Entwicklung eines Systems zur Erstellung und Bearbeitung von Programmieraufgaben für Java-Einsteiger

Development of a system for creation of and working on programming exercises for Java-novices

Bachelor-Thesis von Jan Skorvan aus Wetzlar

Tag der Einreichung:

1. Gutachten: Dr. Guido Rößling
2. Gutachten: M.Sc. Michael Stein

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Studiendekanat

Entwicklung eines Systems zur Erstellung und Bearbeitung von Programmieraufgaben für Java-Einsteiger
Development of a system for creation of and working on programming exercises for Java-novices

Vorgelegte Bachelor-Thesis von Jan Skorvan aus Wetzlar

1. Gutachten: Dr. Guido Rößling
2. Gutachten: M.Sc. Michael Stein

Tag der Einreichung:

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Jan Skorvan, die vorliegende Bachelor-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Thesis Statement pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Jan Skorvan, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

Datum/Date: Unterschrift/Signature:

Zusammenfassung

Der Einstieg in die Programmiersprache Java hat bei einigen Studienanfänger*innen bereits vor dem Studium stattgefunden. Der verhältnismäßig hohe Kenntnisstand dieser Gruppe könnte bewirken, dass diejenigen Studierenden zurückfallen, denen Java und allgemein die objektorientierte Programmierung völlig neu sind. Um diese Studierenden zu unterstützen wurde die vorliegende Arbeit verfasst und das zugehörige Programm entwickelt.

Andere Forschungsarbeiten sind sich in der Hinsicht einig, dass Fehlermeldungen beim Lernprozess eine zentrale Rolle spielen. Die Formulierung dieser Nachrichten bedarf in pädagogischen Anwendungen weitaus größerer Aufmerksamkeit, als sie bei der Entwicklung von Programmen zur professionellen Java-Programmierung erfährt. Aus diesem Grund wurden sämtliche Fehlermeldungen des Java-Compilers neu formuliert, sodass sie verdeutlichen, welcher Fehler zu den Meldungen geführt hat, damit Studierende schneller herausfinden können, was zu diesem Fehler geführt hat.

Weil es bereits eine große Herausforderung für eine*n Anfänger*in sein kann, ein kleines, aber vollständiges Programm zu schreiben, besteht in diesem Programm die Möglichkeit, in einer Aufgabe nur einen Ausschnitt eines Programms zu verlangen; das können wenige Zeilen sein, eine oder mehrere Methoden oder auch eine ganze Klasse. Diese Einstufung in verschiedene Komplexitätsgrade ermöglicht einen besser angepassten Anstieg der Anforderungen.

Das Programm zu dieser Arbeit verwendet JavaFX, um eine moderne grafische Oberfläche bereitzustellen. Es können in einer ZIP-Datei verpackte Aufgaben importiert, bearbeitet und getestet werden. Eine Wissensdatenbank gibt Hilfestellungen, wenn ein Programmierkonzept nicht verstanden wurde. Aufgaben und Wissensdatenbank können mit einem zweiten Programm für Lehrende auf dem neuesten Stand gehalten werden.

Die die Entwicklung im Rahmen dieser Arbeit abschließende Evaluation des Programms zeigt, dass das Grundkonzept stimmig ist und gut aufgenommen wird. Verbesserungen können sicherlich noch vorgenommen werden und mit weiterem zeitlichen Aufwand könnte das Programm auch noch erweitert werden.

Abstract

Many students already started using Java before attending university. The proportionally better knowledge of this group might lead to other students falling behind who are new to Java and object-oriented programming in general. This thesis was created along the accompanying program in order to support these students.

Different scientific papers agree that error messages are highly important during the learning process. The wording of these messages requires higher attention in pedagogical context than it receives during development of programs professionally used for writing Java code. Thus, all of the Java-compiler's error messages were rewritten, so they clarify which exact error led to the message, in order for students to find out easier what mistake led to that error.

As it can already be a difficult challenge for a beginner to write a small but complete program, this tool offers the possibility for a task to demand only a portion of a whole program. That might be only a few lines of code, one or more methods or a whole class. This classification of tasks into distinct complexities enables a better suited increase of requirements.

The program developed in this thesis uses JavaFX to deliver a modern graphical user interface. Tasks packed into a ZIP file can be imported, worked on and tested. A knowledge database offers help in case a programming concept is unclear. Tasks and the knowledge database can be kept up to date with a second program for teachers.

The evaluation finalizing the development as part of this thesis shows that the basic concept is working well and received positive feedback. Improvements surely can be made and with more time expenses the program could still be developed further.

Inhaltsverzeichnis

1 Einleitung

Studienanfänger*innen des Fachs Informatik lernen an der TU Darmstadt in der zweiten Hälfte der Erstsemesterveranstaltung „Funktionale und objektorientierte Programmierung“ die Programmiersprache Java. Gerade zu Beginn des Studiums sind die bestehenden Vorkenntnisse sehr unterschiedlich. Einige Studierende hatten bereits in der Sekundarstufe II Informatikunterricht, manche einen Programmierkurs oder eine AG in der Sekundarstufe I, anderen ist der fachliche Inhalt völlig neu.

Das Kerncurriculum gymnasiale Oberstufe des Landes Hessen für Informatik [6] sieht vor, dass in der Einführungsphase Grundlagen der objektorientierten Programmierung vermittelt werden. Hierfür soll eine objektorientierte Programmiersprache verwendet werden. In der Qualifikationsphase 1 werden Algorithmik und objektorientierte Programmierung behandelt. Auch in den Lehrplänen für Sachsen [22], Berlin [23], Brandenburg und Mecklenburg-Vorpommern [2] ist die objektorientierte Programmierung enthalten. Selbst wenn eine andere Programmiersprache als Java verwendet wird, bestehen durch diese Kenntnisse bereits große Vorteile gegenüber Studierenden, die zum ersten Mal mit Informatik als Studien- oder Unterrichtsfach in Berührung kommen.

Diesen Unterschied des Wissensstandes zu überwinden, sollte unter anderem Aufgabe der Einführungsveranstaltungen an der Universität sein. In Vorlesungen und Übungen wird das Thema den Studierenden näher gebracht. Um jedoch ein tiefgreifendes Verständnis für die Programmiersprache Java zu entwickeln, müssen Studierende auch selbst Lösungen finden und die Sprache und ihre Regeln durchschauen. Dies ist auf Anhieb in einer Vorlesung oder auch in der Übung nicht immer vollständig möglich. Vor allem zurückhaltende oder schüchterne Personen möchten nur ungern auf ihr fehlendes Verständnis aufmerksam machen und erhalten so nicht die Hilfe, die sie benötigen, um Java zu lernen. Abgegebene Hausübungen funktionieren aufgrund fehlenden Wissens nicht richtig, die Studierenden erhalten entsprechend schlechte Bewertungen und gehen durch Frustration mit weniger Motivation in die nächste Woche. Ohne gezielte Förderung kann das für das restliche Studium essenzielle Verständnis der Programmiersprache Java vollständig ausbleiben.

Um diesem Umstand entgegenzuwirken, entstand die Idee, das in dieser Arbeit vorgestellte Programm zu entwickeln. Das Ziel soll es sein, das Verständnis der grundlegenden Konzepte von Java mit einfachen, kurzen Aufgaben zu testen und gegebenenfalls durch Erklärungen zu verbessern. Der Fokus soll ausschließlich auf der Programmierung liegen, ohne dass eine komplexe Entwicklungsumgebung aufgesetzt und verwendet werden muss und ohne dass Code geschrieben werden muss, den man noch nicht versteht, der aber angeblich einfach hingeschrieben werden muss, damit das Programm funktioniert. Dieser Fokus auf die Programmierung soll möglichst viele ablenkenden und frustrierenden Faktoren ausblenden, damit Studierende effizient an den eigenen Fähigkeiten arbeiten können.

Im zweiten Kapitel dieser Arbeit werden bereits bestehende Programme mit einer ähnlichen Zielsetzung vorgestellt, analysiert und bewertet. Daraufhin werden Arbeiten aus der Forschung betrachtet und ausgewertet, um deren Einfluss auf das Programm zu verdeutlichen. Außerdem werden Richtlinien entnommen oder abgeleitet, die für das Verfassen von Fehlermeldungen und Aufgaben einen positiven Effekt zeigen. Auf dieser Basis wird dann bewertet, wie aussagekräftig und hilfreich die Fehlermeldungen des Java-Compilers sind. Schließlich folgt eine kurze Einführung in JavaFX, den Nachfolger von Swing. JavaFX soll zur Erstellung der grafischen Benutzeroberfläche verwendet werden.

Das dritte Kapitel befasst sich mit den Anforderungen, die an das Programm gestellt werden. Dabei wird eine Aufteilung des Programms vorgenommen. Studierende sollen Aufgaben bearbeiten und Fehler erklärt bekommen, während Lehrende entsprechende Aufgaben und Erklärungen erstellen und verfassen sollen. Da beide Teile einen beachtlichen Schnittbereich haben, werden zunächst die Anforderungen an diese gemeinsamen Eigenschaften definiert. Daraufhin werden die für beide Teile spezifischen Anforderungen analysiert. Am Ende des Kapitels wird diskutiert, auf welche Art die Verteilung von neu erstellten Aufgaben an Studierende, die das Programm bereits verwenden, am effizientesten ist.

Einen Blick aus der Sicht der Anwender*innen liefert Kapitel 4. Das Programm für Studierende und das Programm für Lehrende werden in ihrer Erscheinung in Form der grafischen Benutzeroberfläche beschrieben. Dabei werden auch Designentscheidungen erläutert und gerechtfertigt. Die Anwendung des Programms wird erklärt und somit dient dieses Kapitel ebenfalls als Grundlage zum tieferen Verständnis des Programms. Außerdem werden die im Programm verwendeten und auf die Ergebnisse aus Kapitel zwei gestützten Fehlermeldungen tabellarisch aufgeführt.

Kapitel 5 gibt einen tieferen Einblick in die tatsächliche Implementierung des Programms. Dabei wird zum einen der grobe Aufbau des Programms verdeutlicht und zum anderen werden verwendete Algorithmen gezeigt und erklärt. Auch die verwendeten externen Bibliotheken werden erläutert und ihre Funktion im Programm wird verdeutlicht.

Das sechste Kapitel beschreibt die Durchführung und die Ergebnisse einer kleinen Nutzerstudie als Evaluation. Die Ergebnisse zeigen allgemein eine gute Bewertung des Programms, weisen jedoch auch auf Schwachstellen hin. Einige Fehler wurden direkt behoben, andere diskutiert und vorerst zurückgestellt.

Mit diesen zurückgestellten Fehlern und weiterführenden Funktionen und Möglichkeiten des Programms befasst sich das siebte und letzte Kapitel. Nach einer kurzen rückblickenden Zusammenfassung wird der Blick auf die Weiterarbeit gerichtet und aufgezeigt, welche Bereiche verbessert, verändert, ersetzt oder neu erschaffen werden können. Auf dieser Grundlage werden Ansätze für die zukünftige Weiterentwicklung des Programms vorgestellt.

2 Verwandte Arbeiten und Grundlagen

In diesem Kapitel werden verwandte Arbeiten und Anwendungen analysiert. Verschiedene Artikel befassen sich mit vergleichbaren Themen aus der Lehre und enthalten wertvolle Informationen und Gedankengänge.

Das Ziel dieses Kapitels ist es, erfolgreiche Maßnahmen zur Erleichterung des Einstiegs in die Programmierung zu erfassen sowie die technische Umsetzung zu bewerten, um den Einfluss auf die vorliegende Arbeit zu verdeutlichen.

Im letzten Teil werden Grundlagen behandelt, die für die Entwicklung und das Verständnis dieser Arbeit wichtig sind. Zuerst werden die Fehlermeldungen des Java-Compilers betrachtet, darauf folgt eine Einführung in die API JavaFX, welche den bisherigen Standard Swing ersetzt.

2.1 Programmierumgebungen für Einsteiger

Dieser Abschnitt beschäftigt sich mit bereits bestehenden Anwendungen, insbesondere solchen, die den Einstieg in Java erleichtern sollen. Es soll dargelegt werden, wie gut dieses Ziel erreicht wird, welche Mittel dafür zum Einsatz kommen und welche dieser Mittel in diese Arbeit übernommen werden können. Zudem werden die Programme kritisch betrachtet, um etwaige Fehler oder Mängel nicht zu übernehmen.

Untersucht wird die Webseite CodingBat, die die Programmierfähigkeiten der Anwender mit einfach gestellten Aufgaben und automatischen Tests prüft und verbessert. Die Editoren BlueJ und Java-Editor werben besonders damit, den Einstieg in Java zu erleichtern, und verfolgen dieses Ziel auf unterschiedliche Weise.

2.1.1 CodingBat

Die Website CodingBat [19] wurde von Nick Parlante erstellt, bietet Coding-Probleme online an und wirbt damit, Programmierfähigkeiten der Nutzer zu entwickeln und zu verbessern. Zu diesem Zweck werden kurze Aufgabenstellungen präsentiert, die direkt im Browser bearbeitet und bewertet werden können. Eine Vielzahl an Aufgaben aus grundlegenden Bereichen der Programmierung wie Manipulation von Strings und Datenstrukturen, Schleifen und Logik sollen helfen, diese kleinen Probleme des Programmierens zu meistern, um sich dann besser den großen Problemen zuwenden zu können. „Jemand, der mit Schleifen, Logik etc. kämpft, hat keine Zeit für das große Ganze“ [18].

Übersicht

Die Website ermöglicht es, Code zu schreiben, ohne dass dabei etwas anderes in den Weg kommt. Es wird eine Umgebung bereitgestellt, bei der man direkt mit der Aufgabenstellung anfangen kann, ohne einen Editor zu öffnen, ein neues Projekt anzulegen oder eine neue Klasse zu erstellen.

Nachdem eine Aufgabe geöffnet wurde, werden die Aufgabenstellung sowie ein Code-Bereich angezeigt. Mit einem Go-Knopf kann der eingegebene Code gespeichert, kompiliert und ausgeführt (getestet) werden. Wurde kein Benutzerkonto erstellt, wird der Code nur für die Dauer einer Sitzung gespeichert. Wurde der Code erfolgreich kompiliert, werden nach dem Testen die Testergebnisse angezeigt. Bei einfachen Aufgaben besteht die Möglichkeit, sich einen Lösungsvorschlag anzeigen zu lassen. Sind alle Tests erfolgreich, wird eine weitere Aufgabe zur Bearbeitung angeboten.

Die Aufgabenstellungen sind zwar nicht anwendungsbezogen, ermöglichen es aber, in einer kurzen Zeit viel Übung im Programmieren zu bekommen. Nachdem der*die Nutzer*in sich angemeldet hat, wird der bisherige Gesamtfortschritt (auch über mehrere Sitzungen hinweg) in einem Graphen festgehalten. Dieser Graph gibt an, zu welchem Zeitpunkt welche Aufgabe gelöst wurde und wie oft das Ergebnis falsch war. Geschriebener Code wird dauerhaft gespeichert, sodass auch über mehrere Sitzungen hinweg gearbeitet und zuvor geschriebener Code noch einmal aufgefrischt werden kann.

Code-Bereich

Der Code-Bereich (siehe Abbildung 2.1 auf der nächsten Seite) enthält beim erstmaligen Öffnen der Aufgabe bereits einen Methodenkopf, sodass nur der Methodenkörper programmiert werden muss. Es besteht theoretisch auch die Möglichkeit, eigene Hilfsmethoden zu schreiben, was in manchen Aufgaben explizit empfohlen oder gefordert wird. Bestimmte Schlüsselwörter wie `static`, `class` oder die `System`-Klasse sind im Code nicht erlaubt, die Verwendung führt zur Ausgabe „Bad code“. Außerdem besteht eine Beschränkung, welche bestehenden Klassen verwendet werden dürfen.

Das Syntax-Highlighting unterscheidet die Gruppen Java-Schlüsselwörter und Primitive (violett), Zahlen und Strings (blau), arithmetische und logische Operatoren (hellgrau), Kommentare (grün) und die Klasse String (dunkelgrau). Zusammengehörige

Klammern werden bei Berührung mit dem Cursor hervorgehoben. Die aktuell angewählte Zeile ist grau hinterlegt, allerdings gibt es keine Zeilennummern.

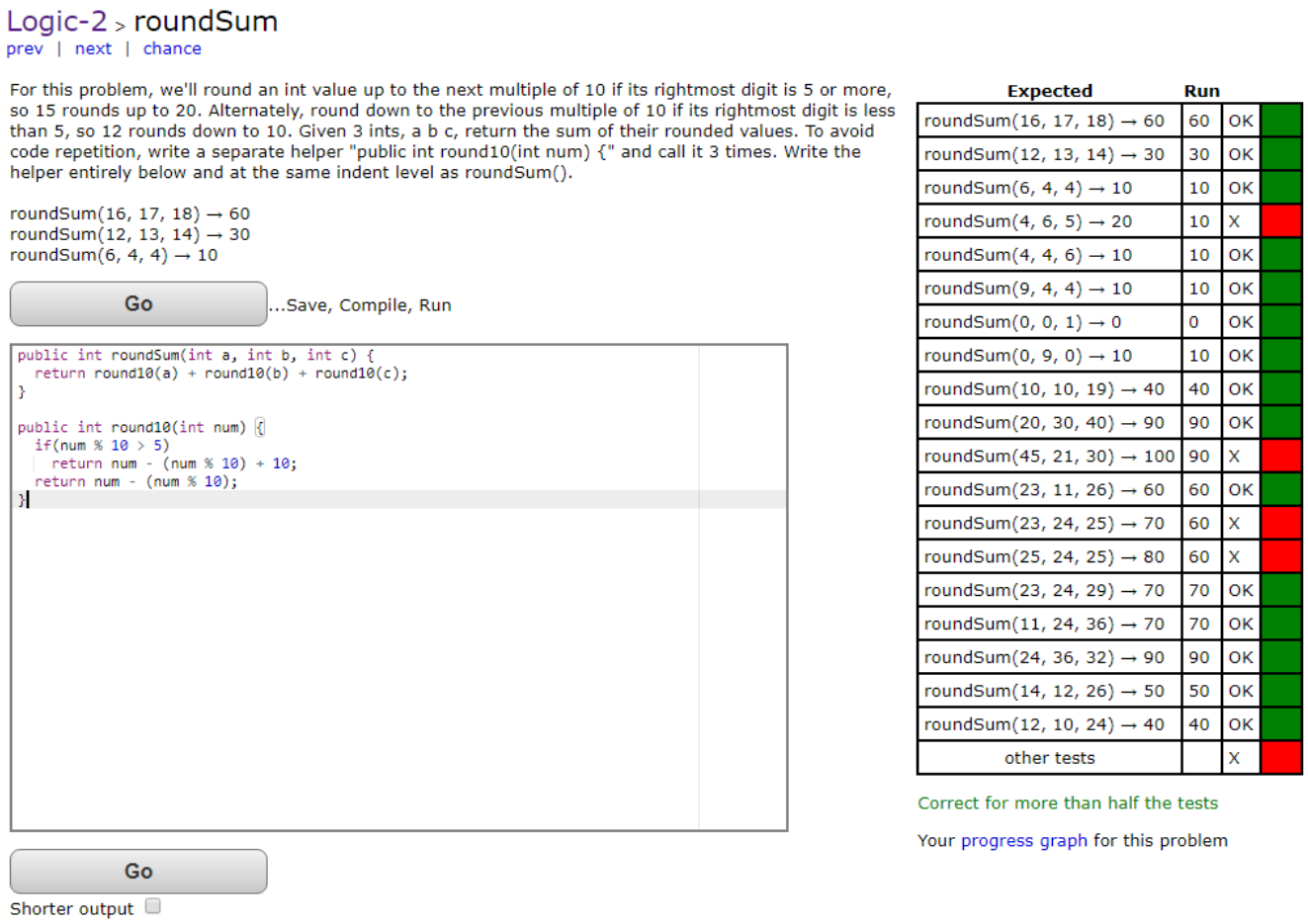

Abbildung 2.1: Beispielaufgabe auf der Seite CodingBat mit Testergebnissen. Anmerkung für Druck in Graustufen: Bei „OK“ ist das Feld dahinter grün, bei „X“ ist es rot.

Kompilierfehler und Tests

Falls beim Kompilieren des Codes Fehler auftreten, wird die entsprechende Javac-Fehlermeldung mit Zeilenangabe angezeigt. Diese Fehlermeldungen sind sehr kurz und geben einem Neuling möglicherweise zu wenig Informationen, um den Fehler zu verstehen. Vor allem gibt es keine Möglichkeit, sich den Ursprung oder die Art des Fehlers erklären zu lassen. Die Aussagekraft dieser Fehlermeldungen wird in Abschnitt 2.3.1 auf Seite 14 diskutiert.

Die Ausführung der Tests erfolgt automatisch nach dem erfolgreichen Kompiliervorgang. Es wird für jeden öffentlichen ausgeführten Test angezeigt, mit welchen Parametern die Methode aufgerufen wurde, welches das erwartete und welches das tatsächliche Ergebnis waren.

Schlägt der Test fehl, wird dies durch ein rotes Feld und ein „X“ angezeigt. Wird bei der Ausführung eine Exception geworfen, so werden der Klassenname dieser Exception und eine kurze Beschreibung mit Zeilenangabe mitgeteilt. Wie viel Neulinge aus dem bloßen Namen einer Exception lesen können, wird in Abschnitt 2.3.2 auf Seite 17 behandelt. Bei einigen Aufgaben gibt es auch private Tests, die zusammengefasst nur vermitteln, ob sie erfolgreich waren oder nicht.

Bewertung

CodingBat kommt dem Ziel der vorliegenden Arbeit sehr nahe. Einfache Aufgaben, die ohne großen Aufwand bearbeitet werden können und automatisch auf Korrektheit überprüft werden, erleichtern den Einstieg in Java deutlich. Der bisherige Fortschritt kann gespeichert werden, jeder Haken für eine erfolgreich bearbeitete Aufgabe belohnt die aktive Nutzung.

Einer der wenigen negativen Aspekte ist, dass für die Verwendung der Seite eine Internetverbindung benötigt wird. Auf der Zugfahrt von der oder zur Universität lässt sich damit nicht ohne Weiteres arbeiten. Außerdem werden die Ergebnisse online gespei-

chert und dürfen für Forschungszwecke anonymisiert verwendet werden [20], was manche Personen davon abhalten könnte, hier die eigenen, möglicherweise als „peinlich schlecht“ empfundenen Programmierfähigkeiten auszuprobieren.

Zuletzt werden Kompilierfehler und Exceptions lediglich angezeigt, nicht aber erklärt. Bevor beim Vorkommen von unverständlichen Fehlermeldungen Google bemüht werden muss oder man sich die Vorlesungsfolien noch einmal anschaut, würde es Zeit sparen, diese Fehler direkt auf der Seite zu erklären.

2.1.2 BlueJ

Die Entwicklungsumgebung BlueJ [11] wurde von einer Forschungsgruppe bestehend aus Mitgliedern der University of Kent in England und der Deakin University in Australien mit der Unterstützung von Oracle entworfen und umgesetzt. BlueJ wurde speziell für die Lehre entwickelt und wirbt mit einem einfachen Interface, Interaktivität mit Objekten und direkter Ausführung von Java-Ausdrücken [9]. Das Programm ist für die Betriebssysteme Windows, macOS und Ubuntu/Debian erhältlich. Außerdem ist der Download als JAR-Datei möglich, mit der das Programm unter jedem Betriebssystem ausgeführt werden kann, das über ein hinreichend aktuelles JDK verfügt.

Übersicht

Öffnet man BlueJ das erste Mal, wird man von der GUI aufgefordert, ein Projekt zu öffnen oder ein neues Projekt zu erstellen. Mit einem Button oder dem „Bearbeiten“-Menü lässt sich eine neue Klasse hinzufügen. Diese wird grafisch im UML-Stil angezeigt, allerdings ohne Methoden und Attribute.

Mit einem Rechtsklick auf eine Klasse lässt sich eine neue Instanz erzeugen, die im unteren Teil des Fensters erscheint. Auf dieser Instanz lassen sich dann die Methoden der Klasse ausführen und die Attribute einsehen (siehe Abbildung 2.2). Der Code einer Klasse lässt sich in einem neuen Fenster bearbeiten. Weitere Merkmale der Umgebung sind für diese Arbeit nicht relevant und werden daher nicht weiter angeführt.

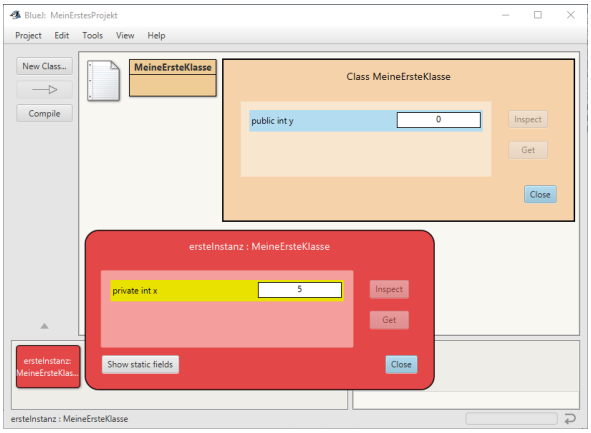

Abbildung 2.2: Die BlueJ-GUI mit einer Klasse, einer Instanz und den jeweiligen Anzeigen für Variablen bzw. Attribute

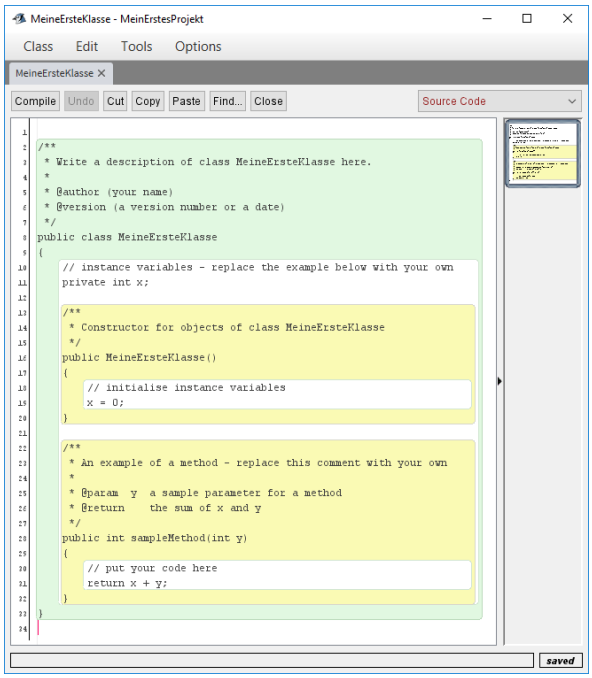

Abbildung 2.3: Der in mit BlueJ neu erstellten Klassen standardmäßig enthaltene Code. Syntax-Highlighting ist deaktiviert.

Code-Bereich

Jede neu erstellte Klasse enthält bereits den gleichen, vorgegebenen Beispielcode (siehe Abbildung 2.3). Diese enthält JavaDoc, ein privates Attribut, einen parameterloser Konstruktor und die Methode `beispielMethode(int)`. Diese Elemente überfordern Neulinge schnell. Ohne Unterweisung durch eine*n Lehrer*in oder andere Hilfestellungen ist es schwierig, eine Stelle zu finden, an

der man endlich die frisch gelernten Programmierkenntnisse ausprobieren kann. Jedoch findet sich im Menü „Ansicht“ die Option „Direkteingabe anzeigen“, mit der man (wie beworben) Java-Ausdrücke direkt ausführen kann.

Syntax-Highlighting ist auch in BlueJ vorhanden, kann aber deaktiviert werden: Modifikatoren, Schleifen und if/else (weinrot), andere Schlüsselwörter und primitive Datentypen (rot), JavaDoc (blau), Kommentare (grau), Strings (grün) und Wahrheitswerte (türkis) werden standardmäßig farblich hervorgehoben. Der Scope von Klassen, Methoden und Schleifen wird durch einen farbigen Hintergrund symbolisiert. Die Transparenz des Hintergrundes ist einstellbar.

Schon während des Programmierens, spätestens nachdem man den Button „Übersetzen“ betätigt hat, werden Kompilierfehler im Code rot markiert. Die Fehlermeldungen entsprechen den Javac-Meldungen, die in Abschnitt 2.3 auf Seite 13 genauer betrachtet werden.

Bewertung

BlueJ ist eine vereinfachte, aber sehr spezialisierte Entwicklungsumgebung, die für Schulen gut geeignet ist. Vor allem objektorientierte Programmierung lässt sich damit gut verdeutlichen. Allerdings ist die Komplexität der Einarbeitung für die Einzelarbeit höher als gewünscht. Mit einem Tutorial für BlueJ (zum Beispiel [10], 39 Seiten) verbringt man viel Zeit mit den Grundlagen des Programms, die man auch für erste Programmieraufgaben verwenden könnte.

Hat man die Benutzung des Programms verinnerlicht, fehlen immer noch Übungsaufgaben, um die tatsächliche Programmierung zu erlernen. Diese müssten in einem zweiten Fenster geöffnet werden. Falls die Aufgaben nicht auf BlueJ zugeschnitten sind, kann es zu Diskrepanzen in den Anweisungen kommen. Das eher grafische Interface von BlueJ, um neue Instanzen zu erstellen oder Methoden aufzurufen, lässt sich anders anwenden als eine selbst geschriebene `main`-Methode.

Genau diese Art der Anwendung von BlueJ wird von der Forschung kritisiert [7]. Dass keine `main`-Methode geschrieben werden muss, erspare zwar Erklärungen wie „Schreiben Sie das einfach, Sie werden es später verstehen“, jedoch müssten Schüler*innen möglicherweise Java-Grundlagen neu erlernen, wenn später eine andere Umgebung verwendet wird.

2.1.3 Java-Editor

Der Java-Editor [21] wurde von dem Informatiklehrer Gerhard Röhner entwickelt. Das Ziel war es, eine Entwicklungsumgebung für die Schule zu erstellen, die ohne große Leistungsanforderungen läuft und kostenlos verfügbar ist, sodass Schüler*innen sie auch zu Hause verwenden können. Der Java-Editor wurde für Windows entwickelt und kann nur unter Zuhilfenahme von Emulatoren unter Linux und macOS zum Laufen gebracht werden.

