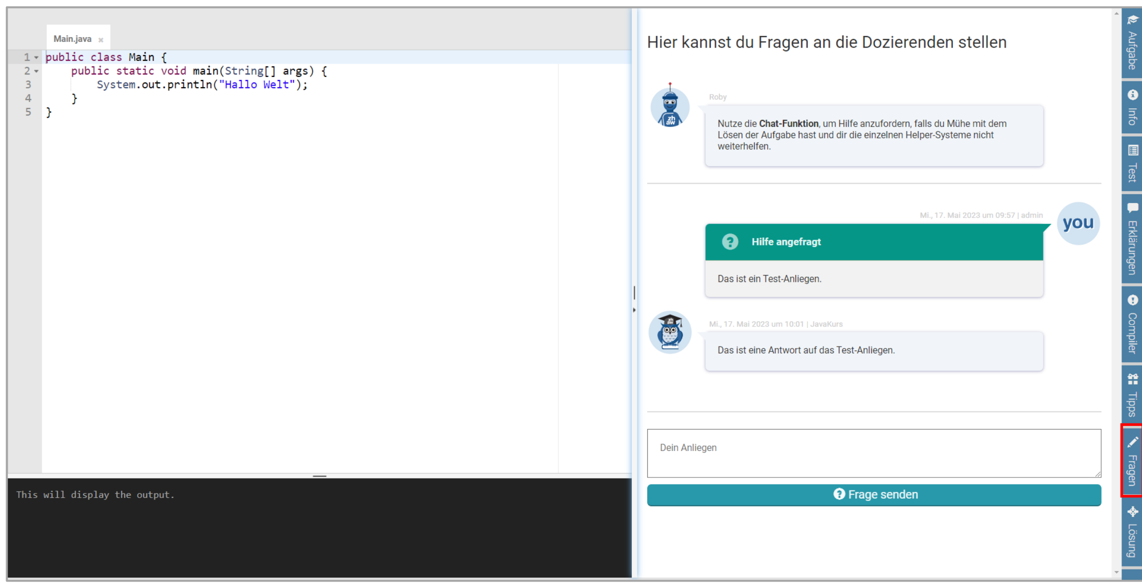


Abbildung 9-5: Manuelle Fragen – finale Version

10 Evaluation

Für eine erste Analyse der neuen Version des Codeboards wurden Usability-Tests mit fünf Probanden durchgeführt, welche alle ähnliche Programmierkenntnisse aufweisen. Im Vordergrund stand das Lösen von vordefinierten Aufgaben sowie die Beantwortung von einem Fragebogen. Das grundlegende Ziel von Usability-Tests ist es, Informationen über die Nutzbarkeit von Produkten durch die Erhebung von Daten zu sammeln (Rubin & Chisnell, 2008). Als Kriterien für die Auswertung wurden die 5E's von Whitney Quesenbery's verwendet. Diese setzen sich aus folgenden fünf Eigenschaften zusammen (Barnum, 2011, S. 108):

- *Efficient*: Dabei wird geprüft, wie schnell eine Aufgabe gelöst werden kann.
- *Effective*: Es soll geprüft werden, ob die Aufgabe vollständig abgeschlossen und die Ziele erreicht werden können.
- *Engaging*: Es erfolgt eine Prüfung, ob die Benutzerinnen und Benutzer im Allgemeinen eine positive Erfahrung mit der Applikation gemacht haben.
- *Error tolerant*: Es wird geprüft, ob bei der Benutzung Fehler auftreten und wie sich diese auf die Benutzung auswirken.
- *Easy to learn*: Dabei wird geprüft, ob sich die Applikation so verhält, wie die Benutzerinnen und Benutzer es erwarten. Zusätzlich erfolgt eine Prüfung, ob die Verwendung des Codeboards/Helpersysteme ohne Hilfe möglich ist.

10.1 Vorgehensweise

In den nachfolgenden Kapiteln wird der Ablauf der Evaluation beschrieben. Zuerst werden die Aufgaben, welche die Probanden lösten, erläutert. Im Anschluss erfolgt eine Beschreibung von Punkten, welche bei der Beobachtung relevant waren, sowie der Befragung, welche am Ende der Evaluation durchgeführt wurde.

10.1.1 Aufgaben

In den nachfolgenden drei Abschnitten werden die Aufgaben beschrieben, welche die Probanden während der Evaluation lösten. Die Aufgaben wurden vorgegeben, um der Herausforderung, dass Probanden das Produkt erkunden statt konkret ausprobieren, begegnen zu können (Hertzum, 2020, S. 38). Der Autor erwähnt, dass das Erkunden von Produkten zu oberflächlichen Resultaten führt. Zusätzlich erwähnt der Autor, dass mittels vorgegebenen Aufgaben identifiziert werden kann, ob Nutzende vom Weg abgekommen

sind oder ob sie Informationen zur Aufgabenlösung übersehen (Hertzum, 2020, S. 38). Die Aufgabenstellungen sowie dazugehörige Musterlösungen sind dem Anhang 4.1 zu entnehmen.

Überblick über die Helpersysteme: Zu Beginn sollten sich die Probandinnen und Probanden einen Überblick über die einzelnen Helper-Systeme verschaffen. Sie durften die einzelnen Systeme selbstständig ausfindig machen und sich anschliessend Überlegungen zum Anwendungszweck der einzelnen Helpersysteme machen. Im Anschluss erfolgte eine Befragung der Probanden, wobei diese die einzelnen Systeme in eigenen Worten erklären sollten. Diese Aufgabe wurde durchgeführt, um evaluieren zu können, ob Nutzende die einzelnen Helpersysteme ohne Anweisungen oder ein zusätzliches Briefing bedienen und verstehen können.

Aufgabe zur Fehlerbehebung (Fehler finden): In dieser Aufgabe mussten die Teilnehmenden in einem vorgegebenen Programm Fehler ausfindig machen und diese beseitigen. Dabei wurde das Vorgehen beobachtet sowie evaluiert, welches Helpersystem am häufigsten genutzt wurde und wie lange die Studierenden für die Lösung der Aufgabe brauchten. Dabei hatten die Probanden 5 Minuten Zeit, um diese Aufgabe zu lösen. Durch diesen Test konnte examiniert werden, wie effektiv die Nutzung der einzelnen Helpersysteme beim Finden und Beseitigen von Fehlern ist. Ziel dabei war, die Zweckmässigkeit der Integration des Coding-Assistant zu belegen. Es wurde davon ausgegangen, dass der Coding-Assistant am schnellsten und effizientesten zur Lösung von Fehlern im Programmcode beiträgt.

Allgemeine Aufgabe (Code-Ergänzung): Diese Aufgabe beinhaltete vergebenen Code, welcher zu ergänzen war. Die Probandinnen und Probanden mussten diesen so ergänzen, dass das effektive Verhalten des Programms mit dem zu erwartendem (vordefiniertem) Verhalten übereinstimmte. Der vorgegebene Code beinhaltete keine Fehler und musste lediglich erweitert werden, um das gewünschte Verhalten zu erreichen. Für die Lösung dieser Aufgabe standen den Teilnehmenden 10 Minuten zur Verfügung. Durch diese Aufgabe konnte geprüft werden, wie die Studierenden beim Lösen einer Aufgabe vorgehen und in welchem Szenario sie welches Helpersystem nutzen.

10.1.2 Beobachtung

Die Probanden wurden während der Evaluation beobachtet. Zusätzlich wurden Bildschirmaufnahmen gemacht, um die Interaktion der Probanden mit dem Codeboard im

Nachhinein analysieren zu können. Durch diese Beobachtung konnte evaluiert werden, wie die Studierenden beim Lösen der Aufgabe vorgegangen sind, was falsch gelaufen ist oder was den Probanden gefiel. Für die Beobachtung waren zusätzlich folgende Punkte relevant:

- Wie geht der Nutzende beim Lösen der Aufgabe vor?
- Wie geht der Nutzende bei Unklarheiten oder Problemen vor?
- Wird der Info-Tab vom Nutzenden aufgerufen?
- Ist dem Nutzenden klar, wie die einzelnen Helper-Systeme aufgerufen werden können?
- Wo in der Applikation sucht der Nutzende nach den einzelnen Helpersystemen?
- Wird der „Ich brauche Hilfe“-Button genutzt oder werden die Helpersysteme direkt über die einzelnen Tabs aufgerufen?
- Nutzt der Benutzer alle Helpersysteme?
- In welchem Szenario ruft der Nutzende die einzelnen Helpersysteme auf?
- Wie oft werden die einzelnen Helpersysteme genutzt?
- Welches Helpersystem wird am häufigsten genutzt?
- Versteht der Nutzende das Feedback der einzelnen Helpersysteme?
- Setzt der Nutzende die Outputs der einzelnen Helpersysteme um?
- Welches Helpersystem wird am meisten verwendet, um auf Fehler im Code aufmerksam gemacht zu werden?
- Wie schnell können Fehler im Code durch die Nutzung der Helper-Systeme gelöst werden?
- Wie lange brauchen die Probanden, um die vorgegebene Aufgabe zu lösen?

10.1.3 Befragung

Nach dem Test erfolgte eine schriftliche Befragung der Teilnehmenden. Dabei konnten Stärken und Schwächen des Produkts sowie Eindrücke und Meinungen der Probanden zur Benutzerfreundlichkeit und zum Lerneffekt evaluiert werden (Rubin & Chisnell, 2008). Der Fragebogen kann dem Anhang 4.2 entnommen werden.

10.2 Ergebnisse der Evaluation

In den nachfolgenden drei Kapiteln werden die Ergebnisse der Evaluation dargelegt. Vorab erfolgt eine kurze Beschreibung zur Auswertung der gelösten Aufgaben. Danach werden die wichtigsten Erkenntnisse, welche aus der Beobachtung der Probanden

gewonnen werden konnten, erläutert. Im Anschluss erfolgt eine Analyse der Auswertung der Antworten auf den Fragebogen.

10.2.1 Aufgaben

An dieser Stelle sei auf die Auswertung der Aufgabe hingewiesen, welche sich im Anhang 4.3 befindet. Eine grundsätzliche Erkenntnis war, dass die Aufgaben inhaltlich anspruchsvoll waren. Bis auf eine Person konnte niemand die Aufgaben in der vorgegebenen Zeit lösen. Grundsätzlich wurden jedoch zufriedenstellende Ergebnisse erzielt. Da keinerlei Vergleichsmöglichkeiten vorhanden sind (wie beispielsweise eine Auswertung von Aufgaben, welche mit Hilfe der ursprünglichen Version des Codeboards gelöst wurden), werden diese nicht weiter erläutert. Um die Kriterien, *Efficient* sowie *Effective* (Kapitel 10) stichhaltig bewerten zu können, ist eine umfassendere Evaluation erforderlich.

10.2.2 Beobachtung

Dieses Kapitel hat zum Ziel, die prägnantesten Erkenntnisse, welche aus der Beobachtung der Probanden gewonnen werden konnten, darzulegen.

Eine grundlegende Erkenntnis war, dass bei der Aufgabe *Überblick über die Helpersysteme* zu Beginn alle Probanden den Info-Tab aufgerufen und die Informationen betreffend den einzelnen Helpersystemen durchgelesen haben. Dadurch war für alle Teilnehmenden bereits anfänglich klar, welchen Anwendungszweck die einzelnen Systeme abdecken und wie diese aufgerufen werden können. Es ist anzumerken, dass der "Ich brauche Hilfe"-Button von keiner Person genutzt wurde. Nichtsdestotrotz konnten alle Teilnehmenden die Funktionalität sowie den Anwendungszweck der einzelnen Helpersysteme am Ende der ersten Aufgabe erläutern. An diesem Punkt gilt es anzumerken, dass zusätzlich die Buttons betreffend *beautify* und *Gültigkeitsbereiche von Variablen* übersehen wurden. Eine Ursache für diese Beobachtung könnte die Position dieser Buttons im Info-Tab sein. Die Chatboxen, welche die Erklärungen der einzelnen Helpersysteme beinhalten, sind alle unmittelbar erkenntlich. Um jedoch die Informationen zu den Buttons einzusehen, muss man in diesem Tab nach unten scrollen. Dies könnte begründen, weshalb nur eine Person diese Buttons im Verlauf des Testing genutzt hat.

Eine weitere Feststellung betraf das Vorgehen beim Lösen der einzelnen Aufgaben. Grundsätzlich haben alle Probanden zu Beginn der Aufgabe ohne Hilfe der Helpersysteme gearbeitet. Bei Unklarheiten oder Problemen haben sie anschliessend von den einzelnen Systemen Gebrauch gemacht. Dabei ist zu erwähnen, dass alle Probanden

unterschiedliche Helpersysteme zum Lösen der zweiten und dritten Aufgabe genutzt haben. Aus diesem Grund kann nicht gesagt werden, welches Helpersystem am häufigsten genutzt wurde. Bei der Fehlerbehebungs-Aufgabe wurde beobachtet, dass die Warnsymbole, welche fehlerhafte Code-Zeilen im Editor markieren, einen äusserst positiven Effekt hatten. Prinzipiell war den Teilnehmenden somit direkt ersichtlich, in welcher Code-Zeile sich ein Fehler eingeschlichen hatte. Dementsprechend kann die Annahme, dass der Coding-Assistant eine schnelle und effiziente Fehlerermittlung ermöglicht, bestätigt werden. Dennoch ist anzumerken, dass Fehler, welche vom Coding-Assistant nicht erkannt wurden, lediglich von einer Person behoben werden konnten. Diese hat dafür das Helpersystem *Compiler-Meldungen* genutzt und anhand dessen Erklärungen die Fehler beheben können.

Zum Schluss sei noch auf drei ergänzende Beobachtungen eingegangen. Für eine Person war es verwirrend, dass nachdem alle verfügbaren Tipps abgerufen wurden, der Button, um weitere Tipps anzufordern, verschwand. Dies wurde zur Kenntnis genommen und der Code so angepasst, dass der Button in diesem Fall deaktiviert und nicht entfernt wird. Eine weitere Erkenntnis war, dass die Erklärungen für die Compiler-Meldungen teilweise nicht korrekt angezeigt wurden. In vereinzelten Fällen konnte der fehlerhafte Code nicht korrekt erklärt werden, was zu Verwirrung führte. Diese Problematik ist jedoch nur bei einer Person aufgetreten, weswegen diese im Anschluss versucht hat, die Fehlermeldungen aus der Konsole zu verstehen und umzusetzen. Im Allgemeinen sind während der Interaktion mit dem Codeboard indes keine weiteren Fehler aufgetreten. Dementsprechend gilt das zu prüfende Kriterium *error tolerant* als erfüllt.

Zusammenfassend kann gesagt werden, dass die einzelnen Helpersysteme ihren Zweck erfüllten und eine positive Auswirkung auf die Lernerfahrung der Teilnehmenden hatten. Zudem konnten die Teilnehmenden das Feedback der Helpersysteme in den meisten Fällen erfolgreich umsetzen. Diese Aussagen werden zusätzlich mittels nachfolgend erläuterter Auswertung der Befragung bekräftigt.

10.2.3 Befragung

In diesem Kapitel werden die wichtigsten Erkenntnisse aus der Befragung zusammenfassend erläutert. Die vollständige Auswertung der Umfrage ist dem Anhang 4.4 zu entnehmen.

Erfreulich war, dass alle Teilnehmenden das Codeboard und die dazugehörigen Helpersysteme bereits nach einer kurzen Eingewöhnungszeit korrekt nutzen konnten (Abbildung 10-1 & 10-2). Es ist anzumerken, dass das Codeboard um einige neue Tabs sowie Buttons ergänzt wurde. Aus diesem Grund ist es umso erfreulicher, dass die Verwendung der einzelnen Funktionalitäten grundsätzlich selbsterklärend ist.

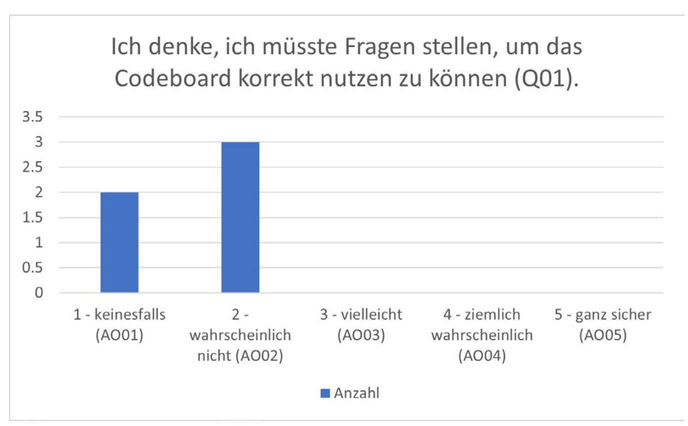


Abbildung 10-1: Auswertung – Frage Q01

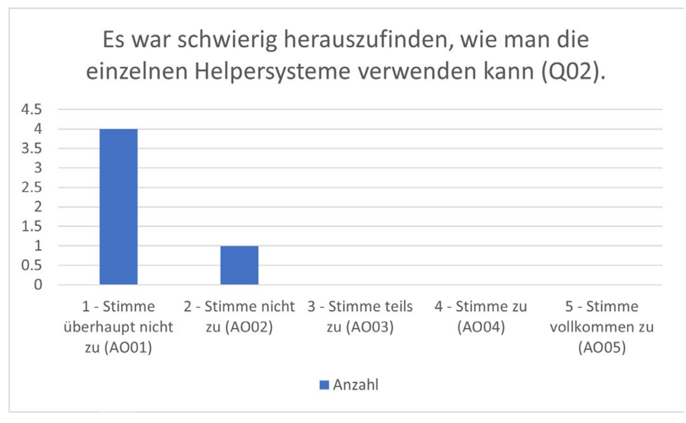


Abbildung 10-2: Auswertung – Frage Q02

Zusätzlich untermauert die nachfolgende Abbildung die vorab erläuterte Erkenntnis, denn für alle Probanden, abgesehen von einer Person, war klar, in welcher Situation sie welches Helpersystem nutzen mussten. Das Kriterium *easy to learn* gilt somit als erfüllt.