Das Programm besteht unter anderem aus einem normalen Code-Editor. Mit einem „Neu“-Knopf wird eine neue, leere Java-Datei erstellt. Ein Klassen-Modellierer ermöglicht es, den Aufbau einer Objektklasse mit einer grafischen Benutzeroberfläche zu erstellen, ohne Code eingeben zu müssen. Mit dem interaktiven Interpreter können eingegebene Befehle direkt hintereinander ausgeführt werden. Der Debugger erlaubt es, während der Ausführung eines Programms die Werte der Variablen auszulesen.

Code-Editor

Erstellt man eine neue Klasse (leer oder mit dem Klassen-Modellierer), wird der Code-Editor geöffnet (siehe Abbildung 2.4 auf der nächsten Seite). Das Syntax-Highlighting ist komplett frei einstellbar. Standardmäßig ist es sehr dezent: Schlüsselwörter fett, Strings und Zahlen blau, Kommentare und Dokumentation dunkelblau und kursiv. Geradezu knallig bunt wirkt dagegen die Markierung der Scopes in einem blassen grün, gelb, rot und blau.

Kompilierfehler werden im Code nicht markiert. Zum Kompilieren wird ein Knopf betätigt, danach werden die Fehler in einer Konsole ausgegeben. Wird ein Programm ausgeführt, öffnet sich eine Windows-Eingabeaufforderung. Hier wird das Programm mit dem Konsolenbefehl `java ...` ausgeführt. Exceptions werden daher von Java als Stack Trace ausgegeben und nicht weiter erläutert. Wie gut Neulinge nur anhand der Standard-Fehlermeldungen aus ihren Fehlern lernen und die Meldungen verstehen können, wird in Abschnitt 2.3 diskutiert.

Klassen-Modellierer

Öffnet man den Klassen-Modellierer (siehe Abbildung 2.5), erscheint ein Fenster, in dem man verschiedene Eigenschaften der Klasse verändern kann. Im ersten Reiter „Klasse“ werden der Name der Klasse und die Vererbung eingestellt und ausgewählt, ob die Klasse abstrakt ist. Im Reiter „Attribute“ können ebensolche erstellt werden mit den ausgewählten Modifikatoren (Zugriffstyp/static/final), dem angegebenen Namen, einem der vorgegebenen Datentypen (der auch eine selbst erstellte Klasse sein kann) und optional mit einem Wert. Außerdem können automatisch `get()`- und `set()`-Methoden erzeugt werden.

Der Reiter „Methoden“ bietet die größte Funktionalität. Es wird unterschieden zwischen Konstruktoren, Prozeduren und Funktionen. Die Optionen (Zugriffstyp/static/abstract), der Name (nicht bei Konstruktoren) und der Rückgabetyp (nur bei Funktionen) werden ausgewählt. Es können außerdem beliebig viele Übergabeparameter mit ausgewähltem Namen und Typ hinzugefügt werden.

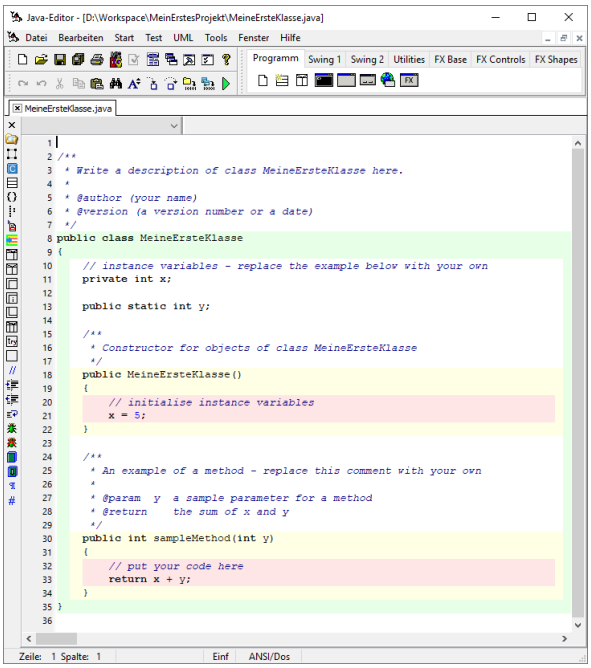

Abbildung 2.4: Der Java-Editor mit Beispielcode

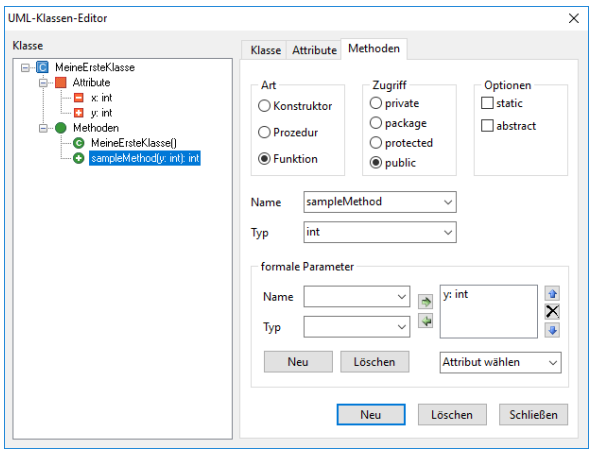

Abbildung 2.5: Der Klassen-Modellierer des Java-Editors

Debugger und Interpreter

Der Debugger ermöglicht es, den bestehenden Code Zeile für Zeile durchzugehen und dabei zu beobachten, wie sich Instanzen, Attribute, Variablen und der Aufrufstack verändern. Dies ist zur Veranschaulichung der Funktionsweise sowie der Fehlerbehebung hervorragend geeignet, wird jedoch typischerweise nur bei längerem Code benötigt.

Der Interpreter führt eingegebene Codezeilen aus und zeigt in einer Tabelle an, welche Variablen dabei erstellt wurden sowie deren Wert. Dies ist sehr nützlich, um die Auswirkung verschiedener Befehle zu beobachten. Man spart sich eine Konsolenausgabe im Code und kann auch mehrere Variablen gleichzeitig verfolgen. Der Interpreter hat einige hier nicht näher erläuterte Bugs, die allerdings bei einfachen Anwendungen nicht auftreten sollten.

Ein gravierender Fehler ist jedoch, dass die Syntax einiger Anweisungen nicht zwangsläufig stimmen muss. Um eine neue Variable oder Instanz zu erzeugen, ist kein abschließendes Semikolon nötig, bei anderen Befehlen jedoch schon. Außerdem funktioniert die Ausführung einer mehrzeiligen Anweisung nur umständlich. Ein String kann zudem nicht auf den Wert `null` gesetzt werden.

Bewertung

Der Java-Editor bietet die Möglichkeit, mit dem einfachen Editor schnell mit dem Programmieren in einer Datei anzufangen. Dabei muss jedoch gleich eine ganze Klasse erstellt werden, bis man sie ausführen kann. Abhilfe schafft der Interpreter, mit dem man einzelne Codezeilen ausführen und dabei auch Variablen und Instanzen erstellen kann. Die Fehler, die der Interpreter hat, machen ihn jedoch ungeeignet für die Einführung in die Programmierung, da eigentlich richtige Befehle zu Fehlern führen („Ich dachte ich darf einen String auf `null` setzen“) und eigentlich falsche Befehle trotzdem ein Ergebnis liefern („Man kann Variablen ohne Semikolon instanziieren?“).

Der Klassen-Modellierer ist ein nettes Tool, das allerdings keine Arbeit am Code selbst erfordert. Das Verständnis, welcher Code welchen Teil der Klasse bewirkt, bleibt möglicherweise aus (vergleiche hierzu auch den Abschnitt „Bewertung“ in 2.1.2 auf der vorherigen Seite). Symbolleisten am linken und oberen Rand des Code-Bereiches bieten viele Funktionalitäten, die möglicherweise Zeit beim Programmieren sparen, jedoch den Fähigkeiten der Nutzenden schaden können. In Klausuren an der TU Darmstadt gab es bereits Anmerkungen seitens der Studierenden wie „In Eclipse hätte ich jetzt ... gemacht“[1], die darauf hinweisen, dass derartige Unterstützungen für den Lernerfolg auch abträglich sein können.

Der Java-Editor bietet eine Entwicklungsumgebung, die, auch bezogen auf diese Arbeit, größeren Entwicklungsumgebungen wie Eclipse nur drei Dinge voraushat: die einfache Erstellung einer neuen Klasse (im Gegensatz zur vorherigen Auswahl von Workspace und Projekt), die geringere Leistungsanforderung und der Interpreter für die direkte Ausführung von Code. Erstere sind keine Vorteile, durch die für komplette Neulinge der Einstieg in Java deutlich erleichtert wird. Der Interpreter bietet zwar eine Mög-

[1] Die Eclipse Java IDE [24] bietet in dieser Hinsicht noch weitaus mehr automatisierte Code-Erzeugung.

lichkeit zum Ausprobieren von Code, verleitet dabei aber eher zum Herumprobieren und zur Fehlersuche als zum zielorientierten Programmieren.

2.2 Forschungsarbeiten zur Verbesserung der Lehre

Auch die Forschung beschäftigt sich schon seit längerer Zeit mit der Vereinfachung des Einstiegs in die Programmierung. Besonders sticht dabei die Arbeit von Brown und Altadmri [3] hervor: Die häufigsten Fehler vieler Programmierenden zu kennen, ist schließlich die wichtigste Voraussetzung, um diese Fehler in einem Programm zu erkennen und über sie aufzuklären. Auch sehr wichtig für diese Arbeit ist der Artikel von Marceau, Fisler und Krishnamurthi [12], in dem die Aussagekraft von Fehlermeldungen betrachtet wird. Es wird untersucht, wie Neulinge Fehlermeldungen und Markierungen im Code aufnehmen, und es wird vorgeschlagen, wie die Fehlermeldungen angepasst werden müssen, um diese Zielgruppe zu unterstützen.

2.2.1 Brown und Altadmri: Novice Java Programming Mistakes

Neil C. C. Brown und Amjad Altadmri von der University of Kent beschreiben in ihrem 2017 veröffentlichten Artikel [3] häufige Programmierfehler von Java-Neulingen und wie sich die Analyse von großen Datenmengen mit der Ansicht von Lehrenden vereinen lässt. Der Artikel enthält unter anderem die Beschreibung verschiedener Fehlerarten, ihrer Häufigkeit und Behebungsdauer sowie die Änderung dieser Statistiken über Zeit. Für die vorliegende Arbeit ist vor allem das Erkennen der vorgestellten Fehlerarten wichtig, um den Lernfortschritt durch erklärende und verständliche Fehlermeldungen zu verbessern.

Programmierfehler

Die Autoren nennen 18 Fehler aus einer vorherigen Arbeit, die sie in drei Gruppen aufteilen: falsch verstandene (oder vergessene) Syntax, Typenfehler und andere semantische Fehler. Für eine Auflistung aller Fehler siehe Tabelle 2.1 auf der nächsten Seite. Die Autoren weisen darauf hin, dass der Begriff „Fehler“ keine Unterscheidung darstellt zwischen „langfristigen Missverständnissen“ und „gelegentlichen Ausrutschern“. Das bedeutet, dass aus der Analyse nicht hervorgeht, ob es sich um einen typischen Flüchtigkeits- oder Tippfehler handelt oder ob die Lernenden das Programmieren tatsächlich noch nicht gut genug beherrschen.

Der Blackbox-Datensatz

Für die Analyse der Häufigkeit der Fehler wurde der Blackbox-Datensatz verwendet, der aus fast 100 Millionen Kompiliervorgängen besteht, die mit der Entwicklungsumgebung BlueJ (siehe Abschnitt 2.1.2 auf Seite 8) gesammelt wurden. Nur vier der 18 Fehler konnten rein aus den Javac-Fehlermeldungen der Kompilierfehler erkannt werden. Diese Erkenntnis bekräftigt die in Abschnitt 2.3 auf Seite 13 herausgearbeiteten Schlüsse. Bei den restlichen Fehlern, die zum Teil gar keinen Kompilierfehler hervorrufen, wurde die Erkennung mithilfe eines Parsers durchgeführt.

Aus der Häufigkeit der Fehler lässt sich - wegen der fehlenden Unterscheidung zwischen Missverständnissen und Ausrutschern mit Abstrichen - ablesen, welche Fehler im Code besonders behandelt werden müssen, worauf die Neulinge hingewiesen werden müssen und wo vermutlich der größte Erklärungsbedarf besteht. Gerade die häufigsten Fehler im Programm zu erkennen, soll ein Ziel dieser Arbeit sein, damit sich diese Fehler nicht einschleifen.

Ein weiterer Aspekt, den der Artikel untersucht ist die Dauer, bis ein Kompiliervorgang stattfindet, bei dem ein Fehler behoben wurde. Dies gibt Aufschluss darüber, wie schwierig es ist, falschen Code zu identifizieren und zu beheben. Dieser Wert sticht besonders bei Fehlern hervor, die keinen Kompilierfehler erzeugen, sondern semantischen Ursprungs sind. Diese Fehler sind besonders frustrierend, wenn man sich einfach nicht erklären kann, warum ein bestimmter Test nicht funktioniert, wo doch alle Bedingungen erfüllt zu sein scheinen. Um diese Frustration zu vermeiden, soll das hier beschriebene Programm ebensolche Fehler finden und darauf hinweisen.

Als Gesamtmaß der Schwere eines Fehlers schlagen die Autoren eine Kombination von Häufigkeit und Behebungsdauer vor. Tabelle 2.1 auf der nächsten Seite ist daher sortiert nach der gesamten Behebungsdauer aller Instanzen eines Fehlers, was die häufigsten und schwierigsten Fehler nach oben bringen soll.

Tabelle 2.1: Fehlerarten im Blackbox Datensatz, sortiert nach der von Brown und Altadmri definierten Schwere

Nr	Gruppe	Beschreibung
1	Syntax	unnötige, fehlende oder an dieser Stelle falsche Klammern (rund/eckig/geschweift) bzw. Anführungszeichen (einfach/doppelt)
2	Typ	Methoden werden mit falschen Argumenten aufgerufen (falsche Typen/Anzahl)
3	Semantik	Verwendung von '==' statt '.equals()'
4	Semantik	der Rückgabewert einer nicht-void-Methode wird ignoriert[2]
5	Syntax	der Zuweisungsoperator (=) wird mit dem Vergleichsoperator (==) verwechselt
6	Semantik	das Ende einer nicht-void-Methode wird ohne ein return erreicht
7	Semantik	eine Klasse soll ein Interface implementieren, enthält aber nicht alle nötigen Methoden
8	Syntax	falsches Semikolon direkt nach einem if-, for- oder while-Header, die ungewollt den Körper bildet
9	Syntax	Verwechslung von Booleschen Operatoren (&& und ||) mit Bit-Operatoren (& und |)
10	Semantik	Aufruf nicht-statischer Methoden als ob sie statisch wären
11	Syntax	Falsches Semikolon direkt nach dem Methodenkopf
12	Syntax	Angabe der Parametertypen vor den Parametern beim Aufruf einer Methode
13	Typ	inkompatible Typen zwischen dem Rückgabewert der Methode und der Variable, in der das Ergebnis gespeichert werden soll
14	Syntax	Klammern nach einem Methodenaufruf vergessen
15	Syntax	falsche Trennzeichen in for-Schleifen (Kommata statt Semikolons)
16	Syntax	falsche Schreibweise bei größer/kleiner gleich (=>, =< statt >=, <=)
17	Syntax	Verwendung von Schlüsselwörtern als Variablen- oder Methodenname
18	Syntax	Verwendung von geschweiften statt runden Klammern für eine if-Bedingung

2.2.2 Marceau, Fisler und Krishnamurthi: Mind Your Language

Die Autoren dieses Artikels [12] befassen sich mit der Aussagekraft von Fehlermeldungen und der Auswirkung auf die Korrekturfähigkeiten von Programmierneulingen. Sie stellen fest, dass Fehlermeldungen als pädagogisches Hilfsmittel helfen sollten, das Problem zu verstehen, das zu diesem Fehler geführt hat. Aus einer eigenen Studie von 2009 fassen sie zusammen, dass Schwierigkeiten mit dem Vokabular der Fehlermeldungen bestanden und die Markierungen im Code durch farbliche Kennzeichnung von Ausdrücken oft missverstanden wurden. Weiter behandeln sie die Frage, warum es Neulingen schwer fällt, effektiv auf Fehlermeldungen zu reagieren.

Es werden zwar Fehlermeldungen aus DrRacket behandelt, jedoch lassen sich die daraus geschlossenen Ergebnisse auch auf Java übertragen.

Markierungen im Code

Aus Interviews mit Studierenden erschließe sich, dass Markierungen im Code gleichbedeutend seien mit „hier bearbeiten“ oder „hier nach dem Problem suchen“. Dies sei unter anderem auch in der starken visuellen Präsenz von Markierungen begründet. Auch in der Fehlermeldung werde gezielt nach einem Lösungsvorschlag gesucht.

Die Autoren führen an, dass Markierungen im Code mehr Semantik benötigen. Es entstehe schnell der Eindruck, dass Programmieren ein planloses Raten sei - genau das Gegenteil des Lehrziels. Wie gut die Fehlermarkierungen im Java-Compiler zum Fehler passen, wird in Abschnitt 2.3.1 auf Seite 14 diskutiert.

Vokabular

Bezüglich des Vokabulars der Fehlermeldungen sei aufgefallen, dass Begriffe von Studierenden falsch genutzt oder umständlich umschrieben worden seien, dass der korrekte fachliche Ausdruck also nicht verwendet wurde. Dass das Fachvokabular schlecht beherrscht werde, untergrabe die Fähigkeit, auf Fehlermeldungen zu reagieren.

Um zu untersuchen, wie gut Studierende das Vokabular von DrRacket beherrschen, wurde an mehreren Universitäten ein Quiz durchgeführt. Es musste anhand eines Fachbegriffes eine Instanz dieses Begriffes in einem Beispielcode markiert werden. DrRacket war zuvor schon seit mindestens einigen Monaten genutzt worden. Das Ergebnis zeigt, dass nur vier von 15 abgefragten Begriffen von mehr als 50% richtig identifiziert wurden. Das werfe die Frage auf, ob die Studierenden die Bedeutung von Fehlermeldungen überhaupt herausfinden können.

[2] Dies ist nicht immer ein Fehler, z.B. wenn eine remove-Methode das entfernte Element zurückgibt, man es aber nicht benötigt.

Daraufhin wurden die Lehrenden befragt, ob die Begriffe aus dem Quiz in den Lehrveranstaltungen enthalten waren. Es stellte sich heraus, dass das Vokabular der Veranstaltungen teilweise nicht an das der Entwicklungsumgebung angepasst war. Wurde das passende Wort verwendet, seien beim Quiz um 13,8% bessere Ergebnisse erzielt worden. Hieraus schließen die Autoren, dass die Verbindung zwischen Lehrplan und Entwicklungsumgebungen enger sein müsse.

Empfehlungen

Es wird empfohlen, dass Fehlermeldungen keine Lösungsvorschläge enthalten sollten. Obwohl manche Fehler auf den ersten Blick einfache Lösungen hätten, würden davon nicht immer alle Fälle abgedeckt. Außerdem sollten die Markierungen im Code nicht dazu verleiten, falsche Änderungen am Code vorzunehmen. Es müsse sorgfältig überprüft werden, wie eine Markierung von Neulingen aufgenommen werde.

Ein weiterer angeführter Punkt ist die Vereinfachung von Vokabular. Obwohl die Präzision von Ausdrücken in manchen Fällen leide, sei es einfacher, Fehlermeldungen mit vertrauten Begriffen zu versehen. Die Komplexität der Fehlermeldungen müsse an den Kursfortschritt angepasst werden: Zu Beginn müssten die Formulierungen stark vereinfacht sein und mit Fortschreiten des Kurses komplexer und dadurch präziser werden. Zur Vereinfachung des Vokabulars wird eine Tabelle angeführt, die sich jedoch nur zu geringen Teilen auf Java übertragen lässt.

Außerdem seien Konstrukte, die von Nutzenden selbst erstellt worden seien, eine weitere mögliche Fehlerquelle, die in Fehlermeldungen oft übergangen werde. Wenn eine Funktion falsch definiert worden sei und dann aufgerufen werde, wie sie eigentlich sein solle, so deute die Fehlermeldung meistens auf den eigentlich richtigen Aufruf. Entwicklungsumgebungen in der Lehre müssten hier vermeiden, eine beeinflussende Wirkung durch die Fehlermeldung zu haben. Eine Ausnahme seien vordefinierte Methoden, deren Definition mit großer Wahrscheinlichkeit korrekt ist, sodass der Aufruf tatsächlich die Fehlerquelle ist.

Allgemein müsse die Interaktion mit Entwicklungsumgebungen stärker in die Lehre integriert werden. Tutoren, die bei Fehlermeldungen helfen, die Begriffe mit den tatsächlichen Stellen im Code zu verknüpfen, seien eine gute Hilfe. Außerdem sollten Markierungen im Code, Fehlermeldungen und die Reaktion darauf als zentraler Teil des Kurses behandelt werden. Um Neulinge hier gezielt zu unterstützen, empfehle es sich möglicherweise, einen eigenen Kurs zur Behandlung von Fehlermeldungen anzubieten. Entwicklungsumgebungen müssten schließlich ein Handbuch über die Semantik von Fehlermeldungen bereitstellen, um das Verständnis dafür zu verbessern.

2.2.3 Weitere Arbeiten

Hristova et al. geben eine Liste von häufigen Fehlern in Java an [7]. Diese Liste wurde auf das Anwendungsgebiet der Lehre gekürzt und dient als Grundlage für die Analyse des Blackbox-Datensatzes wie in Abschnitt 2.2.1 auf Seite 11 behandelt. Zudem wird beschrieben, dass für das Programm „Expresso“, das diese Fehler erkennen soll, der eingegebene Code als Vektor von Worten verarbeitet wird. Diese Herangehensweise erscheint für diese Arbeit überholt, zumal es bereits fertige Bibliotheken gibt, die Quellcode in einen abstrakten Syntaxbaum umwandeln können.

Einige interessante Denkanstöße liefern McCall und Kölling [13]. Die Autoren stellen fest, dass viele bisherige Arbeiten nur die Fehlermeldungen analysieren, jedoch eigentlich an den Fehlern selbst interessiert seien. Der Grund hierfür sei, dass die Verarbeitung von Fehlermeldungen sich einfach automatisieren lasse. Das Problem bestehe darin, dass der gleiche Fehler in anderem Kontext eine andere Fehlermeldung produzieren könne, jedoch umgekehrt auch die gleiche Fehlermeldung von komplett unterschiedlichen Fehlern hervorgerufen werden könne. Zudem seien bei der Verfassung der Fehlermeldungen eines Compilers üblicherweise keine pädagogischen Überlegungen eingeflossen. Deshalb seien auf diesen Meldungen basierte Forschungsarbeiten nur beschränkt aussagekräftig.

Im Laufe der Arbeit wurde manuell eine Kategorisierung von Fehlern erstellt. Mit zehn dieser Kategorien werden fast 59% aller Fehler abgedeckt. Zweimal wird von McCall und Kölling erwähnt, dass mit einer Verbesserung von Mitteilung und Verständnis nur dieser Fehler eine signifikante Auswirkung für pädagogische Zwecke erzielt werden könne. Diese Aussage war der Grund dafür, in dieser Arbeit nur die häufigsten Fehler selbst zu erkennen und zu behandeln, während für seltenere Fälle lediglich eine Präzisierung der Javac-Fehlermeldung vorgenommen wurde. Schließlich wird darauf hingewiesen, dass die Suche nach dem tatsächlichen Fehler von „Unknown Identifier“-Meldungen oft präziser sei, wenn nach ähnlich geschriebenen Bezeichnern gesucht würde. Dieser Denkanstoß lässt sich in zukünftigen Arbeiten sicher gut umsetzen.

2.3 Bewertung der Java-Fehlermeldungen

Dieser Abschnitt befasst sich mit der Aussagekraft, pädagogischen Qualität und Präzision der Java-Fehlermeldungen. Betrachtet werden hierbei Kompilierfehler sowie die häufigsten Laufzeitfehler. Anhand einiger Beispiele wird verdeutlicht, wieso das Verständnis von Neulingen für die Fehlerquelle in einigen Fällen behindert wird. Der Abschnitt 4.1.5 auf Seite 30 enthält die angepassten Fehlermeldungen, die pädagogisch stärker unterstützen sollen.

Eine wertvolle Grundlage zur Vervollständigung der Fehlermeldungen war das Dokument von Ben-Ari [1], in dem alle Fehlermeldungen des Java-Compilers und alle „Kern“-Laufzeitfehler aufgelistet werden. Es wird ebenfalls verdeutlicht, welche verschiedenen Ursachen zu den aufgelisteten Fehlermeldungen führen können. Codebeispiele und Beschreibungen wurden teilweise hieraus entnommen.

2.3.1 Kompilierfehler

Generell ist anzumerken, dass ein einziger Fehler im Code gleich mehrere Kompilierfehler hervorrufen kann. Im erwähnten Artikel erfolgt daher der Hinweis, dass nur versucht werden solle, den ersten Fehler zu beheben, um dann erneut zu kompilieren. Für sehr einfach gehaltene und kurze Programme, wie sie von Neulingen geschrieben werden, ist dieses Verfahren durchaus sinnvoll und wird eine Erwähnung in den Meldungen erhalten. Es ist wichtiger, zuerst korrekt zu programmieren, als sofort die höchste Effizienz anzupeilen.

Syntaxfehler

Der Fehler `... expected` wird zum Beispiel hervorgerufen, wenn eine Klammer oder ein Semikolon fehlt. Diese Meldung tritt leider nicht immer genau an der Stelle auf, an der der tatsächliche Fehler liegt. Außerdem kann die Meldung auf Zeichen hinweisen, die gar nicht fehlen und sogar falsch wären. Im Gegensatz zur Empfehlung, keine Lösungsansätze zu bieten [12], wird hier indirekt eine Lösung vorgeschlagen. Es ist jedoch bei dieser Fehlerart meist sehr wichtig, mindestens die ganze Zeile genau auf Korrektheit zu überprüfen.

String-Literale müssen in Java in Anführungszeichen eingeschlossen sein. Trifft dies nicht zu, meldet der Compiler `unclosed string literal`. Diese Fehlermeldung ist präzise und sagt genau aus, welches der Fehler ist - davon ausgehend, dass ein String absichtlich geöffnet wurde.

Die Meldung `illegal start of expression/type` tritt auf, wenn ein bestimmter Ausdruck erwartet, aber nicht gefunden wurde. Fehlerquellen für diese Meldung gibt es viele, was den Nutzen der Meldung für Neulinge stark verringert. Beispiele sind das Fehlen des neuen Wertes nach einem Zuweisungsoperator, die Verwendung von Modifikatoren vor Variablen in einer Methode, das Fehlen eines zweiten + oder − beim in- oder dekrementieren oder unausgeglichene Klammern.

```
1  public int foo() {
2    int a =              // fehlender Wert für Zuweisung
3    public int b = 0;    // falscher public-Modifikator
4    b-;                  // zweites - fehlt
5    return a + b; )      // Klammer nach einer Anweisung
6  }
```

Die gleichermaßen wenig aussagekräftig erscheinende Meldung `not a statement` kann bei vielen Ursachen auftreten und sagt Neulingen somit wenig darüber aus, worin genau der Fehler liegt. Sie weist lediglich darauf hin, dass ein Statement an dieser Stelle stehen soll, allerdings keines gefunden wurde. Jedoch gibt es einige Beispiele, bei denen die Lösungssuche durch diese Beschreibung nicht gerade unterstützt wird: das Fehlen eines Namens für eine Variable, eine `for`-Schleife mit falsch angeordneten Ausdrücken, eine falsche Aneinanderreihung von Bedingungen oder der Versuch, in einer Methode mehrere Werte zurückzugeben.

```
1  public int bar() {
2    int a = 1;
3    int = 1;                        // Name der Variablen fehlt
4    for(int i = 0; i++; i < a) {    // Test und Fortsetzung vertauscht
5      if (i < 1) {
6        ...
7      } else (i >= 1) {             // falscher boolescher Ausdruck nach else
8        ...
9      }
10   }
11   return a; b; c;                 // Versuch, 3 Variablen zurückzugeben
12 }
```

Bezeichner

Wurde ein Bezeichner falsch geschrieben oder schlicht nicht definiert, wird die Meldung `cannot find symbol` ausgegeben. Die Stelle, an der der Fehler auftritt, ist meist auch die Fehlerquelle. Die einzige Ausnahme ist, dass eine Variable oder Methode

bei der Definition falsch geschrieben, aber mit richtiger Schreibweise aufgerufen wurde. Lösen lässt sich dieses Problem meistens, wenn die Schreibweise von Anwendung und (falls selbst geschrieben) Definition genau überprüft wird.

```
1 public int foo(int frist) {       // Variable frist wurde falsch geschrieben aber nicht
2                                   // als Fehler markiert
3   itn second = "abc".lenght();    // Klasse itn und Methode lenght wurden nicht gefunden
4   return first + secnod;          // Variablen first und secnod wurden nicht gefunden
5 }
```

Der Versuch, zwei Variablen mit dem gleichen Namen zu deklarieren, oder einfach ein Missverständnis, wie man bereits deklarierte Variablen verwendet, führt zur Meldung `... is already defined in ...`. Dabei wird jeder Deklarationsversuch über den ersten hinaus als Fehler markiert. Die Meldung ist zwar präzise und zeigt den Ort des Fehlers gut an, jedoch lässt sie in pädagogischer Hinsicht die Möglichkeit außer Acht, dass ein grundlegendes Missverständnis über die Nutzung von Variablen besteht.

```
1 public int bar(int a, int b) {
2   int a = a * b;                  // Deklaration statt einfacher Zuweisung
3   return a;
4 }
```

Wird versucht, eine Variable als Array zu behandeln, die keines ist, resultiert daraus die Meldung `array required but ... found`. Diese Meldung ist in bestimmtem Kontext leicht misszuverstehen, wie das folgende Beispiel anhand möglicher Interpretationen der Meldung zeigt.

```
1 public int foo(int a) {
2   if (a[0])                 // muss etwa ein Array in eine if-Bedingung?
3     return a[0];            // muss die Methode ein Array statt einem int zurückgeben?
4   return 0[0];              // welcher int ist hier falsch?
5 }
```

Die Meldung `... has private access in ...` kommt bei Neulingen vermutlich eher selten vor. Erst mit der Verwendung externer oder selbst geschriebener Klassen wird die Sichtbarkeit von Variablen und Methoden relevant. Die Meldung deutet darauf hin, dass die Verwendung der Variable falsch ist, wobei bei selbst geschriebenen Klassen die Möglichkeit besteht, dass `get()`- und `set()`-Methoden fehlen oder die Sichtbarkeit falsch definiert wurde.

Berechnungen

Diese Gruppe von Fehlern ist syntaktisch korrekt, verletzt jedoch die Semantik von Java, insbesondere mit der Typprüfung verwandte Regeln.

Ein häufiger Fehler ist die Verwendung einer lokalen Variable, bevor diese initialisiert wurde. Die Meldung `variable ... might not have been initialized` weist auf das Problem hin, gibt allerdings keine genauere Erklärung, die für Neulinge nützlich sein könnte.

```
1 int i;
2 i++;     // i wurde noch nicht initialisiert
```

Wird eine Methode, für die es nur eine Definition gibt, mit der falschen Anzahl von Parametern aufgerufen, meldet der Compiler `method ... in class ... cannot be applied to given types`. Die Meldung gibt zusätzlich Informationen darüber, welche Argumenttypen übergeben und welche erwartet wurden.

Wird hingegen eine Methode, die für verschiedene Parameterlisten definiert ist, mit der falschen Anzahl an Parametern aufgerufen, lautet die Nachricht `no suitable method found for ...`. Die gleiche Meldung erscheint, wenn die übergebene Anzahl der Parameter mehrfach vorhanden ist, jedoch keine Definition passende Typen erwartet. Diese Meldung listet sogar alle vorhandenen Definitionen der Methode mit den erwarteten Parametertypen auf und jeweils einen Grund, warum die übergebenen Parameter nicht passen.

Die letzte Möglichkeit ist, eine Methode, die für eine Parameterzahl n nur einmal definiert ist, mit n Parametern aufzurufen, von denen mindestens einer den falschen Typ hat. Die Fehlermeldung lautet dann `incompatible types` und geht weiter mit `... cannot be converted to ...` oder `possible lossy conversion from ... to ...`, je nachdem, welcher dieser Fälle eintritt.[3] Genau dieselbe Fehlermeldung wird angezeigt, wenn man einer Variable einen Wert zuweisen

[3] Mit der Compiler-Option `-Xdiags:verbose` werden diese (vereinfachten) Nachrichten durch die zuvor beschriebenen ersetzt.

will, der nicht zum Typ der Variable passt. Das kann zum einen die Zuweisung von komplett inkompatiblen Typen sein, zum anderen eine Zuweisung, bei der der zugewiesene Wert an Präzision verlieren könnte.

Die ersten vier Möglichkeiten können durch zwei verschiedene Fehler verursacht worden sein: Entweder der Aufruf der Methode wurde falsch programmiert oder die Definition ist fehlerhaft. Beide dieser Fehler können wiederum aus zahlreichen möglichen Fehleinschätzungen oder Missverständnissen entstanden sein, sodass es schier unmöglich scheint, eine genaue Fehlerangabe zu liefern. Es wäre aber sinnvoll, die Möglichkeit einer fehlerhaften Definition bei durch den*die Programmierer*in selbst geschriebenen Methoden in Betracht zu ziehen [12].

Die letzten beiden Fehlermeldungen zeigen, dass hier komplett verschiedene Fehler zur gleichen Meldung führen. Das erschwert die Reaktion auf eine solche Fehlermeldung, da Neulinge unter Umständen nicht wissen, welchen der Fehler sie gemacht haben. Es empfiehlt sich daher, die Compiler-Option `-Xdiags:verbose` zu verwenden, um Zuweisung und Methodenaufruf in den Kompilierfehlern unterscheiden zu können.

Einige Operatoren in Java sind auf bestimmte Datentypen beschränkt. Wird diese Beschränkung nicht beachtet, folgt daraus die Meldung `bad operand type(s) for ...`. Einen Sonderfall bilden die unären Operatoren zum Inkrementieren und Dekrementieren. Werden diese bei einem Wert statt einer Variable verwendet, lautet die Meldung `unexpected type` mit einer Angabe, welcher Typ erwartet und welcher gefunden wurde. Diese Meldungen sind präzise und sagen aus, was für diesen Operanden falsch ist. Pädagogisch gesehen fehlt aber die Betrachtung der Möglichkeit, dass der falsche Operator verwendet wurde oder es gar keinen gibt, der den gewünschten Zweck erfüllt.

```
1 String c = "abc" - "bc";    // Operator - ist nicht für Strings definiert
2 int a = 5++;                // Operator ++ erfordert eine Variable
```

Fehler bei `return`-Ausdrücken

Falls in Methoden mit einem Rückgabewert nicht jede mögliche Kombination von Bedingungen zu einem `return`-Ausdruck führt, wird die Meldung `missing return statement` angezeigt. Außer der fehlenden Erklärung, dass jeder Pfad durch die Methode einen solchen Ausdruck haben muss, und der Betrachtung der Möglichkeit, dass der Methodenkopf falsch definiert wurde und die Methode gar keinen Rückgabewert haben soll, ist an dieser Meldung nichts auszusetzen.

Sollte bei einem `return`-Ausdruck kein Rückgabewert angegeben sein, obwohl die Methode laut der Deklaration einen solchen haben sollte, erscheint die Fehlermeldung `incompatible types: missing return value`. Wie vorhin beschrieben fehlt hier lediglich eine Erklärung und die Erwähnung der Möglichkeit, dass die Methode tatsächlich keinen Rückgabewert haben soll, dies aber im Kopf der Methode falsch angegeben wurde.

Weiterhin gibt es den Fehler, einen Rückgabewert zu liefern, wenn der Rückgabetyp der Methode `void` ist. In der entsprechenden Meldung `incompatible types: unexpected return value` wird nicht deutlich, dass das Schlüsselwort `void` aussagt, dass die Methode keinen Rückgabetyp haben soll.

Die Meldung `invalid method declaration; return type required` weist darauf hin, dass bei der Deklaration der Methode der Rückgabetyp fehlt. Das kann verschiedene Gründe haben: Entweder wurde der Rückgabetyp schlicht vergessen oder aber die Absicht war es, einen Konstruktor zu schreiben, und der Klassenname wurde falsch geschrieben. Gerade für letzteren Fall ist die Fehlermeldung nicht hilfreich, da sie in die falsche Richtung lenkt und dazu drängt, einen Rückgabetyp einzufügen. Dieses Vorgehen führt schnell zu planlosem Raten anhand der Kompilierfehler [12].

Falls nach einem `return`-Ausdruck, der nicht an eine Bedingung geknüpft ist, weiterer Code steht, so meldet der Compiler `unreachable statement`. Diese Meldung bedarf lediglich einer ausführlicheren Erklärung, dass die Ausführung der Methode nach dem `return`-Ausdruck beendet wird und dass jeglicher Code danach nie ausgeführt wird.

Statische Methoden und Variablen

Werden nicht-statische Methoden oder Variablen aus einem statischen Kontext aufgerufen, erscheint die Meldung `non-static method/variable ... cannot be referenced from a static context`. Die Ursache für einen solchen Fehler kann zum einen sein, dass bei der Definition der aufgerufenen Methode oder Variable der `static`-Modifikator vergessen wurde. Es ist auch möglich, dass der Kontext, in dem der Fehler entstanden ist, statisch ist, ohne dass dies beabsichtigt war. Schließlich besteht die Möglichkeit, dass das ganze Konzept von statischen und nicht-statischen Methoden und Variablen noch nicht verinnerlicht wurde oder eine grundlegende Fehlinformation besteht. Um alle Möglichkeiten abzudecken, muss eine pädagogisch orientierte Formulierung der Fehlermeldung die ersten beiden Möglichkeiten in Betracht ziehen und zusätzlich weiterführende Hilfe über das `static`-Schlüsselwort anbieten.

Kaskadierende Fehlermeldungen

Ein weiteres großes Problem für Neulinge ist, dass ein kleiner Fehler im Code mitunter eine ganze Reihe von Fehlermeldungen erzeugen kann. Diese Wand von Text, die sich scheinbar gegen den selbst geschriebenen Code richtet, kann einschüchternd wirken und den Lernfortschritt hemmen, indem die Motivation eingeschränkt wird. Ein Extrembeispiel mit den entsprechenden Fehlermeldungen ist hier aufgeführt.

```
1 int foo(int f) {
2   if (int f[4] ==[5] _[6] 5)[7] {[8]
3     return f - 2;
4   }
5   return[9] _[10] f +[11] _ 2[12];
6 }
```

Der einzige Fehler ist das überflüssige `int` in Zeile 2, wodurch die neun angemerkten Fehlermeldungen erscheinen und der Compiler zusätzlich davon ausgeht, dass die letzte geschweifte Klammer die Klasse schließt statt der Methode.

2.3.2 Exceptions und Errors

Obwohl ein Programm erfolgreich kompiliert wurde, kann es während der Ausführung zu Fehlern kommen. Werden diese Fehler nicht korrekt behandelt oder von vornherein vermieden, so wird die Ausführung des Programms an der Stelle des Fehlers gestoppt oder sogar das Programm beendet. Es folgt eine Auflistung von Exceptions und Errors, die bei der Verwendung des zu erstellenden Programms häufig auftreten könnten und mit denen Java-Neulinge vertraut gemacht werden müssen. Es wird auch die Aussagekraft der in der Exception enthaltenen weiterführenden Informationen diskutiert. Auch für diesen Teil war das Dokument von Ben-Ari [1] hilfreich.

Exceptions mit Indizes

Abstrakte Datenstrukturen, vor allem Arrays und Strings, haben eine bestimmte Zahl an Elementen. Wird versucht, auf ein Element zuzugreifen, das es nicht gibt, kommt es zu einer Exception. Drei Typen dieser Exceptions sind in der Java-API enthalten.

Der erste allgemeine Typ ist die `IndexOutOfBoundsException`. Diese erhält eine Beschreibung, aber keine konkreten Angaben zu Index und Anzahl der enthaltenen Elemente. Diese Information wäre für die Fehlerfindung hilfreich, wird von der Klasse jedoch nicht erzwungen. In der Nachricht sind diese Angaben aber möglicherweise vorhanden.

Von dieser Exception erben `ArrayIndexOutOfBoundsException` und `StringIndexOutOfBoundsException`, die beide für die jeweiligen Datenstrukturen geworfen werden. Im Gegensatz zur `IndexOutOfBoundsException` verfügen diese Klassen über einen Konstruktor, der einen Index als Parameter erhält und daraus eine Nachricht generiert. Aus Sicht des hier zu erstellenden Programms wäre es nützlich, den Index auch als Parameter zu speichern, statt ihn nur in die Nachricht aufzunehmen.

Aus pädagogischer Sicht bedürfen diese Exceptions keiner aufwendigen Erklärung. Ist die Bedeutung von „IndexOutOfBounds“ erst bekannt, kann der Fehler meist schnell identifiziert werden. Ein Hinweis, dass eine Überprüfung auf die Anzahl der vorhandenen Elemente sinnvoll ist, wäre jedoch hilfreich und wünschenswert.

Eine selten vorkommende Exception ist die `NegativeArraySizeException`, die nur eine kurze Erklärung benötigt: Arrays dürfen keine negative Anzahl an Elementen haben.

Dereferenzierung von `null`

Wenn ein Zeiger als Attribut neu deklariert wird oder als Parameter einer Methode `null` übergeben wird, dürfen diese Zeiger nicht dereferenziert werden. Das Ergebnis ist eine `NullPointerException`. Hier besteht durchaus etwas Erklärungsbedarf.

4 '.class' expected
5 illegal start of expression
6 ';' expected
7 illegal start of expression
8 ';' expected
9 illegal start of type
10 ';' expected
11 illegal start of type
12 <identifier> expected

Es könnte unklar sein, was genau der Ausdruck `null` bedeutet, wieso ein Zeiger an dieser Stelle diesen Wert hat, was ein Zeiger überhaupt ist und vor allem wie man die Exception vermeiden kann. Diese Informationen muss ein auf die Vereinfachung des Einstiegs ausgerichtetes Programm liefern.

Fehler bei Berechnungen

Wird bei einer Ganzzahldivision durch 0 geteilt oder die Restklasse modulo 0 bestimmt, wirft das Programm eine `ArithmeticException`. Die dabei angezeigte Nachricht lautet „/ by zero“. Da diese Exception einen einfachen Grund hat, ist es leicht, eine gute Erklärung zu liefern, wie sich dieser Fehler vermeiden lässt.

Der Versuch, einen String zu einer Zahl umzuwandeln, der keine erlaubte Zahl darstellt, führt zu einer `NumberFormatException`. Dies kann durch eine zu große oder zu kleine Zahl oder unerlaubte Zeichen im String verursacht werden. Diese Exception beschreibt recht eindeutig, welcher Fehler vorliegt, und gibt in der Nachricht auch den String an, der zum Fehler geführt hat. Allerdings ist in der Nachricht nicht enthalten, welches Zeichen genau zum Fehler geführt hat. So ist beispielsweise die Anweisung `Double.valueOf("1,0")` nicht erlaubt, da Gleitkommazahlen mit einem Punkt getrennt werden müssen. Dieser Hinweis fehlt und sollte der Dokumentation beigefügt werden.

Speicherfehler

Bei einer zu tiefen Rekursion von Methodenaufrufen kommt es zwangsläufig zu einem `StackOverflowError`. Dieser benötigt nur eine kurze Beschreibung, vermutlich fehlt im geschriebenen Code ein Rekursionsanker oder die gleiche Methode wird versehentlich wieder aufgerufen. Möglich ist auch, dass der Rekursionsschritt nicht korrekt ausgeführt wird und die Parameter nie den Rekursionsanker erreichen.

Ähnlich verhält es sich mit dem `OutOfMemoryError`, bei dem zu wenig Speicherplatz verfügbar ist, um die angeforderte Aktion auszuführen. Dies liegt im hier vorliegenden Anwendungsfeld meist an zu großen Arrays oder anderen Datenstrukturen, weshalb dieser Fehler recht einfach zu bestimmen ist. Eine kurze Erläuterung sollte hier deshalb ausreichen.

Andere Fehler

Ein weiterer Fehler entsteht, wenn ein Objekt konvertiert werden soll, ohne dass die gewünschte Konvertierung möglich ist. Es resultiert eine `ClassCastException`, in deren Nachricht darüber informiert wird, welche Typen nicht kompatibel sind. Hier ist es für ein pädagogisches Programm wichtig, eine Einführung zu liefern, wie die Konvertierung von Klassen funktioniert und wann sie möglich ist.

Die `UnsupportedOperationException` wird geworfen, wenn ein Interface eine Methode deklariert, die eine erbende Klasse jedoch nicht implementiert. Ein Beispiel ist der Aufruf der Methode `add(E)` des Interfaces `List<E>`, wenn die Instanz durch `Arrays.asList(E...)` erzeugt wurde. Hier gibt es wenige Informationen, die man allgemein liefern kann, um das Problem zu lösen. Die einzige Möglichkeit ist ein Verweis auf die Dokumentation der verwendeten Klassen und dass die Implementierung einer Methode vom dynamischen, nicht vom statischen Typ abhängt.

Der letzte hier behandelte Fehler ist die `ConcurrentModificationException`. Diese tritt auf, wenn während der Verwendung eines `Iterators` die zugrundeliegende Datenstruktur verändert wird. Dies betrifft auch die Verwendung der foreach-Schleife. Hier ist es pädagogisch wichtig zu erwähnen, dass es alternative Möglichkeiten gibt, eine Datenstruktur zu iterieren. Eine `for`-Schleife, die die Indizes iteriert, ist zwar nicht immer performant, aber meistens weniger fehleranfällig, wenn die Indizierung beim Löschen oder Hinzufügen von Elementen korrekt geändert wird. Außerdem ist es wichtig darauf hinzuweisen, dass die Verwendung eines `ListIterators` statt einer foreach-Schleife die Methoden `add`, `set` und `remove` des `ListIterators` offenlegt. Hierbei ist jedoch die Änderung der Position des aktuellen Zeigers zu beachten.

2.4 Vorstellung von JavaFX

JavaFX ist eine Zusammensetzung von Grafik- und Medienpaketen, mit dem plattformübergreifend konsistente Anwendungen erstellt werden können. Seit Java SE 7 ist es in JDK und JRE enthalten und bietet eine moderne Alternative zu AWT und Swing. Die Verwendung von CSS erlaubt es, Elemente getrennt vom Code zu designen. Da JavaFX die neueste grafische Oberfläche für Java ist, erscheint es sinnvoll, von Swing auf JavaFX umzusteigen.

Jede JavaFX-Anwendung erbt von der Klasse `javafx.application.Application`. Um die Anwendung zu starten, wird die Methode `launch(String[])` aufgerufen. Hier beginnt der Lebenszyklus der Anwendung. Es folgt ein Aufruf der Methode `init()`, gefolgt vom Aufruf der Methode `start(javafx.stage.Stage)`. Letztere Methode erhält als Parameter bereits eine Instanz der zentralen Klasse `Stage`. Diese Klasse bildet die „Bühne“, auf der die Anwendung angezeigt wird, und erscheint nach dem Aufruf der Methode `show()` als Fenster auf dem Bildschirm. Die Anwendung läuft von hier, bis das

letzte Fenster geschlossen oder die Methode `Platform.exit()` aufgerufen wird. Dann folgt ein letzter Aufruf der Methode `stop()`, die von der Klasse `Application` geerbt wird und überschrieben werden kann. Die einzige abstrakte Methode der Klasse `Application` ist `start(Stage)`, weshalb diese Methode überschrieben werden muss. Ein Minimalbeispiel einer JavaFX-Anwendung zeigt Abbildung 2.6. Bei Ausführung des Programms wird ein leeres Fenster ohne Titel angezeigt.

```
1  package de.tu_darmstadt.informatik.skorvan.example;
2
3  import javafx.application.Application;
4  import javafx.stage.Stage;
5
6  public class JavaFXExample extends Application {
7
8      @Override
9      public void start(Stage primaryStage) throws Exception {
10         primaryStage.show();
11     }
12
13     public static void main(String[] args) {
14         launch(args);
15     }
16
17 }
```

Abbildung 2.6: Code für ein minimales Beispielprogramm für JavaFX

Für die weiteren vorgestellten Elemente zeigt Abbildung 2.7 auf der nächsten Seite ein kleines Beispielprogramm. Alle folgenden Zeilenangaben beziehen sich auf den dort gezeigten Code. Die Darstellung des Programmes ist in den Abbildungen 2.8 bis 2.11 auf Seite 21 gezeigt.

2.4.1 Die Methode `start(Stage)`

Diese Methode dient der Initialisierung der grafischen Oberfläche. Die übergebene Instanz der Klasse `Stage` enthält einige Methoden, die für die Darstellung und Verwendung des gesamten Fensters relevant sind. Wichtig sind vor allem die Methode `setTitle(String)` (Zeile 40), mit der der Name oder Titel des Fensters festgelegt wird, die Methode `setScene(Scene)` (Zeile 41), die eine auf dem Fenster oder der „Bühne“ anzuzeigende Szene übergibt, und die Methode `show()` (Zeile 42), die wie zuvor erwähnt das Fenster sichtbar macht.

Eine neue Szene wird mit dem Konstruktor `Scene(Parent, double, double)` instantiiert (Zeile 37). Das zweite und dritte Argument sind die Breite und Höhe der Szene, an die sich die Bühne anpasst, der die Szene zugewiesen wird. Sie bestimmen also die Größe des Fensters. Das erste Argument ist der Wurzelknoten des sogenannten Szenegraphen. Diesem Wurzelknoten können als Kinder weitere Elemente hinzugefügt werden, deren Anordnung durch die Implementierung des jeweiligen direkten Elternknotens festgelegt wird. Im Szenegraphen hat jeder Knoten außer dem Wurzelknoten genau einen Elternknoten. Ein Beispiel für einen Wurzelknoten ist die Klasse `javafx.scene.layout.VBox`, instantiiert in Zeile 17. Dieser Knoten ordnet alle Kindknoten vertikal untereinander an. Die Kindknoten sind in einer `ObservableList<Node>` gespeichert, die mit der Methode `getChildren` abgerufen werden kann. Dieser können dann mit den Methoden `add(Node)` oder `addAll(Node...)` weitere Knoten hinzugefügt werden. Dies geschieht in den Zeilen 27, 31 und 35.

Für die Verwendung von CSS wird der `ObservableList<String>`, die durch `Scene.getStylesheets()` abgerufen wird, mit der Methode `add(String)` eine URL zur CSS-Datei hinzugefügt (Zeile 38). Alle so hinzugefügten Dateien werden beim Design der Szene berücksichtigt. Die Datei „style.css“ zeigt Abbildung 2.12 auf Seite 21. Dort wird das Design festgelegt, das in den Abbildungen 2.8 bis 2.11 auf Seite 21 sichtbar ist. Eine Gesamtübersicht über alle möglichen CSS-Anweisungen und die Bezeichnungen der verschiedenen Knoten liefert der JavaFX CSS Reference Guide [17].

2.4.2 Die Klasse `Node` und wichtige Erben

`javafx.scene.Node` ist die Basisklasse für alle Knoten des Szenegraphen. Knoten können als Teil einer Szene auf einer Bühne dargestellt werden, also auf dem Fenster erscheinen. Folgende Erben dieser Klasse sind häufig verwendete Beispiele.

Die Klasse `javafx.scene.control.Button` stellt einen Knopf dar, mit dem Nutzer*innen eine Aktion bewirken können. Im Beispiel wird der Knopf in Zeile 21 mit einer Beschriftung initialisiert. Die Aktion, die bei einem Klick ausgeführt werden soll, wird in den Zeilen 22 bis 24 als Lambda-Ausdruck angegeben. Standardmäßig sind Knöpfe so klein wie möglich, sodass sie die

Beschriftung noch anzeigen können, falls genug Platz vorhanden ist. Die Anordnung ist standardmäßig oben links. Um beides zu ändern, müssen die Methoden `setPrefSize(double, double)` und `setAlignment(javafx.geometry.Pos)` verwendet werden (Zeilen 25 und 26).

Mit einem `javafx.scene.control.Label` kann ein einfacher Text angezeigt werden. Die Initialisierung in Zeile 29 des Beispiels enthält bereits den Text. Da der Text die Breite des Fenster überschreiten würde, wird dieser zunächst nicht vollständig angezeigt. Abhilfe schafft die Methode `setWrapText(boolean)`, mit der eingestellt werden kann, ob der Text am Zeilenende automatisch umgebrochen werden soll.

Das `javafx.scene.control.TextField` ist eine einfache Möglichkeit, eine kurze Eingabe zu tätigen. Nach der Initialisierung in Zeile 33 wird mit der Methode `setPromptText` (Zeile 34) ein Hinweis eingestellt, welcher Inhalt hier erwartet wird.

Eine vollständige Übersicht über alle Funktionen von JavaFX findet sich in der offiziellen Dokumentation [16]. JavaFX enthält seit der Java-Version 8u40 (März 2015) ebenfalls die Möglichkeit, anpassbare Dialoge anzuzeigen. Eine sehr gute Übersicht liefert der Informatiklehrer Marco Jakob [8].

```
1  package de.tu_darmstadt.informatik.skorvan.example;
2
3  import javafx.application.Application;
4  import javafx.geometry.Insets;
5  import javafx.geometry.Pos;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Button;
8  import javafx.scene.control.Label;
9  import javafx.scene.control.TextField;
10 import javafx.scene.layout.VBox;
11 import javafx.stage.Stage;
12
13 public class JavaFXExample extends Application {
14
15     @Override
16     public void start(Stage primaryStage) throws Exception {
17         VBox vbox = new VBox();
18         vbox.setPadding(new Insets(20d));
19         vbox.setSpacing(10d);
20
21         Button button = new Button("Klick' Mich!");
22         button.setOnAction((evt) -> {
23             button.setText("Danke.");
24         });
25         button.setPrefSize(100d, 40d);
26         button.setAlignment(Pos.TOP_CENTER);
27         vbox.getChildren().add(button);
28
29         Label label = new Label("Dies ist ein Label mit einer Nachricht.");
30         label.setWrapText(true);
31         vbox.getChildren().add(label);
32
33         TextField textField = new TextField();
34         textField.setPromptText("Eingabe");
35         vbox.getChildren().add(textField);
36
37         Scene scene = new Scene(vbox, 220, 180);
38         scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());
39
40         primaryStage.setTitle("TestTitle");
41         primaryStage.setScene(scene);
42         primaryStage.show();
43     }
44
45     public static void main(String[] args) {
46         launch(args);
47     }
48
49 }
```

Abbildung 2.7: Beispielcode zur Verwendung von JavaFX

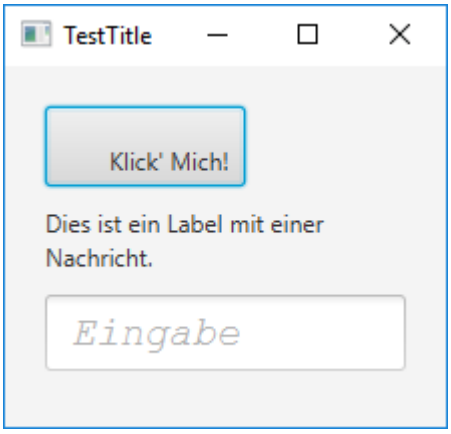

Abbildung 2.8: Das Beispielprogramm nach dem Start

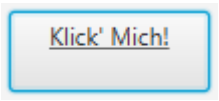

Abbildung 2.9: Der Knopf, während die Maus darüber liegt (`hover`)

Abbildung 2.10: Der Knopf, während die Maus gedrückt wird (`armed`)

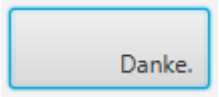

Abbildung 2.11: Der Knopf nach dem Mausklick

```
.button {
    -fx-alignment: bottom-right;
}

.button:hover {
    -fx-alignment: top-center;
    -fx-underline: true;
}

.button:armed {
    -fx-text-fill: #dd4455;
}

.text-field {
    -fx-font: italic 20 monospace;
}
```

Abbildung 2.12: Die Datei „style.css“, die vom Beispielcode in Abbildung 2.7 auf der vorherigen Seite verwendet wird. Das Design ist sichtbar in den Abbildungen 2.8 bis 2.11.

3 Anforderungsanalyse

Dieses Kapitel befasst sich mit den Anforderungen, die das Programm erfüllen muss, um den Zielen dieser Arbeit gerecht zu werden. Es unterscheidet dabei zwischen dem (Haupt-)Programm, mit dem der Zielgruppe der Einstieg erleichtert werden soll, dem Programm für Lehrende, um Aufgaben zu erstellen, und den gemeinsamen Eigenschaften, die beide Programme miteinander verbinden.

3.1 Gemeinsame Eigenschaften

Sowohl das Programm für Lehrende als auch das Programm für Studierende müssen in bestimmten Punkten übereinstimmen, damit von ersterem erstellte Aufgaben auch von letzterem gelesen werden können. Das Gleiche gilt für Einträge in die Wissensdatenbank, die weiterführende Erklärungen liefern sollen.

3.1.1 Eigenschaften der Aufgaben

Für die eindeutige Identifikation und Unterscheidung von Aufgaben, sowohl bei der Verteilung als auch bei der Bearbeitung, muss jede Aufgabe eindeutig identifiziert werden können. Wenn eine Aufgabe anhand dieses Identifikators als Duplikat erkannt wurde, soll Nutzenden eine Meldung angezeigt werden, die eine Auswahl anbietet, welches der beiden Duplikate gelöscht werden soll.

Jede Aufgabe soll einen kurzen Titel enthalten, der den Inhalt oder das bearbeitete Thema der Programmiersprache beschreibt. Nutzende sollten im Idealfall daran erkennen können, ob die Bearbeitung dieser Aufgabe für sie zu diesem Zeitpunkt sinnvoll erscheint. Eine detaillierte Beschreibung der Aufgabenstellung in Textform soll ebenfalls enthalten sein. Mit dieser Aufgabenstellung und eventuell weiteren Hilfestellungen aus der Wissensdatenbank soll es allen Nutzenden, die sich mit der Hilfestellung beschäftigen, möglich sein, die Aufgabe erfolgreich zu lösen.

Grundsätzlich soll das Programm drei Arten von Aufgaben unterstützen: Für die ersten Schritte in Java bietet es sich an, nur wenige zu schreibende Codezeilen zu fordern. Dabei sollte noch nicht einmal ein Methodenkopf geschrieben werden. Diese Art erfordert es, dass Informationen darüber vorliegen, wie der eingereichte Code in ein Framework integriert werden muss, um der Aufgabe und den Tests zu entsprechen.

Der nächste Schritt ist, eine ganze Methode selbst zu schreiben, die dann getestet und ausgeführt werden kann. Da Methoden in Java nicht einzeln geschrieben werden dürfen, ist hier wieder ein Framework erforderlich, das mindestens eine Klasse erstellt, mit der die geschriebene Methode dann getestet werden kann.

Am Ende der Verwendung dieses Programmes sollen Neulinge eine ganze Klasse selbst schreiben können. Die Erstellung mehrerer Klassen, Vererbung und weitere objektorientierte Konzepte werden in dieser Arbeit nicht behandelt. Daher muss auch die Anweisung, in welchem Package eine Klasse liegt, nicht eingebracht werden. Das Ziel des Programms ist es, lediglich eine anfängliche Einführung zu geben, bis auf eine umfangreichere, aber auch weniger pädagogisch unterstützende Entwicklungsumgebung wie zum Beispiel Eclipse [24] umgestiegen wird. Als Zwischenschritt zur weiteren Einführung in die Objektorientierung stellt BlueJ [11] (behandelt in Abschnitt 2.1.2) eine gute Entwicklungsumgebung dar.

Es ist für die ersten beiden beschriebenen Typen wichtig anzumerken, dass die Verwendung von externen oder API-Klassen, die einen `import`-Ausdruck benötigen, bei der Erstellung des Frameworks zu berücksichtigen ist. Wenn der von Neulingen geschriebene Code in ein Framework eingefügt wird, besteht im Nachhinein keine Möglichkeit, einen import-Ausdruck einzufügen. Es ist wichtig, sowohl Neulinge als auch Aufgabenersteller*innen darüber in Kenntnis zu setzen.

Um eine Lösung auf Korrektheit zu überprüfen, muss jede Aufgabe eine Testdatei zur Verfügung stellen, die nach der erfolgreichen Kompilierung ausgeführt wird. Es sollen sowohl öffentliche als auch private Tests möglich sein. Das bedeutet, dass beim Fehlschlag eines Tests nicht immer die übergebenen Parameter, das tatsächliche und das erwartete Ergebnis angezeigt werden, damit Nutzende nicht dazu verleitet werden, alle Tests durch eine Reihe von Abfragen zu lösen, die die getesteten Werte erwarten und das entsprechende Ergebnis zurückgeben. Auch ist es sinnvoll, die Testdateien zu verschlüsseln, damit nicht durch Öffnen der Datei die Tests ausgelesen werden können.