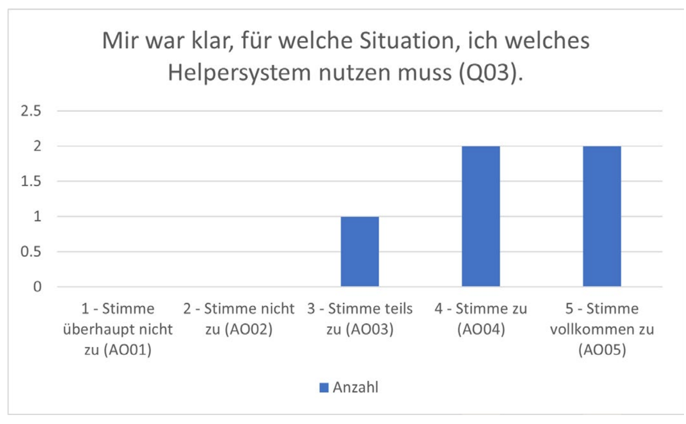


Abbildung 10-3: Auswertung – Frage Q03

Betreffend der Frage, ob die einzelnen Helpersysteme den Studierenden beim Lösen der Aufgabe Hilfe geboten haben, waren sich die Probanden grundsätzlich einer Meinung. Bis auf eine Person stimmen alle Personen dieser Aussage zu (Abbildung 10-4). Zusätzlich sind 75% der Teilnehmenden der Meinung, dass die einzelnen Helpersysteme ihre Lernerfahrung deutlich verbessert haben (Abbildung 10-5). Alle Teilnehmenden beantworteten die Frage, ob sie das Codeboard einem technisch weniger versiertem Kollegen weiterempfehlen würden, um diesem beim Erlernen der Programmiersprache Java zu unterstützen, mit "Ja". Dies sind sehr erfreuliche Ergebnisse, da das grundlegende Ziel dieser Arbeit war, das Codeboard so zu konfigurieren und zu ergänzen, um den Studierenden eine optimale Lernunterstützung zu bieten. An diesem Punkt sei noch auf das Helpersystem eingegangen, das seinen Anwendungszweck aus Sicht der Teilnehmenden am besten erfüllt. Dabei handelt es sich um den *Coding-Assistant*. Die Auswertungen, welche diese Aussage bestärken, können dem Anhang 4.4 entnommen werden.

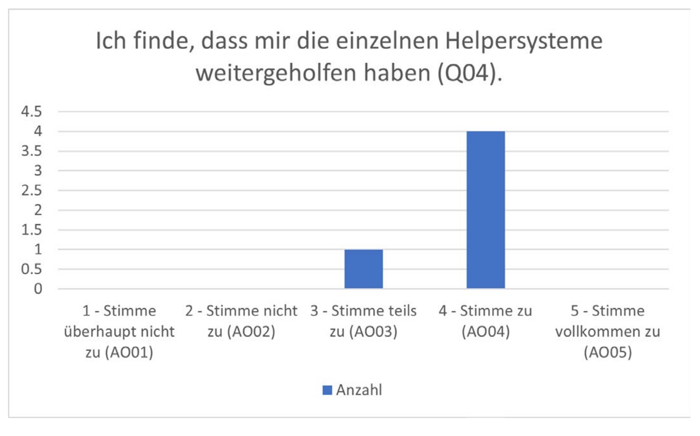


Abbildung 10-4: Auswertung – Frage Q04

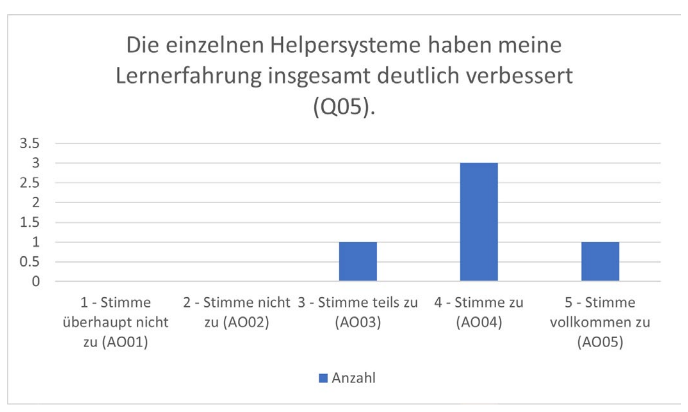


Abbildung 10-5: Auswertung – Frage Q05

Das Kriterium *engaging* kann ebenfalls als erfüllt betrachtet werden. Einerseits wird dies durch die gesamten Aussagen in diesem Kapitel sowie aus den grundlegenden Erkenntnissen gemäss Kapitel 10.2.2 begründet. Andererseits waren alle Teilnehmenden mit dem Aussehen und der Handhabung des Codebords zufrieden respektive äusserst zufrieden.

Zum Abschluss dieses Kapitels sei noch auf vereinzelte Punkte eingegangen, welche den Nutzenden bei der Interaktion mit dem Codeboard gefehlt haben. Eine Person wünschte sich genauere Erklärungen zu den Compiler-Meldungen. Zusätzlich wurde die Ausgabe von mehr Tipps gewünscht. Hierzu ist zu erwähnen, dass für die Fehlerbehebungs-Aufgabe ein Hinweis hinterlegt war und für die Code-Ergänzungsaufgabe zwei Tipps, welche bei Bedarf abhängig vom aktuellen Stand der Lösung abgerufen werden konnten. Weitere Punkte sowie Funktionalitäten, welche den Teilnehmenden beim Arbeiten mit dem Codeboard gefallen haben, sind ebenso im Anhang 4.4 ersichtlich.

10.3 Resümee

Zusammenfassend können die Evaluationsergebnisse erfolgreich taxiert werden. Das Codeboard und dessen Helpersysteme boten den Teilnehmenden eine Lernunterstützung bei Unklarheiten oder Problemen und generell beim Lösen von Aufgaben. Es gilt dennoch anzumerken, dass für stichhaltigere Ergebnisse eine umfangreichere Evaluation nötig wäre. Da zukünftig eine Produktivschaltung dieses Systems geplant ist, war diese Evaluation allerdings ein wichtiger Meilenstein, um eine erste Beurteilung der Usability und der Funktionalität zu erhalten.

11 Zusammenfassung und Ausblick

Zum Abschluss dieser Arbeit werden die wichtigsten und aussagekräftigsten Ergebnisse zusammengefasst dargelegt. Zudem ist eine resümierende Beantwortung der Forschungsfrage sowie der dazugehörigen Arbeitsfragen Bestandteil dieses Kapitels. Im Kapitel Integrationsvorbereitung werden Punkte erläutert, welche vor einer Produktivschaltung des Systems beachtet werden müssen. Abgerundet wird diese Thesis mit Handlungsempfehlungen für die zukünftige Forschung.

11.1 Fazit

Ziel dieser Arbeit war es, den Coding-Assistant in das Codeboard zu integrieren und die bestehenden Helpersysteme zu optimieren und zu erweitern. Das Ergebnis dieser Arbeit hat gezeigt, dass der Coding-Assistant in das Codeboard integriert und die bereits vorhandenen Helpersysteme mit Erfolg optimiert und erweitert werden konnten. Wie eine Analyse des Forschungsstandes ergab, gibt es aktuell keine Tools, welche die Vorteile der im Codeboard genutzten Helpersysteme übertreffen. Die neue Version des Codeboards ist bis auf wenige Anpassungen grundsätzlich bereit, die in den Kursen aktuell genutzte Version des Codeboards abzulösen. Die in dieser Arbeit durchgeführte Evaluation hat ebenfalls gezeigt, dass Lernende mit der Applikation problemlos interagieren können. In den nachfolgenden Abschnitten erfolgt eine zusammenfassende Beantwortung der Arbeitsfragen.

1. *Was sind die Anforderungen an die einzelnen Helpersysteme?*

   Die wichtigsten Anforderungen konnten in Zusammenarbeit mit dem Auftraggeber identifiziert und mit Erfolg umgesetzt werden. Eine umfassende Beschreibung dieser Anforderungen kann den Kapiteln 5, 6, 7 sowie 8 entnommen werden.

2. *Wie können die einzelnen Helpersysteme am effektivsten ergänzt und aufeinander abgestimmt werden?*

   Das Kapitel 4 zeigt Ansätze auf, wie die in der ursprünglichen Version des Codeboards vorhandenen Helpersysteme sowie der Coding-Assistant effektiv ergänzt und aufeinander abgestimmt werden können. Eine grundlegende Erkenntnis in diesem Zusammenhang war das Trennen der einzelnen Systeme für eine optimale Usability. Dadurch konnte gewährleistet werden, dass jedes Helpersystem seinen eigenen Anwendungszweck hat. Zusätzlich zeigte die Evaluation, dass sich gerade der Coding-

Assistant und das Helpersystem Compiler-Meldungen optimal ergänzen. Beispielsweise kann der Coding-Assistant Studierende auf Fehler im Code hinweisen, ohne dass diese den Code vorerst ausführen müssen. Für komplexere Fehler, welche vom Coding-Assistant nicht erkannt werden, können hingegen die Compiler-Meldungen verwendet werden.

3. *Wie kann die Wahl der zu ergänzenden und neu zu implementierenden Helpersysteme im Vergleich zu anderen Lösungen (KI) begründet werden?*

Eine präzise Beantwortung dieser Arbeitsfrage ist Teil des Kapitels 3.3. Zusammenfassend lässt sich sagen, dass die aktuell bekannten Systeme die Vorteile der in dieser Arbeit behandelten Helpersysteme nicht übertreffen. Aus diesem Grund erwies es sich als sinnvoller diese Systeme weiterhin zu verwenden und im Rahmen dieser Arbeit weiterzuentwickeln. Dennoch sei an dieser Situation anzumerken, dass gerade Systeme, welche auf künstlicher Intelligenz beruhen, zukünftig nicht ausgeblendet werden dürfen. Aus diesem Grund empfiehlt es sich, kontinuierlich zu prüfen, ob solche Systeme die Vorteile der aktuell genutzten Tools übertreffen.

4. *Wie muss die Benutzeroberfläche des Codeboards gestaltet werden, um den vollen Nutzen aller Helpersysteme ausschöpfen zu können?*

Die Überlegungen betreffend dieser Thematik erschliessen sich aus der Gesamtheit dieser Arbeit. Mit den in Kapitel 4 beschriebenen Mockups erfolgte eine erste visuelle Darstellung dieser Gedanken. Durch das aus dem kontinuierlichen Austausch mit dem Auftraggeber gewonnene Feedback konnten diese Überlegungen weiter verbessert werden. Das Kapitel 9 zeigt schlussendlich das Resultat der Umsetzung dieser zentralen Überlegungen und somit die finale Lösung. Dass die Benutzeroberfläche erfolgreich gestaltet wurde, kann der Auswertung der Evaluation in Kapitel 10 entnommen werden. Diese hat gezeigt, dass Studierende problemlos mit dem Codeboard interagieren können und beim Lösen der Aufgaben erfolgreich durch die einzelnen Helpersysteme unterstützt wurden.

5. *Welche Möglichkeiten gibt es, die einzelnen Systeme zu testen?*

Die einzelnen Systeme wurden mittels Black-Box-Tests geprüft. Dabei wurde jeweils getestet, ob das effektive Verhalten der einzelnen Tools mit dem erwarteten Verhalten übereinstimmte. Die durchgeführten Tests haben ergeben, dass die einzelnen Systeme ihren Anwendungszweck erfüllen und praktisch keinerlei Probleme bei der Nutzung

auftauchen. Eine Beschreibung der durchgeführten Tests ist Bestandteil der Kapitel 5, 6, 7 und 8, wobei eine detaillierte Dokumentation dem Anhang 3 entnommen werden kann.

6. *Wie kann ein möglicher Beitrag zur verbesserten Lernunterstützung durch die neue Version des Codeboards evaluiert und ausgewertet werden?*

Zur Beantwortung dieser Frage wurde eine Evaluation mit fünf Probanden durchgeführt. Dabei mussten diese in vorgegebener Zeit diverse vordefinierte Aufgaben lösen sowie einen anschliessenden Fragebogen ausfüllen. Diese Evaluation lieferte wertvolle Ergebnisse, um eine erste Analyse der neuen Version zu ihrer Tauglichkeit in einer realen Situation durchzuführen. Zudem ermöglichten diese Usability-Tests, Erkenntnisse zu gewinnen, welche massgeblich zur Beantwortung der Forschungsfrage beigetragen haben. Zusammenfassend kann das Ergebnis der Evaluation als erfolgreich beurteilt werden. Die Teilnehmenden haben den Anwendungszweck der einzelnen Helpersysteme korrekt erfasst und konnten diese Tools in passenden Situationen einsetzen.

Im nachfolgenden Abschnitt, welcher zugleich dieses Kapitel 11.1 abrundet, erfolgt eine zusammenfassende Beantwortung der Forschungsfrage.

> *Wie ist das bestehende Codeboard zu konfigurieren und zu erweitern, um den Studierenden eine bestmögliche Lernunterstützung und Hilfestellung bei Unklarheiten oder Problemen bieten zu können?*

Die Integration des Coding-Assistant bietet Studierenden eine ganzheitliche neue Hilfestellung bei Unklarheiten oder Problemen. Zusätzlich wurden die bereits vorhandenen Helpersysteme optimiert und erweitert, um Studierenden eine optimierte Lernunterstützung bieten zu können. Gerade die Evaluation hat gezeigt, dass die neue Version des Codeboards Studierenden bei Problemen oder Unklarheiten durchaus Unterstützung bieten kann. Dies verdeutlicht, dass bei der Konfiguration sowie der Erweiterung des bestehenden Codeboards die korrekten Entwicklungsschritte vollzogen worden sind. Obschon diese Evaluation mit einer geringen Anzahl Probanden durchgeführt wurde, konnten wertvolle Ergebnisse erzielt werden.

11.2 Integrationsvorbereitung

Angesichts des beschränkten Rahmens der Bachelorarbeit war es nicht möglich alle erhobenen Anforderungen umzusetzen sowie einige Probleme, welche sich aus der Implementation ergaben, zu beheben. Aus diesem Grund werden in diesem Kapitel wichtige Punkte genannt, welche vor einer Produktivschaltung der neuen Version des Codeboards beachtet und allenfalls verbessert werden müssen.

Ergänzung der Sprachelemente: Wie bereits erwähnt, kann der Coding-Assistant alle Sprachelemente der ersten fünf Semesterwochen des Moduls Software Engineering 1 erkennen und erklären. Daraus ergibt sich die Notwendigkeit, die aktuelle Konfiguration, um alle in den Kursen verwendeten Sprachelemente zu erweitern. Damit kann sichergestellt werden, dass der Coding-Assistant diese Elemente erklären kann und nicht korrekten Code als inkorrekt markiert.

Performance-Probleme: Ein weiterer Punkt betrifft die in Kapitel 6.2.3 erläuterten Performanceprobleme, welche auftreten, sobald viele Code-Zeilen vom Coding-Assistant erklärt werden müssen. Es muss daher eine Möglichkeit gefunden werden, wie mit dieser Problematik umgegangen werden kann.

Gültigkeitsbereich von Variablen: Bei dieser Funktionalität gibt es Komplikationen, welche vor einer Produktivschaltung geprüft werden müssen. Einerseits werden die Balken, welche die Gültigkeitsbereiche darstellen, beim Zoomen in der Applikation teilweise nicht mehr korrekt dargestellt. Andererseits werden Variablendeklarationen mit identischem Variablennamen in zwei oder mehr verschiedenen Methoden vom Coding-Assistant fälschlicherweise als inkorrekt identifiziert. Eine detaillierte Beschreibung dieser Herausforderungen kann dem Kapitel 6.3.3 entnommen werden.

Nichtsdestotrotz gilt anzumerken, dass diese Punkte eine sehr geringe Auswirkung auf die Usability haben. Das Codeboard und dessen dazugehörigen Funktionalitäten funktionieren grundsätzlich fehlerlos.

Zum Abschluss dieses Kapitels werden noch einige Erweiterungsmöglichkeiten aufgezeigt, um die Funktionalität der einzelnen Systeme weiter zu verbessern.

Die nachfolgenden drei Anforderungen betreffen den *Coding-Assistant*. In der ursprünglichen Version war es aus Usability-Gründen nicht möglich, lange Erklärungen zu verwenden. Durch die Verwendung von Chatboxen, welche die Erklärungen beinhalten,

ergibt sich die Möglichkeit, die Erklärungen betreffend Detaillierungsgrad weiter zu verbessern. Neben den Erklärungen könnte die Funktionalität *Erkennen von fehlerhaftem Code* um die nachfolgend erwähnten Punkte erweitert werden. Einerseits könnte die Konfiguration so angepasst werden, dass der Coding-Assistant die unausgewogene Nutzung von eckigen Klammern erkennt und Studierende darauf aufmerksam macht. Des Weiteren könnte implementiert werden, dass automatisch der Tab, welcher die Code-Erklärungen beinhaltet, angezeigt wird, sobald der Code einen Fehler aufweist.

Beim Helpersystem *Compiler-Meldungen* könnten ebenfalls die dazugehörigen Erklärungen weiter verbessert werden. Die aktuelle Konfiguration deckt sicherlich die am häufigsten auftretenden Fehlermeldungen ab, sollte allerdings um weitere Fehlermeldungen und dazugehörige Erklärungen ergänzt werden.

Eine weitere Anforderung, welche umgesetzt werden sollte, betrifft die Markierung des aktuell geöffneten Tabs. Da die neue Version des Codeboards doch einige Tabs aufweist, wäre es von Vorteil, wenn der aktuell geöffnete Tab markiert wird, damit ersichtlich ist, in welchem Tab man sich gerade befindet.

11.3 Handlungsempfehlungen

Die Entwicklung dieser neuen Version des Codeboards schafft grundlegende Ansätze für zukünftige Forschung, welche in diesem Kapitel erläutert werden.

Sicherlich sollte eine umfassendere Evaluation mit einer grösseren Anzahl Teilnehmenden durchgeführt werden. Dies würde einerseits ermöglichen, stichhaltigere Ergebnisse zur Frage, ob das Codeboard den Studierenden eine bestmögliche Lernunterstützung bieten kann, zu erhalten. Andererseits könnte die Usability umfassender bewertet werden, um allenfalls Herausforderungen oder Verbesserungsmöglichkeiten zu identifizieren, welche im Rahmen der in dieser Arbeit durchgeführten Evaluation nicht erkannt wurden.

Zur Frage, ob die Nutzung der neuen Version des Codeboards zu einer geringeren Anzahl manuell gestellter Fragen durch Studierende führt, müssten ebenfalls geeignete Analysen durchgeführt werden.

Zum Abschluss dieser Thesis sei noch einmal die Thematik betreffend künstlicher Intelligenz aufgegriffen. Es erachtet sich als wichtig in Zukunft kontinuierlich zu prüfen, ob eine Integration von Systemen, welche unter anderem im Stand der Technik erläutert

wurden, die Vorteile der aktuell genutzten Helpersysteme übertreffen. Sollte dies der Fall sein, empfiehlt es sich einen Austausch der Systeme in Betracht zu ziehen.