Für die korrekte Erzeugung einer kompilierbaren Java-Datei aus der gelösten Aufgabe ist es notwendig, den erwarteten Klassennamen anzugeben, damit eine entsprechende JAVA-Datei erstellt werden kann. Der Klassenname muss ohnehin in Testdateien und bei den ersten beiden Aufgabenarten im Framework angegeben werden und diese Eingabe ist gleichzeitig eine weitere Kontrolle, ob Fehler bei der Erstellung oder Änderung einer Aufgabe gemacht wurden.

3.1.2 Einträge in der Wissensdatenbank

Um Fehlermeldungen, Exceptions und andere Konstrukte benutzerfreundlich und verständlich zu erklären, sind mitunter mehrere Sätze nötig. Beispiele unterstützen dabei, das Erklärte besser zu verstehen. Würde dies alles der Fehlermeldung selbst hinzugefügt

werden, wären Nutzer*innen schnell von einer Wand aus Text überwältigt. Zudem würde bei mehreren Instanzen der gleichen Fehler der gleiche Text angezeigt. Diese Überlegung führte zu dem Entschluss, eine Wissensdatenbank anzulegen, die Einträge über wichtige Themen der Java-Programmierung enthält.

Die Wissensdatenbank soll primär an Stellen weiterhelfen, an denen der Detailgrad der Fehlermeldungen oder die eigenen Programmierkenntnisse nicht ausreichen, um eine Aufgabe erfolgreich zu lösen. Wünschenswert ist, dass es eine Verknüpfung von Fehlermeldungen und Aufgabenstellungen zur Datenbank gibt, sodass bei einem unbekannten oder unklaren Begriff sofort der passende Eintrag aufgerufen werden kann.

Es soll aber auch möglich sein, die Wissensdatenbank aufzurufen, ohne dass es eine Verlinkung dorthin gibt. Nicht alle Aufgabenstellungen enthalten immer die passenden Schlüsselwörter, und auch die Fehlermeldungen führen nicht immer an die Themen, die zur Lösung der Aufgabe nötig sind. Daher soll auch anderweitig eine Möglichkeit bestehen, die Datenbank zu öffnen und zu durchstöbern.

Jeder Eintrag benötigt einen Titel, mit dem dieser eindeutig identifiziert wird. Es ist nicht erwünscht, dass es mehrere Artikel mit dem gleichen Titel gibt, da dies zu Verwirrung führen kann, welcher der Artikel nun der ist, der bei der Bearbeitung oder dem Verständnis weiterhilft. Der wichtigste Teil des Eintrages ist der tatsächliche Inhalt, der in Textform das Thema erklären soll. Im Idealfall soll der Artikel alleine ausreichen, um dieses Thema der Programmiersprache zu verstehen.

Um den Einträgen eine Ordnung zu geben, soll es möglich sein, sie zu gruppieren. Dabei werden mehrere Einträge unter einem Eintrag, der die ganze Gruppe beschreibt, zu einer Gruppe zusammengefasst. Durch diese Ordnung entsteht eine Baumstruktur, die die Orientierung innerhalb der Datenbank erleichtert und die Übersichtlichkeit verbessert. Für die grafische Umsetzung erscheint es sinnvoll, die Gruppen ein- und ausklappen zu können.

Schließlich soll es möglich sein, jedem Artikel Schlüsselwörter hinzuzufügen. Diese Schlüsselwörter sollen dafür genutzt werden, in Fehlermeldungen und Aufgabenstellungen auf den entsprechenden Artikel zu verweisen. Dies eröffnet die Möglichkeit, weiterführende Hilfestellungen und Informationen zu liefern, ohne dabei auf einen Schlag zu überfordern. Sind von Nutzenden mehr Informationen erwünscht, stehen diese auf Abruf zur Verfügung, werden jedoch nicht bei jeder Fehlermeldung gleich angehängt.

3.2 Das Programm für Lernende

Für Lernende soll das Programm so übersichtlich wie möglich aufgebaut sein. Die Benutzeroberfläche soll eine Liste aller verfügbaren Aufgaben anzeigen, die zur Bearbeitung ausgewählt werden können. Diese Liste zeigt die jeweiligen Titel der Aufgaben, um diese zu identifizieren. Nachdem eine Aufgabe ausgewählt wurde, wird die Aufgabenstellung angezeigt. In einem Textfeld lässt sich dann Code schreiben, um die Aufgabenstellung zu bearbeiten.

Beim Schreiben von Code sind keine Shortcuts oder automatische Vervollständigungen gewünscht. Studierende müssen in Klausuren per Hand auf Papier programmieren und sollen das manuelle Ausschreiben von Code auch in diesem Programm schon lernen. Wie schon in Kapitel 2.1.3 im Abschnitt „Bewertung“ erwähnt, hat eine solche Zeitersparnis nicht immer die gewünschte Auswirkung. Eine visuelle Unterstützung für das Code-Verständnis soll ein Syntax-Highlighting darstellen, das vor allem Schlüsselwörter hervorhebt.

Wenn der*die Nutzer*in den Code fertig geschrieben hat, soll durch einen Knopf der Code kompiliert werden. Treten dabei Fehler auf, sollen diese in einer Konsole angezeigt werden. Dabei soll eine verständliche Beschreibung ebenso vorhanden sein wie Links zu weiterführenden Hilfestellungen in der Wissensdatenbank. Im Code soll der fehlerhafte Teil markiert werden. Zeilennummern in der Fehlermeldung und neben dem Code sollen die Zuordnung erleichtern.

War der Kompiliervorgang erfolgreich, werden alle der Aufgabe beiliegenden Tests ausgeführt. Treten dabei Fehler auf, so werden bei öffentlichen Tests die Parameter des Testaufrufs und das erwartete sowie das tatsächliche Ergebnis angezeigt. Bei privaten Tests soll lediglich eine Meldung angezeigt werden, dass nicht alle Tests erfolgreich waren. Außerdem soll in dem Code nach bestimmten semantischen Fehlern gesucht werden, die nicht unbedingt Kompilierfehler erzeugen. Waren alle Tests erfolgreich, soll eine Meldung angezeigt werden. Danach kann mit einer weiteren Aufgabe fortgefahren werden.

Es soll bei jeder Aufgabe die Möglichkeit bestehen, den eingegebenen Code mit einem Knopf zu speichern, sodass dieser bei der nächsten Bearbeitung der Aufgabe wieder geladen wird. Dies soll es ermöglichen, die Lösung einer erfolgreich bearbeiteten Aufgabe später noch einmal zu rekapitulieren sowie eine fehlgeschlagene Bearbeitung später fortzusetzen. Außerdem soll durch einen weiteren Knopf der bisherige Fortschritt bei dieser Aufgabe zurückgesetzt werden - der gespeicherte Code soll gelöscht werden, damit die Bearbeitung erneut begonnen werden kann.

3.3 Das Tool für Lehrende

Das Tool für Lehrende beinhaltet die Funktionen der Erstellung, Anpassung und Verteilung von Aufgaben sowie Artikeln der Wissensdatenbank. Hier wird beschrieben, welche Anforderungen an die Funktionalität bestehen.

3.3.1 Funktionen für Aufgaben

Der erste zentrale Aspekt der Anwendung für Lehrende ist die Erstellung, die Änderung und das Testen von Aufgaben. Die erstellten Aufgaben müssen zur weiteren Bearbeitung oder als Backup an einer beliebigen Stelle gespeichert werden können. Abschließend sollen die fertig gestellten Aufgaben so exportiert werden, dass sie an die Lernenden verteilt werden können.

Um Aufgaben zu erstellen, müssen die in Abschnitt 3.1 auf Seite 22 beschriebenen Eigenschaften von Aufgaben eingegeben werden. Titel, Aufgabenstellung, Framework und Klassenname sind per Eingabe in ein Textfeld umsetzbar. Für die Testdatei soll per Pfad angegeben werden, wo die Testdatei liegt. Dies sollte durch einen „Datei öffnen“-Dialog erleichtert werden.

Um die Erstellung einer lauffähigen Aufgabe zu erleichtern, ist es empfehlenswert, eine Klasse mit einer gelösten Aufgabe in einer beliebigen, gewohnten Entwicklungsumgebung zu schreiben. Auch die Testklasse sollte so erstellt werden. Aus dem geschriebenen Code lassen sich die Eingaben für die entsprechenden Textfelder dann einfach in die Aufgabenerstellung einfügen. Dabei wird im Code einfach der Teil ausgelassen, der bei der Bearbeitung der Aufgabe geschrieben werden soll. Der Pfad zur erstellten und an einer beliebigen Stelle gespeicherten Testdatei wird eingefügt, um die Aufgabe zu vervollständigen.

Sobald eine Aufgabe mindestens einen Titel, eine Beschreibung, einen Klassennamen und eine Testdatei hat, kann sie in einer beliebigen XML-Datei gespeichert werden. Hierfür muss beim ersten Speichervorgang ein „Datei speichern“-Dialog geöffnet werden. Danach kann die Aufgabe automatisch in derselben Datei gespeichert werden. Genauso soll eine bestehende Aufgabe aus einer Datei geladen, dann bearbeitet und schließlich in einer beliebigen oder der gleichen Datei wieder gespeichert werden können. Es ist empfehlenswert und wird sogar dringend empfohlen, jede Aufgabe in einem eigenen Ordner zu speichern, um eine bessere Übersicht zu bewahren. Diese Ordnerstruktur wirkt sich auch auf die Exportierung aus, da jede Aufgabe jeweils in den Ordner gepackt wird, in dem sie gespeichert ist.

Wurde eine Aufgabe gespeichert, kann sie auch getestet werden. Dabei wird das Programm für Lernende geöffnet und die gerade erstellte Aufgabe als einzige angezeigt. Es besteht dann die Möglichkeit, die Aufgabe aus der Sicht der Lernenden zu betrachten und probeweise zu bearbeiten oder zu lösen. Dieser Schritt wird für jede Aufgabe dringend empfohlen, bevor die Aufgabe veröffentlicht wird, um den Aufwand für Korrekturen und erneutes Verteilen zu vermeiden.

Die angegebene Testdatei soll beim Speichervorgang verschlüsselt werden, um ein einfaches Auslesen der Tests zu verhindern. Anschließend muss sie in das Verzeichnis der XML-Datei kopiert werden, damit die Testdatei im richtigen Ordner und im richtigen Format vorliegt. Für die Exportierung werden die gespeicherten Dateien in eine ZIP-Datei gepackt, um so verteilt werden zu können.

3.3.2 Funktionen für Artikel der Wissensdatenbank

Um einen neuen Artikel für die Wissensdatenbank zu schreiben, müssen zuerst der Titel eingegeben und der bestehende Artikel, unter dem der neue Artikel eingeordnet werden soll, ausgewählt werden. Daraufhin kann der Inhalt des Artikels verfasst werden, zusammen mit einer Liste von Schlüsselwörtern, die zu diesem Artikel führen sollen. Bei den Schlüsselwörtern ist zu beachten, dass von mehreren Artikeln, die das gleiche Schlüsselwort verwenden, nur einer das Ziel einer Verlinkung ist. Die Vergabe von Schlüsselwörtern sollte also einmalig geschehen.

Bearbeitete Artikel sollen automatisch gespeichert werden. Das Ziel der Wissensdatenbank ist es nicht, gezielte Artikel zu exportieren, sondern den gesamten Inhalt zu verteilen. Daher sind die Ordnerstruktur und der Speicherort nicht wichtig für Lehrende; beides kann vom Programm vorgegeben werden. Wichtig ist nur, dass Lehrende die Wissensdatenbank exportieren können, um sie an Lernende zu verteilen, sollte die Notwendigkeit einer Aktualisierung bestehen. Da dies nicht regelmäßig der Fall sein dürfte, besteht vermutlich keine Notwendigkeit, diese Funktionalität vereinfacht in die Anwendung aufzunehmen.

Bereits erstellte Artikel können auch wieder gelöscht werden, falls das gewünscht ist. Hier ist auch von der Möglichkeit auszugehen, dass der Artikel andere Artikel als Kinder in der Baumstruktur enthält. In diesem Fall werden diese Kinder ebenfalls gelöscht. Alle gelöschten Artikel werden auch aus der Dateistruktur gelöscht, damit sie beim nächsten Ladevorgang nicht wieder eingelesen werden. Hierauf sind Lehrende beim Löschen explizit hinzuweisen. Möglicherweise empfiehlt es sich sogar, eine doppelte Bestätigung zu verlangen.

Es ist denkbar, dass sich während der Erstellung der Datenbank die gewünschte Struktur ändert. Daher soll es möglich sein, bereits bestehende Artikel in der Baumstruktur zu bewegen. Hierbei ist zu bedenken, dass ein Artikel nicht als sein eigenes Kind registriert werden kann und auch als kein Kind, das ein Nachfahre des Artikels selbst ist. Hierbei würde es zu einer zirkulären Abhängigkeit kommen, die zum einen in einer Baumstruktur verboten ist und zum anderen nicht im Sinne der Sortierung und Ordnung der Datenbank wäre.

3.4 Verteilung von Programm und Aufgaben

Bei der technischen Umsetzung der Verteilung von Programm und Aufgaben gibt es mehrere Ansätze. Die Vor- und Nachteile von zwei Ansätzen werden im Folgenden diskutiert.

3.4.1 Bereitstellung als eine JAR-Datei

Eine Möglichkeit ist es, alle Aufgaben in das Programm zu integrieren und eine einzige JAR-Datei zum Download anzubieten.

Vorteilhaft daran ist, dass Nutzer*innen dann nur diese Datei ausführen müssen und direkt alle bestehenden Aufgaben bearbeiten können. Für die Einfügung neuer Aufgaben muss dann nur diese JAR-Datei neu heruntergeladen werden und alle hinzugefügten und/oder aktualisierten Aufgaben sind darin enthalten. Es müssen keine weiteren Dateien heruntergeladen, entpackt oder verschoben werden, bevor das Programm voll einsatzbereit ist.

Ein Nachteil ist hingegen, dass bei der Verwendung der JAR-Datei weitere Dateien erzeugt werden müssen. Um diese über mehrere Sitzungen hinweg zugänglich zu haben, wie es Konfiguration und gespeicherter Fortschritt erfordern, müssten diese im gleichen Verzeichnis wie die JAR-Datei angelegt werden, was eventuell den Ordner, in dem diese gespeichert ist, ungewollt mit anderen Dateien füllt. Möglich wäre auch, die Dateien in einem anderen, eigens dafür erstellten Ordner zu speichern, was die Kontrolle der Nutzer über die Speicherverwendung einschränkt. Soll das Programm wieder gelöscht werden, müssen diese Dateien ebenfalls entfernt werden, um eine vollständige Deinstallation zu gewährleisten.

Hinzu kommt, dass die Erstellung von Aufgaben eine Bearbeitung des gesamten Projektes erfordert. Um eine neue Aufgabe hinzuzufügen, müsste der Quellcode des Programmes selbst geändert werden, um dann eine neue JAR-Datei zu packen und zu verteilen. Zudem lässt sich anhand einer einzelnen JAR-Datei schlecht erkennen, welche Aufgaben darin enthalten sind, ohne dass man diese Datei entpackt oder ausführt. Nutzer*innen können daher möglicherweise nicht erkennen, ob sie eine veraltete Version nutzen, oder ob sie schon die neuesten Aufgaben haben. Eine Versionsnummer würde hier Abhilfe schaffen, allerdings wäre die schrittweise Verteilung von Aufgaben gemäß den Vorlesungsinhalten deutlich erschwert.

3.4.2 Gepackte ZIP-Datei mit JAR-Datei und Ordnern

Die zweite Möglichkeit ist, eine ZIP-Datei zu stellen, die das Programm in einer JAR-Datei sowie Ordner mit ersten Aufgaben enthält. Das Programm selbst kann immer noch mit einem Doppelklick ausgeführt werden, jedoch muss zunächst die ZIP-Datei entpackt werden.

Verglichen mit der vorhergehenden Herangehensweise hat dieser Ansatz den Nachteil, dass neue Aufgaben immer als gepackte Datei heruntergeladen werden müssen, um in den entsprechenden Ordner extrahiert zu werden. Dieser weitere Arbeitsschritt muss in jedem Fall durchgeführt werden. Eine Erleichterung würde es hier darstellen, wenn man im Programm selbst eine Datei auswählen könnte, die neue Aufgaben enthält, und wenn diese Aufgaben dann in den richtigen Ordner entpackt würden. Hierbei wäre es für Anwender*innen irrelevant, in welchem Ordner das Programm gespeichert ist, sobald eine Verknüpfung besteht, um es zu starten.

Der Vorteil ist, dass bei der Installation sofort klar wird, dass weitere Dateien im Verzeichnis vorhanden sind. Nutzer*innen können darauf reagieren und einen Unterordner erstellen, falls sie das wünschen. Alle von der Anwendung erstellten Dateien befinden sich in diesem einen Ordner und können so leicht wieder gelöscht werden, sobald dies gewünscht ist.

Sind die Aufgaben in der ZIP-Datei in einer gut verständlichen Ordnerstruktur vorhanden, lässt sich anhand der Ordnernamen erkennen, welche Aufgaben in welcher ZIP-Datei vorhanden sind. Somit lässt sich auch leicht erkennen, ob neue Aufgaben hinzugekommen sind oder ob schon alle Aufgaben zur Bearbeitung bereitstehen. Das Hinzufügen neuer Aufgaben im Programm erfordert außerdem nicht unbedingt eine IDE, sondern lässt sich mit einem eigens dafür erstellten Programm einfach umsetzen.

4 Konzeptueller Aufbau des Programms

Wie in Kapitel 3 beschrieben, gibt es zwei Programme, die einander ergänzen. Das erste Programm ist für Studierende gedacht, die an Aufgaben arbeiten und ihre Programmierfähigkeiten verbessern wollen. Das zweite Programm ist für Lehrende gedacht, die diese Aufgaben und weitere Hilfsmittel erstellen.

Dieses Kapitel befasst sich mit dem Aufbau der grafischen Benutzeroberfläche beider Programme. Für einige Elemente wird dargelegt, welche Überlegungen zu der gewählten Art der Umsetzung geführt haben. Dieses Kapitel stellt somit auch eine Bedienungsanleitung dar, die in kürzerer Form auch dem Programm beigefügt ist und von dort geöffnet werden kann.

4.1 Das Programm für Studierende

In Abschnitt 3.4 auf Seite 24 wurde diskutiert, auf welche Arten das Programm und die Aufgaben verteilt werden können. Für die Verteilung dieses Programmes wurde eine ZIP-Datei gewählt. Diese Datei enthält eine ausführbare JAR-Datei, um das Programm zu starten, eine JAR-Datei, die JUnit enthält, und Ordner für Aufgaben, Wissensdatenbank und Hilfe. Nachdem die ZIP-Datei entpackt wurde, kann das Programm zum ersten Mal gestartet werden.

4.1.1 Erster Start des Programms

Direkt nach dem ersten Start erscheint eine Meldung (Abbildung 4.1), dass keine Konfigurationsdatei gefunden wurde. Nach einem Klick auf „OK“ öffnet sich ein Fenster mit den Konfigurationseinstellungen (Abbildung 4.2). Hier werden einmalig notwendige Einstellungen vorgenommen, um die Lauffähigkeit des Compilers und der Tests zu gewährleisten. Unter Java Version 9 ist dabei das Eingabefeld für den JDK-Pfad deaktiviert, da in diesem Fall das Programm bereits mit dem JDK gestartet werden muss.

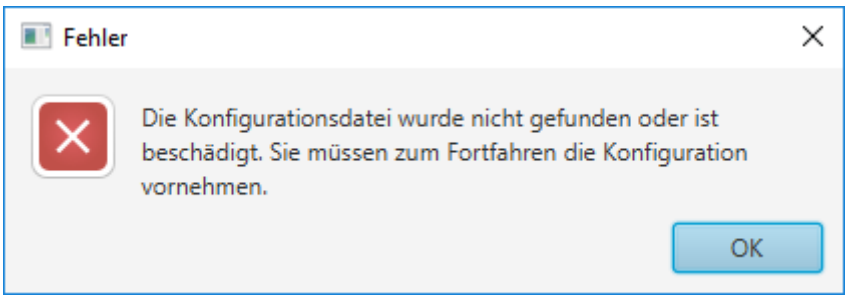

Abbildung 4.1: Die Meldung nach dem ersten Start des Programms

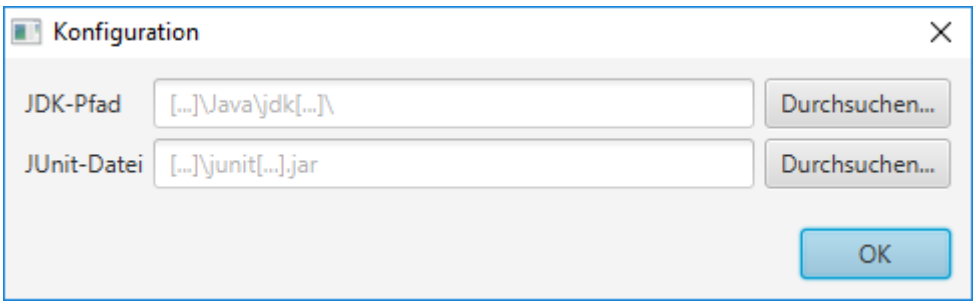

Abbildung 4.2: Das Fenster der Konfigurationseinstellungen unter Windows mit Java Version 8

4.1.2 Die Konfiguration

Unter Java Version 8 müssen in der Konfiguration zwei Pfade eingetragen werden. Beide werden beim Kompiliervorgang benötigt, um den Code und die Tests kompilieren zu können. Daher ist es wichtig, dass diese Pfade korrekt gesetzt sind. Unter Java Version 9 fällt die Konfiguration des JDK-Pfades weg, da die Laufzeitumgebung nicht mehr programmatisch geändert werden kann.

Der erste Pfad muss zum JDK-Ordner führen. Damit ist nicht der darin enthaltene *bin*-Ordner gemeint, sondern der Ordner, in dem der *bin*-Ordner liegt. Dieser Ordner liegt typischerweise im Verzeichnis „Java“ und enthält im Namen die Version des JDKs. Da die Version nicht auf jedem Gerät einheitlich ist und sich zudem verändern kann, kann dieser Pfad nicht fest gesetzt werden. Von der automatischen Erkennung des korrekten Pfades wurde abgesehen, da der Installationspfad sich abhängig von Gerät, Betriebssystem und Nutzerpräferenz unterscheiden kann.

Der zweite Pfad muss zum JUnit-Archiv führen. Diese ist in der ZIP-Datei des Programms enthalten und wurde somit schon heruntergeladen. Die Datei muss nur noch verlinkt werden. Beide Pfade können mithilfe des „Durchsuchen“-Knopfes in einem Dateibrowser ausgewählt werden. Diese Vorgehensweise erleichtert das Auffinden des Pfades und stellt die korrekte Eingabe sicher.

4.1.3 Übersicht über das Programm

Das Hauptfenster des Programms beinhaltet alle Elemente, die für die Arbeit an Aufgaben wichtig sind. Abbildung 4.3 zeigt dieses Fenster mit Markierungen und Nummerierungen, die im Folgenden erläutert werden. Am oberen Rand des Fensters befindet sich die Toolbar (4) mit allen Knöpfen, die für die Bedienung des Programmes nötig sind. Tabelle 4.1 auf Seite 29 liefert eine Übersicht, welcher Knopf welche Funktionalität hat.

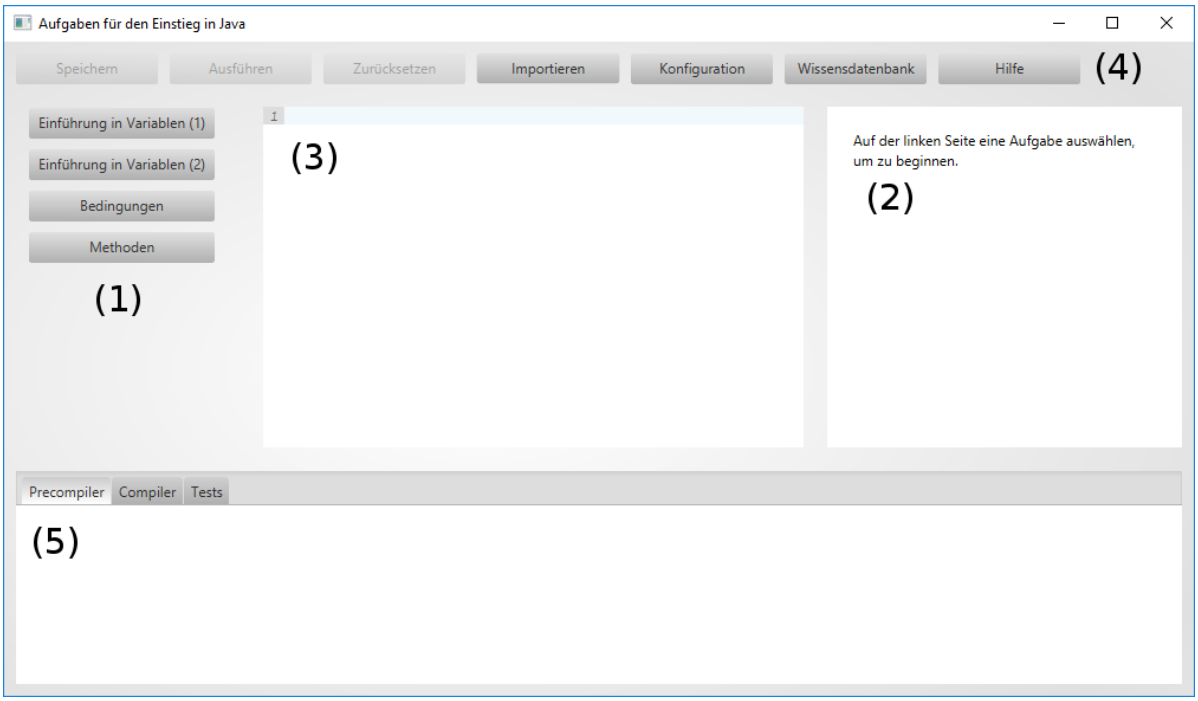

Abbildung 4.3: Das Hauptfenster des Programms für Studierende mit Nummerierungen zur Erläuterung der Elemente

Aufgabenübersicht

Eine Übersicht über alle Aufgaben, die momentan geladen sind, zeigt die Liste im linken Teil (1). Jede Aufgabe wird durch einen Knopf dargestellt. Dies erlaubt es, eine Aufgabe anzuklicken, um sie zu öffnen. Eine andere Möglichkeit der Umsetzung wäre es gewesen, eine `ListView` zu verwenden, in der man ein Element auswählen kann. Diese wäre aber grafisch weniger ansprechend und mit mehr Aufwand in der Umsetzung verbunden gewesen. Zudem ist das Aussehen eines Knopfes bekannt und vertraut, sodass die Möglichkeit einer Interaktion durch Klicken ohne Erklärung bewusst ist.

Die Breite dieses Bereichs und aller Knöpfe richtet sich nach dem Knopf mit dem längsten Titel. Dies wird in Abbildung 4.4 auf der nächsten Seite an einem Beispiel verdeutlicht. Durch diese Maßnahme wird Platz gespart, wenn alle Titel verhältnismäßig kurz sind. Unter einer horizontalen Scrollbar hätte die Lesbarkeit gelitten, wenn Titel eine voreingestellte Breite überschreiten.

Aufgabenstellung

Wurde eine Aufgabe ausgewählt, so wird die Aufgabenstellung dazu im rechten Teil des Fensters (2) angezeigt. Direkt nach dem Start des Programms wurde noch keine Aufgabe ausgewählt, deshalb steht hier zu diesem Zeitpunkt der Hinweis, eine Aufgabe auf der linken Seite auszuwählen.

Abbildung 4.4: Vergleich der Breite der Aufgabenübersicht bei verschiedenen Maximallängen der Titel

Benötigt der Text der Aufgabenstellung mehr Platz als vorhanden ist, wird eine vertikale Scrollbar eingefügt. Wie bei der Aufgabenübersicht wird auch hier keine horizontale Scrollbar angezeigt, um die Lesbarkeit nicht einzuschränken. Eine Besonderheit dieses Textfeldes ist, dass Schlüsselwörter aus der Wissensdatenbank als Link markiert werden. Ein Beispiel hierfür zeigt Abbildung 4.5, in der eine Aufgabe geladen wurde.

Code-Bereich

Um eine Aufgabe zu lösen, muss im Code-Bereich in der Mitte des Fensters (3) Code geschrieben werden, der die Aufgabenstellung erfüllt. Schlüsselwörter werden violett und fett geschrieben. Runde und geschweifte Klammern werden dezent farblich markiert, Strings werden blau dargestellt. Kommentare erscheinen in einem hellen Blaugrün und treten vor dem Rest des Codes in den Hintergrund, um nicht abzulenken. Das Syntax-Highlighting soll das Verständnis für Schlüsselwörter und Syntax erhöhen, ohne dabei zu stark abzulenken. Um die aktuelle Position des Cursors schneller finden zu können, wird die ganze Zeile in einem leichten Blau hinterlegt.

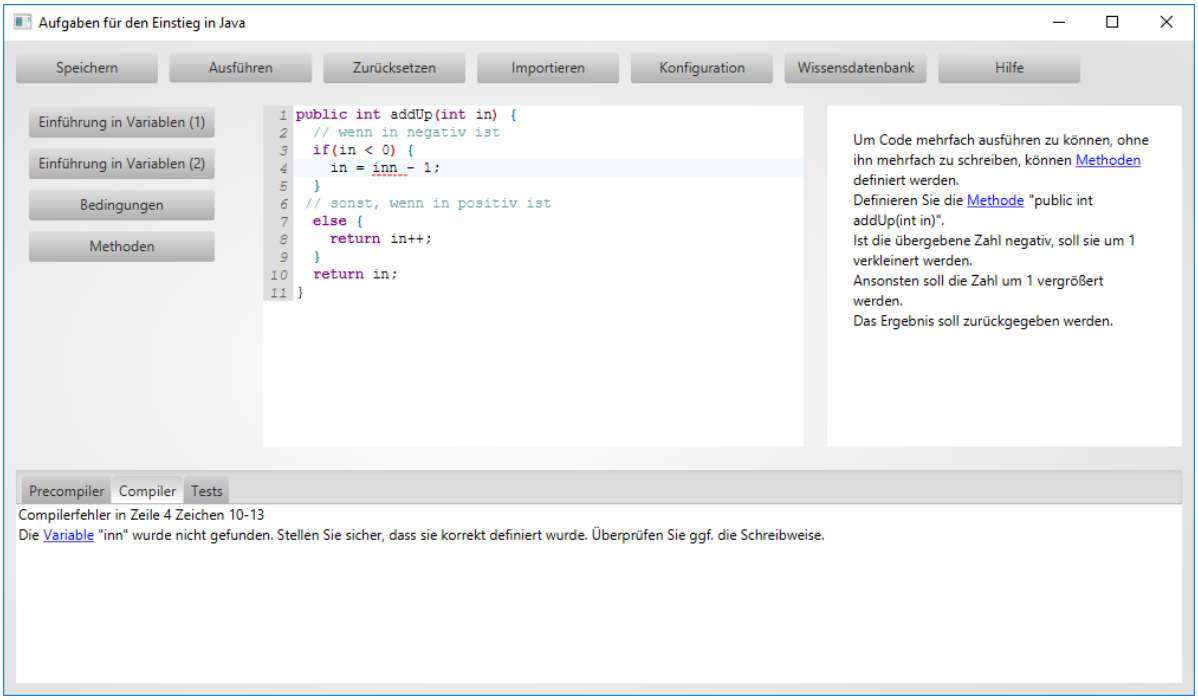

Abbildung 4.5: Das Hauptfenster des Programms für Studierende mit fehlerhaftem Beispielcode, nachdem dieser ausgeführt wurde

Sofern nach dem Kompilieren Fehler gefunden wurden, die eine eindeutige Stelle im Code betreffen, wird der Code dort unterstrichen. Bei schwerwiegenden Kompilierfehlern, die die Ausführung behindern, ist die Farbe rot. Semantische Fehler, die keinen Kompilierfehler verursachen, aber trotzdem zu unbeabsichtigtem Verhalten des Programms führen können, werden gelb unterstrichen. Highlighting und Fehlermarkierungen sind in Abbildung 4.5 beispielhaft dargestellt.

Beim Schreiben des Codes gibt es keine Hilfestellungen. Automatische Vervollständigung kann, wie in Abschnitt 2.1.3 auf Seite 10 angemerkt, dem Lernprozess entgegenwirken, weil der Code nicht selbst verinnerlicht, sondern auf die verwendete Tastenkombination reduziert wird.

Tabelle 4.1: Knöpfe der Toolbar und ihre Funktion, entsprechend den Anforderungen in Abschnitt 3.2 auf Seite 23

Beschriftung	Funktion
Speichern	Der geschriebene Code wird gespeichert und beim nächsten Start des Programms wieder geladen. Dies ermöglicht es, zu einem späteren Zeitpunkt mit der Bearbeitung fortzufahren oder eine Lösung zu rekapitulieren.
Ausführen	Es wird versucht, den geschriebenen Code zu kompilieren. Falls vorhanden, werden Fehler angezeigt und markiert. Die angezeigten Fehlermeldungen werden in Abschnitt 4.1.5 auf der nächsten Seite vorgestellt. War der Kompiliervorgang erfolgreich, werden alle Tests ausgeführt. Dann wird über das Ergebnis berichtet.
Zurücksetzen	Der selbst geschriebene Code wird gelöscht. Die Bearbeitung der Aufgabe kann erneut begonnen werden, um den Lerneffekt durch Wiederholung zu verbessern.
Importieren	Es wird ein Dialog geöffnet, mit dem sich eine ZIP-Datei auswählen lässt, die neue Aufgaben enthält. Die neuen Aufgaben werden entpackt und in das Programm eingebunden. So wird sichergestellt, dass neu erstellte Aufgaben einfach in bereits bestehende Systeme eingebunden werden können.
Konfiguration	Die Konfiguration (siehe Abbildung 4.2 auf Seite 26) wird geöffnet. Es können Änderungen an den dort gespeicherten Werten vorgenommen werden.
Wissensdatenbank	Die Wissensdatenbank wird geöffnet. Es besteht somit jederzeit Zugriff auf weiterführende Hilfe, ohne dass ein Stichwort in Aufgabenstellung oder Fehlermeldung erscheinen muss.
Hilfe	Es wird eine Anleitung für das Programm geöffnet, die dessen Benutzung erklärt. Für die erste Nutzung des Programms wird das Tutorial empfohlen.

4.1.4 Wissensdatenbank

Die Wissensdatenbank enthält - wie in Abschnitt 3.1.2 auf Seite 22 erklärt - Einträge über die Programmiersprache Java. Diese Einträge sollen dabei helfen, neue oder noch nicht verinnerlichte Konzepte zu lernen. Dies beinhaltet auch die Erklärung von Fachbegriffen, die in Fehlermeldungen vorkommen. Dies wurde wie in Abschnitt 2.2.2 auf Seite 13 beschrieben in der Arbeit von Marceau, Fisler und Krishnamurthi [12] empfohlen.

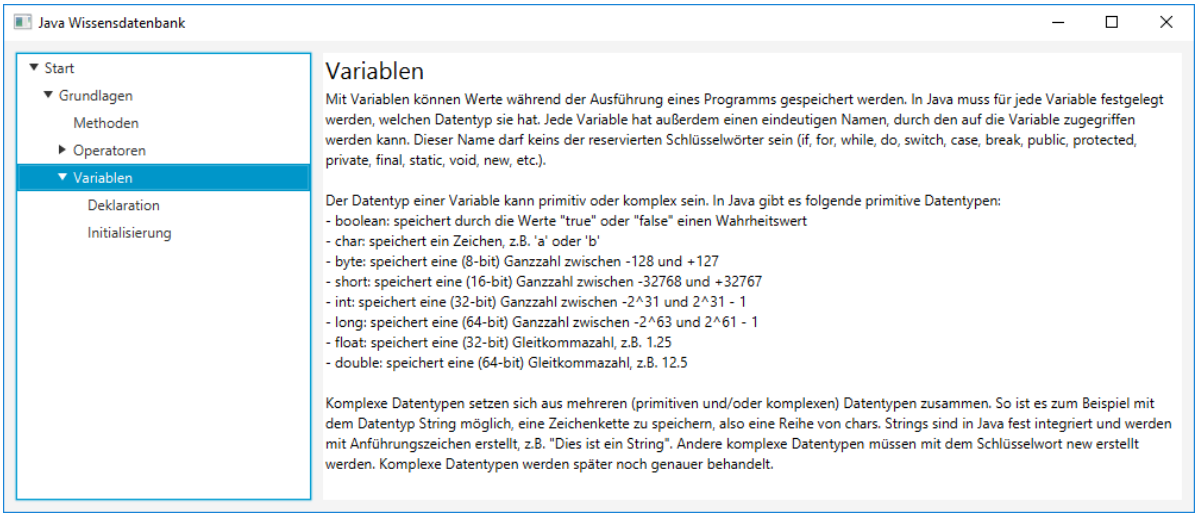

Abbildung 4.6: Ein Beispielartikel in der Wissensdatenbank für Studierende

Die Wissensdatenbank kann auf zwei Arten geöffnet werden. Die erste Möglichkeit ist, einen Link anzuklicken. Dies öffnet die Wissensdatenbank und direkt den verlinkten Artikel. Die zweite Möglichkeit ist der Knopf „Wissensdatenbank“ im oberen Teil des Fensters. Abbildung 4.6 zeigt einen beispielhaften Artikel in der Wissensdatenbank.

Auf der linken Seite befindet sich ein Verzeichnis über alle vorhandenen Artikel. Diese sind in einer Hierarchie angeordnet, in der eine gesamte Kategorie ein- und ausgeklappt werden kann. Die Kategorie selbst ist ebenfalls ein Artikel, der zumindest eine kurze Beschreibung enthalten sollte.

Mit einem Klick auf einen Eintrag im Verzeichnis wird dieser auf der rechten Seite des Fensters angezeigt. Um die Lesbarkeit zu verbessern, werden die Zeilen automatisch umgebrochen. Das kann zwar bei Codebeispielen zu einem Umbruch im Code führen, jedoch schadet eine horizontale Scrollbar der Lesbarkeit von Fließtext und würde häufiger auftreten. Bei der Erstellung der Artikel sollte jedoch immer bedacht werden, dass die Überschreitung einer gewissen Zeilenlänge zu einem Umbruch führen kann.

4.1.5 Fehlermeldungen und Testergebnisse

Im unteren Teil des Fensters erscheinen Meldungen über Fehler und Testergebnisse. Es wird automatisch der Tab angezeigt, in dem es zuletzt zu einer Ausgabe kam. Alle Schlüsselwörter, die in der Wissensdatenbank eingetragen sind, werden als Link markiert und öffnen den verlinkten Eintrag. Es gibt drei Reiter, die verschiedene Arten von Meldungen anzeigen.

Der erste Reiter „Precompiler“ enthält Fehlermeldungen, die der Java-Compiler nicht oder in anderer Form ausgibt. Die Fehlertypen, die erkannt werden, sind entnommen aus Tabelle 2.1 auf Seite 12 - sofern es keinen Kompilierfehler gibt, der diesen Fehler beschreibt. Tabelle 4.2 auf der nächsten Seite enthält alle Fehler, die der Precompiler erkennt, und die entsprechenden Meldungen.

Der zweite Reiter „Compiler“ gibt alle Meldungen des Compilers aus. Diese Meldungen werden aus den Meldungen des im JDK enthaltenen Compilers übersetzt, enthalten Zeilen- und Spaltennummern (soweit vorhanden) und werden gegebenenfalls mit weiteren zur Verfügung stehenden Informationen versehen. Abbildung 4.5 auf Seite 28 zeigt hierfür ein Beispiel. Die Variable „in“ aus Zeile 1 wurde in Zeile 4 als „inn“ falsch geschrieben. Der Hinweis auf korrekte Definition und Überprüfung der Schreibweise stellt sicher, dass keine der vorhandenen Schreibweisen grundsätzlich als falsch gemeldet wird. Dies entspricht der Empfehlung von Marceau, Fisler und Krishnamurthi [12], wie in Abschnitt 2.2.2 auf Seite 13 beschrieben. Tabelle 4.3 auf Seite 32 zeigt alle Fehlermeldungen, die im Compiler-Reiter erscheinen können.

Die Ergebnisse der Testausführung werden im dritten Reiter „Tests“ angezeigt. Dies tritt nur ein, wenn der Kompiliervorgang erfolgreich war und der Code somit ausgeführt werden konnte. Eine beispielhafte Meldung zeigt Abbildung 4.7. Die Meldung beginnt mit einer Zusammenfassung aller Tests. Diese lautet entweder „Alle Tests waren erfolgreich!“ oder „Es sind . . . von . . . Tests fehlgeschlagen.“ In beiden Fällen folgt eine Angabe der Laufzeit aller Tests: „Die Laufzeit betrug . . . ms.“

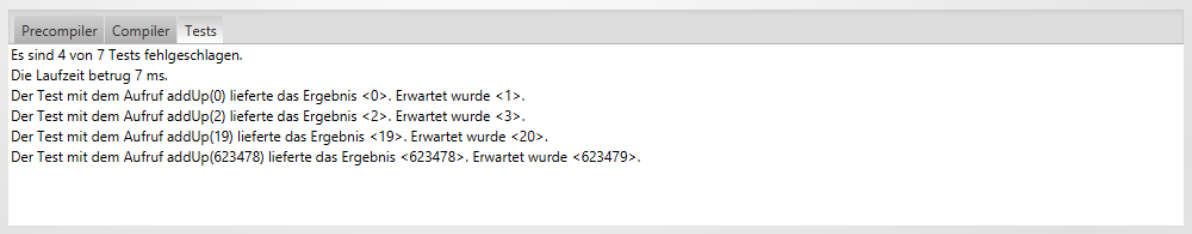

Abbildung 4.7: Ein beispielhafter Test-Reiter nach der Ausführung der Tests

Jeder ausgeführte Test kann drei mögliche Ergebnisse liefern. Im besten Fall läuft er fehlerfrei durch und der Test ist bestanden. Dies spiegelt sich in der Zusammenfassung wider. Kommt es bei einem Test zu einer Exception oder einem Error im ausgeführten Code, so wird dieser mit einer Zeilenangabe angegeben. Der gesamte Stack Trace würde den Reiter überfüllen und Anfänger*innen verwirren, daher ist nur die fehlerhafte Zeile im selbst geschriebenen Code aufgeführt. Jeder häufig auftretende Fehler sollte zudem einen Eintrag in der Wissensdatenbank haben, wo dieser genauer erklärt wird.

Für einen aufgrund des Ergebnisses fehlgeschlagenen Test wird unterschieden, ob dieser Test privat oder öffentlich ist. Bei privaten Tests folgt nur die Meldung „Die privaten Tests waren nicht erfolgreich.“ Öffentliche Tests geben mehr Informationen über den Aufruf sowie das erwartete und tatsächliche Ergebnis: „Der Test mit dem Aufruf . . . lieferte das Ergebnis <. . . > . Erwartet wurde <. . . >.“

Tabelle 4.2: Fehlermeldungen des Precompilers. Empfehlungen von Marceau, Fisler und Krishnamurthi [12] haben zur Formulierung beigetragen. Diese Fehlermeldungen erscheinen bei den häufigsten Fehlern aus dem Blackbox-Datensatz [3], die nicht oder nicht genau genug durch Kompilierfehler erkannt oder beschrieben werden.

Beschreibung des Fehlers	Fehlermeldung des Precompilers
= und == verwechselt	Eventuell haben Sie den Zuweisungsoperator "=" mit dem Vergleichsoperator "==" verwechselt.
Vergleich von Strings mit ==	Sie scheinen zwei Strings mit dem Operator "==" zu vergleichen. Oft liefert jedoch nur die Methode a.equals(b) das erwartete Ergebnis beim Vergleichen von zwei Strings a und b.
; direkt nach if (...) oder for (...)	Ein Semikolon ";" direkt nach "[if|for]" stellt einen leeren Ausdruck dar, sodass nichts ausgeführt wird.
; direkt nach while (...)	Ein Semikolon ";" direkt nach "while" stellt einen leeren Ausdruck dar, sodass nichts ausgeführt wird. Eine Ausnahme bildet die do-while-Schleife.
& oder | beim Vergleich von booleans	Die Bit-Operatoren "&" und "|" haben eine andere Funktionsweise als die Booleschen Operatoren "&&" und "||". Verwenden Sie erstere nur, wenn Sie sich über die genaue Funktionsweise im Klaren sind.
falsche Klammer nach if, for oder while	Die Bedingung einer if-/while-/for-Anweisung wird in runden Klammern "()" angegeben.
mehr öffnende als schließende Klammern	Es wurden {x} [runde|geschweifte|eckige] Klammern "()/{}/[]" mehr geöffnet als geschlossen wurden. Vermutlich liegt hier ein Fehler vor.
mehr schließende als öffnende Klammern	Es wurden {x} [runde|geschweifte|eckige] Klammern "()/{}/[]" mehr geschlossen als geöffnet wurden. Vermutlich liegt hier ein Fehler vor.
ungerade Anzahl von " oder '	Es wurde eine ungerade Anzahl von [Anführungszeichen "|Apostrophen '] gefunden. Vermutlich liegt hier ein Fehler vor.
mehr als ein Zeichen in einem char	Ein char speichert genau ein Zeichen. Innerhalb der Apostrophe ' ' darf daher nur genau ein Zeichen stehen. Ausnahmen sind Angaben in Unicode oder Sonderzeichen.
Fehler beim Parsen (unerwarteter Ausdruck)	Fehler beim Parsen: Der Ausdruck "{...}" scheint an dieser Stelle falsch zu sein. Eventuell fehlt in der vorherigen Zeile ein Semikolon ";" oder eine geschweifte Klammer "}".
lexikalischer Fehler	Unerwartetes Zeichen an dieser Stelle

Tabelle 4.3: Fehlermeldungen des Compilers. Alle Meldungen beginnen mit einer Zeilen- und Spaltenangabe, die der Java-Compiler ausgegeben hat. Empfehlungen von Marceau, Fisler und Krishnamurthi [12] haben zur Formulierung beigetragen. Diese Fehlermeldungen erscheinen bei vom Java-Compiler erzeugten Fehlern. Zur Vervollständigung und zum Verständnis der tatsächlichen Fehlerquellen hat das Dokument von Ben-Ari [1] beigetragen.

Beschreibung des Fehlers/Originalnachricht	Fehlermeldung im Compiler-Reiter
unbekannter Bezeichner	Die [Variable|Methode|Klasse] "..." wurde nicht gefunden. Stellen Sie sicher, dass sie korrekt definiert wurde. Überprüfen Sie ggf. die Schreibweise.
nicht initialisierte Variable	Die Variable "..." wird verwendet, bevor sie initialisiert wurde.
...expected	Es wurde ...erwartet. Überprüfen Sie im Zweifelsfall auch umliegende Zeilen auf Fehler.
falsche Argumente für Methode	Die Methode ... erwartet laut Definition .... Gefunden wurde .... Überprüfen Sie, ob der Aufruf korrekt ist. Falls Sie die Methode ... selbst definiert haben, ist möglicherweise die Definition fehlerhaft.[1]
keine passende Methode für Parameter	Für den Aufruf ... wurde keine passende Methodendefinition gefunden. Folgende Definitionen konnten nicht angewendet werden: *vorhandene Definition der Methode*: [Anzahl von Parametern in Definition und Aufruf unterscheidet sich|... kann nicht zu ... konvertiert werden] Überprüfen Sie, ob der Aufruf korrekt ist. Falls Sie die Methode ... selbst definiert haben, ist möglicherweise die Definition fehlerhaft.
fehlender return-Ausdruck	Es fehlt ein "return"-Ausdruck, der den Rückgabewert der Methode enthält. Stellen Sie sicher, dass für alle Auswertungspfade ein return-Ausdruck erreicht wird. Falls die Methode keinen Wert zurückgeben soll, können Sie den Typ der Methode zu "void" ändern.
nicht-statischer Aufruf aus statischem Kontext	Die nicht-statische [Variable|Methode] ... kann nicht aus einem statischen Kontext aufgerufen werden.
Semikolon nach dem Methodenkopf	Ein Semikolon nach der Methodendeklaration ist nur bei einer abstrakten Methode erlaubt. Bei allen anderen Methoden muss stattdessen ein Methodenkörper folgen, der in geschweiften Klammern {} steht.
reached end of file while parsing	Das Ende der Datei wurde vorzeitig erreicht. Überprüfen Sie, ob alle geschweiften Klammern wieder geschlossen werden.
illegal start of expression	Ungültiger Ausdruck. Der "Precompiler"-Reiter enthält möglicherweise weitere Informationen.
variable declaration not allowed here	Es ist an dieser Stelle nicht erlaubt, eine Variable zu deklarieren.
illegal start of type	Ungültiger Beginn einer Deklaration. Möglicherweise wurde dieser Fehler durch einen anderen Fehler hervorgerufen. Beachten Sie auch den "Precompiler"-Reiter.
nicht konvertierbare Typen	Inkompatible Typen: ... kann nicht automatisch zu ... konvertiert werden. Sie können einen Cast verwenden, wenn Sie sicher sind, dass die Typen explizit konvertierbar sind.
verlustbehaftete Konvertierung	Die Konvertierung von ... zu ... verursacht möglicherweise einen Präzisionsverlust. Stellen Sie sicher, dass Sie diese Möglichkeit betrachtet haben. Eine Konvertierung lässt sich mit einem Cast explizit anweisen.
mehrfache Verwendung eines Bezeichners	Die [Variable|Methode] ... wurde bereits in der [Methode|Klasse] ... definiert.
unreachable statement	Dieser Code wird niemals ausgeführt. Grund dafür kann sein, dass im Code davor eine Endlosschleife oder ein return-Ausdruck vorhanden ist.

[1] Falls die Anzahl der übergebenen Parameter mit der Anzahl der Parameter der gefundenen Methode übereinstimmt, erscheint zusätzlich folgende Meldung: Es besteht ebenfalls die Möglichkeit, dass beim Aufruf eine Konvertierung fehlgeschlagen ist.

4.2 Das Programm für Lehrende

Das Programm für Lehrende beinhaltet das Programm für Studierende. Es ist allerdings eine JAR-Datei enthalten, die einen anderen Einstiegspunkt in das Programm hat. Es muss vor der ersten Verwendung also auch eine ZIP-Datei entpackt werden.

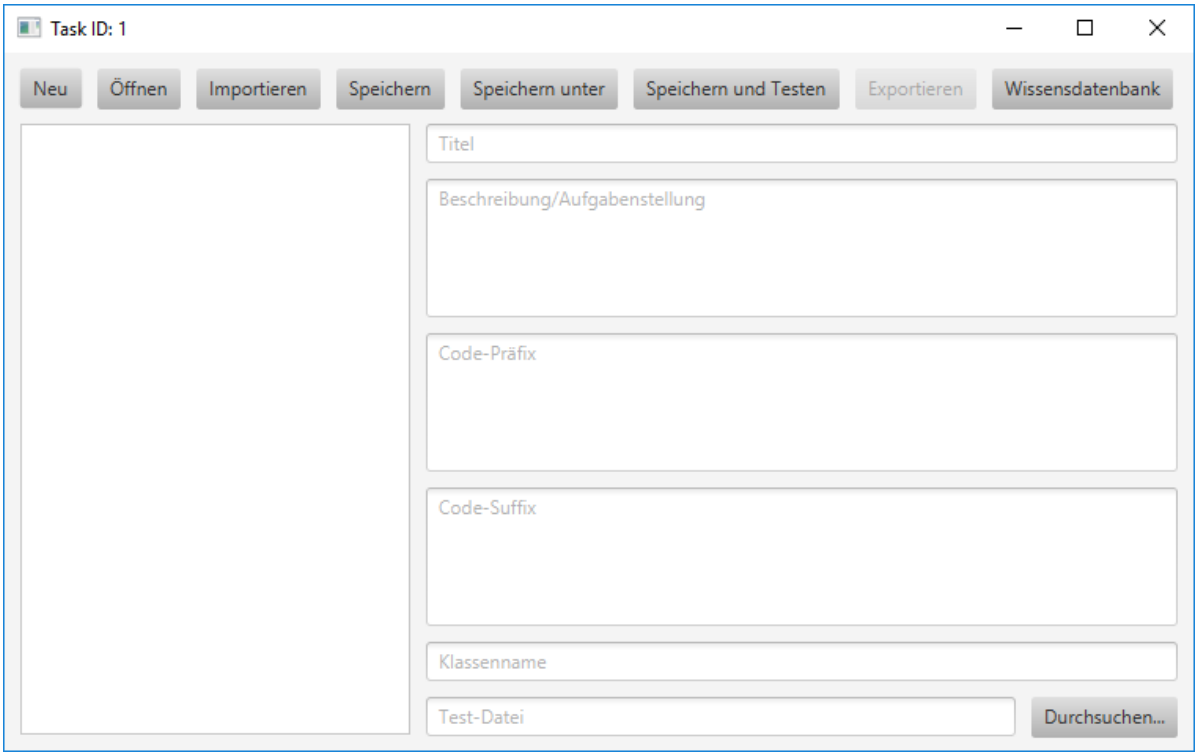

Abbildung 4.8: Das Programm für Lehrende nach dem ersten Start

4.2.1 Übersicht über das Programm

Abbildung 4.8 zeigt das Programm direkt nach dem Start. Eine Übersicht über die Knöpfe in der Leiste oben liefert Tabelle 4.4 auf der nächsten Seite. Auf der linken Seite steht eine Liste mit geladenen Aufgaben. Da standardmäßig keine Aufgabe geladen wird, ist sie hier leer. Auf der rechten Seite befinden sich Eingabefelder für die Eigenschaften einer Aufgabe. Diese Felder entsprechen den in Abschnitt 3.3.1 auf Seite 24 beschriebenen Eigenschaften und erfüllen damit den Zweck der Aufgabenerstellung wie gefordert. Jedes Feld zeigt einen Hinweis an, welcher Inhalt sich hier befinden soll, solange kein Text darin steht und das Feld nicht angewählt ist. Der Knopf „Durchsuchen...“ neben dem Textfeld „Test-Datei“ öffnet einen Dateibrowser, mit dem sich eine Java-Datei auswählen lässt, die die Tests beinhaltet.

Für die Verfassung von Aufgabenstellungen sollte darauf geachtet werden, dass Stichwörter aus der Wissensdatenbank zu Verlinkungen zum entsprechenden Artikel werden. Soll der geschriebene Text eine Verlinkung zu gewissen Artikeln haben, muss also auf die Verwendung des korrekten Stichworts geachtet werden. Eine andere Möglichkeit ist es, am Ende der Aufgabenstellung eine Liste von Stichwörtern anzugeben, die alle für diese Aufgabe relevanten Artikel angibt.

Liste der geladenen Aufgaben

Während der Nutzung des Programms werden neue Aufgaben erstellt oder bestehende Aufgaben geladen. Diese erscheinen in einer Liste im linken Teil des Fensters. Jeder Eintrag in der Liste beinhaltet die ID sowie den Titel der Aufgabe. Um eine Aufgabe anzuzeigen, muss sie in der Liste angeklickt werden. Dies ermöglicht es, diese Aufgabe zu bearbeiten, zu testen oder zu speichern. Wurde eine angezeigte Aufgabe bearbeitet und wird dann eine neue Aufgabe ausgewählt, erstellt oder geöffnet, wird angeboten, die Änderungen vorher zu speichern (Abbildung 4.9 auf der nächsten Seite).

In der Liste der geladenen Aufgaben können auch mehrere Aufgaben ausgewählt werden. Dies ermöglicht es, mit der Funktion „Exportieren“ (siehe Tabelle 4.4 auf der nächsten Seite) mehrere Aufgaben auf einmal zu exportieren.

Tabelle 4.4: Knöpfe der Toolbar im Programm für Lehrende

Beschriftung	Funktion
Neu	Alle Textfelder werden geleert, um eine neue Aufgabe erstellen zu können. Die ID der neuen Aufgabe wird automatisch festgelegt.
Öffnen	Ein Dateibrowser erscheint, mit dem eine XML-Datei geöffnet werden kann, die eine Aufgabe enthält. Diese Aufgabe erscheint auf der linken Seite des Fensters als geladene Aufgabe und wird im rechten Teil angezeigt.
Importieren	Mit einem Dateibrowser kann eine ZIP-Datei ausgewählt werden, die Aufgaben enthält. Alle Aufgaben werden in ein temporäres Verzeichnis entpackt und dann geladen. Diese Aufgaben werden so behandelt, als hätten sie keinen Speicherort.
Speichern	Die aktuell angezeigte Aufgabe wird in der XML-Datei gespeichert, aus der sie geladen wurde. Hat eine Datei keinen Speicherort, ist die Funktionalität die gleiche wie bei „Speichern unter“.
Speichern unter	Es wird ein Dateibrowser geöffnet, um einen Speicherort für die aktuell angezeigte Aufgabe auszuwählen. Die Aufgabe wird in dieser Datei gespeichert.
Speichern und Testen	Zuerst wird die Funktionalität von „Speichern“ ausgeführt. Dann wird das Programm für Studierende geöffnet, in dem die angezeigte Aufgabe als einzige geladen wird und somit aus der Sicht von Studierenden getestet werden kann.
Exportieren	Sobald mindestens eine Aufgabe in der Liste ausgewählt ist, kann dieser Knopf betätigt werden. Alle ausgewählten Aufgaben werden in eine ZIP-Datei gepackt, die dann an Studierende verteilt werden kann. Studierende können sie mit dem Knopf „Importieren“ im Programm für Studierende laden.
Wissensdatenbank	Die Wissensdatenbank wird geöffnet, sodass sie bearbeitet werden kann. Eine genauere Erklärung liefert Abschnitt 4.2.2.

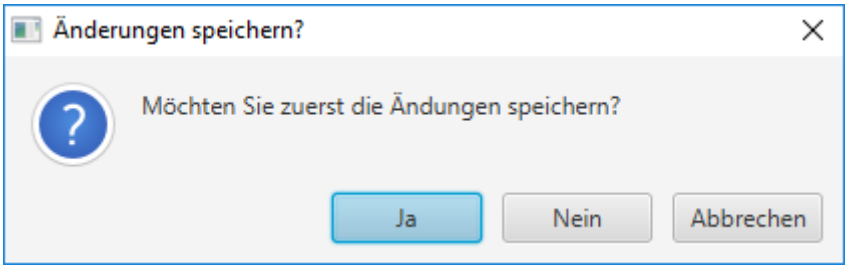

Abbildung 4.9: Anfrage, ob vor der nächsten Aktion die aktuelle Aufgabe gespeichert werden soll. Ja speichert sie (siehe Tabelle 4.4) und führt die Aktion aus, Nein speichert sie nicht und führt die Aktion aus, Abbrechen speichert sie nicht und bricht die Aktion ab.

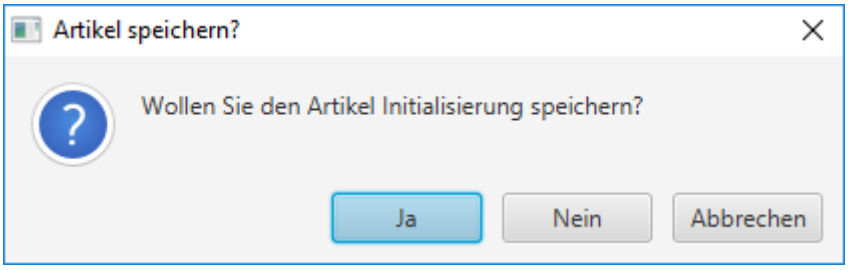

Abbildung 4.10: Wenn ein neuer Artikel geöffnet wird, nachdem der zuletzt geöffnete geändert wurde, ohne zu speichern, wird diese Frage angezeigt.

4.2.2 Editor für die Wissensdatenbank

Mit dem Knopf „Wissensdatenbank“ im Programm für Lehrende öffnet sich ein neues Fenster. Hier können Artikel aus der Wissensdatenbank erstellt, geändert, gelöscht und verschoben werden. Ein Beispiel zeigt Abbildung 4.11 auf der nächsten Seite. Links im Fenster werden alle vorhandenen Artikel angezeigt. Die Anzeige gleicht der im Programm für Studierende, beschrieben in Abschnitt 4.1.4 auf Seite 29, um schon bei der Erstellung einen Eindruck zu bekommen, wie die Datenbank für Studierende aussehen würde. Ein Artikel kann im Editor angezeigt werden, indem er in dieser Liste angeklickt wird. Wurde der zuvor angezeigte Artikel geändert, wird zuerst gefragt, ob dieser gespeichert werden soll (Abbildung 4.10).