12 Literaturverzeichnis

Ace. (o. J.). *Ace - The High Performance Code Editor for the Web*. https://ace.c9.io/

Ace. (2023). *Packaged version of Ace code editor (Ajax.org Cloud9 Editor)*. https://github.com/ajaxorg/ace-builds

Altadmri, A., & Brown, N. C. C. (2015). 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (S. 522–527). ACM. https://doi.org/10.1145/2676723.2677258

AngularJS - docs. (o. J.). *AngularJS: Developer Guide: Conceptual Overview*. https://docs.angularjs.org/guide/concepts

AngularJS - ngRepeat. (o. J.). *ngRepeat - directive in module ng*. https://docs.angularjs.org/api/ng/directive/ngRepeat

AngularJS - ngStyle. (o. J.). *ngStyle - directive in module ng*. https://docs.angularjs.org/api/ng/directive/ngStyle

Antonucci, P. (2014). *AutoTeach: Incremental hints for programming exercises* [Master Thesis]. https://doi.org/10.3929/ETHZ-A-010255485

Antonucci, P., Estler, C., Nikolić, D., Piccioni, M., & Meyer, B. (2015). An Incremental Hint System For Automated Programming Assignments. In *ITICSE '15: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (S. 320–325). ACM. https://doi.org/10.1145/2729094.2742607

Assiri, F. Y., & Elazhary, H. (2020). Automated Java exceptions explanation using natural language generation techniques. *Computer Applications in Engineering Education*, *28*(3), 626–644. https://doi.org/10.1002/cae.22232

Balzert, H. (2009). *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum Akademischer Verlag. https://doi.org/10.1007/978-3-8274-2247-7

Barnum, C. M. (2011). *Usability testing essentials: Ready, Set ...Test!* Morgan Kaufmann Publishers.

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P.-M., Pearce, J. L., & Prather, J. (2019). Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *ITiCSE '19: Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (S. 177–210). ACM. https://doi.org/10.1145/3344429.3372508

Becker, B. A., Denny, P., Prather, J., Pettit, R., Nix, R., & Mooney, C. (2021). Towards Assessing the Readability of Programming Error Messages. In *ACE '21: Australasian Computing Education Conference* (S. 181–188). ACM. https://doi.org/10.1145/3441636.3442320

Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., & Mooney, C. (2016). Effective compiler error message enhancement for novice programming students. *Computer Science Education*, *26*(2–3), 148–175. https://doi.org/10.1080/08993408.2016.1225464

Berander, P., & Andrews, A. (2005). Requirements Prioritization. In A. Aurum & C. Wohlin (Hrsg.), *Engineering and Managing Software Requirements* (S. 69–94). Springer-Verlag. https://doi.org/10.1007/3-540-28244-0_4

Bonar, J., & Soloway, E. (1985). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human–Computer Interaction*, *1*(2), 133–161. https://doi.org/10.1207/s15327051hci0102_3

Brennan, K. (2009). *A guide to the Business analysis body of knowledge (BABOK guide)* (2. Aufl.). International Institute of Business Analysis.

Code GPT. (o. J.). *Code Explanation*. https://code-gpt-docs.vercel.app/docs/tutorial-features/explain

Codeanywhere. (o. J.). *Codeanywhere Docs*. https://docs.codeanywhere.com/

Codeboard.io. (o. J.). *A web-based IDE to teach programming in the classroom*. https://codeboard.io/

Coding Rooms. (o. J.). *Developer Training & Enablement Platform*. https://www.codingrooms.com/

Coding Rooms - Help. (o. J.). *Help Center*. https://app.codingrooms.com/app/course/how-to-guides-FEG3-tW/b/creating-courses-jBh11IY

Corbett, A. T., & Anderson, J. R. (2001). Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In *CHI '01: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (S. 245–252). https://doi.org/10.1145/365024.365111

Denny, P., Prather, J., Becker, B. A., Mooney, C., Homer, J., Albrecht, Z. C., & Powell, G. B. (2021). On Designing Programming Error Messages for Novices:

Readability and its Constituent Factors. In *CHI '21: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (S. 1–15). ACM. https://doi.org/10.1145/3411764.3445696

Ettles, A., Luxton-Reilly, A., & Denny, P. (2018). Common logic errors made by novice programmers. In *ACE 2018*: *Proceedings of the 20th Australasian Computing Education Conference* (S. 83–89). ACM. https://doi.org/10.1145/3160489.3160493

Everett, G. D., & McLeod, R. (2007). *Software testing: Testing across the entire software development life cycle*. IEEE Press. https://doi.org/10.1002/9780470146354

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020* (S. 1536–1547). ACL. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Gallego-Romero, J. M., Alario-Hoyos, C., Estévez-Ayres, I., & Delgado Kloos, C. (2020). Analyzing learners' engagement and behavior in MOOCs on programming with the Codeboard IDE. *Educational Technology Research and Development*, *68*(5), 2505–2528. https://doi.org/10.1007/s11423-020-09773-6

Gerdes, A., Heeren, B., Jeuring, J., & van Binsbergen, L. T. (2017). Ask-Elle: An Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education*, *27*(1), 65–100. https://doi.org/10.1007/s40593-015-0080-x

GitHub Copilot. (o. J.). *Your AI Pair Programmer*. https://github.com/features/copilot

GitHub - About branches. (o. J.). *About Branches*. https://ghdocs-prod.azurewebsites.net/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches

GitHub - About issues. (o. J.). *About Issues*. https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues

GitHub - About repositories. (o. J.). *About Repositories*. https://docs.github.com/en/repositories/creating-and-managing-repositories/about-repositories

Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). What would other programmers do: Suggesting solutions to error messages. In *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (S. 1019–1028). ACM. https://doi.org/10.1145/1753326.1753478

Hattie, J. (2010). *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. Routledge.

Hatton, S. (2008). Choosing the Right Prioritisation Method. In *19th Australian Conference on Software Engineering (Aswec 2008)* (S. 517–526). IEEE. https://doi.org/10.1109/ASWEC.2008.4483241

Hertzum, M. (2020). *Usability testing: A practitioner's guide to evaluating the user experience*. Springer. https://doi.org/10.1007/978-3-031-02227-2

Höhn, R., & Höppner, S. (Hrsg.). (2008). *Das V-Modell XT*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-30250-6

IEEE830. (1998). *IEEE Recommended Practice for Software Requirements Specifications*. IEEE. https://doi.org/10.1109/IEEESTD.1998.88286

International Organization for Standardization. (2019). *Ergonomics of human-system interaction-Part 210: Human-centred design for interactive systems* (ISO Standard No. 9241-210:2019). https://www.iso.org/standard/77520.html

Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016). Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (S. 2073–2083). ACL. https://doi.org/10.18653/v1/P16-1195

Js-beautify. (2022). *Js-Beautify*. https://www.npmjs.com/package/js-beautify

Karlsson, J., Wohlin, C., & Regnell, B. (1998). An evaluation of methods for prioritizing software requirements. *Information and Software Technology*, *39*(14–15), 939–947. https://doi.org/10.1016/S0950-5849(97)00053-0

Keuning, H., Jeuring, J., & Heeren, B. (2019). A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education*, *19*(1), 1–43. https://doi.org/10.1145/3231711

Kong, S. C., & Abelson, H. (Hrsg.). (2022). *Computational thinking education in K-12: Artificial intelligence literacy and physical computing*. The MIT Press. https://doi.org/10.7551/mitpress/13375.001.0001

Kowal, B., Włodarz, D., Brzychczy, E., & Klepka, A. (2022). Analysis of Employees' Competencies in the Context of Industry 4.0. *Energies*, *15*(19), 7142. https://doi.org/10.3390/en15197142

Kuang, L., Zhou, C., & Yang, X. (2022). Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems. *Automated Software Engineering*, *29*(2), 43. https://doi.org/10.1007/s10515-022-00341-1

Lehtola, L., Kauppinen, M., & Kujala, S. (2004). Requirements Prioritization Challenges in Practice. In F. Bomarius & H. Iida (Hrsg.), *Product Focused Software Process Improvement* (S. 497–508). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-24659-6_36

Lin, L., Huang, Z., Yu, Y., & Liu, Y. (2022). Multi-Modal Code Summarization with Retrieved Summary. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation* (S. 132–142). IEEE. https://doi.org/10.1109/SCAM55253.2022.00020

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., … Liu, S. (2021). *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. arXiv. http://arxiv.org/abs/2102.04664

McBroom, J., Koprinska, I., & Yacef, K. (2021). A Survey of Automated Programming Hint Generation: The HINTS Framework. *ACM Computing Surveys*, *54*(8), 1–27. https://doi.org/10.1145/3469885

McZara, J., Sarkani, S., Holzer, T., & Eveleigh, T. (2015). Software requirements prioritization and selection using linguistic tools and constraint solvers - A controlled experiment. *Empirical Software Engineering*, *20*(6), 1721–1761. https://doi.org/10.1007/s10664-014-9334-8

MDN - Map. (2023). *Map Object*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Meyer, B., di Milano, P., & Meyer, B. (2017). *Fourteen years of software engineering at ETH Zurich*. https://doi.org/10.48550/arxiv.1712.05078

Miller, C. S. (2014). Metonymy and reference-point errors in novice programming. *Computer Science Education*, *24*(2–3), 123–152. https://doi.org/10.1080/08993408.2014.952500

Naik, K., & Tripathy, P. (2008). *Software testing and quality assurance: Theory and practice*. John Wiley & Sons. https://doi.org/10.1002/9780470382844.ch1

Ott, C., Robins, A., & Shephard, K. (2016). Translating Principles of Effective Feedback for Students into the CS1 Context. *ACM Transactions on Computing Education*, *16*(1), 1–27. https://doi.org/10.1145/2737596

Pears, A., & Seidman, S. (2007). *A survey of literature on the teaching of introductory programming*. *39*(4), 204–223.

Pflüger, A., & Queins, S. (2014). Agile und andere Vorgehensweisen. In C. Rupp (Hrsg.), *Requirements-Engineering und -Management: Aus der Praxis von klassisch bis agil* (6. Aufl., S. 51–69). Hanser. http://doi.org/10.3139/9783446443136

Pohl, K., & Rupp, C. (2015). *Requirements engineering fundamentals: A study guide for the certified professional for requirements engineering exam, foundation level, IREB compliant* (2. Aufl.). Rocky Nook.

Price, T. W., Dong, Y., Zhi, R., Paaßen, B., Lytle, N., Cateté, V., & Barnes, T. (2019). A Comparison of the Quality of Data-Driven Programming Hint Generation Algorithms. *International Journal of Artificial Intelligence in Education*, *29*(3), 368–395. https://doi.org/10.1007/s40593-019-00177-z

Replit. (o. J.). *Welcome to Replit Docs*. https://docs.replit.com/

Ricca, F., Scanniello, G., Torchiano, M., Reggio, G., & Astesiano, E. (2010). On the effectiveness of screen mockups in requirements engineering: Results from an

internal replication. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (S. 1–10). ACM. https://doi.org/10.1145/1852786.1852809

Rice, W. (2015). *Moodle E-Learning course development: A complete guide to create and develop engaging e-learning courses with Moodle* (3. Aufl.). Packt Publishing.

Rivero, J. M., Grigera, J., Rossi, G., Robles Luna, E., Montero, F., & Gaedke, M. (2014). Mockup-Driven Development: Providing agile support for Model-Driven Web Engineering. *Information and Software Technology*, *56*(6), 670–687. https://doi.org/10.1016/j.infsof.2014.01.011

Robertson, S., & Robertson, J. (2013). *Mastering the requirements process: Getting requirements right* (3. Aufl.). Addison-Wesley.

Rubin, J., & Chisnell, D. (2008). *Handbook of usability testing: How to plan, design, and conduct effective tests* (2. Aufl.). Wiley Pub. https://doi.org/10.1177/106480469500300408

Shute, V. J. (2008). Focus on Formative Feedback. *Review of Educational Research*, *78*(1), 153–189. https://doi.org/10.3102/0034654307313795

Strickroth, S., & Pinkwart, N. (2017). Automatisierte Bewertung in der Programmierausbildung – Eine Übersicht. In O. J. Bott, P. Fricke, U. Priss, & M. Striewe (Hrsg.), *Automatisierte Bewertung in der Programmierausbildung* (S. 17–37). Waxmann Verlag GmbH.

Svensson, R. B., Gorschek, T., Regnell, B., Torkar, R., Shahrokni, A., Feldt, R., & Aurum, A. (2011). Prioritization of quality requirements: State of practice in

eleven companies. In *2011 IEEE 19th International Requirements Engineering Conference* (S. 69–78). IEEE. https://doi.org/10.1109/RE.2011.6051652

W3Schools—Arrays. (o. J.). *JavaScript Arrays*. https://www.w3schools.com/js/js_arrays.asp

W3Schools—HTML. (o. J.). *HTML DOM Element InnerHTML Property*. https://www.w3schools.com/jsref/prop_html_innerhtml.asp

W3Schools—RegExp. (o. J.). *JavaScript RegExp Reference*. https://www.w3schools.com/jsref/jsref_obj_regexp.asp

Wiegers, K. E., & Beatty, J. (2013). *Software requirements* (3. Aufl.). Microsoft Press.

Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., & Zhang, L. (2022). An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (S. 39–51). ACM. https://doi.org/10.1145/3533767.3534390

Zhang, M., Zhou, G., Yu, W., Huang, N., & Liu, W. (2023). GA-SCS: Graph-Augmented Source Code Summarization. *ACM Transactions on Asian and Low-Resource Language Information Processing*, *22*(2), 1–19. https://doi.org/10.1145/3554820

13 Anhang

Anhangsverzeichnis

Anhang 1: Anforderungen – Changelog ... 107

Anhang 1.1: Codeboard ... 107

Anhang 1.2: Hinweissystem – Coding-Assistant ... 108

Anhang 1.3: Hinweissystem – Compiler-Meldungen ... 109

Anhang 1.4: Hinweissystem – Tipps ... 110

Anhang 2: Implementationsdokumentation ... 111

Anhang 2.1: Codeboard ... 111

Anhang 2.2: Coding-Assistant ... 115

Anhang 2.2.1: JSON-Datei – explanations.json ... 120

Anhang 2.2.2: JSON-Datei – colors.json ... 122

Anhang 2.3: Compiler-Meldungen ... 122

Anhang 2.4: Tipps ... 124

Anhang 3: Tests ... 127

Anhang 3.1: Codeboard ... 127

Anhang 3.2: Coding-Assistant ... 129

Anhang 3.2.1: Erklärungen ... 129

Anhang 3.2.2: Gültigkeitsbereiche ... 132

Anhang 3.3: Compiler-Meldungen ... 135

Anhang 3.4: Tipps ... 138

Anhang 4: Evaluation ... 140

Anhang 4.1: Aufgaben ... 140

Anhang 4.1.1: Aufgabe – Fehlerbehebung ... 140

Anhang 4.1.2: Aufgabe – Code-Ergänzung ... 143

Anhang 4.2: Fragebogen... 145

Anhang 4.3: Auswertung der Aufgaben.. 147

Anhang 4.3.1: Aufgabe – Fehlerbehebung... 147

Anhang 4.3.2: Aufgabe – Code-Ergänzung.. 152

Anhang 4.4: Auswertung der Umfrage.. 157

Anhang 1: Anforderungen – Changelog

Dieser Anhang veranschaulicht den iterativ-inkrementellen Prozess der Anforderungserhebung. Dabei wird zwischen folgenden Status unterschieden:

- *Change*: Dieser Status wird für Anforderungen verwendet, welche erweitert oder angepasst wurden. Die Änderungen sind jeweils in blauer Farbe dokumentiert.
- *Remove*: Dieser Status wird verwendet, falls eine Anforderung nicht umgesetzt wurde. Die dazugehörige Anforderung wird jeweils durchgestrichen angezeigt.
- *New*: Falls im iterativen Prozess eine neue Anforderung erhoben wurde, wird ihr dieser Status zugewiesen.

Anhang 1.1: Codeboard

Req. #	Status	Requirement
CB-A-1	Change	Die Funktionalität der einzelnen Helpersysteme soll in einem zusätzlichem Tab erläutert werden. *Zusätzlich sollen die Helpersysteme direkt aus diesem Tab aufrufbar sein.*
CB-A-4	Remove	~~Fehler im Code sollen durch ein rotes Kreuz im Code-Editor in der entsprechenden Zeile markiert werden. Falls man mit dem Mauszeiger über das Kreuz fährt, soll eine mögliche Lösung angezeigt werden.~~
CB-A-5	Change	Der Code soll mittels eines Buttons formatiert werden. ~~Dieselbe Funktionalität soll mittels Button, welcher die Code-Blöcke visuell darstellt, gewährleistet sein.~~
CB-A-6	New	Der Zugriff auf die manuellen Fragen sowie Antworten auf die Fragen von Dozierenden soll über einen eigenen Tab in der rechten Navigationsleiste gewährleistet sein.
CB-A-7	Remove	~~Der Ace-Editor soll auf die neuste Version aktualisiert werden.~~

Tabelle 1-1: Changelog – Codeboard

Anhang 1.2: Hinweissystem – Coding-Assistant

	Req #	Status	Requirement
Sprachelemente und Integrationsvorbereitung	CA-SI-A-2	Remove	~~Der Coding-Assistant soll um die Sprachelemente der Semesterwochen 6 bis 14 aus dem Modul Software Engineering 1 ergänzt werden.~~
	CA-SI-A-3	Remove	~~Die bereits implementierten Sprachelemente sollen bezüglich Prägnanz, Detaillierungsgrad und Länge verbessert werden.~~
	CA-SI-A-5	Remove	~~Die Konfiguration der Gültigkeitsbereiche von Variablen soll um passende Sprachelemente erweitert werden.~~
Erklärungen	CA-E-A-4	Change	Falls die Code-Zeile Fehler aufweist, soll in der Chatbox auf diese aufmerksam gemacht werden. Zum Beispiel: «In dieser Zeile hat sich ein Fehler eingeschlichen. Bitte korrigiere den Code, damit ich ihn erklären kann.» *Zusätzlich zur Erklärung soll die dazugehörige Code-Zeile rot markiert werden. Des Weiteren soll die Chatbox erst angezeigt werden, wenn man auf die nächste Zeile wechselt*.
	CA-E-A-7	Remove	~~Chatboxen sollen auf- und zugeklappt werden können.~~
	CA-E-A-8	New	Falls kein Code im Editor vorhanden ist, soll eine statische Chatbox mit folgendem Inhalt angezeigt werden: "Bitte öffne eine Java-Datei oder schreibe Code, damit ich den Coder erklären kann".

Gültigkeitsbereich von Variablen	CA-GB-A-4	New	Marker für Variablennamen sollen nur bei Einblendung des Gültigkeitsbereichs angezeigt werden.
	CA-GB-A-5	Remove	~~Falls das Fenster mit den Gültigkeitsbereichen eingeblendet wird, soll der Button, um die Code-Blöcke zu visualisieren, deaktiviert werden (Gültigkeitsbereiche A-1).~~
	CA-GB-A-6	New	Falls der Gültigkeitsbereich eingeblendet wird, sollen für alle Variablennamen im Code-Editor Marker eingeblendet werden, damit klar ist, welche Variable zu welcher Visualisierung des Gültigkeitsbereichs gehört.
	CA-GB-A-7	Remove	~~Falls der Gültigkeitsbereich aktiviert ist, sollen Marker für neu deklarierte Variablen dynamisch hinzugefügt werden.~~
	CA-GB-A-8	New	Synchrones Scrollen im Code-Editor und im Fenster für Gültigkeitsbereiche von Variablen muss gewährleistet sein, damit Variablen und dazugehörige Balken auf einer Linie bleiben.
	CA-GB-NFA-2	Remove	~~Das Fenster, welches den Gültigkeitsbereich darstellt, soll gleich wie das Fenster des Ace-Editors aufgebaut sein.~~
Code Blöcke	CA-CB-A-1	Remove	~~Die Visualisierung soll mittels eines Buttons ein- und ausblendbar sein. Dabei werden die Code-Blöcke direkt im Code-Editor farblich hervorgehoben~~
	CA-CB-A-2	Remove	~~Falls die Visualisierung von Codeblöcken eingeblendet wird, soll der Button, um den Gültigkeitsbereich von Variablen anzuzeigen, deaktiviert werden (Gültigkeitsbereiche A-1).~~
	CA-CB-NFA-1	Remove	~~Für die Visualisierung von Code-Blöcken sollen Farben gewählt werden, welche die Lesbarkeit des Codes nicht beeinflussen.~~

Tabelle 1-2: Changelog – Coding-Assistant

Anhang 1.3: Hinweissystem – Compiler-Meldungen

Req. #	Status	Requirement
CM-A-2	Remove	~~Die Erklärungen der Compiler-Meldungen sollen so angepasst werden, so dass diese häufiger zur Lösung des Fehlers beitragen.~~

CM-A-3	Change	Nach der Ausführung des Programms ~~sollen alle~~ *soll nur der erste* Fehler im Code in Textform (Chatboxen) erklärt werden.
CM-A-4	Change	~~Es soll eine statische Chatbox mit folgendem Inhalt vor den Erklärungen angezeigt werden, falls Änderungen im Code vorhanden sind: «Nachfolgend findest du die Erklärungen für die vorhandenen Syntax Fehler in deinem Code. Falls du den Code angepasst hast, führe den Code erneut aus, um deine Änderungen zu überprüfen».~~ *Es soll eine statische Chatbox eingeblendet werden, wenn Änderungen im Code vorhanden sind. Diese soll die Studierende auf die erneute Ausführung des Programms, zur Überprüfen der Änderungen, aufmerksam machen.*
CM-A-6	New	Falls der Code nach der Einblendung der Erklärungs-Chatbox geändert wird, soll diese ausgegraut werden.

Tabelle 1-3: Changelog – Compiler-Meldungen

Anhang 1.4: Hinweissystem – Tipps

Req. #	Status	Old Requirement
T-NFA-1	Remove	~~Der Reiter, welche die Tipps beinhaltet, soll folgendermassen aufgebaut sein:~~ ~~1. Statische Chatbox mit Beschreibung~~ ~~2. Chatboxen für Tipps und Button, um Tipp anzufordern~~ ~~Eingabefeld, um manuelle Fragen stellen zu können~~
T-A-3	New	Falls kein Tipp für die aktuelle Lösung des Studierenden relevant ist, soll ein entsprechendes Pop-up-Fenster angezeigt werden, welches den Studierenden darauf aufmerksam macht.

Tabelle 1-4: Changelog - Tipps

Anhang 2: Implementationsdokumentation

Dieser Anhang bietet eine Übersicht über angepasste Dateien & deren Funktionalitäten in der Codeboard Applikation. Dabei werden für jede umgesetzte Anforderung die vorgenommenen Anpassungen aufgezeigt. Dies soll eine mögliche Integration vereinfachen, da somit nachvollzogen werden kann, welche Anpassungen im Codeboard vorgenommen wurden. Zusätzlich ist das GitHub-Projekt, welches zusätzlich ebenso eine stichwortartige Implementationsdokumentation beinhaltet, über folgenden Link aufrufbar.

> https://github.com/users/trunisam/projects/1/views/1

Das GitHub-Repository befindet sich unter nachfolgendem Link.

> https://github.com/trunisam/codeboard

Anhang 2.1: Codeboard

In der nachfolgenden Tabelle erfolgt eine zusammenfassende Beschreibung der angepassten oder neu erstellen Files betreffend dem Codeboard. Dabei wird in der Spalte *Req. #* auf die dazugehörige Anforderung verwiesen. In der Spalte *Verzeichnis\File* ist der Pfad zur angepassten, oder neu hinzugefügten Datei angegeben. Schlussendlich erfolgt in der Spalte *Implementation* eine zusammenfassende Beschreibung der vorgenommenen Anpassungen.

Req. #	Verzeichnis\File	Implementation
CB-A-1	app\views\partials\navBarRight\navBarRightInfo.html	Dieses Template dient der Anzeige der Erklärungen der einzelnen Helper-Systeme. Der dazugehörige Tab lautet "Info".
	app\scripts\controllers\ide\navBarRight\ideNavBarRightInfoCtrl.js	In diesem File werden die Chatboxen für den "Info" Tab generiert.
	app\scripts\controllers\ide.js	Die vorhandenen Tabs wurden um den Tab "`info`" ergänzt. Der Case "`help`" in der Funktion `$scope.navBarClick()` wurde angepasst, um den "Info" Tab zu öffnen.
	app\styles\ide.css	Das Stylesheet wurde um die Klassen "`.button-row`", "`.button-description`" und "`.info-heading`" ergänzt.

	app\views\partials\chat\chat.html	Erweiterung der `ng-switch-when` Attribute, um `ng-switch-when "info"`.
	app\views\partials\chat\chatLine.html	`<button>` Element, welches `openHelpTab()` Funktion aufruft, wenn auf den Button geklickt wird.
	app\scripts\directives\chatLine.js	Das Directive "`chatLine`" wurde um die Funktion `$scope.openHelpTab`() ergänzt, um aus dem Info-Tab jeweils den richtigen Tab zu öffnen. Das isolierte Scope im Directive "`chatLine`" wurde um eine `tab`-property erweitert.
CB-A-2	app\views\partials\ide.html	`ng-if` Attribut des Buttons "Hilfe Anfordern" wurde angepasst, um diesen auszublenden, falls der Info-Tab nicht angezeigt wird.
	app\scripts\controllers\ide.js	Die Methode `$http.post()` wurde angepasst, dass sich der "Compiler-Meldungen" Tab öffnet, falls ein Fehler im Code vorhanden ist. Die `$scope.requestHelp()` Funktion wurde angepasst, um den "Info" Tab zu öffnen.
CB-A-3	app\scripts\controllers\ide\navBarRight\ideNavBarRightTestCtrl.js	`$rootScope.$broadcast('compilerError')` und `$rootScope.$broadcast('noCompilerError')` in der `$scope.doTheIoTesting()` Funktion, um einen Event zu broadcasten, falls der Test Kompilierungsfehler aufweist oder keine Fehler aufweist. Die `$scope.doTheIoTesting()` Funktion wurde angepasst, um den "Compiler-Meldungen" Tab zu öffnen, falls der Code Fehler aufweist. Zusätzlich wird eine neue Chatbox mit dem Typ "`compilerTest`" erzeugt. Die `$scope.onMessageRating()` Funktion wurde entfernt, da diese nicht mehr in diesem Controller benötigt wird.
	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	Die `filterCompilerChatLines()` Funktion wurde angepasst, um Chatboxen des Typs "`compilerTest`" zu berücksichtigen. `$scope.$on('compilerError', function () {})` wurde angepasst, um Chatboxen des Typs "`compilerTest`" zu berücksichtigen.

		Die `$scope.onMessageRating()` Funktion wurde diesem Controller hinzugefügt.
	app\views\partials\chat\chat.html	Erweiterung der `ng-switch-when` Attribute, um `ng-switch-when "compilterTest"`.
	app\scripts\directives\chatLine.js	Das isolierte Scope im Directive "`chat`" wurde um eine `onMessageRating`-property erweitert.
	app\views\partials\navBarRight\navBarRightCompiler.html	on-message-rating Attribut im `<chat>` Element.
	app\views\partials\navBarRight\navBarRightTest.html	Der Code bezüglich Generierung einer Fehler-Chatbox wurde entfernt, da dieser nicht mehr benötigt wird. `ng-show="state === states.inProgress"` Attribut, um eine entsprechende Nachricht im Test-Tab anzuzeigen. `ng-show="compilation.status === ‘fail’"` Attribut, um eine entsprechende Nachricht im Test-Tab anzuzeigen.
CB-A-5	app\views\index.html	`beautify.js <script>` wurde importiert.
	app\views\partials\ide.html	Button "`codeBeautifyButton`", welcher die Funktion `navBarClick()` aufruft.
	app\scripts\controllers\ide.js	Die `navBarClick()` Funktion wurde um den Case "`beautify_code`" ergänzt, um den Code im Code-Editor zu formatieren. Das Objekt `disabledActions` wurde um die property "`beautify`" ergänzt, um den Button bei Bedarf zu deaktivieren.
	app\scripts\services\codingAssistantCodeMatchSrv.js	Der Code unter "start or end of the blocks" wurde angepasst, um die Gültigkeitsbereiche korrekt anzuzeigen, falls der Code formatiert wird (Erkennung von "} else {" & "} else if () {".
	app\db_codingassistant\explanations.json	Die regulären Ausdrücke für "newVariableSubstringMethodRegex", "newStartValueTwoDimensionalArrayRegex", "redeclareArrayRegex", "elseIfRegex" sowie "elseRegex" wurden angepasst, um erkennt zu werden.
CB-A-6	app\views\partials\navBarRight\navBarRightQuestions.html	Dieses Template dient der Anzeige der Fragen von Studierenden sowie der Antworten durch die Dozierenden. Der dazugehörige Tab lautet "Fragen".

	app\scripts\controllers\ide.js	Die vorhandenen Tabs wurden um den Tab "`questions`" ergänzt.
	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	`$scope.filteredHelpRequestChatLines` Array, um Chatboxen des Typs "`helpRequest`" und "`helpRequestAnswer`" zu speichern. `$scope.init = function() {}` Funktion wurde angepasst, dass sich der Tab "Fragen" öffnet, wenn die User-Rolle "help" ist. Die Funktion `addChatLine()` wurde angepasst, um Chatboxen des Typs "`helpRequest`" zu erkennen und zu erzeugen. Die Funktion `$scope.answerHelpRequest()` wurde angepasst, um eine Chatbox des Typs "`helpRequestAnswer`" zu erstellen. `filterHelpChatLines()` Funktion, um Chatboxen des Typs "`helpRequest`" und "`helpRequestAnswer`" zu filtern. `$scope.$watch('chatLines', function() {})`, um Änderungen im "`chatLines`" Array zu erkennen und Filter-Funktion aufzurufen.
	app\scripts\services\chatSrv.js	Die Funktion `addChatLineCard()` wurde angepasst, um Chatboxen des Typs "`helpRequest`" hinzuzufügen.
	app\views\partials\chat\chat.html	Erweiterung der `ng-switch-when` Attribute, um `ng-switch-when "helpRequest"`. Erweiterung der `ng-switch-when` Attribute, um `ng-switch-when "helpRequestAnswer"`.

Tabelle 2-1: Implementationsdokumentation – Codeboard

Anhang 2.2: Coding-Assistant

Die Files, welche betreffend dem Helpersystem Coding-Assistant angepasst, oder neu erstellt wurden, finden sich in der nachfolgenden Tabelle. Zusätzlich erfolgt in den Kapiteln 2.2.1 und 2.2.2 eine Beschreibung der JSON-Files, welche die Erklärungen, sowie die Farben für Gültigkeitsbereiche und Marker beinhalten.

Req. #	Verzeichnis\File	Implementation
CA-E-A-1	app\views\partials\navBarRight\navBarRightExplanation.html	Dieses Template dient der Anzeige der dynamisch erzeugten Chatboxen für Erklärungen von korrektem oder nicht-korrektem Code sowie der statischen Chatbox, welche angezeigt wird, falls kein Code im Code-Editor vorhanden ist. Der dazugehörige Tab lautet "Erklärungen".
	app\scripts\controllers\ide.js	Die vorhandenen Tabs wurden um den Tab "explanations" ergänzt.
CA-E-A-2	app\views\partials\chat\chatLineExplanation.html	Ein Template für Chatboxen, welche Erklärungen für syntaktisch oder semantisch korrekten Code beinhalten.
	app\views\partials\chat\chat.html	Erweiterung der ng-switch-when Attribute, um ng-switch-when "explanation".
	app\scripts\directives\chatLine.js	Neues directive "chatLineExplanation", um eine Erklärungschatbox anzuzeigen.
	app\scripts\controllers\codingAssistantMainCtrl.js	Beinhaltet das Auslesen der Daten aus dem JSON-File "explanations.json" und des Codes aus dem Code-Editor, jedes Mal, wenn der Code angepasst wird, oder ein neues File geöffnet wird. Zusätzlich werden die Chatboxen basierend aus der Rückgabe des codingAssistantCodeMatchSrv.js generiert.
	app\scripts\controllers\ide.js	`$rootScope.$broadcast('fileOpenend')` als Eventlistener, falls ein File geöffnet wird.
	app\scripts\services\aceEditorSrv.js	`service.aceChangeListener = function (aceEditor, callback) {})`, welche aufgerufen wird, falls Anpassungen im Code-Editor gemacht werden.

	app\scripts\services\codingAssistantCodeMatchSrv.js	Die gesamte Logik für die Generierung des Inhalts der Chatboxen befindet sich in diesem File.
	app\db_codingassistant\explanations.json	In diesem File befinden sich alle regulären Ausdrücke und die dazugehörige Erklärungen.
CA-E-A-3	app\scripts\controllers\codingAssistantMainCtrl.js	Die gesamte Logik für die Generierung der angepassten Chatboxen befindet sich in diesem File.
	app\scripts\services\codingAssistantCodeMatchSrv.js	Die gesamte Logik für die Anpassung des Inhalts oder das Löschen der Chatboxen befindet sich in diesem File.
CA-E-A-4	app\views\partials\chat\chatLineError.html	Ein Template für Chatboxen, welche Erklärungen für syntaktisch oder semantisch nicht korrekten Code beinhalten.
	app\views\partials\chat\chat.html	Erweiterung der `ng-switch-when` Attribute, um `ng-switch-when` "`error`".
	app\views\partials\chat\chatLine.html	<span> Element, welches "Achtung Fehler" anstelle der "Docs" im Header anzeigt, wenn es sich um eine "error"-Chatbox handelt.
	app\scripts\directives\chatLine.js	Neues directive "`chatLineError`", um eine Erklärungschatbox anzuzeigen.
	app\scripts\controllers\codingAssistantMainCtrl.js	Die Generierung der Chatboxen für fehlerhaften Code ist Teil dieses Files. Die Funktionalität betreffend Anzeige von Fehler-Chatboxen, wenn man auf eine neue Zeile wechselt, ist ebenso Teil dieses Files. Des Weiteren werden die Warn-Symbole, welche im Code-Editor bei einem Fehler angezeigt werden, in diesem File generiert, falls eine Code-Zeile einen Fehler aufweist.
	app\scripts\services\codingAssistantCodeMatchSrv.js	Die Funktionalität zur Generierung des Inhaltes der Chatboxen für fehlerhaften Code befindet sich in diesem File.

	app\styles\chat.css	`.chat-line-error` Klasse, um die Fehler-Chatbox von der Erklärungs-Chatbox differenzieren zu können.
CA-E-A-5	app\views\partials\chat\chatLineExplanation.html	Beinhaltet ein `ng-class` Attribut, welches der Chatbox die CSS-Klasse "`highlight`" hinzufügt, wenn die Bedingung erfüllt ist. Zusätzlich sorgt das Attribut "`id`" dafür, dass zur korrekten Chatbox gescrollt wird, falls man im Code-Editor in eine Zeile klickt.
	app\views\partials\chat\chatLineError.html	Beinhaltet ein `ng-class` Attribut, welches der Chatbox die CSS-Klasse "`highlight-error`" hinzufügt, wenn die Bedingung erfüllt ist. Zusätzlich sorgt das Attribut "`id`" dafür, dass zur korrekten Chatbox gescrollt wird, falls man im Code-Editor in eine Zeile klickt.
	app\views\partials\navBarRight\navBarRightExplanation.html	`cursor-position` Attribut, um den Wert der Variable für den Scope verfügbar zu machen.
	app\scripts\controllers\codingAssistantMainCtrl.js	In diesem File wurden folgende Anpassungen vorgenommen: - `AceEditorSrv.mouseDownListener(aceEditor, function (e) {})` & `AceEditorSrv.enterKeyListener(aceEditor, function (e) {})` Funktionen, um die Position des Cursors zu erfassen. - Die Methode `scrollIntoView()`, um beim Klick in eine Code-Zeile die dazugehörige Chatbox hervorzuheben.
	app\scripts\services\aceEditorSrv.js	`service.mouseDownListener = function (aceEditor, callback) {})` & `service.enterKeyListener = function (aceEditor, callback) {})` für das Erfassen der Kursorposition.
	app\scripts\directives\chatLine.js	Das isolierte Scope im Directive "`chat`" wurde um eine `cursorPosition`-property erweitert.
	app\styles\chat.css	Darin befinden sich die Klassen "`.highlight`" und "`.highlight-error`"