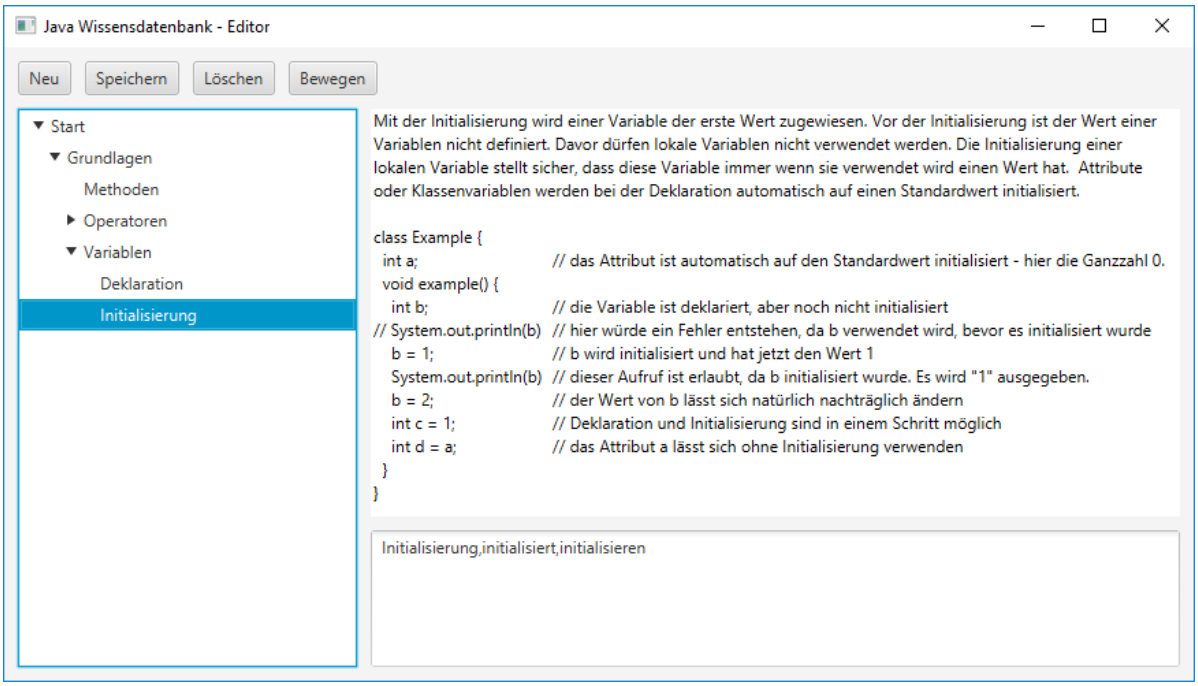

Abbildung 4.11: Ein im Editor für die Wissensdatenbank geöffneter Beispielartikel

In der oberen Leiste befinden sich Knöpfe zur Bedienung, wie aus den anderen Fenstern bekannt. Der erste Knopf „Neu“ öffnet ein Fenster, in dem der Name des neuen Artikels eingegeben werden muss (Abbildung 4.12). Daraufhin öffnet sich ein Fenster, in dem der Artikel ausgewählt werden kann, unter dem der neue Artikel abgelegt werden soll (Abbildung 4.13). Soll der neue Artikel auf dem obersten Level abgelegt werden, so muss als Elternartikel „Start“ ausgewählt werden. Das gleiche Fenster wird geöffnet, wenn der vierte Knopf „Bewegen“ betätigt wird. Dieser ist nur aktiviert, wenn ein Artikel aus der Liste ausgewählt ist, also auf der rechten Seite angezeigt wird. Der angezeigte Artikel wird dann als Unterartikel des im Auswahlfenster gewählten Artikels gespeichert.

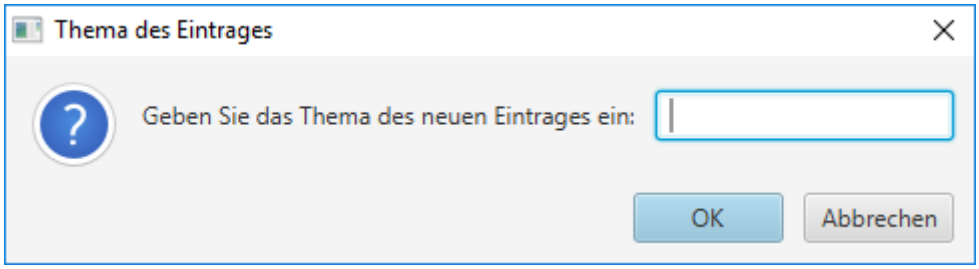

Abbildung 4.12: Das Eingabefenster für den Titel des neuen Artikels

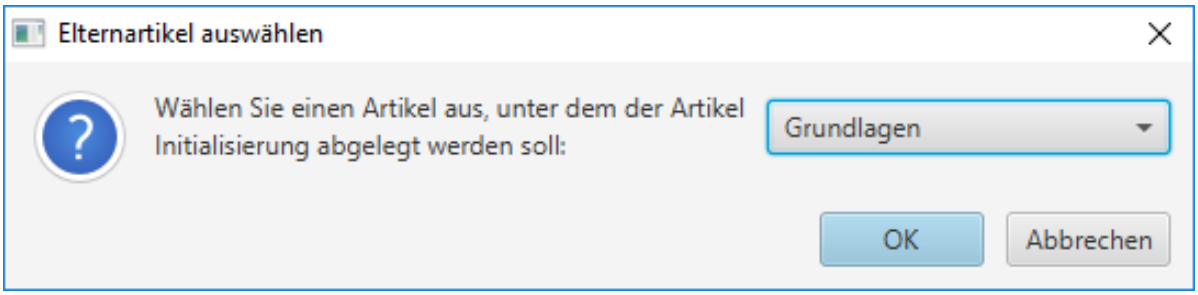

Abbildung 4.13: Das Auswahlfenster für die Verschiebung eines Artikels

Der zweite Knopf „Speichern“ speichert den angezeigten Artikel. Der Speicherort kann nicht gewählt werden, da die Wissensdatenbank nicht zur stückweisen Verteilung gedacht ist, sondern bestenfalls bei der ersten Version des Programms vollständig ist.

Es muss also keine Möglichkeit bestehen, bestimmte Teile der Datenbank an anderen Stellen zu speichern. Sollte eine Aktualisierung der Datenbank nötig sein, ist es empfehlenswert, eine komplett neue Version des Programms zu verteilen, die die aktualisierte Wissensdatenbank enthält. Auf diese Weise ist sichergestellt, dass die Wissensdatenbank in sich immer konsistent ist. Eine andere Möglichkeit wäre es gewesen, einen Speicherort manuell auszuwählen und die Datenbank „kapitelweise“ zu verteilen. Dies würde bei einer gewissen Anzahl von Artikeln jedoch zu erheblichem Arbeitsaufwand führen. Das Vorhandensein der Wissensdatenbank steht dem Lernfortschritt weit weniger im Weg als ein Überfluss an Aufgaben, weshalb Aufgaben im Gegensatz zu Einträgen der Wissensdatenbank einzeln oder in Gruppen verteilt und importiert werden können.

Der Knopf „Löschen“ entfernt den angezeigten Artikel aus der Datenbank. Dieser Vorgang ist endgültig, weshalb es zwei Bestätigungen gibt, bevor ein Artikel gelöscht wird (Abbildungen 4.14 und 4.15).

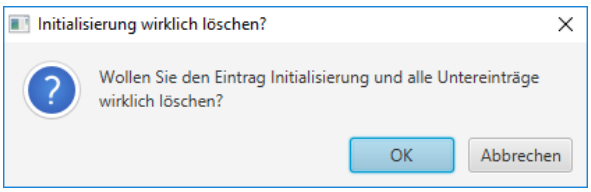

Abbildung 4.14: Die erste Bestätigung, bevor ein Artikel gelöscht wird

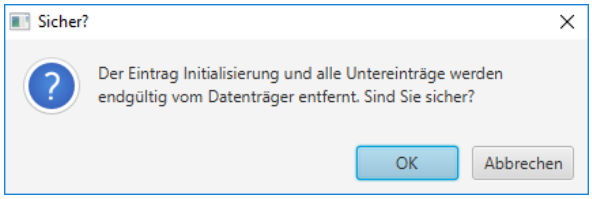

Abbildung 4.15: Die zweite Bestätigung, bevor ein Artikel gelöscht wird

Der rechte Teil des Fensters beinhaltet zwei Textfelder. Im oberen Textfeld wird der Artikel selbst verfasst. Dabei ist vor allem die Verwendung von Tabulatoren hilfreich, um eine geordnete Struktur zu ermöglichen. Die Textformatierung stimmt mit der endgültigen Anzeige überein, sodass Zeilenlängen und Zeichenbreiten gleich bleiben und bei der Bearbeitung schon berücksichtigt werden können. So ist es zum Beispiel möglich, bestimmte Zeichen oder Wörter direkt über- und untereinander zu platzieren. Im gezeigten Artikel betrifft dies die Anordnung der Kommentare im Codebeispiel, die alle auf einer Höhe beginnen, und die Einrückung des Codes selbst. Um Platz zu sparen, wurde eine Einrückung von zwei Leerzeichen gewählt. Dies ist selbstverständlich frei wählbar, sollte aber für alle Codebeispiele einheitlich gehalten werden. Die Tabulatorbreite ist technisch bedingt auf acht Leerzeichen festgelegt.

Das untere Textfeld enthält alle Stichwörter, die zu diesem Artikel führen sollen. Diese werden durch ein Komma ohne Leerzeichen getrennt. Es ist theoretisch möglich, mehrere Wörter in Folge als Stichwort zu verwenden (zum Beispiel die Folge „initialisieren Sie“). Dies mag bei bestimmten Artikeln nützlich sein, schränkt aber die Universalität der Stichwörter ein, da der Begriff "Sie" nichts mit dem Thema "Initialisierung" zu tun hat. Es ist wichtig, die Stichwörter der Artikel an das Vokabular der Fehlermeldungen anzupassen. Erst diese Verknüpfung ermöglicht es, eine tiefer gehende Erklärung zu Fehlermeldungen zu liefern, wie sie von Marceau, Fisler und Krishnamurthi [12] empfohlen wird.

5 Implementierung

Die Implementierung lässt sich in zwei Teile aufteilen. Aus dem ersten Teil kann nur das Programm für Studierende gestartet werden; erst in Verbindung mit dem zweiten Teil ist das Programm für Lehrende enthalten. So lässt sich vermeiden, dass Studierende mit dem Programm Aufgaben erstellen können. Es liegt außerdem eine Trennung von Benutzeroberfläche und Programmlogik vor. Der folgende Abschnitt befasst sich mit der Implementierung des allgemeinen Logikteils, wie er für das Programm für Studierende benötigt wird. Es folgt ein Abschnitt über den Code der Benutzeroberfläche für Studierende. Zuletzt wird die Implementierung der Oberfläche für Lehrende behandelt, wobei auch weitere Logik enthalten ist. Bei der Erklärung werden zudem einige Herangehensweisen an Probleme genauer betrachtet.

5.1 Logik

Eine Übersicht über alle Klassen, die sich mit der Programmlogik befassen, liefert Abbildung 5.4 auf Seite 40. Diese Klassen werden in den folgenden Abschnitten beschrieben.

5.1.1 Die Klasse `StudentMain`

Der Einstiegspunkt des Programms für Studierende ist die Methode `StudentMain.main(String[])`. Da diese Klasse eine JavaFX-Anwendung (siehe Abschnitt 2.4 auf Seite 18) ist, befindet sich der Code für die Initialisierung in der Methode `start(Stage)`. Von hier werden die GUI-Klassen `StudentGui`, `KnowledgeBase` und `Help` initialisiert (siehe Abschnitt 5.2 auf Seite 42). Außerdem werden mit der Methode `Task.loadAll(Path, boolean)` alle im Ordner *tasks* vorhandenen Aufgaben geladen. Dieser Codeabschnitt ist dargestellt in Abbildung 5.1. Der Parameter `temp` ist dabei nur für das Programm für Lehrende relevant. Mithilfe der Methode `Files.walk(Path)` wird ein `Stream` von `Paths` erstellt, der alle Dateien und Ordner im angegebenen Verzeichnis enthält. Dieser `Stream` wird gefiltert auf die Dateiendung .xml - das Format, in dem Aufgaben gespeichert werden. Daraufhin wird jede solche Datei durch die von der Methode `Task.load(Path, boolean)` erzeugte `Task`-Instanz ersetzt, in einem Array gesammelt und zurückgegeben. Jede Instanz der Klasse `Task` repräsentiert also eine Aufgabe, die die in Abschnitt 3.1.1 auf Seite 22 beschriebenen Anforderungen erfüllt.

```
194    public static Task[] loadAll(Path taskFolder, boolean temp) {
195        Task[] res = null;
196        if (taskFolder.toFile().exists()) {
197            try (Stream<Path> paths = Files.walk(taskFolder)) {
198                res = paths.filter(p -> p.getFileName().toString().endsWith(".xml")).map(p -> load(p, temp))
199                        .toArray(Task[]::new);
200            } catch (IOException e) {
201                e.printStackTrace();
202            }
203            return res;
204        }
205        return new Task[0];
206    }
```

Abbildung 5.1: Die Methode `Task.loadAll(Path, boolean)` zum Laden aller Aufgaben im angegebenen Verzeichnis

Daraufhin wird die Konfigurationsdatei geladen. Ist diese nicht vorhanden oder unvollständig, wird eine neue Datei erzeugt, wie in Abschnitt 4.1.1 auf Seite 26 beschrieben. Aus dieser Konfigurationsdatei wird unter Java Version 8 der Pfad zum JDK-Ordner verwendet, um die Systemvariable `java.home` zu ändern. Dies bewirkt, dass der Java-Compiler bei der Ausführung zur Verfügung steht. Die Zeile zum Setzen dieser Variable ist in Abbildung 5.2 gezeigt.

```
System.setProperty("java.home", Config.getJDKPath() + File.separator + "jre");
```

Abbildung 5.2: Anweisung zum Setzen der Systemvariable `java.home` auf die im JDK 8 enthaltene private JRE

Schnittstelle zur grafischen Benutzeroberfläche

Die grafische Benutzeroberfläche verwendet das Interface `StudentCallback`, um auf die Logik zuzugreifen. So sind nur die wirklich nötigen Methoden erreichbar. Das Interface definiert die Methoden `loadTask(Task)`, `importTasks(List`

`<File>`, `save()`, `reset()` und `run()`. Die Verwendung dieser Methoden wird in Abschnitt 5.2 auf Seite 42 erläutert. Für die Programmlogik ist die Methode `run()` besonders interessant, da hier der von Studierenden geschriebene Code kompiliert und getestet wird.

Zunächst wird der geschriebene Code zusammen mit dem Präfix und Suffix aus der Aufgabe in eine Java-Datei geschrieben. Der `ErrorMessageWriter` wird initialisiert, um die Nachrichten von `Precompiler` und `Compiler` in eine verständliche Form zu bringen. Daraufhin werden `Precompiler` und `Compiler` aufgerufen. Diese werden genauer in Abschnitt 5.1.2 beschrieben. War der Kompiliervorgang erfolgreich, wird die Testdatei mit der Klasse `Crypto` entschlüsselt. Die Tests werden von der Klasse `TestRunner` ausgeführt (beschrieben in Abschnitt 5.1.3 auf der nächsten Seite), der eine Instanz von `TestResultWriter` erhält, um das Ergebnis formatiert auszugeben. Die entschlüsselte Testdatei wird sofort wieder gelöscht. Auf diese Weise können die Tests nicht ausgelesen werden.

5.1.2 Die Klassen `Precompiler` und `Compiler`

Diese beiden Klassen werden verwendet, um von Studierenden geschriebenen Code auf Fehler zu überprüfen. Der `Precompiler` verwendet einen abstrakten Syntaxbaum und Pattern-Matching, um syntaktische und semantische Fehler zu finden. Der `Compiler` verwendet den Java-Compiler und interpretiert dessen Fehlermeldungen.

Der `Precompiler`

Diese Klasse stellt nur eine Methode zur Verfügung: `checkForErrors(String, String, Consumer<String>)`. Die Parameter sind der von Studierenden geschriebene Code, ein `String`, der zur erstellten Java-Datei führt, und ein `Consumer`, der die generierten Fehlermeldungen erhält und verarbeitet. Dieser `Consumer` ist eine Instanz der Klasse `ErrorMessageWriter`, die die Fehlermeldung anhand eines Codes erkennt und lesbar auf der grafischen Benutzeroberfläche anzeigt.

Der erste Teil der Methode befasst sich mit dem übergebenen Code. Mithilfe von regulären Ausdrucken, die in der Klasse `Pattern` gespeichert sind, werden Klammern, Apostrophe und Anführungszeichen gezählt. Liegt eine ungerade Anzahl vor, wird ein entsprechender Fehler ausgegeben. Außerdem wird der Fall erkannt, dass direkt nach `if`, `for` oder `while` keine runde Klammer folgt[1]. Es wird ebenfalls überprüft, ob ein `char`-Ausdruck zu viele Zeichen zwischen den Apostrophen hat, da der Java-Compiler in diesem Fall keine verständliche Meldung erzeugt.

Da mit regulären Ausdrücken nur sehr wenige Fehler gefunden werden können und die Erstellung der Ausdrücke sehr aufwendig und fehleranfällig ist, wird im zweiten Teil der Methode das Projekt JavaParser [4] verwendet. Der JavaParser erstellt aus einer Java-Datei einen abstrakten Syntaxbaum, auf dem weitere Operationen ausgeführt werden können. Dabei wird das Visitor-Pattern verwendet. Abbildung 5.3 zeigt eine Methode zur Analyse von binären Ausdrücken.

```
173        @Override
174        public void visit(BinaryExpr n, Consumer<String> out) {
175            // DETECT TYPE 3 ERROR (comparing Strings with == instead of equals)
176            if (n.getOperator() == Operator.EQUALS) {
177                if (ExpressionResolver.isString(n.getLeft()) && ExpressionResolver.isString(n.getRight())) {
178                    StringBuilder msg = new StringBuilder();
179                    msg.append(PRECOMPILER + SEP + B_STR_EQ);
180                    Optional<Range> range = n.getRange();
181                    if (range.isPresent()) {
182                        Range r = range.get();
183                        msg.append(SEP + r.begin.line + SEP + r.begin.column + SEP + r.end.line + SEP + r.end.column);
184                    } else {
185                        System.err.println("no range found for " + n.toString());
186                    }
187                    out.accept(msg.toString());
188                }
189            }
190            // DETECT TYPE 9 ERROR (confusing bit-wise with short-circuit
191            // operators
192            else if ((n.getOperator() == Operator.BINARY_AND || n.getOperator() == Operator.BINARY_OR)
193                    && ExpressionResolver.isBoolean(n.getRight()) && ExpressionResolver.isBoolean(n.getLeft())) {
194                out.accept(PRECOMPILER + SEP + D_AND_OR + SEP + convRange(n.getRange().get()));
195            }
196
197            super.visit(n, out);
198        }
```

Abbildung 5.3: Diese Methode wird vom JavaParser [4] für jeden binären Ausdruck im betrachteten Code ausgeführt.

Zuerst wird die Verwendung des Operators „==“ analysiert. Mithilfe der Klasse `ExpressionResolver` wird überprüft, ob beide Seiten des Ausdrucks vom Typ `String` sind. Ist dies der Fall, wird eine Fehlermeldung erzeugt, die darauf hinweist, dass `Strings` mit der Methode `equals()` verglichen werden sollten.

[1] Ein Semikolon nach `while` ist im Rahmen einer `do-while`-Schleife erlaubt.

Der zweite Teil der Methode erkennt die Verwendung der bitweisen Operatoren „&“ und „|“. Da dieses Programm für die Nutzung durch Neulinge gedacht ist, ist die Verwendung von bitweisen Operatoren meist unnötig oder sogar falsch. Es sollte daher gerade von Anfänger*innen die Verwendung von booleschen Operatoren bevorzugt werden.

Der `Compiler`

Diese Klasse dient als Schnittstelle zur Klasse `javax.tools.JavaCompiler`. Die Methode `setClasspath(String)` erlaubt es, den Ordner, in dem eine Aufgabe gespeichert ist, als Standardpfad zu verwenden. Außerdem können eigene JAR-Dateien - in diesem Fall JUnit - hinzugefügt werden. Die Methoden `compile(...)` und `compileTests(...)` unterscheiden sich in der Ausführlichkeit ihrer Fehlermeldungen. Die Methode `compileTests(...)` wird von der Klasse `TestRunner` verwendet, um die Tests zu kompilieren. Dabei ist die Ausgabe der Fehlermeldungen weniger detailliert. Die Methode `compile(...)` wird verwendet, um die von Studierenden erzeugte Klasse zu kompilieren. Die Fehlermeldungen werden hierbei um Zeilen- und Spaltennummern ergänzt und mit einem Fehlercode versehen, sodass der übergebene `ErrorMessageWriter` die erhaltene Meldung dem Compiler zuordnen kann.

5.1.3 Die Klasse `TestRunner`

Die Ausführung der JUnit-Tests einer Aufgabe übernimmt die Methode `runTests(...)`. Sie erhält einen Pfad zur entschlüsselten Testdatei und fügt dieser die `package`-Information hinzu. Außerdem wird jedem Test die Option `(timeout=1000)` hinzugefügt, sodass die Ausführung nach einer Sekunde mit einem Fehler endet. Hierdurch werden Endlosschleifen abgefangen. Die so erzeugte Datei wird vom `Compiler` kompiliert. Sollten hierbei Kompilierfehler auftreten, werden diese entsprechend angezeigt. Daraufhin wird die Testklasse ausgeführt und das `Result` der Methode `display(...)` übergeben.

Diese Methode überprüft, ob alle Tests erfolgreich waren. Ist dies der Fall, wird mit dem übergebenen `Consumer<String>` - eine Instanz von `TestResultWriter` - eine entsprechende Meldung erzeugt. Sind Tests fehlgeschlagen, wird überprüft, welcher Fehler zum Fehlschlag geführt hat. Ein falsches Ergebnis führt zu einem `AssertionError`. Ist der Test öffentlich, werden vom `TestResultWriter` Aufruf und erwartetes Ergebnis ausgegeben. Der Aufruf muss dazu innerhalb von „<>“ als Nachricht im JUnit-Test angegeben werden. Ist der Test privat, erfolgt lediglich ein Hinweis, dass die privaten Tests nicht erfolgreich waren. Ein privater Test wird durch die Nachricht „<private>“ im JUnit-Test erkannt. Tritt bei der Ausführung des Codes ein anderer Fehler auf, wird der Stack Trace ausgelesen und darin die entsprechende Zeile im Code gesucht. Die Ausgabe enthält dann den Namen der Exception oder des Errors und die Zeile, in der sie oder er aufgetreten ist.

5.1.4 Laden und Speichern

Zum Laden und Speichern von Aufgaben, Einträgen in der Wissensdatenbank und der Konfigurationsdatei wird das XML-Format verwendet. Im nächsten Abschnitt wird beschrieben, wie dies umgesetzt wurde. Der darauffolgende Abschnitt befasst sich mit den Klassen `Zipper`, `FileUtils` und `Crypto`, die weitere Funktionen liefern.

Laden und Speichern im XML-Format

Für die Serialisierung und Deserialisierung von Objekten im XML-Format wurde die XOM-API [5] verwendet. Diese bietet eine sehr einfache Anwendung, die in den folgenden Beispielen sichtbar wird. Um eigene Objekte in eine XML-Datei zu konvertieren, muss ein `Element` erstellt werden, das alle nötigen Informationen enthält, um das Objekt später wiederherzustellen. Diese beiden Funktionen wurden im Interface `XMLConvertable` in den Methoden `Element convertToXML()` und `convertFromXML(Element)` festgehalten. Jedes Objekt, das als XML-Datei serialisiert werden soll, muss dieses Interface implementieren. Dabei erstellt die erste Methode ein `Element` mit allen Informationen, während die zweite Methode aus diesem `Element` alle Informationen in die Instanz übernimmt. In diesem Projekt implementieren die Klassen `Task`, `KnowledgeBaseArticle` und `Config` das Interface.

Die (De-)Serialisierung selbst ist in der Klasse `XMLSerializer` enthalten. Abbildung 5.5 auf Seite 41 zeigt den sehr kurzen Code, mit dem XOM für beides verwendet wird. Es werden jeweils ein `Path xmlFile` und ein `XMLConvertable xmlConvertable` übergeben. Beim Speichern wird daraus ein `Element` erzeugt, das für ein `Document` verwendet wird. Der `Serializer` schreibt dieses `Document` in einen `FileOutputStream`, der die Zieldatei erhält. Beim Laden wird mit dem `Builder` die übergebene Datei eingelesen, deren Wurzelelement das zuvor gespeicherte `Element` ist. Der Erfolg des Vorgangs wird mit dem Rückgabewert `true` angegeben. Abbildung 5.6 auf Seite 41 zeigt eine beispielhaft erzeugte XML-Datei, die zur Verbesserung der Lesbarkeit nachträglich formatiert wurde.

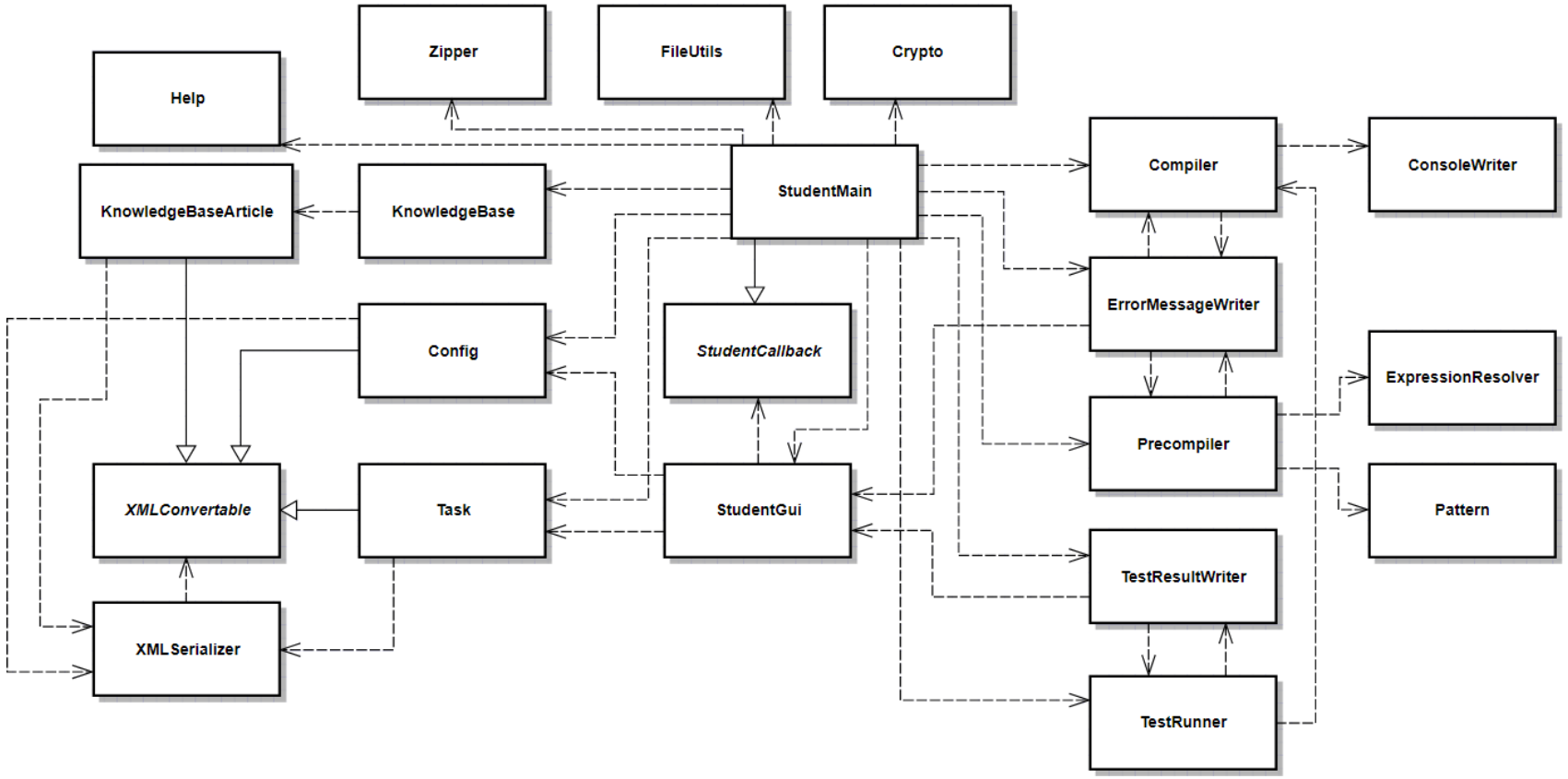

Abbildung 5.4: Vererbungs- und Nutzungsdiagramm der Programmlogik. Zur Vereinfachung sind die Klassen nur benannt, eine Beschreibung ist im Text vorhanden. Gestrichelte Pfeile stellen eine Verwendung dar.

```
39          Document doc = new Document(xmlConvertable.convertToXML());
40          try (FileOutputStream fos = new FileOutputStream(xmlFile.toFile())) {
41              Serializer ser = new Serializer(fos, "ISO-8859-1");
42              ser.write(doc);
43          } catch (IOException e) {
44              System.err.println("Serialization error for XML file " + xmlFile);
45              System.err.println(e.getMessage());
46              return false;
47          }
48          return true;
61          Builder builder = new Builder();
62          try {
63              xmlConvertable.convertFromXML(builder.build(xmlFile.toFile()).getRootElement());
64          } catch (ParsingException | IOException e) {
65              System.err.println("Deserialization error for XML file " + xmlFile);
66              System.err.println(e.getMessage());
67              return false;
68          }
69          return true;
```

Abbildung 5.5: Der Code zur Serialisierung und Deserialisierung von XML-Dateien mithilfe der XOM-API

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <task id="20">
3    <description>Um Code mehrfach ausführen zu können, ohne ihn mehrfach zu schreiben, können Methoden definiert werden.
4  Definieren Sie die Methode "public int addUp(int in)".
5  Ist die übergebene Zahl negativ, soll sie um 1 verkleinert werden.
6  Ansonsten soll die Zahl um 1 vergrößert werden.
7  Das Ergebnis soll zurückgegeben werden.</description>
8    <title>Methoden</title>
9    <code_prefix>public class Methods {</code_prefix>
10   <code_suffix>}</code_suffix>
11   <class_name>Methods</class_name>
12   <test_file>MethodsTest</test_file>
13 </task>
```

Abbildung 5.6: Eine beispielhaft mit der XOM-API erzeugte XML-Datei, die nachträglich zur Verbesserung der Lesbarkeit formatiert wurde

Die für das Entpacken von ZIP-Dateien verwendete Klasse ist `Zipper`. Diese verwendet das Paket `java.util.zip`, um gepackte Dateien zu entpacken, enthält aber auch eine Methode zum Verpacken, die vom Programm für Lehrende verwendet wird (siehe Abschnitt 5.3.2 auf Seite 48). Abbildung 5.7 zeigt die Methode `unzip(...)`. Zunächst wird mit einem `ZipInputStream` eine ZIP-Datei eingelesen. Dann werden alle `ZipEntrys` iteriert und die entsprechenden Ordner oder Dateien im Zielordner erstellt. Der Erfolg des Vorgangs wird mit dem Rückgabewert `true` bestätigt.