CA-E-A-6	app\views\partials\chat\chat.html	Das `<chat-line>` Element wurde um das Attribut "`link`" erweitert, um die Links zu den weiterführenden Dokumentationen anzeigen zu können.
	app\views\partials\chat\chatLine.html	Das Template wurde um ein `<a>` Element ("Docs") erweitert, dass angezeigt wird, wenn der Typ der Chatbox "`explanation`" ist.
	app\scripts\directives\chatLine.js	Das isolierte Scope im Directive "`chatLine`" wurde um eine `link`-property erweitert.
CA-GB-A-1	app\views\partials\ide.html	Button "`varScopeButton`" with `ng-click` Attribut, um die Funktion `navBarClick()` aufzurufen. Zusätzlich wurde das Button-Element um die Attribute `ng-if` und `ng-disabled` ergänzt, um den Button bei Bedarf zu deaktivieren. Das `<div>` Element "`RightPartofMiddlePart`" wurde um ein `ng-style` Attribut ergänzt, um das Layout der Tabs anzupassen (Korrekte Anzeige der Tabs, wenn das Fenster, welches die Gültigkeitsbereiche beinhaltet, auf- oder zugeklappt wird).
	app\scripts\controllers\ide.js	Die `navBarClick()` Funktion wurde um den Case "`show_var_scope`" ergänzt, um die `toggleMarkers()` Funktion in codingAssistantCodeMatchSrv.js und die `$rootScope.toggleVarScope()` Funktion in ide.js aufzurufen. `$rootScope.toggleVarScope()` Funktion im Controller "`RightBarCtrl`", um das Fenster für den Gültigkeitsbereich ein- und auszublenden und das Layout der Tabs anzupassen. Das Objekt `disabledActions` wurde um die property "`varScope`" ergänzt, um den Button bei Bedarf zu deaktivieren.
CA-GB-A-2	app\views\partials\ide.html	Neuer `kendo-splitter` "`innerSplitter`", um das Fenster für den Gültigkeitsbereich von Variablen und den Code-Editor horizontal zu splitten. Neues `<div>` Element "`ideVarScopePartOfMiddlePart`" mit `ng-repeat` Attribut, um die Blöcke der Gültigkeitsbereiche basierend auf der "`variableMap`" zu erzeugen.

	app\styles\ide.css	Das Stylesheet wurde um die Klassen "`.ideVarScopePartOf-MiddlePart`", "`.varScope`", "`.varScopeBlocks`" und "`.ide-Tabs-expanded`" ergänzt.
	app\scripts\control-lers\codingAssistant-MainCtrl.js	`$rootScope.variableMap = Object.fromEntries(re-sult.variableMap)`, um die Blöcke für die Gültigkeitsberei-che für die View verfügbar zu machen.
	app\scripts\services\cod-ingAssistantCodeMatch-Srv.js	Beinhaltet die Funktionalität zur Generierung der Inhalte der "`variableMap`".
	app\db_codingassis-tant\colors.json	Dieses JSON-File beinhaltet die Farben als Hexacodes, welche für die Gültigkeitsbereiche von Variablen benötigt werden.
CA-GB-A-3	app\scripts\services\co-dingAssistantCode-MatchSrv.js	Variablen "`redeclareVarErr`" und "`declareVarErr`", welche prüfen, ob eine Variable zweimal deklariert wurde, oder aus-serhalb ihres Scopes aufgerufen wird. Im Anschluss erfolgen basierend auf dem Wert der Variable spezielle Erklärungen durch den Coding-Assistant.
CA-GB-A-4	app\scripts\services\cod-ingAssistantCodeMatch-Srv.js	`service.toggleMarkers()` Funktion, welche Marker für Vari-ablennamen einblendet, falls der Button "`varScopeButton`" gedrückt wird.
CA-GB-A-6	app\scripts\services\cod-ingAssistantCodeMatch-Srv.js	Array "`storedMarkers`", welches Marker für alle Variablenna-men speichert. Variable"`styleCss`", welches das Stylesheet "`variableMar-ker.css`" beinhaltet. `styleCss.deleteRule()` Methode, um die Styles für die Mar-ker aus dem Stylesheet "`variableMarker.css`" zu löschen. `styleCss.insertRule()` Methode, welche Styles für die Marker dem Stylesheet "`variableMarker.css`" hinzufügt.
	app\db_codingassis-tant\colors.json	Dieses JSON-File beinhaltet die Farben als Hexacodes, welche für die Marker von Variablennamen benötigt werden.
CA-GB-A-8	app\scripts\services\cod-ingAssistantVar-ScopeSrv.js	Neues Directive "`syncScroll`", um synchrones Scrollen im Code-Editor und dem Fenster, welches die Gültigkeitsbereiche von Variablen beinhaltet, zu gewährleisten.

	app\views\partials\ide.html	Das `<div>` Element "`ideEditorAndConsole`" wurde um ein `sync-scroll` Attribut ergänzt.

Tabelle 2-2: Implementationsdokumentation – Coding-Assistant

Anhang 2.2.1: JSON-Datei – explanations.json

Im JSON-File `explanations.json` werden alle Erklärungen und dazugehörige reguläre Ausdrücke gespeichert. Die Datei ist in mehrere Arrays unterteilt, welche in der folgenden Tabelle erläutert werden.

Array	Beschreibung
"lines"	Enthält diverse Erklärungen.
"notCheckedLines"	Enthält Bindewörter, welche für Schleifen verwendet werden.
"expressions"	Enthält die Erklärungen für zahlreiche Operatoren.
"print"	Enthält die Erklärungen für Print-Statements.
"printExpressions"	Enthält diverse Erklärungen betreffend den Ausgaben auf die Konsole.
"varScope"	Enthält diverse Erklärungen bezüglich der Deklaration von Variablen.
"redeclareVar"	Enthält diverse Erklärungen bezüglich den Neuzuweisungen von Variablen.
"randomExpressions"	Enthält Erklärungen betreffend java.util.Random.
"scannerExpressions"	Enthält Erklärungen betreffend java.util.Scanner.
"bitOperationExpression"	Enthält Erklärungen für Bit-Operationen.
"logicOperatorExpressions"	Enthält Erklärungen für logische Operationen.

Tabelle 2-3: JSON-Datei – explanations.json (1)

Die einzelnen Erklärungen werden jeweils als Objekt im dazugehörigen Array gespeichert. Die Objekte setzten sich aus folgenden Eigenschaften zusammen.

Eigenschaft	Beschreibung
"name"	Der Name, welcher die Erklärung eindeutig identifiziert.

"regex"	Der reguläre Ausdruck, welcher dazu dient, den im Code-Editor eingegebene Code zu erkennen.
"answer"	Die Erklärung, welche bei erfolgreicher Erkennung des Codes in der Chatbox angezeigt wird. Dabei stellen die Zahlen in der Erklärung z.B. `\"'1'\"` sogenannte Capture Groups dar. Diese haben als Zweck Code aus dem Editor, wie beispielsweise einen Variablennamen, auszulesen und in der Erklärung wiederzuverwenden.
"link"	Der Link, welche auf weiterführende Dokumentationen zur dazugehörigen Erklärung verweist.
"block"	Dieser Eigenschaft können die Ausdrücke `true` oder `false` zugewiesen werden. Der Wert `true` signalisiert, dass die erklärte Code-Zeile einen neuen Code-Block startet. Dies ist für die Visualisierung der Gültigkeitsbereiche von Variablen relevant.
"type"	Der zur Erklärung dazugehörige Typ. Dieser kann verwendet werden, falls eine andere Speichermethode verwendet wird, um die einzelnen Erklärungen zu filtern. Aktuell sind folgende Typen vorhanden: - array - arrayList - bitOperator - class - condition - exception - hashMap - keyword - loop - mathMethod - method - object - operator - package - print - printExpression - random - scanner - stringMethod - variable

Tabelle 2-4: JSON-Datei – explanations.json (2)

Anhang 2.2.2: JSON-Datei – colors.json

In diesem File werden die Farben, welche für die Markierung von Variablennamen sowie der Einfärbung der Balken, welche die Gültigkeitsbereiche darstellen, benötigt werden, gespeichert. Das File besteht aus einem einzelnen Array `colors`, welchem die Farben in Form von Hexacodes hinzugefügt werden können. Die Farben werden gemäss der definierten Reihenfolge verwendet.

Anhang 2.3: Compiler-Meldungen

In der nachfolgenden Tabelle erfolgt eine Beschreibung der angepassten oder neu hinzugefügten Dateien betreffend der Implementation des Helpersystems Compiler-Meldungen.

Req. #	Verzeichnis\File	Implementation
CM-A-1	app\views\partials\navBarRight\navBarRightCompiler.html	Dieses Template dient der Anzeige der Erklärungen für Compiler-Fehlermeldungen sowie der Fehlermeldungen, welche beim Testen des Programms ausgelöst werden. Der dazugehörige Tab lautet "Compiler".
	app\views\partials\chat\chat.html	Erweiterung der ng-switch-when Attribute, um ng-switch-when "compiler".
	app\scripts\controllers\ide.js	Die vorhandenen Tabs wurden um den Tab "compiler" ergänzt. Der Typ der Chatboxen für Compiler-Fehlermeldungen wurde auf "compiler" in $http.post() angepasst.
	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	$scope.filteredCompilerChatLines Array, um Chatboxen des Typs "compiler" zu speichern. filterCompilerChatLines() Funktion, um Chatboxen des Typs "compiler" zu filtern. $scope.$watch('chatLines', function() {}), um Änderungen im "chatLines" Array zu erkennen und Filter-Funktion aufzurufen.
CM-A-4	app\views\partials\navBarRight\navBarRightCompiler.html	<before-chat> Element, welches statische Chatbox beinhaltet. message-type Attribut in <chat-line-simple> Element.

		ng-show="showCompilerIntroMessage" Attribut, um Intro-Nachricht anzuzeigen. ng-show="showCompilerIntroMessage" Attribut, um Info-Nachricht, wenn sich der Code ändert, anzuzeigen. ng-show="showCompilerIntroMessage" Attribut, um Fehler-Nachricht anzuzeigen, falls ein Fehler vorhanden ist. ng-show="showCompilerIntroMessage" Attribut, um Erfolg-Nachricht anzuzeigen, falls kein Fehler vorhanden ist.
	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	Variable "$scope.showCompilerIntroMessage", welche prüft, ob Intro-Nachricht angezeigt werden muss. Variable "$scope.showCompilerInfoMessage", welche prüft, ob Info-Nachricht angezeigt werden muss. Variable "$scope.showCompilationErrorMessage", welche prüft, ob Fehler-Nachricht angezeigt werden muss. Variable "$scope.showCompilerNoCompilationErrorMessage", welche prüft, ob Erfolg-Nachricht angezeigt werden muss. AceEditorSrv.aceChangeListener($scope.ace.editor, function() {}), welche ausgeführt wird, wenn es Änderungen im Code-Editor gibt.
	app\views\partials\chat\chatLineSimple.html	ng-class Attribut, welches der statischen Chatbox die Klasse ".highlight" oder ".highlight-error" hinzufügt, falls der Code Fehler/keine Fehler aufweist.
	app\scripts\directives\chatLine.js	Dem Directive "chatLineSimple" wurde ein isoliertes scope hinzugefügt, welches um eine messageType-property erweitert.
CM-A-5	app\scripts\controllers\ide.js	$scope.broadcast('compilerError') Event, um anzuzeigen, dass während der Kompilierung ein Fehler aufgetaucht ist. $scope.broadcast('noCompilerError') Event, um anzuzeigen, dass während der Kompilierung kein Fehler aufgetaucht ist.

	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	Variable "lastCompilerChatboxIndex", um den Index der letzten Compiler-Meldung zu speichern. $scope.$on(‘compilerError’, function() {}), welche letzte Chatbox vom Typ "compiler" entfernt, falls der Code erneut ausgeführt wird und noch immer einen Fehler beinhaltet. $scope.$on(‘noCompilerError’, function() {}), welche letzte Chatbox vom Typ "compiler" entfernt, falls der Code erneut ausgeführt wird und keinen Fehler beinhaltet.
CM-A-6	app\views\partials\navBarRight\navBarRightCompiler.html	show-compiler-info-message Attribut im <chat> Element.
	app\views\partials\chat\chatLineCard.html	ng-class Attribut, welches der Chatbox die Klasse ".gray-out-compiler-chatbox" hinzufügt, falls der Code geändert wird.
	app\scripts\directives\chatLine.js	Das isolierte scope im Directive "chat" wurde um eine showCompilerInfoMessage-property erweitert.
	app\styles\chat.css	".gray-out-compiler-chatbox" Klasse.

Tabelle 2-5: Implementationsdokumentation – Compiler-Meldungen

Anhang 2.4: Tipps

Der Anhang *Implementationsdokumentation* wird mit nachfolgender Tabelle abgeschlossen, welche die vorgenommenen Anpassungen betreffend dem Helpersystem Tipps beinhaltet.

Req. #	Verzeichnis\File	Implementation
T-A-1	app\scripts\controllers\ide.js	Die vorhandenen Tabs wurden um den Tab "tips" ergänzt.
	app\views\partials\navBarRight\navBarRightTips.html	Dieses Template dient der Anzeige von Tipps. Der dazugehörige Tab lautet "Tipps".

	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	`$scope.init = function() {}` Funktion wurde angepasst, dass sich der Tab "Tipps" öffnet. `$scope.filteredTipChatLines` Array, um Chatboxen des Typs "`hint`" zu speichern. Die Funktion `getChatLineAvatar()` wurde angepasst, um Chatboxen des Typs "`hint`" zu erkennen. Die Funktion `getNumTipsAlreadySent()` wurde angepasst, um Chatboxen des Typs "`hint`" zu erkennen. Die Funktion `addChatLine()` wurde angepasst, um Chatboxen des Typs "`hint`" zu erkennen. `filterTipChatLines()` Funktion, um Chatboxen des Typs "`hint`" zu filtern. `$scope.$watch('chatLines', function() {})`, um Änderungen im "`chatLines`" Array zu erkennen und Filter-Funktion aufzurufen.
	app\scripts\services\chatSrv.js	Die Funktion `addChatLineCard()` wurde angepasst, um Chatboxen des Typs "`hint`" hinzuzufügen.
	app\views\partials\chat\chat.html	Erweiterung der `ng-switch-when` Attribute, um `ng-switch-when` app\views\partials\chat\chatLine.html`hint`". Anpassung des Templates, um eine <chat-line-simple> anzuzeigen.
	app\views\partials\chat\chatLine.html	Das Template wurde um ein `<span>` Element erweitert, welches den Namen des Tipps anzeigt falls es sich um eine Tipp-Chatbox handelt.
	app\scripts\directives\chatLine.js	Das isolierte scope im Directive "`chatLine`" wurde um eine `cardType`-, `cardReference`- und `cardTitle`-property erweitert.
T-A-2	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	Der Zugriff auf den Ace-Editor wird über die Variable `aceEditor` sichergestellt. Die `$scope.init` Funktion wurde angepasst, um angezeigten Tipps eine `sent`-property hinzuzufügen, dass sie nicht 2x während der aktiven Sitzung angezeigt werden. Zusätzlich wurde

		`$scope.chatLines.forEach` hinzugefügt, um die `sent`-property von bereits gesendeten Tipps auf `true` zu setzen. Die Funktion `$scope.askForTip()` wurde angepasst, um Tipps abhängig vom Stand der Lösung anzuzeigen. Des Weiteren wird bei hinzugefügten Tipps die property `sent` in der Datenbank auf `true` gesetzt und die Indexposition gespeichert. Zusätzlich wurde die Funktion angepasst, dass ein Modalfenster angezeigt wird, falls kein relevanter Tipp vorhanden ist.
	app\scripts\services\chatSrv.js	Die Funktion `addChatLineCard()` wurde um 2 Parameter ("`aStatus`", "`aTipIndex`") erweitert.
	codeboard.json	Die einzelnen Tipps in der Konfigurationsdatei wurden um die propertys `mustMatch` (`true`/`false`) und `matching` (Regex) ergänzt.
T-A-3	app\scripts\controllers\ide\navBarRight\ideNavBarRightHelpCtrl.js	Die Funktion `$scope.askForTip()` wurde angepasst, um ein Modalfenster anzuzeigen, falls kein relevanter Tipp vorhanden ist.
	app\views\partials\navBarRight\navBarRightTips.html	Das Template für das Modalfenster befindet sich in diesem File.

Tabelle 2-6: Implementationsdokumentation – Tipps

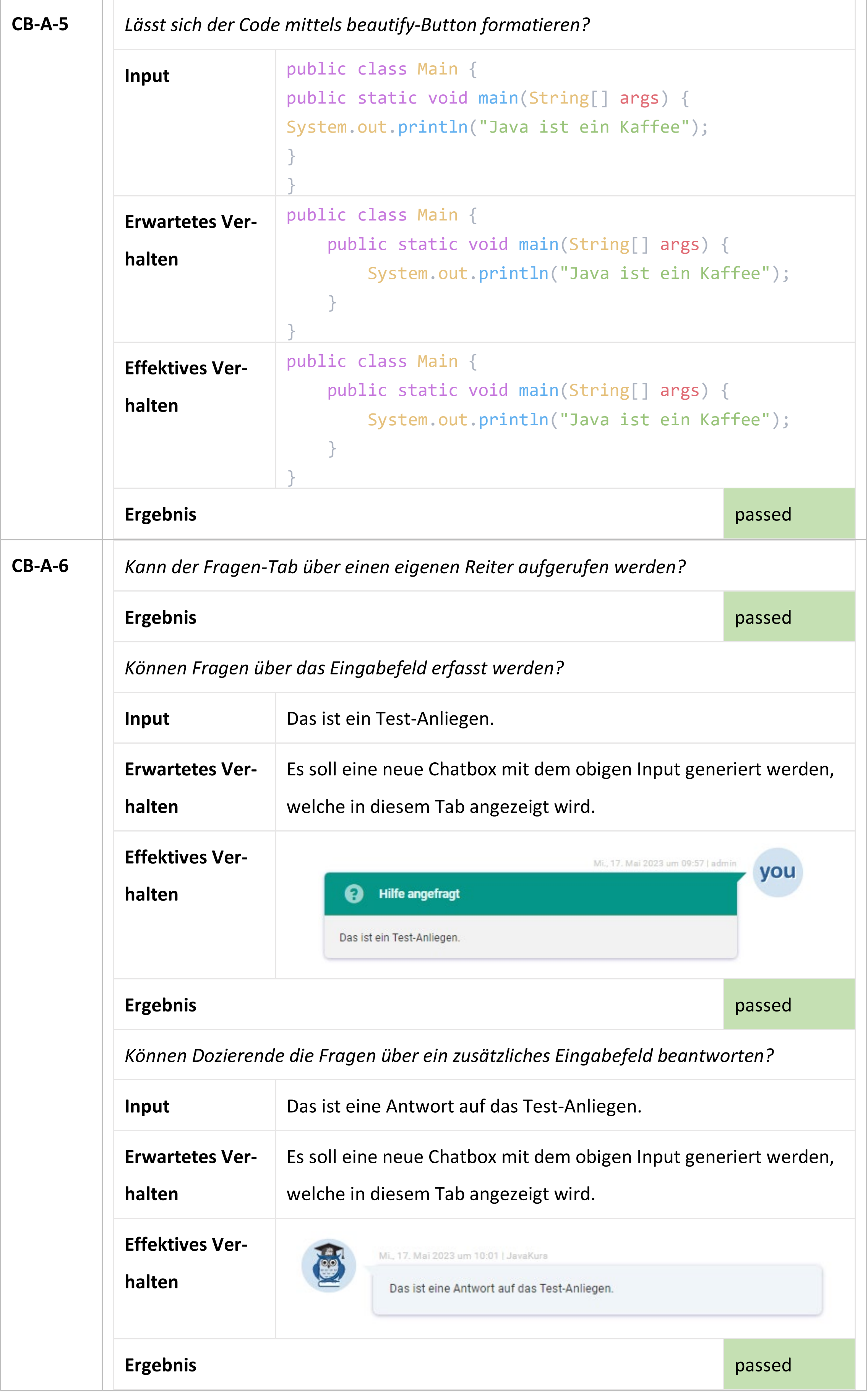
Anhang 3: Tests

In den nachfolgenden vier Kapitel werden die durchgeführten Tests betreffend dem Codeboard sowie den drei Helpersystemen veranschaulicht. Dabei werden alle erhobenen und umgesetzten Anforderungen getestet. Bei relevanten Anforderungen werden zusätzlich der Input, das erwartete Verhalten sowie das effektive Verhalten dokumentiert.

Anhang 3.1: Codeboard

In diesem Anhang werden alle erhobenen und umgesetzten Anforderungen betreffend dem Codeboard auf ihre Korrektheit überprüft.

<table>
<tr><th>Req. #</th><th colspan="2">Test</th></tr>
<tr><td rowspan="2">CB-A-1</td><td colspan="2">Lassen sich die einzelnen Tabs aus dem Info-Tab aufrufen?</td></tr>
<tr><td>Ergebnis</td><td>passed</td></tr>
<tr><td rowspan="4">CB-A-2</td><td colspan="2">Wird der Test-Tab nach Klick auf den Test-Button (obere Navigationsleiste) angezeigt und der Test ausgeführt?</td></tr>
<tr><td>Ergebnis</td><td>passed</td></tr>
<tr><td colspan="2">Wird der Compiler-Tab und die Erklärung zur Compiler-Meldung angezeigt, falls fehlerhafter Code ausgeführt wird?</td></tr>
<tr><td>Ergebnis</td><td>passed</td></tr>
<tr><td rowspan="4">CB-A-3</td><td colspan="2">Wird der Compiler-Tab und die Erklärung zur Compiler-Meldung angezeigt, falls der Code getestet wird und Fehler aufweist?</td></tr>
<tr><td>Ergebnis</td><td>passed</td></tr>
<tr><td colspan="2">Wird das IO-Testing durchgeführt (im Test-Tab), falls der Code getestet wird und keine syntaktischen oder semantischen Fehler aufweist?</td></tr>
<tr><td>Ergebnis</td><td>passed</td></tr>
</table>



CB-A-5	*Lässt sich der Code mittels beautify-Button formatieren?*	

	Input	`public class Main {`
		`public static void main(String[] args) {`
		`System.out.println("Java ist ein Kaffee");`
		`}`
		`}`

	Erwartetes Verhalten	`public class Main {`
		`    public static void main(String[] args) {`
		`        System.out.println("Java ist ein Kaffee");`
		`    }`
		`}`

	Effektives Verhalten	`public class Main {`
		`    public static void main(String[] args) {`
		`        System.out.println("Java ist ein Kaffee");`
		`    }`
		`}`

	Ergebnis	passed
CB-A-6	*Kann der Fragen-Tab über einen eigenen Reiter aufgerufen werden?*	
	Ergebnis	passed
	Können Fragen über das Eingabefeld erfasst werden?	
	Input	Das ist ein Test-Anliegen.
	Erwartetes Verhalten	Es soll eine neue Chatbox mit dem obigen Input generiert werden, welche in diesem Tab angezeigt wird.
	Effektives Verhalten	Mi., 17. Mai 2023 um 09:57 | admin you Hilfe angefragt Das ist ein Test-Anliegen.
	Ergebnis	passed
	Können Dozierende die Fragen über ein zusätzliches Eingabefeld beantworten?	
	Input	Das ist eine Antwort auf das Test-Anliegen.
	Erwartetes Verhalten	Es soll eine neue Chatbox mit dem obigen Input generiert werden, welche in diesem Tab angezeigt wird.
	Effektives Verhalten	Mi., 17. Mai 2023 um 10:01 | JavaKurs Das ist eine Antwort auf das Test-Anliegen.
	Ergebnis	passed

Tabelle 3-1: Testdokumentation – Codeboard

Anhang 3.2: Coding-Assistant

In den nachfolgenden Abschnitten befindet sich eine Dokumentation der durchgeführten Tests betreffend dem Helpersystem *Coding-Assistant*.

Anhang 3.2.1: Erklärungen

In der nachfolgenden Tabelle sind die durchgeführten Tests betreffend *Erklärungen* aufgelistet.

<table>
<tr><th>Req. #</th><th colspan="3">Test</th></tr>
<tr><td rowspan="2">CA-E-A-1</td><td colspan="3">Können die Erklärungen über den Tab "Erklärungen" aufgerufen werden?</td></tr>
<tr><td colspan="2">Ergebnis</td><td>passed</td></tr>
<tr><td rowspan="5">CA-E-A-2</td><td colspan="3">Wird für jede Code-Zeile im Code-Editor eine neue Erklärungs-Chatbox angezeigt?</td></tr>
<tr><td>Input</td><td colspan="2"><pre>public class Main {
    public static void main(String[] args) {
        System.out.println("Java ist ein Kaffee");
    }
}</pre></td></tr>
<tr><td>Erwartetes Verhalten</td><td colspan="2">Es sollen drei Chatboxen erzeugt werden.</td></tr>
<tr><td>Effektives Verhalten</td><td colspan="2">Roby erklärt Zeile 1 | Docs
Klasse "Main" wird deklariert und ist public
Roby erklärt Zeile 2 | Docs
Die "main-method" wird erstellt (Ausführung des Programms beginnt hier)
Roby erklärt Zeile 3 | Docs
""Java ist ein Kaffee"" wird auf der Konsole ausgegeben (mit Zeilenumbruch)</td></tr>
<tr><td colspan="2">Ergebnis</td><td>passed</td></tr>
<tr><td rowspan="4">CA-E-A-3</td><td colspan="3">Passt sich der Inhalt der Chatboxen an Änderungen im Code an?</td></tr>
<tr><td colspan="2">Ergebnis</td><td>passed</td></tr>
<tr><td colspan="3">Wird die dazugehörige Chatbox entfernt, falls der Code auf einer Zeile gelöscht wird?</td></tr>
<tr><td colspan="2">Ergebnis</td><td>passed</td></tr>
</table>

CA-E-A-4		
	Erkennt der Coding-Assistant Fehler im Code und macht Studierende darauf aufmerksam?	

	Input	`public class Main {`
		`    public static void main(String[] args)`
		`        System.out.println("Java ist ein Kaffee")`
		`    }`
		`}`

	Erwartetes Verhalten	Es sollen zwei Chatboxen erzeugt werden, welche die Studierenden auf Fehler im Code (Zeile 2 und 3 hinweisen).
	Effektives Verhalten	Roby erklärt Zeile 1 | Docs Klasse "Main" wird deklariert und ist public Roby erklärt Zeile 2 | Achtung Fehler In dieser Zeile hat sich ein Fehler eingeschlichen. Bitte korrigiere den Code, damit ich ihn erklären kann! Roby erklärt Zeile 3 | Achtung Fehler In dieser Zeile hat sich ein Fehler eingeschlichen. Bitte korrigiere den Code, damit ich ihn erklären kann!
	Ergebnis	passed
	Wird zusätzlich zur Fehler-Chatbox die fehlerhafte Code-Zeile direkt im Code-Editor rot markiert?	

	Input	`public class Main {`
		`    public static void main(String[] args)`
		`        System.out.println("Java ist ein Kaffee")`
		`    }`
		`}`

	Erwartetes Verhalten	Die Zeilen 2 und 3 sollen im Editor rot markiert werden.

	Effektives Verhalten	`1 public class Main {`
		`2     public static void main(String[] args)`
		`3         System.out.println("Java ist ein Kaffee")`
		`4     }`
		`5 }`

	Ergebnis	passed
	Werden die Chatboxen und Markierungen erst angezeigt, wenn Studierende auf eine neue Code-Zeile wechseln?	
	Ergebnis	passed

CA-E-A-5	*Wird die dazugehörige Erklärungs-Chatbox bei Klick in eine Code-Zeile hervorgehoben?*		

	Input	`public class Main {`
		`    public static void main(String[] args) {`
		`        System.out.println("Java ist ein Kaffee");`
		`    }`
		`}`	

	Erwartetes Verhalten	Falls man im Editor in Zeile 2 klickt, soll, die dazugehörige Chatbox farblich hervorgehoben werden.	
	Effektives Verhalten	Roby erklärt Zeile 1 | Docs Klasse "Main" wird deklariert und ist public Roby erklärt Zeile 2 | Docs Die "main-method" wird erstellt (Ausführung des Programms beginnt hier) Roby erklärt Zeile 3 | Docs '"Java ist ein Kaffee"' wird auf der Konsole ausgegeben (mit Zeilenumbruch)	
	Ergebnis		passed
	Wird die dazugehörige Fehler-Chatbox bei Klick in eine Code-Zeile hervorgehoben		

	Input	`public class Main {`
		`    public static void main(String[] args)`
		`        System.out.println("Java ist ein Kaffee");`
		`    }`
		`}`	

	Erwartetes Verhalten	Falls man im Editor in Zeile 2 (Fehler) klickt soll die dazugehörige Chatbox farblich hervorgehoben werden.	
	Effektives Verhalten	Roby erklärt Zeile 1 | Docs Klasse "Main" wird deklariert und ist public Roby erklärt Zeile 2 | Achtung Fehler In dieser Zeile hat sich ein Fehler eingeschlichen. Bitte korrigiere den Code, damit ich ihn erklären kann! Roby erklärt Zeile 3 | Docs '"Java ist ein Kaffee"' wird auf der Konsole ausgegeben (mit Zeilenumbruch)	
	Ergebnis		passed
	Wird bei vielen Chatboxen bei Klick in eine Zeile zur entsprechenden Chatbox gescrollt?		

	Ergebnis	passed
CA-E-A-6	*Werden beim Klick auf das Wort "Docs" im Header der Chatbox die dazugehörigen Links zu weiterführenden Dokumentationen aufgerufen?*	
	Ergebnis	passed
CA-E-A-8	*Wird eine statische Chatbox angezeigt, falls kein Code im Editor vorhanden ist?*	
	Input	-
	Erwartetes Verhalten	Die statische Chatbox soll angezeigt werden.
	Effektives Verhalten	Roby Bitte öffne eine Java-Datei oder schreibe Code, damit ich den Coder erklären kann
	Ergebnis	passed
CA-E-NFA-1	*Werden die Chatboxen mit einer möglichst geringen Verzögerung angezeigt?*	
	Ergebnis	passed

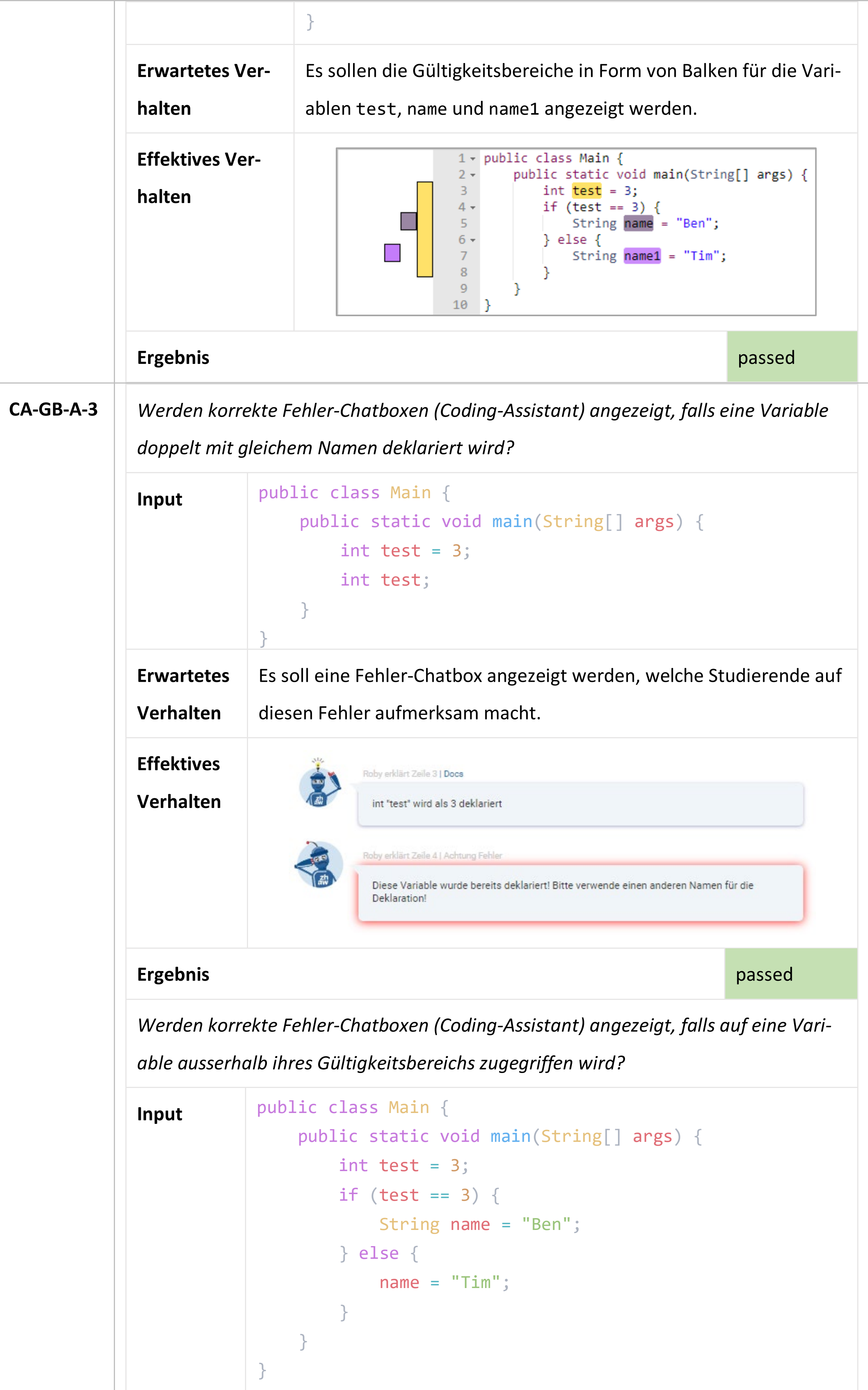
Tabelle 3-2: Testdokumentation – Coding-Assistant – Erklärungen

Anhang 3.2.2: Gültigkeitsbereiche

In den folgenden Tabellen befinden sich die durchgeführten Tests betreffend der Funktionalität *Gültigkeitsbereiche von Variablen.*

Req. #	Test	
CA-GB-A-1	*Lässt sich das Fenster, welches die Gültigkeitsbereiche beinhaltet dynamisch mittels eines Buttons ein- und ausblenden?*	
	Ergebnis	passed
CA-GB-A-2	*Werden die Gültigkeitsbereiche korrekt dargestellt?*	
	Input	(siehe Code unten)

```
public class Main {
    public static void main(String[] args) {
        int test = 3;
        if (test == 3) {
            String name = "Ben";
        } else {
            String name1 = "Tim";
        }
    }
```



```
}
```

Erwartetes Verhalten

Es sollen die Gültigkeitsbereiche in Form von Balken für die Variablen `test`, `name` und `name1` angezeigt werden.

Effektives Verhalten

```
1  public class Main {
2      public static void main(String[] args) {
3          int test = 3;
4          if (test == 3) {
5              String name = "Ben";
6          } else {
7              String name1 = "Tim";
8          }
9      }
10 }
```

Ergebnis: passed

CA-GB-A-3

Werden korrekte Fehler-Chatboxen (Coding-Assistant) angezeigt, falls eine Variable doppelt mit gleichem Namen deklariert wird?

Input

```
public class Main {
    public static void main(String[] args) {
        int test = 3;
        int test;
    }
}
```

Erwartetes Verhalten

Es soll eine Fehler-Chatbox angezeigt werden, welche Studierende auf diesen Fehler aufmerksam macht.

Effektives Verhalten

Ergebnis: passed

Werden korrekte Fehler-Chatboxen (Coding-Assistant) angezeigt, falls auf eine Variable ausserhalb ihres Gültigkeitsbereichs zugegriffen wird?