```
32    public static boolean unzip(File zip, Path targetFolder) {
33        try (ZipInputStream in = new ZipInputStream(new FileInputStream(zip))) {
34            ZipEntry entry;
35            while ((entry = in.getNextEntry()) != null) {
36                File target = targetFolder.resolve(entry.getName()).toFile();
37                if (!entry.isDirectory()) {
38                    if (!target.exists())
39                        target.createNewFile();
40                    if (!writeToFile(in, target)) {
41                        System.err.println("could not write to file " + target.toString());
42                        return false;
43                    }
44                } else if (!target.mkdirs()) {
45                    System.err.println("could not create folder " + target.toString());
46                    return false;
47                }
48                in.closeEntry();
49            }
50        } catch (IOException e) {
51            e.printStackTrace();
52            return false;
53        }
54        return true;
55    }
```

Abbildung 5.7: Die Methode `Zipper.unzip(...)`, mit der gepackte ZIP-Dateien entpackt werden können

Die Klasse `FileUtils` stellt lediglich die Methode `cleanUp(Path)` zur Verfügung. Diese Methode löscht alle leeren Ordner im angegebenen Verzeichnis und in allen Unterverzeichnissen, um nach der Entfernung von Aufgabenduplikaten einen besseren Überblick über den Ordner *tasks* zu behalten.

Die Methode `decrypt(File, File)` in der Klasse `Crypto` wird verwendet, um verschlüsselte Testdateien zu entschlüsseln. Dazu wird die Methode `crypto(...)` verwendet, die in Abbildung 5.8 gezeigt ist. Der Parameter `cipherMode` entscheidet, ob ver- oder entschlüsselt wird. Dann wird ein Schlüssel aus einem String generiert und der Inhalt der Eingabedatei mit dem AES-Algorithmus ver- oder entschlüsselt. Das Ergebnis wird in die Ausgabedatei geschrieben.

```
61    private static void crypto(int cipherMode, File inFile, File outFile) {
62        try (FileInputStream in = new FileInputStream(inFile); FileOutputStream out = new FileOutputStream(outFile)) {
63            Key secretKey = new SecretKeySpec(Arrays.copyOf(key.getBytes(), 16), "AES");
64            Cipher cipher = Cipher.getInstance("AES");
65            cipher.init(cipherMode, secretKey);
66
67            byte[] inputBytes = new byte[(int) inFile.length()];
68            in.read(inputBytes);
69
70            byte[] outputBytes = cipher.doFinal(inputBytes);
71
72            out.write(outputBytes);
73        } catch (IOException | GeneralSecurityException e) {
74            e.printStackTrace();
75        }
76    }
```

Abbildung 5.8: Die Methode `Crypto.crypto(...)`, die zur Ver- und Entschlüsselung verwendet wird

5.2 Benutzeroberfläche für Studierende

Einen Überblick über die Klassen der grafischen Benutzeroberfläche für Studierende liefert Abbildung 5.9 auf der nächsten Seite. Die Schnittstellen der Klasse `StudentGui` zum Logikteil wurden bereits in Abschnitt 5.1.1 auf Seite 37 behandelt. Die auf dem Diagramm gezeigten Klassen werden im Folgenden erläutert.

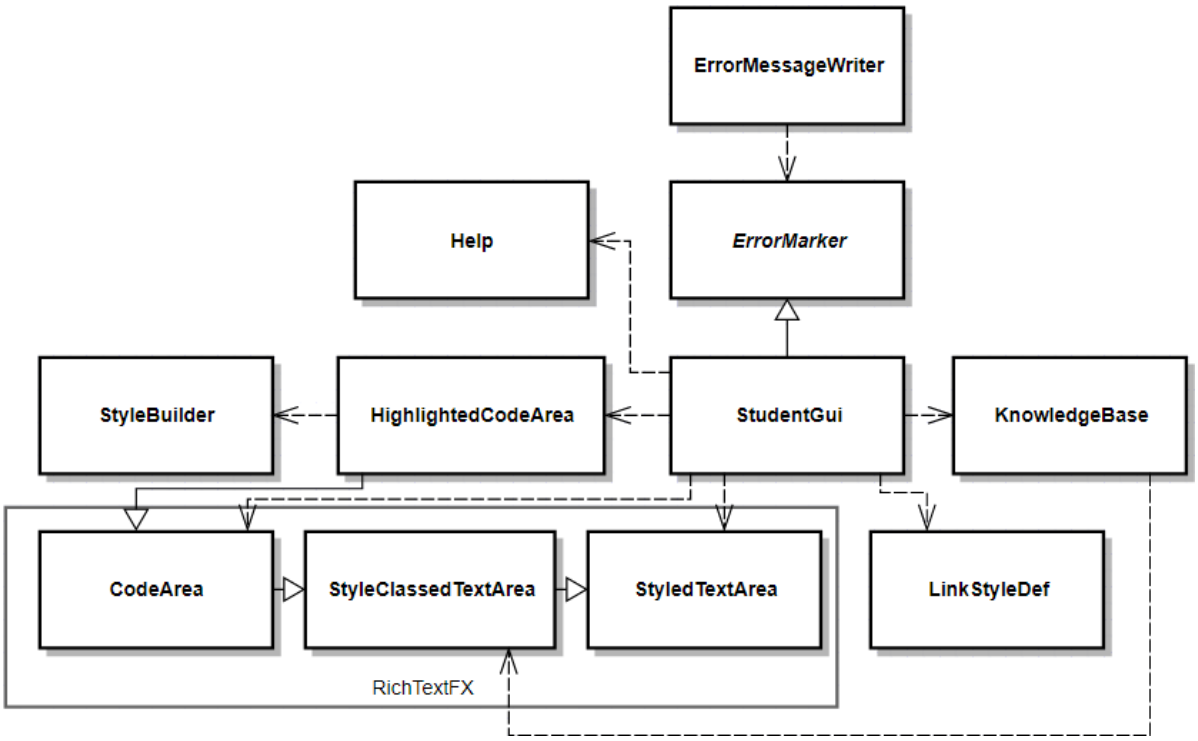

Abbildung 5.9: Verwendungs- und Vererbungsdiagramm der grafischen Benutzeroberfläche für Studierende. Zur Vereinfachung sind die Klassen nur benannt, eine Beschreibung ist im Text vorhanden. Gestrichelte Pfeile stellen eine Verwendung dar.

5.2.1 Initialisierung

Die Initialisierung der grafischen Oberfläche erfolgt jeweils in der Methoden `init(Stage)` in den Klassen `StudentGui`, `Help` und `KnowledgeBase`. Die Klasse `StudentGui` füllt wie in Abschnitt 2.4 auf Seite 18 beschrieben eine JavaFX-Anwendung mit Inhalt. Insbesondere wird die Klasse `HighlightedCodeArea` verwendet, wo später der Code eingegeben wird. Diese Klasse wird im folgenden Abschnitt näher erläutert. Außerdem implementiert `StudentGui` das Interface `ErrorMarker`, das von der Klasse `ErrorMessageWriter` (erwähnt in Abschnitt 5.1.2 auf Seite 38) verwendet wird, um Fehler und Warnungen im Code zu markieren.

Die Klassen `Help` und `KnowledgeBase` haben nur statische Methoden, sodass sie statisch verwendet werden können. Die Klasse `Help` verwendet das Package `javafx.scene.web`, um mit einem Webbrowser die Hilfeseite anzuzeigen. Die Hilfe kann auf diese Weise auch außerhalb der Anwendung in einem beliebigen Browser geöffnet werden. Die Klasse `KnowledgeBase` verwendet die Klasse `javafx.scene.control.TreeView` für die Baumstruktur der Einträge und die Klasse `StyleClassedTextArea`, die im folgenden Abschnitt näher erläutert wird, zum Anzeigen eines Eintrages.

5.2.2 Die Bibliothek RichTextFX

Erweiterte Möglichkeiten zum Gestalten von Textfeldern bietet RichTextFX [14]. Besonders interessant sind dabei die Klassen `CodeArea`, `StyleClassedTextArea` und `StyledTextArea`. Eine grobe Übersicht über die allgemeine Funktionsweise und die Unterschiede dieser Klassen liefert die Readme-Datei auf der Github-Seite des Projekts. Im Folgenden werden nur die für das Programm relevanten Aspekte behandelt.

`StyledTextArea` und `LinkStyleDef`

Die Klasse `StyledTextArea` liefert die allgemeinste Funktionsweise des Projekts. Es kann mit einer eigenen Definition angegeben werden, welche Elemente eines Textes wie angezeigt werden sollen. In der Klasse `StudentGui` wird diese Möglichkeit verwendet, um die Links von Schlüsselwörtern zu Einträgen in der Wissensdatenbank zu ermöglichen. Dafür wird die Klasse `LinkStyleDef` erstellt, die angibt, wie ein bestimmter Text behandelt werden soll. Abbildung 5.10 auf der nächsten Seite zeigt die Klasse, die eine `Runnable`-Instanz ausführt, wenn auf den blau unterstrichenen Text geklickt wurde. Der Cursor ist über dem Link außerdem eine Hand.

```java
package de.tu_darmstadt.informatik.skorvan.student.gui;

import javafx.scene.Cursor;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;

class LinkStyleDef {

    private Runnable action;

    LinkStyleDef() {}

    LinkStyleDef(Runnable action) {
        this.action = action;
    }

    void applyToText(Text text) {
        if(action != null) {
            text.setCursor(Cursor.HAND);
            text.setFill(Color.BLUE);
            text.setUnderline(true);
            text.setOnMouseClicked(click -> action.run());
        }
    }

}
```

Abbildung 5.10: Die Klasse `LinkStyleDef` zur Erstellung von Links

Um den definierten Style in einer `StyledTextArea` anzuwenden, muss diese bei der Definition übergeben werden. Abbildung 5.11 zeigt den Code dafür. Die ersten beiden Parameter beschreiben den Style eines ganzen Abschnitts und werden daher nicht verwendet - der Lambda-Ausdruck ist leer, also wird der Style nicht verändert. Die letzten beiden Parameter beschreiben den Style aller Zeichen im Text. Der dritte Parameter gibt dazu den Default-Style an, der zu Beginn für den gesamten Text verwendet wird. Standardmäßig erhält die Klasse `LinkStyleDef` keine bei einem Klick auszuführende Aktion, weshalb der Style nicht verändert wird. Der vierte Parameter gibt an, wie die als Style übergebene Klasse verwendet werden muss. In diesem Fall ist dies in der Klasse `LinkStyleDef` die Methode `applyToText(Text)`.

```java
        currentTaskLabel = new StyledTextArea<LinkStyleDef, LinkStyleDef>(
                new LinkStyleDef(),
                (a, b) -> {},
                new LinkStyleDef(),
                (text, style) -> style.applyToText(text));
```

Abbildung 5.11: Die Initialisierung der `StyledTextArea`, in der die aktuelle Aufgabe angezeigt wird

Um einen Link tatsächlich zu verwenden, muss der Style der `StyledTextArea` an der gewünschten Stelle geändert werden. Dies wird in Abbildung 5.12 auf der nächsten Seite dargestellt. Zu Beginn der Methode wird überprüft, ob bereits alle Schlüsselwörter aus der Wissensdatenbank geladen wurden. Diese müssen daraufhin zu einem `Pattern` zusammengefügt werden, um sie im zu überprüfenden Text leicht zu finden. Dies wird mithilfe des `Matchers` getan. Beginn und Ende jedes Treffers zeigen an, an welcher Stelle ein Link gesetzt werden muss. Der `String group` beinhaltet das gefundene Schlüsselwort und in der vorher erstellten `Map` kann der Name des entsprechenden Artikels gefunden werden. Der Style ist eine Instanz der Klasse `LinkStyleDef`. Dieser Instanz wird mit einem Lambda-Ausdruck übergeben, welche Aktion bei einem Klick ausgeführt werden soll. Die Methode `KnowledgeBase.show(String)` zeigt den Artikel mit dem übergebenen Titel an.

Es wird deutlich, dass die Klasse `StyledTextArea` eine sehr große Freiheit lässt, wie ein Text angezeigt werden soll. Der Nachteil ist dabei der hohe Codeaufwand. Anders ist es bei den folgenden, spezialisierten Klassen.

```
278    private void checkForLinks(String msg, int length, StyledTextArea<LinkStyleDef, LinkStyleDef> text) {
279        if (linkPattern == null || linkMap == null) {
280            StringBuilder sb = new StringBuilder();
281            linkMap = KnowledgeBase.getKeywordMapping();
282            linkMap.keySet().forEach(keyword -> {
283                if (!keyword.isEmpty())
284                    sb.append("(" + Pattern.quote(keyword) + ")|");
285            });
286            if (sb.length() <= 0)
287                return;
288            sb.deleteCharAt(sb.length() - 1);
289            linkPattern = Pattern.compile(sb.toString());
290        }
291        Matcher matcher = linkPattern.matcher(msg);
292        while (matcher.find()) {
293            int start = matcher.start() + length;
294            int end = matcher.end() + length;
295            String group = matcher.group();
296            text.setStyle(start, end, new LinkStyleDef(() -> {
297                KnowledgeBase.show(linkMap.get(group));
298            }));
299        }
300    }
```

Abbildung 5.12: Die Methode zum Finden und Markieren von Links

`StyleClassedTextArea`

In der Wissensdatenbank soll der erste Absatz jedes Artikels die Überschrift sein und somit großgeschrieben werden. Die Klasse `StyleClassedTextArea` ermöglicht dies, indem eine vorgegebene CSS-Klasse des RichTextFX-Projekts verwendet wird. Diese CSS-Klasse ist in Abbildung 5.13 gezeigt. Es hätte ebenfalls die Möglichkeit gegeben, zwei Instanzen der Klasse `Label` aus JavaFX zu verwenden. Dies hätte allerdings die Umwandlung der gespeicherten Artikel in Titel und Inhalt verkompliziert.

```
1  .paragraph-box:first-paragraph {
2      -fx-font-size: 20px;
3  }
```

Abbildung 5.13: Die Datei kb.css, die den ersten Absatz der `StyleClassedTextArea` vergrößert

`CodeArea` und `HighlightedCodeArea`

RichTextFX liefert die Klasse `CodeArea`, welche bereits eine für Code-Editoren spezialisierte Version der `StyleClassedTextArea` ist. Anhand einer Demo auf der Github-Seite [15] lässt sich leicht erkennen, wie das Highlighting von Schlüsselwörtern funktioniert. Diese Demo wurde als Vorlage gewählt, um die Klasse `HighlightedCodeArea` zu erstellen, die diese und weitere Funktionalitäten enthält.

Außer dem Syntax-Highlighting übernimmt die Klasse `HighlightedCodeArea` auch die Markierung von Fehlern und Warnungen von Compiler und Precompiler. In der Methode `StyleSpans<Collection<String>> computeHighlighting(String)` wird der Style des Textes nach jeder Änderung im Text berechnet. Der Rückgabetyp `StyleSpans` ist eine Klasse des RichTextFX-Projekts, mit der eine `StyleClassedTextArea` angepasst werden kann. Sie enthält Informationen, welche Style-Klassen an welchen Stellen des Textes angewendet werden sollen. Die CSS-Dateien application.css und javakeywords.css (siehe Abbildungen 5.14 und 5.15 auf der nächsten Seite) beschreiben, welche CSS-Eigenschaften die jeweiligen Klassen haben.

Da RichTextFX keine Funktionalität bietet, an beliebigen Stellen einen Style einzufügen, wurde die Klasse `StyleBuilder` erstellt. Diese ermöglicht es, mit der Methode `addStyle(String, int, int)` an beliebiger Stelle eine Style-Klasse hinzuzufügen sowie mit `create()` daraus eine Instanz der Klasse `StyleSpans` zu erzeugen. Sie verwendet dafür die Klasse `StyleSpansBuilder` von RichTextFX. Diese erlaubt es nur, den Style in einem Durchgang von vorne nach hinten zu bauen. Jede Style-Klasse wird durch einen String repräsentiert. Als Codierung wird im `StyleBuilder` jeder verschiedenen Style-Klasse ein Bit eines `shorts` zugewiesen. Dann wird eine `ArrayList<Short>` mit einem Eintrag für jedes Zeichen im Text erstellt, sodass für jedes Zeichen eine beliebige Kombination aus allen bekannten Styles gespeichert wird. Dies ermöglicht eine effiziente Kombination von bis zu 16 Style-Klassen.

Die Methode `create()` - gezeigt in Abbildung 5.16 auf der nächsten Seite - durchläuft dann diese `ArrayList` und fügt für jedes Zeichen die entsprechenden Style-Klassen als `ArrayList<String>` - oder auch keinen Style - zum `StyleSpansBuilder` hinzu. Dafür ruft sie für jedes gesetzte Bit ab, welcher String durch dieses Bit repräsentiert wird.

Diese Information ist in einer `ArrayList<String>` gespeichert, wo der entsprechende `String` unter dem Index gespeichert ist, der gleich der Position des Bits ist.

```
.error {
    -rtfx-underline-color: red;
    -rtfx-underline-width: 1px;
    -rtfx-underline-dash-array: 2 2 3 2;
}

.warning {
    -rtfx-underline-color: gold;
    -rtfx-underline-width: 1px;
    -rtfx-underline-dash-array: 2 2 3 2;
}
```

Abbildung 5.14: Die Datei application.css mit dem Style für Fehler und Warnungen im Code

```
.keyword {
    -fx-fill: purple;
    -fx-font-weight: bold;
}
.semicolon {
    -fx-font-weight: bold;
}
.paren {
    -fx-fill: firebrick;
    -fx-font-weight: bold;
}
.bracket {
    -fx-fill: darkgreen;
    -fx-font-weight: bold;
}
.brace {
    -fx-fill: teal;
    -fx-font-weight: bold;
}
.string {
    -fx-fill: blue;
}

.comment {
    -fx-fill: cadetblue;
}

#codeArea .paragraph-box:has-caret {
    -fx-background-color: #f2f9fc;
}
```

Abbildung 5.15: Die Datei java-keywords.css mit dem Style für Schlüsselwörter, Strings, Kommentare und Sonderzeichen ▷

```
public StyleSpans<Collection<String>> create() {
    if (styleArray == null || styleArray.size() == 0)
        return null;
    StyleSpansBuilder<Collection<String>> sb = new StyleSpansBuilder<>();
    styleArray.forEach(s -> {
        ArrayList<String> style = new ArrayList<>(Short.SIZE);
        for (byte bit = 0; bit < styleValues.size(); bit++) {
            if ((s & (1 << bit)) != 0)
                style.add(styleValues.get(bit));
        }
        sb.add(style, 1);
    });
    return sb.create();
}
```

Abbildung 5.16: Die Methode `create()` der Klasse `StyleBuilder`, mit der der Style des Codes zusammengesetzt wird

5.3 Erweiterungen für Lehrende

Die Möglichkeit, Aufgaben und Einträge in der Wissensdatenbank zu erstellen, zu verändern und für die Verteilung zu verpacken, erfordert eine Erweiterung der Implementierung. Diese Erweiterung wird in den folgenden Abschnitten erläutert. Abbildung 5.17 auf der nächsten Seite zeigt eine Übersicht über die verwendeten Klassen sowie deren Beziehung zu bereits erläuterten Klassen.

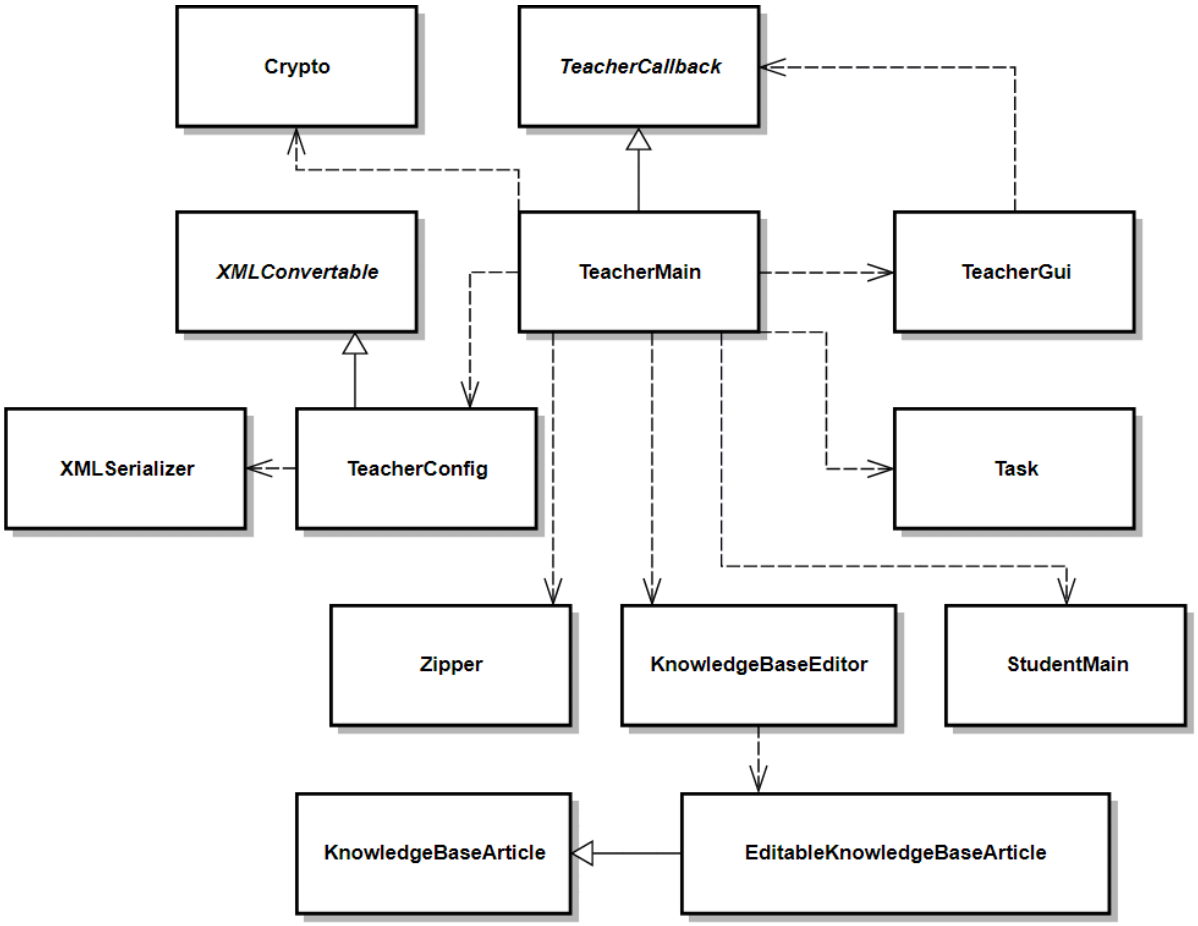

Abbildung 5.17: Die für die Erweiterung für Lehrende verwendeten Klassen und deren Beziehung zueinander und zu bereits erläuterten Klassen. Zur Vereinfachung sind die Klassen nur benannt, eine Beschreibung ist im Text vorhanden. Gestrichelte Pfeile stellen eine Verwendung dar.

5.3.1 Die Klasse `TeacherMain`

Ähnlich wie im Programm für Studierende ist `TeacherMain` der Einstiegspunkt für die Anwendung. Diese Klasse implementiert das Interface `TeacherCallback`, welches von der Klasse `TeacherGui` verwendet wird, um grafische Oberfläche und Logik miteinander zu verbinden. In der Methode `start(Stage)` der Klasse `TeacherMain`, die, wie in Abschnitt 2.4 auf Seite 18 beschrieben, beim Start des Programms aufgerufen wird, wird eine Instanz von `TeacherGui` erstellt und mit der `init(Stage)`-Methode initialisiert. Auch der `KnowledgeBaseEditor` (siehe Abschnitt 5.3.4 auf Seite 50) wird hier statisch initialisiert. Außerdem wird mit der Klasse `TeacherConfig` die Konfigurationsdatei geladen, in der die nächste zu verwendende ID einer Aufgabe gespeichert ist. Wurde keine Konfigurationsdatei gefunden, wird diese neu erstellt und die nächste ID auf 1 gesetzt.

Die Methoden, die im Interface `TeacherCallback` definiert werden, werden nun aufgeführt und kurz erläutert. Die Methode `newTask()` erstellt mithilfe der `Task.TaskFactory` eine neue, leere Aufgabe. Daraufhin kann die Aufgabe in der GUI frei bearbeitet werden, bis entweder `save(String[])` oder `save(String[], Path)` aufgerufen wird. Das `String`-Array enthält jeweils die sechs Eingaben zu den verschiedenen Attributen einer Aufgabe (siehe Abschnitt 3.1.1 auf Seite 22 und Abbildung 4.8 auf Seite 33). Ist ein `Path` gegeben, wird die Aufgabe an dieser Stelle gespeichert. Ansonsten wird sie an der gleichen Stelle gespeichert, an der sie zuletzt gespeichert war. Ist kein vorheriger Pfad bekannt, wird ein Dateibrowser geöffnet, mit dem ein Speicherort ausgewählt werden kann. Beim Speichern wird mithilfe der Methode `Crypto.encrypt(File, File)` die angegebene Testdatei verschlüsselt, falls diese sich seit dem letzten Speichervorgang verändert hat.

Um bereits vorhandene Aufgaben zu öffnen, werden die Methoden `openTasks(List<Path>, boolean)` und `importTasks(List<File>)` verwendet. Die erste Methode (gezeigt in Abbildung 5.18 auf der nächsten Seite) öffnet alle in einem „Datei öffnen“-Dialog ausgewählten Aufgabendateien. Der übergebene `boolean` entscheidet hierbei, ob die Aufgabe aus einer temporären Datei geladen wird oder nicht. Dies beeinflusst, ob die Aufgabe an dieser Stelle auch wieder gespeichert werden soll. Die zweite Methode lädt alle übergebenen ZIP-Dateien, in denen Aufgaben verpackt sind (Abbildung 5.19 auf der nächsten Seite). Hierfür wird die Klasse `Zipper` verwendet (siehe Abschnitt 5.1.4 auf Seite 39). Die Dateien werden in einen temporären

Ordner entpackt, weshalb sie beim Laden mit der Methode `Task.loadAll(Path, boolean)` auch nur temporär geladen werden. Wird eine importierte Aufgabe geändert und dann gespeichert, muss ein neuer Speicherort gewählt werden.

```
58    @Override
59    public boolean openTasks(List<Path> taskFiles, boolean temp) {
60        Task[] tasks = taskFiles.stream().map(p -> Task.load(p, temp)).toArray(Task[]::new);
61        if (tasks != null) {
62            gui.addTasks(tasks);
63            return true;
64        } else {
65            gui.showLoadError();
66            return false;
67        }
68    }
```

Abbildung 5.18: Die Methode `open(List<Path>, boolean)` in der Klasse `TeacherMain`, mit der bereits bestehende Aufgaben geöffnet werden können

```
185    @Override
186    public boolean importTasks(List<File> zips) {
187        try {
188            Path temp = Files.createTempDirectory("importedTasks");
189            zips.forEach(p -> Zipper.unzip(p, temp));
190            Task[] tasks = Task.loadAll(temp, true);
191            if (tasks != null) {
192                gui.addTasks(tasks);
193            }
194        } catch (IOException e) {
195            e.printStackTrace();
196            return false;
197        }
198        return true;
199    }
200 }
```

Abbildung 5.19: Die Methode `import(...)` in der Klasse `TeacherMain`, mit der ZIP-Dateien mit Aufgaben importiert werden können

Wenn eine Aufgabe bereits geladen ist und nur angezeigt werden soll (sie wird in der Liste in der GUI ausgewählt), wird die Methode `showTask(Task)` aufgerufen, damit die Programmlogik weiß, welche Aufgabe gerade angezeigt wird. Bevor die neue Aufgabe jedoch angezeigt wird, wird durch einen Aufruf der Methode `checkChanged(String[])` überprüft, ob die zuvor angezeigte Aufgabe geändert wurde. Es werden einfach alle übergebenen Eingaben mit den vorhandenen Informationen in der gespeicherten Aufgabe verglichen.

Um eine Aufgabe zu testen, muss das Programm für Studierende gestartet werden. Dies übernimmt die Methode `testTask()`. Die aktuell angezeigte Aufgabe wird gespeichert und mit der Methode `setTaskLocation(Path)` einer neuen Instanz der Klasse `StudentMain` übergeben. Daraufhin wird das Programm für Studierende gestartet. Schließlich können gespeicherte Aufgaben mit der Methode `exportTasks(Task[], File)` in eine ZIP-Datei gespeichert werden. Dies erlaubt die Verteilung an Studierende, die diese Datei dann in ihr Programm importieren können. Der folgende Abschnitt gibt eine Erläuterung über die Funktionsweise der Methode `Zipper.zip(File, Iterable<String>, Iterable<Path>)`.

5.3.2 Verpacken von Dateien

Die Methode `zip(File, Iterable<String>, Iterable<Path>)` ermöglicht es, beliebige Dateien in eine ZIP-Datei zu verpacken. Der Quellcode dieser Methode wird in Abbildung 5.20 auf der nächsten Seite gezeigt. Der erste Parameter gibt dabei die Zieldatei an. Es wird ein neuer `ZipOutputStream` erstellt, der in der Java-API enthalten ist. Die weiteren übergebenen Parameter bilden jeweils Paare aus dem Namen der Dateien oder Ordner, wie sie innerhalb der ZIP-Datei vorliegen sollen, und der tatsächlichen Datei oder dem tatsächlichen Ordner, der in die ZIP-Datei geschrieben werden soll. Für jedes dieser Paare wird ein `ZipEntry` erstellt, der einen neuen Eintrag im ZIP-Archiv darstellt. Mit der Methode `readFromFile(ZipOutputStream, Path)` wird die Datei, zu der der `Path` führt, byteweise in den `ZipOutputStream` eingelesen. Ist der `Path` jedoch ein Verzeichnis, wird nichts eingelesen. Daraufhin wird der `ZipEntry`

geschlossen und mit dem nächsten Paar fortgefahren. Tritt an irgendeiner Stelle ein Fehler auf, wird `false` zurückgegeben, damit der Fehler berichtet werden kann.