Input

```
public class Main {
    public static void main(String[] args) {
        int test = 3;
        if (test == 3) {
            String name = "Ben";
        } else {
            name = "Tim";
        }
    }
}
```

ID			
	Erwartetes Verhalten	Es soll eine Fehler-Chatbox für Zeile 7 angezeigt werden, da auf diese Variable nicht zugegriffen werden kann.	
	Effektives Verhalten	Roby erklärt Zeile 7 | Achtung Fehler Du probierst auf eine Variable zuzugreifen, welche nocht nicht deklariert wurde, oder sich ausserhalb des Scopes befindet!	
	Ergebnis		passed
CA-GB-A-4	*Werden Marker für Variablennamen nur bei Einblendung des Fensters, welches die Gültigkeitsbereiche beinhaltet, angezeigt?*		
	Ergebnis		passed
CA-GB-A-6	*Werden alle Variablennamen mit einem Marker markiert (siehe CA-GB-A-2 für eine Visualisierung)?*		
	Ergebnis		passed
CA-GB-A-8	*Ist synchrones Scrollen zwischen dem Code-Editor und dem Fenster für Gültigkeitsbereiche möglich?*		
	Ergebnis		passed
CA-GB-NFA-1	*Stimmen die Farben der Marker mit den Farben der Balken der Gültigkeitsbereiche überein?*		
	Ergebnis		passed

Tabelle 3-3: Testdokumentation – Coding-Assistant – Gültigkeitsbereiche

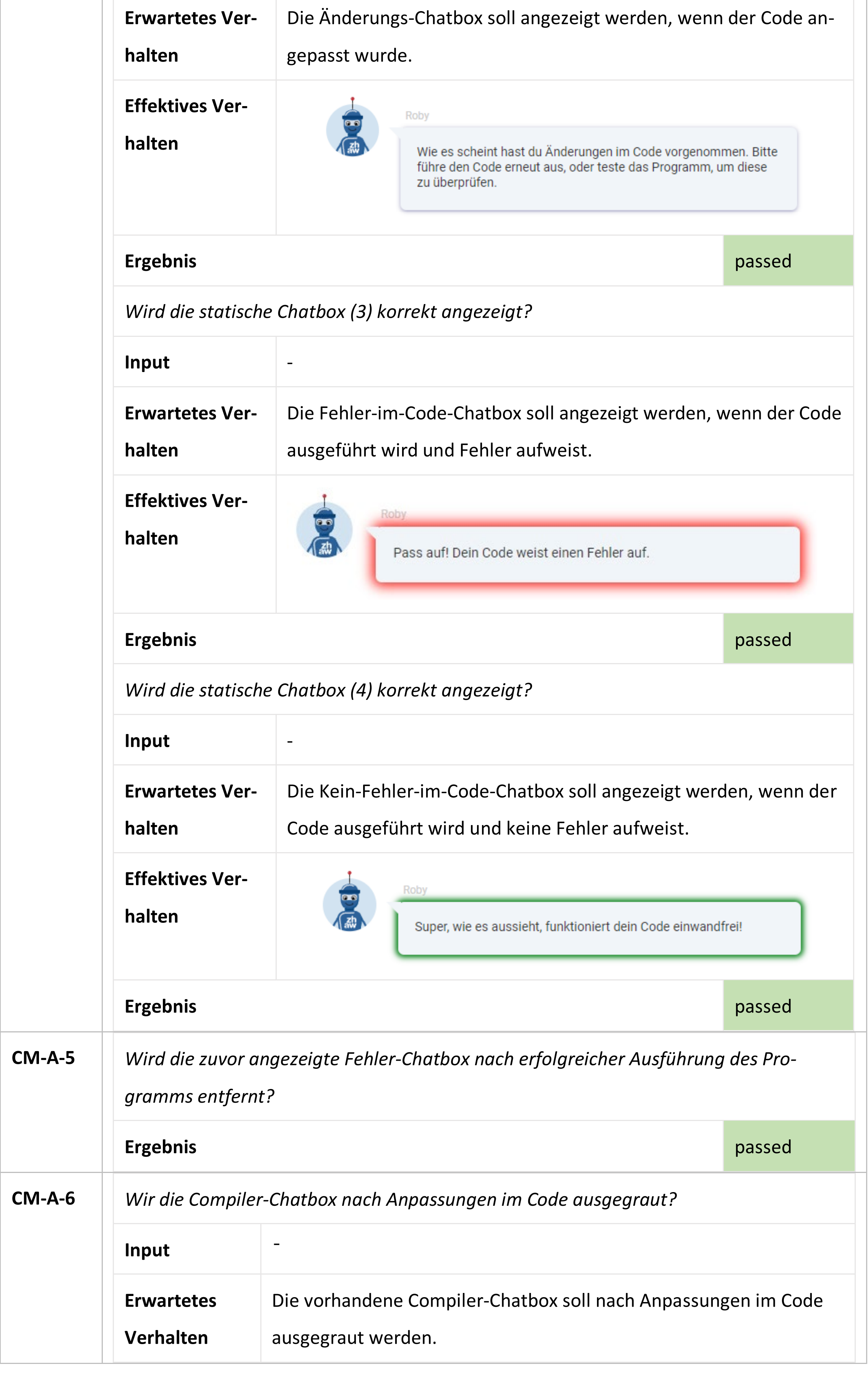
Anhang 3.3: Compiler-Meldungen

Die durchgeführten Tests betreffend dem Helpersystem *Compiler-Meldungen* sind in der nachfolgenden Tabelle dokumentiert.

Req. #	Test		
CM-A-1	*Können die Erklärungen über den Tab "Compiler" aufgerufen werden?*		
	Ergebnis		passed
CM-A-3	*Wird jeweils nur der erste Fehler in der Konsole in Form einer Chatbox erklärt?*		

	Input	`public class Main {`
		`    public static void main(String[ args) {`
		`        System.out.println("Java ist ein Kaffee")`
		`    }`
		`}`	

	Erwartetes Verhalten	Es soll nur der Fehler in Zeile 2 erklärt werden (obschon auch ein Fehler in Zeile 3 vorhanden ist)	
	Effektives Verhalten	Roby Fehler beim Kompilieren Vorsicht! Wie es aussieht ist die Zeile 2 in der Klasse Main fehlerhaft. Das Zeichen] fehlt und der Code kannd deshalb nicht verarbeitet werden. Überprüfe die Zeile und füge das fehlende Zeichen] hinzu.	
	Ergebnis		passed
CM-A-4	*Wird die statische Chatbox (1) korrekt angezeigt?*		
	Input	-	
	Erwartetes Verhalten	Die Info-Chatbox soll angezeigt werden, wenn der Code weder geändert noch ausgeführt wurde.	
	Effektives Verhalten	Roby Nutze diesen Tab, um dir Exceptions erklären zu lassen. Wenn du das Programm ausführst und dieses Fehler beinhaltet, wird automatisch in diesen Tab gewechselt.	
	Ergebnis		passed
	Wird die statische Chatbox (2) korrekt angezeigt?		
	Input	-	



	Erwartetes Verhalten	Die Änderungs-Chatbox soll angezeigt werden, wenn der Code angepasst wurde.
	Effektives Verhalten	Roby Wie es scheint hast du Änderungen im Code vorgenommen. Bitte führe den Code erneut aus, oder teste das Programm, um diese zu überprüfen.
	Ergebnis	passed
	Wird die statische Chatbox (3) korrekt angezeigt?	
	Input	-
	Erwartetes Verhalten	Die Fehler-im-Code-Chatbox soll angezeigt werden, wenn der Code ausgeführt wird und Fehler aufweist.
	Effektives Verhalten	Roby Pass auf! Dein Code weist einen Fehler auf.
	Ergebnis	passed
	Wird die statische Chatbox (4) korrekt angezeigt?	
	Input	-
	Erwartetes Verhalten	Die Kein-Fehler-im-Code-Chatbox soll angezeigt werden, wenn der Code ausgeführt wird und keine Fehler aufweist.
	Effektives Verhalten	Roby Super, wie es aussieht, funktioniert dein Code einwandfrei!
	Ergebnis	passed
CM-A-5	*Wird die zuvor angezeigte Fehler-Chatbox nach erfolgreicher Ausführung des Programms entfernt?*	
	Ergebnis	passed
CM-A-6	*Wir die Compiler-Chatbox nach Anpassungen im Code ausgegraut?*	
	Input	-
	Erwartetes Verhalten	Die vorhandene Compiler-Chatbox soll nach Anpassungen im Code ausgegraut werden.

	Effektives Verhalten	Roby Fehler beim Kompilieren Warte mal. Wie es aussieht ist die Zeile 7 in der Klasse Main fehlerhaft. Du hast, wie es scheint, das Zeichen ; vergessen. Überprüfe die Zeile und füge das fehlende Zeichen ; hinzu.	
	Ergebnis		passed

Tabelle 3-4: Testdokumentation – Compiler-Meldungen

Anhang 3.4: Tipps

Zum Anschluss dieses Anhangs werden in der folgenden Tabelle die durchgeführten Tests betreffend dem Helpersystem *Tipps* erläutert.

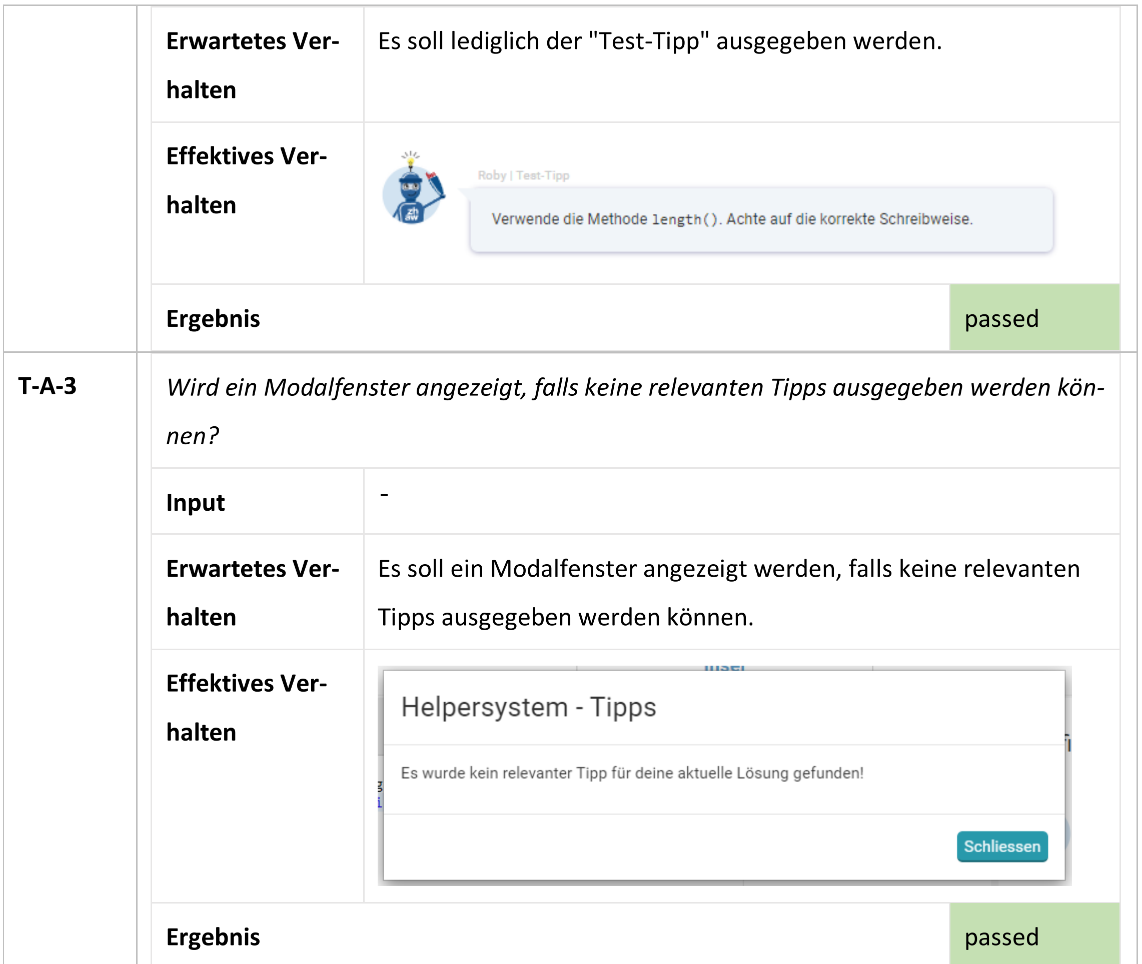
Req. #	Test		
T-A-1	*Können die Tipps über den Tab "Tipps" aufgerufen werden?*		
	Ergebnis		passed
T-A-2	*Werden die Tipps abhängig vom Stand der Lösung ausgegeben?*		
	Input	**Code-Editor:** (siehe unten) **Konfigurationsdatei (codeboard.json):** (siehe unten)	

Code-Editor:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Java ist ein Kaffee");
    }
}
```

Konfigurationsdatei (codeboard.json):

```
{
    "name": "Tipp 1",
    "note": "Der Text in der Print-Anweisung muss in doppelten Anführungszeichen stehen.",
    "mustMatch": true,
    "matching": "System\\.out\\.println\\([\\w\\s]+\\)\\s*;"
},
{
    "name": "Tipp 2",
    "note": "Du musst lediglich die dritte Zeile anpassen. Ersetze dort einfach den Text <i>Java ist eine Insel</i> mit <i>Java ist ein Kaffee</i>. Achte darauf, dass am Anfang und am Ende des Textes doppelte Anführungszeichen stehen.",
    "mustMatch": false,
    "matching": "System\\.out\\.println\\(\"Java ist ein Kaffee\"\\);"
},
{
    "name": "Test-Tipp",
    "note": "Verwende die Methode <tt>length()</tt>. Achte auf die korrekte Schreibweise.",
    "mustMatch": false,
    "matching": "\\w*\\.length\\(\\)"
}
```



	Erwartetes Verhalten	Es soll lediglich der "Test-Tipp" ausgegeben werden.
	Effektives Verhalten	Roby | Test-Tipp Verwende die Methode `length()`. Achte auf die korrekte Schreibweise.
	Ergebnis	passed
T-A-3	*Wird ein Modalfenster angezeigt, falls keine relevanten Tipps ausgegeben werden können?*	
	Input	-
	Erwartetes Verhalten	Es soll ein Modalfenster angezeigt werden, falls keine relevanten Tipps ausgegeben werden können.
	Effektives Verhalten	Helpersystem - Tipps Es wurde kein relevanter Tipp für deine aktuelle Lösung gefunden! Schliessen
	Ergebnis	passed

Tabelle 3-5: Testdokumentation – Compiler-Meldungen

Anhang 4: Evaluation

Dieser Anhang beinhaltet alle Unterlagen betreffend der durchgeführten Evaluation.

Anhang 4.1: Aufgaben

In den nachfolgenden zwei Kapiteln befinden sich die zwei Aufgaben sowie die dazugehörigen Musterlösungen, welche die Probanden im Rahmen der Evaluation lösen mussten.

Anhang 4.1.1: Aufgabe – Fehlerbehebung

In den nachfolgenden Abbildungen finden sich die Aufgabenstellung, die Musterlösung sowie die Fehler im Code betreffend der Aufgabe *Fehlerbehebung*.

Aufgabenstellung:

```
import java.util.Arrays;

public class ErrorFinding {
    public static void main(StrIng[] args) {
        int[ zahlen = {4, 7, 2, 9, 5, 1, 8, 3, 6};

        int summe = calculateSum(zahlen);

        double durchschnitt  calculateAverage(zahlen, summe);

        System.out.printIn("Summe: " + summe);
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 1 ;
        for (int i = 0; i < zahlen.length; i++) {
            sum = summe + zahlen[i];
        }
        return sum
    }

    public static d0uble calculateAverage(int[] zahlen, int summe {
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}
```

Abbildung 4-1: Aufgabe – Fehlerbehebung

Musterlösung:

```java
import java.util.Arrays;

public class ErrorFinding {
    public static void main(String[] args) {
        int[] zahlen = {4, 7, 2, 9, 5, 1, 8, 3, 6};

        int summe = calculateSum(zahlen);

        double durchschnitt = calculateAverage(zahlen, summe);

        System.out.println("Summe: " + summe);
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 0;
        for (int i = 0; i < zahlen.length; i++) {
            sum = sum + zahlen[i];
        }
        return sum;
    }

    public static double calculateAverage(int[] zahlen, int summe) {
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}
```

Abbildung 4-2: Musterlösung – Fehlerbehebung

Fehler im Code:

```
import java.util.Arrays;

public class ErrorFinding {
    public static void main(StrIng[] args) {
        int[ zahlen = {4, 7, 2, 9, 5, 1, 8, 3, 6};

        int summe = calculateSum(zahlen);

        double durchschnitt calculateAverage(zahlen, summe);

        System.out.printIn("Summe: " + summe);
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 1;
        for (int i = 0; i < zahlen.length; i++) {
            sum = summe + zahlen[i];
        }
        return sum
    }

    public static d0uble calculateAverage(int[] zahlen, int summe {
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}
```

String] = l 0 sum ; o)

Abbildung 4-3: Fehler im Code – Fehlerbehebung

Anhang 4.1.2: Aufgabe – Code-Ergänzung

Die nachfolgenden Abbildungen zeigen die Aufgabenstellung sowie die dazugehörige Musterlösung betreffend der Aufgabe *Code-Ergänzung*.

Aufgabenstellung:

```
public class CodeCompletion {

    public static void main(String[] args) {
        String[] personen = {"Alice", "Bob", "Charlie", "David", "Eva"};
        int[] alter = {30, 25, 22, 28, 34};

        double durchschnittsAlter = calculateAverageAge(alter);
        String laengsterName = findLongestName(personen);

        System.out.println("Durchschnittsalter: " + durchschnittsAlter);
        System.out.println("Längster Name: " + laengsterName);
    }

    public static double calculateAverageAge(int[] alter) {
        // TODO: Berechne das durchschnittliche Alter der Personen im Array
alter.
    }

    public static String findLongestName(String[] personen) {
        // TODO: Finde die Person mit dem längsten Namen im Array personen.
    }
}
```

Abbildung 4-4: Aufgabe – Code-Ergänzung

Musterlösung:

```
public class CodeCompletion {

    public static void main(String[] args) {
        String[] personen = {"Alice", "Bob", "Charlie", "David", "Eva"};
        int[] alter = {30, 25, 22, 28, 34};

        double durchschnittsAlter = calculateAverageAge(alter);
        String laengsterName = findLongestName(personen);

        System.out.println("Durchschnittsalter: " + durchschnittsAlter);
        System.out.println("Längster Name: " + laengsterName);
    }

    public static double calculateAverageAge(int[] alter) {
        double summe = 0;
        for (int i = 0; i < alter.length; i++) {
            summe += alter[i];
        }
        return summe / alter.length;
    }

    public static String findLongestName(String[] personen) {
        String name = personen[0];
        for (int i = 0; i < personen.length; i++) {
            if (personen[i].length() > name.length()) {
                name = personen[i];
            }
        }
        return name;
    }
}
```

Abbildung 4-5: Musterlösung – Code-Ergänzung

Anhang 4.2: Fragebogen

In der nachfolgenden Tabelle befinden sich die 15 Fragen, welche für die Umfrage verwendet wurden.

#	Frage
Q01	Ich denke, ich müsste Fragen stellen, um das Codeboard korrekt nutzen zu können. (1 - keinesfalls, 2 - wahrscheinlich nicht, 3 - vielleicht, 4 - ziemlich wahrscheinlich, 5 - ganz sicher)
Q02	Es war schwierig herauszufinden, wie man die einzelnen Helpersysteme verwenden kann. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q03	Mir war klar, für welche Situation, ich welches Helpersystem nutzen muss. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q04	Ich finde, dass mir die einzelnen Helpersysteme weitergeholfen haben. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q05	Die einzelnen Helpersysteme haben meine Lernerfahrung insgesamt deutlich verbessert. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q06	Das Helpersystem "Compiler-Meldungen" lieferte hilfreiche Informationen zur Behebung von Syntaxfehlern. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q07	Das Helpersystem "Coding-Assistant" half mir, meinen Code besser zu verstehen und mich auf Syntax-Fehler aufmerksam zu machen. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q08	Das Helpersystem "Tipps" lieferte nützliche Hinweise, um die Aufgabe lösen zu können. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)

Q09	Das Helpersystem "Test" hat Probleme in meinem Code identifiziert und mir hilfreiches Feed-back gegeben. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q10	Ich bin mit dem "Look & Feel" (Aussehen und Handhabung) des Codeboards zufrieden. (1 - Stimme überhaupt nicht zu, 2 - Stimme nicht zu, 3 - Stimme teils zu, 4 - Stimme zu, 5 - Stimme vollkommen zu)
Q11	Wenn ein technisch weniger versierter Kollege dich bitten würde, ihm ein Programm zu emp-fehlen, dass ihn beim Erlernen der Programmiersprache Java unterstützt, würdest du ihm das Codeboard weiterempfehlen? (Ja, Nein)
Q12	Was ist deiner Meinung nach die wichtigste Funktion, die noch hinzugefügt werden sollte? (Freitext)
Q13	Welches Helper-System würdest du am häufigsten verwenden? (Coding-Assistant, Compiler-Meldungen, Tipps, Test, manuelle Fragen)
Q14	Nenne zwei Punkte, die dir beim Arbeiten mit dem Codeboard gefallen haben. (Freitext)
Q15	Nenne zwei Punkte, die dir beim Arbeiten mit dem Codeboard gefehlt haben. (Freitext)

Tabelle 4-1: Evaluation – Fragenkatalog

Anhang 4.3: Auswertung der Aufgaben

In den nachfolgenden zwei Kapiteln werden die Ergebnisse der Probanden zu den durchgeführten Aufgaben dargestellt.

Anhang 4.3.1: Aufgabe – Fehlerbehebung

Die nachfolgenden Abbildungen visualisieren die Ergebnisse der Probanden bezüglich der Aufgabe *Fehlerbehebung*.

Teilnehmer 1:

```
import java.util.Arrays;

public class ErrorFinding {
    public static void main(String[] args) {
        int[] zahlen = {4, 7, 2, 9, 5, 1, 8, 3, 6};

        int summe = calculateSum(zahlen);

        double durchschnitt = calculateAverage(zahlen, summe);

        System.out.println("Summe: " + summe);
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 1 ;  ❌
        for (int i = 0; i < zahlen.length; i++) {
            sum = summe + zahlen[i];❌
        }
        return sum;
    }

    public static double calculateAverage(int[] zahlen, int summe) {
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}

7/9 Fehler gefunden
```

Abbildung 4-6: Korrektur 1 – Fehlerbehebung

Teilnehmer 2:

```
import java.util.Arrays;

public class ErrorFinding {
    public static void main(StrIng[] args) { ❌

        int[] zahlen = [4, 7, 2, 9, 5, 1, 8, 3, 6]; ❌

        int summe = calculateSum(zahlen);

        double durchschnitt  calculateAverage(zahlen, summe); ❌

        System.out.printIn("Summe: " + summe); ❌
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 1 ; ❌
        for (int i = 0; i < zahlen.length; i++) {
            sum = summe + zahlen[i]; ❌
        }
        return sum ❌
    }

    public static d0uble calculateAverage(int[] zahlen, int summe { ❌
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}

0/9 Fehler gefunden
```

Abbildung 4-7: Korrektur 2 – Fehlerbehebung

Teilnehmer 3:

```
import java.util.Arrays;

public class ErrorFinding {
    public static void main(String[] args){
        int[] zahlen = {4, 7, 2, 9, 5, 1, 8, 3, 6};

        int summe = calculateSum(zahlen);

        double durchschnitt = calculateAverage(zahlen, summe);

        System.out.println("Summe: " + summe);
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 1; ❌
        for (int i = 0; i < zahlen.length; i++) {
            sum = summe + zahlen[i]; ❌
        }
        return sum;
    }

    public static double calculateAverage(int[] zahlen, int summe){
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}

7/9 Fehler gefunden
```

Abbildung 4-8: Korrektur 3 – Fehlerbehebung

Teilnehmer 4:

```
import java.util.Arrays;

public class ErrorFinding {
    public static void main(StrIng[] args) { ❌
        int zahlen = [4, 7, 2, 9, 5, 1, 8, 3, 6]; ❌

        int summe = calculateSum(zahlen);

        double durchschnitt = calculateAverage(zahlen, summe);

        System.out.printIn("Summe: " + summe);❌
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 1 ;❌
        for (int i = 0; i < zahlen.length; i++) {
            sum = summe + zahlen[i];❌
        }
        return sum;
    }

    public double calculateAverage(int[] zahlen, int summe {❌
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}

3/9 Fehler gefunden
```

Abbildung 4-9: Korrektur 4 – Fehlerbehebung

Teilnehmer 5:

```
import java.util.Arrays;

public class ErrorFinding {
    public static void main(String[] args) {
        int[] zahlen = {4, 7, 2, 9, 5, 1, 8, 3, 6};

        int summe = calculateSum(zahlen);

        double durchschnitt = calculateAverage(zahlen, summe);

        System.out.println("Summe: " + summe);
        System.out.println("Durchschnitt: " + durchschnitt);
    }

    public static int calculateSum(int[] zahlen) {
        int sum = 0 ;
        for (int i = 0; i < zahlen.length; i++) {
            sum = sum + zahlen[i];
        }
        return sum;
    }

    public static double calculateAverage(int[] zahlen, int summe) {
        double avg;
        int laenge = zahlen.length;
        avg = summe / laenge;
        return avg;
    }
}

9/9 Fehler gefunden
```

Abbildung 4-10: Korrektur 5 – Fehlerbehebung

Anhang 4.3.2: Aufgabe – Code-Ergänzung

Die nachfolgenden Abbildungen zeigen die Lösungen der Probanden bezüglich der Aufgabe *Code-Ergänzung*.

Teilnehmer 1:

```
public class CodeCompletion {

    public static void main(String[] args) {
        String[] personen = { "Alice", "Bob", "Charlie", "David", "Eva" };
        int[] alter = { 30, 25, 22, 28, 34 };

        double durchschnittsAlter = calculateAverageAge(alter);
        String laengsterName = findLongestName(personen);

        System.out.println("Durchschnittsalter: " + durchschnittsAlter);
        System.out.println("Längster Name: " + laengsterName);
    }

    public static double calculateAverageAge(int[] alter) {
        // TODO: Berechne das durchschnittliche Alter der Personen im Array
alter.

        double average = 0;
        for (int i = 0; i < alter.length; i++) {
            average += alter[i];
        }
        return average / alter.length;
    }

    public static String findLongestName(String[] personen) {
        //TODO: Finde die Person mit dem längsten Namen im Array personen.
        String longestName;
        for (int i = 0; i < personen.length; i++) {
            if (longestName < personen[i]) {
                longestName = personen[i];
            }
        }
        return longestName;
    }
}
```

Abbildung 4-11: Korrektur 1 – Code-Ergänzung

Teilnehmer 2:

```
public class CodeCompletion {

    public static void main(String[] args) {
        String[] personen = {"Alice", "Bob", "Charlie", "David", "Eva"};
        int[] alter = {30, 25, 22, 28, 34};

        double durchschnittsAlter = calculateAverageAge(alter);
        String laengsterName = findLongestName(personen);

        System.out.println("Durchschnittsalter: " + durchschnittsAlter);
        System.out.println("Längster Name: " + laengsterName);
    }

    public static double calculateAverageAge(int[] alter) {
        int sum= 0; ✗
        for(int i=0;i < alter.length;i++){ ✓
            sum += alter[i]; ✓
        }
        return sum/alter.length; ✓
    }

    public static String findLongestName(String[] personen) {
        String longestName = " "; ✗

        for(int i=0;i < personen.length;i++){ ✓
            if(personen[i].length > longestName.length){ ✗
                longestName = personen[i]; ✓
            }
        }
        return longestName; ✓
    }
}
```

Abbildung 4-12: Korrektur 2 – Code-Ergänzung

Teilnehmer 3:

```
public class CodeCompletion {

    public static void main(String[] args) {
        String[] personen = {"Alice", "Bob", "Charlie", "David", "Eva"};
        int[] alter = {30, 25, 22, 28, 34};

        double durchschnittsAlter = calculateAverageAge(alter);
        String laengsterName = findLongestName(personen);

        System.out.println("Durchschnittsalter: " + durchschnittsAlter);
        System.out.println("Längster Name: " + laengsterName);
    }

    public static double calculateAverageAge(int[] alter) {
        // TODO: Berechne das durchschnittliche Alter der Personen im Array
alter.
        double x = 0; ✓
        for(int i=0; alter.length() > i; i++){ ✗
            x += alter[i]; ✓
        }
        x = x / alter.length(); ✗
        return x; ✓
    }

    public static String findLongestName(String[] personen) {
        // TODO: Finde die Person mit dem längsten Namen im Array personen.
        boolean first = true; ✓
        String name = ""; ✓
        int l = 0; ✓

        if(first){
            String name = personen[0]; ✗
            l = personen[0].length(); ✓
            first = false; ✓
        }

        for(int i=1; personen.length() > i; i++){ ✗

            if(personen[i].length() > l){ ✗
                name = personen[i]; ✓

            }
        }

        return name; ✓


    }
}
```

Abbildung 4-13: Korrektur 3 – Code-Ergänzung

Teilnehmer 4:

```
public class CodeCompletion {

    public static void main(String[] args) {
        String[] personen = {"Alice", "Bob", "Charlie", "David", "Eva"};
        int[] alter = {30, 25, 22, 28, 34};

        double durchschnittsAlter = calculateAverageAge(alter);
        String laengsterName = findLongestName(personen);

        System.out.println("Durchschnittsalter: " + durchschnittsAlter);
        System.out.println("Längster Name: " + laengsterName);
    }

    public static double calculateAverageAge(int[] alter) {
        // TODO: Berechne das durchschnittliche Alter der Personen im Array
alter.
        return ((alter[1] + alter[2] + alter[3] + alter[4] + alter [5])/5); ❌
    }

    public static String findLongestName(String[] personen) {
        // TODO: Finde die Person mit dem längsten Namen im Array personen.
        for (personen; i=1; i++) { ❌
            length personen[i] = int[] lengthName; ❌

        }
    }
}
```

Abbildung 4-14: Korrektur 4 – Code-Ergänzung

Teilnehmer 5:

```
public class CodeCompletion {

    public static void main(String[] args) {
        String[] personen = {"Alice", "Bob", "Charlie", "David", "Eva"};
        int[] alter = {30, 25, 22, 28, 34};

        double durchschnittsAlter = calculateAverageAge(alter);
        String laengsterName = findLongestName(personen);

        System.out.println("Durchschnittsalter: " + durchschnittsAlter);
        System.out.println("Längster Name: " + laengsterName);
    }

    public static double calculateAverageAge(int[] alter) {
        double sum = 0;

        for(int i=0;i<alter.length; i++){
            sum = sum + alter[i];
        }

       return sum / alter.length; ✓

        // TODO: Berechne das durchschnittliche Alter der Personen im Array
alter.
    }

    public static String findLongestName(String[] personen) {

        int mostLetters = 0;
        int pos = 0;

         for(int i=0; i<personen.length; i++){

            if(mostLetters<personen[i].length()) {
            mostLetters= personen[i].length();
            pos = i;
        }
         }

        return personen[pos]; ✓

        // TODO: Finde die Person mit dem längsten Namen im Array personen.
    }
}
```

Abbildung 4-15: Korrektur 5 – Code-Ergänzung

Anhang 4.4: Auswertung der Umfrage

Nachfolgende Grafiken veranschaulichen die Auswertung der Umfrage, welche Bestandteil der Evaluation war.

Zusammenfassung für Q01		
Ich denke, ich müsste Fragen stellen, um das Codeboard korrekt nutzen zu können.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - keinesfalls (AO01)	2	40.00%
2 - wahrscheinlich nicht (AO02)	3	60.00%
3 - vielleicht (AO03)	0	0.00%
4 - ziemlich wahrscheinlich (AO04)	0	0.00%
5 - ganz sicher (AO05)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-16: Auswertung der Umfrage – Q01

Zusammenfassung für Q02		
Es war schwierig herauszufinden, wie man die einzelnen Helpersysteme verwenden kann.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	4	80.00%
2 - Stimme nicht zu (AO02)	1	20.00%
3 - Stimme teils zu (AO03)	0	0.00%
4 - Stimme zu (AO04)	0	0.00%
5 - Stimme vollkommen zu (AO05)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-17: Auswertung der Umfrage – Q02

Zusammenfassung für Q03		
Mir war klar, für welche Situation, ich welches Helpersystem nutzen muss.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	0	0.00%
3 - Stimme teils zu (AO03)	1	20.00%
4 - Stimme zu (AO04)	2	40.00%
5 - Stimme vollkommen zu (AO05)	2	40.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-18: Auswertung der Umfrage – Q03

Zusammenfassung für Q04		
Ich finde, dass mir die einzelnen Helpersysteme weitergeholfen haben.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	0	0.00%
3 - Stimme teils zu (AO03)	1	20.00%
4 - Stimme zu (AO04)	4	80.00%
5 - Stimme vollkommen zu (AO05)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-19: Auswertung der Umfrage – Q04

Zusammenfassung für Q05		
Die einzelnen Helpersysteme haben meine Lernerfahrung insgesamt deutlich verbessert.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	0	0.00%
3 - Stimme teils zu (AO03)	1	20.00%
4 - Stimme zu (AO04)	3	60.00%
5 - Stimme vollkommen zu (AO05)	1	20.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-20: Auswertung der Umfrage – Q05

Zusammenfassung für Q06		
Das Helpersystem "Compiler-Meldungen" lieferte hilfreiche Informationen zur Behebung von syntaktischen oder semantischen Fehlern.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	1	20.00%
3 - Stimme teils zu (AO03)	3	60.00%
4 - Stimme zu (AO04)	1	20.00%
5 - Stimme vollkommen zu (AO05)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-21: Auswertung der Umfrage – Q06

Zusammenfassung für Q07		
Das Helpersystem "Coding-Assistant" half mir, meinen Code besser zu verstehen und mich auf Fehler im Code aufmerksam zu machen.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	0	0.00%
3 - Stimme teils zu (AO03)	1	20.00%
4 - Stimme zu (AO04)	2	40.00%
5 - Stimme vollkommen zu (AO05)	2	40.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-22: Auswertung der Umfrage – Q07

Zusammenfassung für Q08		
Das Helpersystem "Tipps" lieferte nützliche Hinweise, um die Aufgabe lösen zu können.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	0	0.00%
3 - Stimme teils zu (AO03)	2	40.00%
4 - Stimme zu (AO04)	3	60.00%
5 - Stimme vollkommen zu (AO05)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-23: Auswertung der Umfrage – Q08

Zusammenfassung für Q09		
Das Helpersystem "Test" hat Probleme in meinem Code identifiziert und mir hilfreiches Feedback gegeben.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	0	0.00%
3 - Stimme teils zu (AO03)	1	20.00%
4 - Stimme zu (AO04)	4	80.00%
5 - Stimme vollkommen zu (AO05)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-24: Auswertung der Umfrage – Q09

Zusammenfassung für Q10		
Ich bin mit dem Look & Feel (Aussehen und Handhabung) des Codeboards zufrieden.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
1 - Stimme überhaupt nicht zu (AO01)	0	0.00%
2 - Stimme nicht zu (AO02)	0	0.00%
3 - Stimme teils zu (AO03)	0	0.00%
4 - Stimme zu (AO04)	1	20.00%
5 - Stimme vollkommen zu (AO05)	4	80.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-25: Auswertung der Umfrage – Q10

Zusammenfassung für Q11		
Wenn ein technisch weniger versierter Kollege dich bitten würde, ihm ein Programm zu empfehlen, dass ihn beim Erlernen der Programmiersprache Java unterstützt, würdest du ihm das Codeboard weiterempfehlen?		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
Ja (AO01)	5	100.00%
Nein (AO02)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-26: Auswertung der Umfrage – Q11

Zusammenfassung für Q12		
Was ist deiner Meinung nach die wichtigste Funktion, die noch hinzugefügt werden sollte?		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
Passt so evtl. noch Rechtschreibfehler anzeigen Lösungsvorschläge für einzelne Teile des Codes ein highlighter für error im code für besseres Analyse konnte nur 1x einen Tipp anfordern	5	100.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-27: Auswertung der Umfrage – Q12

Zusammenfassung für Q13		
Welches Helper-System würdest du am häufigsten verwenden?		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
Coding-Assistant (AO01)	1	20.00%
Compiler-Meldungen (AO02)	3	60.00%
Tipps (AO03)	0	0.00%
Test (AO04)	1	20.00%
manuelle Fragen (AO05)	0	0.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-28: Auswertung der Umfrage – Q13

Zusammenfassung für Q14 [Punkt 1]		
Nenne zwei Punkte, die dir beim Arbeiten mit dem Codeboard gefallen haben.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
Intuitive bedienung Compilermeldunge Einfachheit Coding assistent Compiler Informationen	5	100.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-29: Auswertung der Umfrage – Q14 (1)

Zusammenfassung für Q14 [Punkt 2]		
Nenne zwei Punkte, die dir beim Arbeiten mit dem Codeboard gefallen haben.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
Compiler feedback Gültigkeitsbereich der variablen Man weiss wo man Hilfe bekommt test system Einfache usability	5	100.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-30: Auswertung der Umfrage – Q14 (2)

Zusammenfassung für Q15 [Punkt 1]		
Nenne zwei Punkte, die dir beim Arbeiten mit dem Codeboard gefehlt haben.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
Genauerer Compiler feedback Evtl. Rechtschreibprüfung Internes Google (für Copy-Paste von For-Schleifen) Code highlighter wenn man nicht mehr weiter weiss, vielleicht eine hilfe oder so	5	100.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-31: Auswertung der Umfrage – Q15 (1)

Zusammenfassung für Q15 [Punkt 2]		
Nenne zwei Punkte, die dir beim Arbeiten mit dem Codeboard gefehlt haben.		
Antwort	**Anzahl**	**Brutto-Prozentsatz**
Mehr tipps nichts - bessere Platzierung der Navigations leiste. -	5	100.00%
Keine Antwort	0	0.00%
Gesamt(Brutto)	**5**	**100.00%**

Abbildung 4-32: Auswertung der Umfrage – Q15 (2)