```
98      public static boolean zip(File zip, Iterable<String> names, Iterable<Path> data) {
99          try (ZipOutputStream out = new ZipOutputStream(new FileOutputStream(zip))) {
100             Iterator<String> namesIt = names.iterator();
101             Iterator<Path> dataIt = data.iterator();
102             while (namesIt.hasNext() && dataIt.hasNext()) {
103                 ZipEntry ze = new ZipEntry(namesIt.next());
104                 out.putNextEntry(ze);
105                 if (!readFromFile(out, dataIt.next()))
106                     return false;
107             }
108             out.closeEntry();
109         } catch (IOException e) {
110             e.printStackTrace();
111             return false;
112         }
113         return true;
114     }
```

Abbildung 5.20: Die Methode `Zipper.zip(...)`, mit der Dateien verpackt werden können

5.3.3 Die Klasse `TeacherGui`

Nach der Initialisierung ermöglicht es diese Klasse, die Methoden der Klasse `TeacherMain` durch eine grafische Benutzeroberfläche zu verwenden. Zum Laden von Aufgaben wird der entsprechende Knopf verwendet. Die ausgewählten Dateien werden der Logik übergeben, die daraufhin zurückmeldet, welche Aufgaben angezeigt werden sollen. Diesen letzten Schritt übernimmt die Methode `addTasks(Task[])`. Diese fügt die übergebenen Aufgaben in die Liste im linken Teil des Fensters hinzu und überprüft anschließend, ob Duplikate vorliegen. Dies ist der Fall, wenn zwei Aufgaben die gleiche ID besitzen. Darauf reagiert die grafische Oberfläche, indem für jeden Konflikt ein Dialog angezeigt wird, der beide vorhandenen Versionen beschreibt und dann die gewünschte Reaktion anfragt. Dies kann das Löschen oder temporäre Ignorieren einer der beiden Dateien sein oder das Ignorieren beider Dateien. Letzteres ermöglicht es, anhand der beiden Dateien selbst herauszufinden, welche der beiden die gewünschte ist.

```
287     public boolean checkChanged() {
288         boolean hasChanged = callback
289                 .checkChanged(Arrays.stream(input).map(tip -> tip.getText()).toArray(String[]::new));
290         if (hasChanged) {
291             Alert alert = new Alert(AlertType.CONFIRMATION);
292             alert.setHeaderText(null);
293             alert.setTitle("\u00c4nderungen speichern?");
294             alert.setContentText("M\u00f6chten Sie zuerst die \u00c4ndungen speichern?");
295             alert.getButtonTypes().setAll(ButtonType.YES, ButtonType.NO, ButtonType.CANCEL);
296
297             Optional<ButtonType> result = alert.showAndWait();
298             if (result.get().equals(ButtonType.YES)) {
299                 return callback.saveTask(Arrays.stream(input).map(i -> i.getText()).toArray(String[]::new));
300             }
301             if (result.get().equals(ButtonType.NO))
302                 return true;
303             if (result.get().equals(ButtonType.CANCEL))
304                 return false;
305         }
306         return true;
307     }
```

Abbildung 5.21: Die Methode `checkChanged()` in der Klasse `TeacherGui`, die eine Änderung am Inhalt erkennt und darauf hinweist

Wurde eine Aufgabe geändert und soll daraufhin eine neue Aufgabe angezeigt werden, so wird dies durch die Methode `checkChanged()` erkannt (Abbildung 5.21). Diese sammelt alle vorhandenen Eingaben und übergibt sie an das `StudentCallback`, wo sie mit der gespeicherten Aufgabe verglichen werden. Wurden Änderungen erkannt, wird in einem Dialog gefragt, ob die geänderte Aufgabe zuerst gespeichert werden soll. Die Anfrage, die zur Überprüfung auf Änderungen geführt hat (z.B. eine neue Aufgabe soll angezeigt werden), wird dann entweder abgebrochen oder durchgeführt. Dies gibt der Rückgabe-

wert der Methode an, wobei `true` bedeutet, dass nach ihrer Ausführung mit der Anfrage fortgefahren werden soll, und `false`, dass sie abgebrochen werden soll.

Schließlich beinhaltet diese Klasse die Methode `showFileDialog(int)`, die verschiedene Dialoge anzeigt. Entsprechend der festgelegten Konstanten bestimmt der übergebene `int`, ob eine XML-, Java- oder ZIP-Datei geöffnet oder gespeichert werden soll. Nachdem eine Datei ausgewählt wurde, wird entsprechend damit fortgefahren. Abbildung 5.22 zeigt beispielhaft das Verfahren zum Öffnen einer Java-Datei, die als Testdatei einer Aufgabe dient.

```
313         FileChooser fc = new FileChooser();
314         fc.setInitialDirectory(Paths.get("").toAbsolutePath().toFile());
315         fc.getExtensionFilters().clear();
316         if (type == OPENTEST) {
317             fc.setTitle("Testdatei \u00f6ffnen f\u00fcr Aufgabe " + input[0].getText());
318             fc.getExtensionFilters().add(new ExtensionFilter("Java Source Files (*.java)", "*.java"));
319             File result = fc.showOpenDialog(primaryStage);
320             if (result != null) {
321                 input[5].setText(result.getAbsolutePath().toString());
322                 return true;
323             }
324             return false;
325         }
```

Abbildung 5.22: Ein Ausschnitt der Methode `showFileDialog(int)`, mit dem ein „Datei öffnen“-Dialog für Java-Dateien angezeigt wird

5.3.4 Editor für die Wissensdatenbank

Die Wissensdatenbank wird ebenfalls von der Klasse `TeacherMain` initialisiert. Direkt in der Methode `KnowledgeBaseEditor.init(Stage)` werden mit der Methode `KnowledgeBaseArticle.loadAll(Path)` alle im entsprechenden Ordner gespeicherten Artikel geladen. Sie werden temporär in einer `LinkedList<EditableKnowledgeBaseArticle>` gespeichert. Abbildung 5.23 zeigt den Code, mit dem die Artikel aus der Liste in eine Baumstruktur einsortiert werden. Die Methode `addToChild(TreeItem<String>, TreeItem<String>, List<String>)` (Abbildung 5.24 auf der nächsten Seite) iteriert vom Wurzelelement den Baum rekursiv bis zum richtigen Knoten, indem die Liste der Elternknoten abgearbeitet wird. Sind keine Elternknoten mehr in der Liste vorhanden, ist der rekursiv erreichte „Wurzelknoten“ der gewünschte Elternknoten für das neue Element.

```
71          ListIterator<EditableKnowledgeBaseArticle> it = temp.listIterator(temp.size());
72          while (it.hasPrevious()) {
73              EditableKnowledgeBaseArticle a = it.previous();
74              articles.put(a.getTitle(), a);
75              if (!a.getTitle().equals("Start")) {
76                  TreeItem<String> item = new TreeItem<String>(a.getTitle()){
77                      @Override
78                      public String toString() {
79                          return (String) getValue();
80                      }
81                  };
82                  item.setExpanded(true);
83                  addToChild(item, rootItem, a.getParents());
84              }
85          }
```

Abbildung 5.23: Ein Ausschnitt der Methode `KnowledgeBaseEditor.init(Stage)`, in dem die geladenen Artikel in eine Baumstruktur sortiert werden

Weiterhin hat diese Klasse Methoden zum Erstellen neuer Einträge, die in der Baumstruktur einsortiert werden können, zum Verschieben von Artikeln oder ganzen Gruppen von Artikeln, zum Löschen von Artikeln und Gruppen sowie zum Speichern von Einträgen.

```
286    private static void addToChild(TreeItem<String> item, TreeItem<String> root, List<String> parents) {
287        if (parents == null || parents.isEmpty()) {
288            root.getChildren().add(item);
289            return;
290        }
291        List<TreeItem<String>> children = root.getChildren();
292        TreeItem<String> rightChild = children.stream().filter(t -> t.getValue().equals(parents.get(0))).findFirst()
293                .get();
294        addToChild(item, rightChild,
295                parents.size() <= 1 ? Collections.emptyList() : parents.subList(1, parents.size()));
296    }
```

Abbildung 5.24: Die Methode `KnowledgeBaseEditor.addToChild(TreeItem<String>, TreeItem <String>, List<String>)`, die von der Wurzel des Baumes aus rekursiv die Liste mit Namen der Elternknoten abarbeitet, um am Ende den neuen Knoten einzufügen

6 Evaluation

Nach dem Abschluss des Designs und der Implementierung des Programms wurde eine kleine Nutzerstudie durchgeführt. Leider nahm nur ein kleiner Teil der benachrichtigten Studierenden und Schüler*innen an der Evaluation teil, weshalb eine ausführlichere Bewertung im Rahmen einer Einführungsveranstaltung zu einem späteren Zeitpunkt erfolgen sollte (siehe auch Abschnitt 7 auf Seite 55).

Für die Evaluation wurden alle wichtigen Ordner und Dateien als ZIP-Archiv gepackt und verschickt. Zusätzlich enthielt das Archiv vier in einer gemeinsamen ZIP-Datei gepackte Beispielaufgaben. Für die Evaluation musste ein Fragebogen ausgefüllt werden, der Anweisungen zur Ausführung des Programmes enthielt. Der Fragebogen enthielt Aussagen, bei denen angegeben werden sollte, wie sehr diese zutreffen. Die Bewertung erfolgte von „überhaupt nicht“ (1) bis „voll und ganz“ (5). Die auszuführenden Vorgänge im Programm für Studierende wurden bereits in Kapitel 4 auf Seite 26 genauer beschrieben und mit Screenshots dargestellt.

6.1 Evaluation des Programms für Studierende

Nach dem Start des Programms musste zuerst die Konfiguration durchgeführt werden und daraufhin mussten die Aufgaben importiert werden. Dann konnte mit der Bearbeitung der ersten Aufgabe begonnen werden. Tabelle 6.1 zeigt die Ergebnisse des Fragebogens zu diesem Vorgang. Es wird deutlich, dass die Konfiguration die mit Abstand schlechteste Bewertung erhalten hat. Für die Weiterentwicklung des Programms empfiehlt es sich also, diesen Schritt zu vereinfachen oder sogar eine Möglichkeit zu finden, ihn wegfallen zu lassen. Die intuitive Bedienbarkeit des Programms wurde sehr gut bewertet, sodass dieser Teil des Programms in seiner jetzigen Form beibehalten werden kann.

Aussage	$\varnothing$	σ
Die Konfiguration ist verständlich und einfach.	3.4	1.1
Die grafische Oberfläche gefällt mir.	4.2	1.3
Es ist ersichtlich, welche Funktion welcher Knopf hat.	4.8	0.5
Neue Aufgaben können leicht importiert werden.	4.8	0.5
Die Bearbeitung einer Aufgabe kann leicht begonnen werden.	5.0	0.0

Tabelle 6.1: Die Bewertungen zu den ersten Schritten im Programm für Studierende

Im zweiten Schritt sollten alle importierten Aufgaben bearbeitet werden. Da die meisten Befragten vermutlich sicher im Umgang mit Java waren, erfolgte der Hinweis, auch absichtlich Fehler einzubauen, damit die Fehlermeldungen bewertet werden können. Tabelle 6.2 zeigt die Ergebnisse des Fragebogens zur Bearbeitung von Aufgaben. Auffällig ist hier, dass die Standardabweichung bei den ersten vier Fragen dieser Kategorie deutlich über 1 liegt. Dies zeigt, dass die Formulierung von Aufgabenstellungen und Fehlermeldungen sowie die Hilfe bei Problemen sehr subjektiv aufgenommen wird. Abschnitt 7 auf Seite 55 wird sich mit diesem Thema weiter befassen.

Aussage	$\varnothing$	σ
Die Aufgabenstellung hat mir klar gemacht, was ich zu tun habe.	4.2	1.8
Mir ist aufgrund der Fehlermeldungen klar geworden, was ich falsch gemacht habe.	3.8	1.9
Die Fehlermeldung hat den Fehler treffend beschrieben.	3.0	1.9
Mir wurde gut weitergeholfen, wenn ich nicht wusste, wie ich den Fehler beheben kann.	3.4	2.2
Ich war nach meiner Lösung der Aufgabe zufrieden.	4.0	1.0

Tabelle 6.2: Die Bewertungen zur Bearbeitung von Aufgaben

Schließlich sollte noch die Wissensdatenbank bewertet werden. Diese konnte durch Links in Aufgabenstellungen und Fehlern sowie durch den Knopf „Wissensdatenbank“ erreicht werden. Die Fragen und entsprechenden Bewertungen sind in Tabelle 6.3 auf der nächsten Seite aufgezeigt. Die durchweg guten Bewertungen zeigen, dass das Konzept der Wissensdatenbank sehr positiv aufgenommen wird und eine wertvolle Informationsquelle für die Bearbeitung von Aufgaben darstellt. Dieses Konzept könnte daher

auch in anderen Teilen der Lehre einen positiven Einfluss auf den Lernprozess haben. Besonders die Anordnung der Einträge in einer Baumstruktur wurde als sehr übersichtlich und verständlich aufgenommen, weshalb diese Form der Darstellung sehr geeignet erscheint.

Aussage	$\varnothing$	σ
Die grafische Darstellung der Wissensdatenbank gefällt mir.	4.8	0.5
Die Anordnung der Einträge ist verständlich und die Auswahl funktioniert leicht.	5.0	0.0
Ich habe zu jeder Frage im Kontext der gestellten Aufgaben einen passenden Artikel gefunden.	5.0	0.0
Wenn Sie einen Link verwendet haben: Der als Link markierte Begriff passt inhaltlich zum damit verbundenen Eintrag.	4.7	0.6

Tabelle 6.3: Die Bewertungen zur Wissensdatenbank

Freies Feedback zum Programm konnte in einem Textfeld gegeben werden. Im Folgenden wird dieses Feedback beschrieben und diskutiert.

Es wurde angemerkt, dass die automatische Suche nach einem JDK hilfreich wäre. Da Java Version 9 die programmatische Umstellung auf ein JDK nicht mehr unterstützt, wäre eine solche Funktionalität allerdings nicht hilfreich. Es bestünde dann jedoch die Möglichkeit, den Kommandozeilenbefehl anzuzeigen, mit dem die JAR-Datei des Programms mit dem JDK ausgeführt werden kann.

Weiter sei es nicht deutlich geworden, ob die Aufgaben korrekt gelöst wurden. Die Meldung, dass alle Tests erfolgreich waren (siehe Beschreibung in Abschnitt 4.1.5 auf Seite 30) erscheint in der Tat sehr unauffällig. Daher wird diese Anmerkung in Abschnitt 7 auf Seite 55 aufgegriffen.

Es wurde außerdem ein Fehler berichtet, bei dem der angezeigte Eintrag in der Wissensdatenbank nicht mit dem im linken Teil markierten Thema übereinstimmte. Dieser Fehler wurde aufgrund der geringen Schwere bereits behoben.

Inhaltlich wurde angemerkt, dass der Vergleich von Gleitkommazahlen mit einer Fehlertoleranz nicht in der Wissensdatenbank erklärt wurde. Außerdem sei der Text auf der Startseite der Wissensdatenbank unpräzise. Da die Informationen in der Wissensdatenbank ohnehin konkret auf die Lehre abgestimmt werden sollten (siehe auch Abschnitt 2.2.2 auf Seite 13), obliegt der Lehrkraft die genaue Ausarbeitung der Wissensdatenbank, sodass eine Änderung des Inhalts im Rahmen dieser Arbeit überflüssig ist. Trotzdem wird die Wichtigkeit dieser Inhalte noch einmal bewusst.

6.2 Evaluation des Programms für Lehrende

Die Beantwortung dieses Teils des Fragebogens wurde für die Abgabe nicht benötigt. Dies ist darin begründet, dass dieser Teil des Programms keiner breiten Masse von Personen zur Verfügung gestellt werden muss, sondern von einer kleinen Zahl, vielleicht sogar einer einzigen Person, verwendet wird. Daher ist die personalisierte Anpassung des Programms eine deutlich effizientere Methode der Verbesserung. Aus diesem Grund haben nur drei Befragte den Fragebogen zum Programm für Lehrende ausgefüllt. Die Ergebnisse sind in den Tabellen 6.4 und 6.5 auf der nächsten Seite dargestellt. Aus den Ergebnissen wird aber deutlich, dass die Verwendung des Programms gut aufgenommen wurde.

Aussage	$\varnothing$	σ
Die Eingabefelder sind verständlich und sinnvoll aufgeteilt.	4.3	1.2
Die Erstellung einer neuen Aufgabe ist von der Bedienung (nicht inhaltlich) einfach.	4.0	0.0
Bestehende Aufgaben können einfach bearbeitet werden.	5.0	0.0
Aufgaben können einfach gespeichert und getestet werden.	5.0	0.0
Die Darstellung der Aufgabe im Programm für Studierende (beim Testen) entspricht meiner Vorstellung bei deren Erstellung.	4.5	0.7

Tabelle 6.4: Die Bewertungen zur Erstellung und Bearbeitung von Aufgaben

Wieder zeigt sich, dass die intuitive Bedienung durchweg positive Bewertungen erhielt. Gerade die Bearbeitung sowie das Speichern und Testen von Aufgaben erhielten die beste Bewertung. Die Eingabefelder und die Erstellung einer neuen Aufgabe hingegen wurden jeweils nur mit 4.3 und 4 von 5 Punkten versehen. Daher könnte der Ansatz, alle Eingabefelder auf einer Seite anzuzeigen, noch einmal überdacht werden. Es könnte aber auch genügen, eine ausführliche Anleitung zur Bedienung zur Verfügung zu stellen.

Die Bearbeitung der Wissensdatenbank wurde äußerst positiv aufgenommen. Außer der Funktion von Schlüsselwörtern erhielten alle Aussagen mit 5 Punkten die beste Bewertung. Das Konzept der Bearbeitung der Wissensdatenbank scheint daher ausgereift, wobei die Schlüsselwörter noch einer Erklärung bedürfen.

Das einzige freie Feedback betrifft die Verwendung von Schlüsselwörtern aus der Wissensdatenbank. Es wird darauf hingewiesen, dass die Suche nur für ganze Wörter erfolgen solle und nicht auch als Teil eines Wortes. So solle zum Beispiel das Schlüsselwort „kleiner“ nicht im Wort „verkleinern“ markiert werden. Dies ist ein berechtigter Einwand, der aufgrund der geringen Schwere sofort umgesetzt wurde.

Aussage	$\varnothing$	σ
Die Bearbeitung und das Speichern eines Artikels war einfach.	5.0	0.0
Die Erstellung und das Einsortieren eines neuen Artikels ist intuitiv.	5.0	0.0
Die Funktion von Schlüsselwörtern ist deutlich und diese können leicht verändert werden.	4.0	0.0
Die Darstellung der Einträge im Programm für Studierende entspricht meinen Erwartungen.	5.0	0.0

Tabelle 6.5: Die Bewertungen zur Erstellung und Bearbeitung von Artikeln in der Wissensdatenbank

7 Zusammenfassung und Ausblick

Nachdem bestehende Einstiegsprogramme analysiert, Ergebnisse aus bisherigen Forschungsarbeiten entnommen oder abgeleitet und die Fehlermeldungen des Java-Compilers auf ihre Aussagekraft überprüft wurden, wurde auf dieser Grundlage die Anforderungsanalyse durchgeführt. Diese beschrieb, welche Eigenschaften das fertige Programm erfüllen sollte. Die Umsetzung resultierte in einer optisch positiv aufgenommenen und leicht verständlichen grafischen Oberfläche (siehe Tabelle 6.1 auf Seite 52) unter Verwendung des neuesten Java-Standards für grafische Oberflächen, JavaFX [16].

Die Einführung stark vereinfachter und auf nur einen konkreten Inhalt reduzierter Aufgaben soll den Lernvorgang in der Programmiersprache Java deutlich erleichtern. Mit entsprechend erstellten Aufgaben kann ein einzelnes Themengebiet gezielt erlernt und verbessert werden. Dabei liefern die vereinfacht und auf Deutsch formulierten, an wichtigen Stellen präziseren und ausführlicheren Fehlermeldungen eine bessere Rückmeldung über falsch erlernte oder noch nicht vollständig verstandene Sprachkonstrukte. Mit den richtigen Tests kann die formale Korrektheit des Programms sichergestellt werden.

Durch die Erstellung einer Wissensdatenbank lassen sich Programmierkonzepte thematisch sortiert und strukturiert nachschlagen, sodass Unklarheiten beim Lösen der Aufgaben möglichst schnell beantwortet werden können. Die Verlinkung von Schlüsselwörtern in Aufgabenstellungen und Fehlermeldungen bringt diese Datenbank bei der Anwendung des Programms näher und verleiht ein Gefühl der Sicherheit, wenn ein Begriff, der nicht verstanden wurde, zu einem Eintrag führt, der diesen Begriff erklärt.

Eine technische Grundlage zur Erstellung und Verteilung von Aufgaben besteht mit dem Programm für Lehrende. Die Effektivität der Anwendung des im Rahmen der vorliegenden Arbeit entwickelten Programms beruht jedoch maßgeblich auf der Anpassung der Inhalte; Aufgaben und Einträge der Wissensdatenbank müssen, wie zuvor erwähnt, auf den Inhalt der Vorlesung abgestimmt werden und zwar sowohl in Umfang und Reihenfolge der Themen, als auch in konkreten Formulierungen und geforderten Algorithmen. Die vorliegende Arbeit bietet somit eine Grundlage, aber keine fertige Lösung zur Vereinfachung der Lehre für Einsteiger*innen in die Programmiersprache Java.

Genau wie die Inhalte immer wieder angepasst und überprüft werden müssen, wird auch das Programm Änderungen unterzogen werden müssen. Außerdem konnten einige Verbesserungen aufgrund des Zeitaufwands nicht mehr in die vorliegende Arbeit aufgenommen werden. So wäre es sicher sinnvoll, die Wissensdatenbank nicht auf rein textbasierte Inhalte zu beschränken, sondern auch Bilder, Grafiken, Tabellen und andere Elemente darin darstellen zu können. Auch die Möglichkeit der individuellen Formatierung von Text besteht zurzeit nicht. Diese Eigenschaften könnten den Informationsgehalt vieler Einträge deutlich verbessern. Auch Aufgabenstellungen könnten davon profitieren, wenn andere Elemente darin aufgenommen werden könnten. Eine Grafik verdeutlicht das geforderte Verhalten eines Programms mitunter wesentlich greifbarer als ein Text. Schließlich ergab die Evaluation, dass ein deutlicherer Hinweis nötig ist, wenn eine Aufgabe erfolgreich bearbeitet wurde. Dies ließe sich entweder durch eine Meldung in einem neuen Fenster realisieren oder auch durch farbliche Akzente und Bilder neben der bisherigen Meldung.

Da die Evaluation leider nur in sehr kleinem Umfang stattgefunden hat, wäre es sicherlich auch sinnvoll, weiteres Feedback in einem echten Anwendungsfall zu sammeln. Das würde bedeuten, das Programm als Unterstützung in einer Vorlesungsveranstaltung oder in einem Informatik- oder Programmierkurs in der Oberstufe anzubieten, woraus sich sicher weitere Verbesserungsmöglichkeiten ergeben würden. Vor allem die Verständlichkeit der Fehlermeldungen könnte dann genauer überprüft werden, sodass die Formulierungen gegebenenfalls auch angepasst werden könnten. Mehrere Formulierungen könnten für den gleichen Fehler angeboten werden, sodass bei der Evaluation getestet werden könnte, welche davon die besten Bewertungen erhält. Aus einer erweiterten Evaluation würde ebenfalls hervorgehen, ob es sinnvoll wäre, das Programm mehrsprachig anzubieten. Im bisherigen Rahmen wurde bereits deutlich, dass der Vorgang der Konfiguration der größte Schwachpunkt des Programms ist. Diese sollte daher bei einer Weiterentwicklung des Programms definitiv verbessert werden.

Schließlich erscheint es sinnvoll, jede Aufgabe in einen oder mehrere Themenbereiche der Programmierung einzuordnen. Nach der Bearbeitung einer Aufgabe könnte dann der Hinweis erfolgen, dass es noch weitere Aufgaben zu diesem Thema gibt. Außerdem wäre eine strukturierte Gruppierung der Aufgaben möglich, was die Übersichtlichkeit bei einer großen Zahl von importierten Aufgaben verbessern würde. Für die Verbesserung der Fehlererkennung ist es denkbar, bei unbekannten Bezeichnern nach ähnlich geschriebenen Bezeichnern zu suchen, um auf diese hinzuweisen und die Möglichkeit eines Schreibfehlers in Betracht zu ziehen. Ob diese Umsetzung pädagogisch hilfreich ist, müsste dann nochmals überprüft werden.

Literaturverzeichnis

[1] Ben-Ari, Mordechai: *Compile and Runtime Errors in Java*. Department of Science Teaching, Weizmann Institute of Science, Rehovot 76100 Israel, Januar 2007. http://www.cs.pomona.edu/classes/cs051/handouts/JavaErrorsExplained.pdf (letzter Zugriff: 22.09.2017).

[2] Berlin, Land Brandenburg, Mecklenburg Vorpommern: *Kerncurriculum für die Qualifikationsphase der gymnasialen Oberstufe Informatik*. https://www.bildung-mv.de/export/sites/bildungsserver/downloads/unterricht/Rahmenplaene/Rahmenplaene_allgemeinbildende_Schulen/Informatik/kc-informatik-11-12-gym.pdf (letzter Zugriff: 21.12.2017).

[3] Brown, Neil C. C. und Amjad Altadmri: *Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs*. Transactions on Computing Education, 17(2):7:1–7:21, Mai 2017, ISSN 1946-6226. http://doi.acm.org/10.1145/2994154.

[4] Bruggen, Danny van: *JavaParser*. http://javaparser.org/ (letzter Zugriff: 11.10.2017).

[5] Harold, Elliotte Rusty: *XOM*. http://www.xom.nu/ (letzter Zugriff: 12.10.2017).

[6] Hessisches Kultusministerium: *Kerncurriculum gymnasiale Oberstufe - Informatik*. https://kultusministerium.hessen.de/sites/default/files/media/kcgo-in.pdf (letzter Zugriff: 21.12.2017).

[7] Hristova, Maria, Ananya Misra, Megan Rutter und Rebecca Mercuri: *Identifying and Correcting Java Programming Errors for Introductory Computer Science Students*. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, Seiten 153–156, New York, NY, USA, 2003. ACM, ISBN 1-58113-648-X. http://doi.acm.org/10.1145/611892.611956.

[8] Jakob, Marco: *JavaFX Dialogs*. http://code.makery.ch/blog/javafx-dialogs-official/ (letzter Zugriff: 30.09.2017).

[9] Kölling, Michael: *About BlueJ*. https://www.bluej.org/about.html (letzter Zugriff: 22.09.2017).

[10] Kölling, Michael: *Das BlueJ Tutorial Version 2.0.1*. Mærsk Institute University of Southern Denmark. https://www.bluej.org/tutorial/blueJ-tutorial-deutsch.pdf, Deutsche Übersetzung: Matthias Langlotz und Ingo Rhöse. https://www.bluej.org/tutorial/blueJ-tutorial-deutsch.pdf (letzter Zugriff: 21.12.2017).

[11] Kölling, Michael, Bruce Quig, Andrew Patterson und John Rosenberg: *The BlueJ System and its Pedagogy*. Computer Science Education, 13(4):249–268, 2003. https://www.bluej.org/papers/2003-12-CSEd-bluej.pdf.

[12] Marceau, Guillaume, Kathi Fisler und Shriram Krishnamurthi: *Mind Your Language: On Novices' Interactions with Error Messages*. In: *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, Seiten 3–18, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0941-7. http://doi.acm.org/10.1145/2048237.2048241.

[13] McCall, Davin und Michael Kölling: *Meaningful Categorisation of Novice Programmer Errors*. In: *2014 IEEE Frontiers In Education Conference (FIE)*, Seiten 1–8, October 2014. http://kar.kent.ac.uk/43796/.

[14] Mikula, Tomas: *RichTextFX*. https://github.com/FXMisc/RichTextFX (letzter Zugriff: 07.11.2017).

[15] Mikula, Tomas: *RichTextFX Demo: Automatic highlighting of Java keywords*. https://github.com/FXMisc/RichTextFX#automatic-highlighting-of-java-keywords (letzter Zugriff: 08.11.2017).

[16] Oracle: *JavaFX 8 Overview*. https://docs.oracle.com/javase/8/javafx/api/toc.htm (letzter Zugriff: 30.09.2017).

[17] Oracle: *JavaFX CSS Reference Guide*. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html (letzter Zugriff: 30.09.2017).

[18] Parlante, Nick: *About CodingBat*. http://codingbat.com/about.html (letzter Zugriff: 22.09.2017).

[19] Parlante, Nick: *CodingBat Java*. http://codingbat.com/java (letzter Zugriff: 22.09.2017).

[20] Parlante, Nick: *CodingBat Privacy*. http://codingbat.com/privacy.html (letzter Zugriff: 22.09.2017).

[21] Röhner, Gerhard: *Java-Editor*. http://javaeditor.org/doku.php (letzter Zugriff: 22.09.2017).

[22] Sächsisches Staatsministerium für Kultus und Sport: *Lehrplan Gymnasium - Informatik*. https://www.schule.sachsen.de/lpdb/web/downloads/lp_gy_informatik_2011.pdf?v2 (letzter Zugriff: 21.12.2017).

[23] Senatsverwaltung für Bildung, Jugend und Sport Berlin: *Rahmenlehrplan für die gymnasiale Oberstufe - Informatik*. https://www.berlin.de/sen/bildung/unterricht/faecher-rahmenlehrplaene/rahmenlehrplaene/mdb-sen-bildung-unterricht-lehrplaene-sek2_informatik.pdf (letzter Zugriff: 21.12.2017).

[24] The Eclipse Foundation: *Eclipse IDE for Java Developers*. http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygenr (letzter Zugriff: 22.09.2017).