TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fehlerkategorisierung von Programmierfehlern anhand des Kurses Datenstrukturen

Masterarbeit

zur Erlangung
des akademischen Grades
M.Sc.

Fakultät für Informatik
Professur für Softwaretechnik

Eingereicht von: Nadja Just
Matrikel Nr.: -
Einreichungsdatum: 30.03.2024

Betreuerinnen: Belinda Schantong
Prof. Dr.-Ing. Janet Siegmund

Abstract

Hintergrund: Das Erlernen des Programmierens ist komplex und stellt für viele Programmieranfänger*innen eine große Herausforderung dar. Der Kurs Datenstrukturen an der Technischen Universität ist ein Programmierkurs, der sich an diese richtet.

Ziele: Es wird Programmcode der teilnehmenden Studierenden evaluiert, um Einsichten in die Probleme der Studierenden beim Programmieren zu erhalten. Anhand einer Analyse im Anschluss sollen Erkenntnisse zur Verbesserung der Lehre generiert werden.

Methodik: Die Fehler im Programmcode der Studierenden werden zu diesem Zweck kategorisiert. Für die Auswertung werden schwache und starke Programmieranfänger*innen identifiziert und die Fehler beider Gruppen verglichen.

Ergebnisse: Starke Programmieranfänger*innen zeichnen sich dadurch aus, dass sie vor allem logische Fehler beim Programmieren verursachen, während schwache Programmieranfänger*innen Probleme mit logischen und semantischen Fehlern sowie der Verwendung von Datentypen und Klassen aufweisen. Über den Verlauf des Semesters nimmt die Anzahl der Arten der Programmierfehler sowie ihr Umfang in beiden Gruppen ab. Die Fehler der schwachen Programmieranfänger*innen gleichen sich dabei denen der starken Programmieranfänger*innen im Verlauf des Semesters an.

Implikationen: Die Veränderung der registrierten Fehler im Verlauf des Semesters deutet auf einen Erfolg der Lehre im Kurs Datenstrukturen hin. Probleme wiesen die Studierenden bei der Arbeit mit den Datenstrukturen Klasse, verkettete Liste und Graph auf, diesen sollte in der Lehre des Kurses angemessen begegnet werden.

Keywords: Programmierlehre, Programmieranfänger*innen, Fehlerkategorisierung, Java, Leistungsunterschiede

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ASL Anrechenbare Studienleistung

CS Computer Science

EQ Error Quotient

HP High Performing

HPS High Performing Student

IDE Integrated Development Environment

LP Low Performing

LPS Low Performing Student

PVL Prüfungsvorleistung

SPE Students with Prior Programming Experience

SPNE Students without Prior Programming Experience

UML Unified Modeling Language

1. Einleitung und Hintergrund

Über das Lehren und Lernen des Programmierens gibt es mannigfaltige wissenschaftliche Betrachtungen. Ziel dieser Arbeit ist es, einen Einblick in die Fehler der Studierenden an der Technischen Universität Chemnitz zu erlangen, die den Kurs Datenstrukturen belegen. Dazu werden Arbeiten von Studierenden, die in zwei Kategorien - starke und schwache Programmieranfänger*innen - eingeteilt wurden, auf Fehler ausgewertet und hinsichtlich der aufgetretenen Fehler verglichen.

Eine Einsicht in von den Studierenden verursachten Fehler kann sehr wertvoll für die Gestaltung der Lehre im Kurs sein. Das Lehrpersonal erhält so einen Einblick, welche Themen sich für die Studierenden als schwierig darstellen, sodass es diese anders oder in anderem Umfang lehren kann.

Durch den Vergleich von starken und schwachen Programmieranfänger*innen können Merkmale herausgearbeitet werden, um diese auch im Verlauf des Semesters zu erkennen. Wenn schwache Programmieranfänger*innen bereits früh im Verlauf des Semesters erkannt werden, können diese speziell im Lernprozess unterstützt werden.

Dazu wird in Kapitel 2 ein allgemeiner Überblick über das Forschungsfeld der vorliegenden Arbeit geschaffen. Dies beinhaltet eine Abgrenzung des Begriffs „Programmieranfänger*in“, ein Überblick zur Programmierlehre sowie Forschung zu schwachen und starken Programmieranfänger*innen. Anschließend wird in Kapitel 3 eine Forschungsarbeit zu den Problemen in der Programmierung der beiden Gruppen präsentiert sowie verschiedene Arten der Kategorisierung von Programmierfehlern vorgestellt.

Im Kapitel 4 werden verschiedene Aspekte der Ausführung der Arbeit betrachtet. Dazu gehören unter anderem die gestellten Forschungsfragen, der in der Arbeit betrachtete Kurs Datenstrukturen, die teilnehmenden Studierenden ,aber auch die verwendeten Werkzeuge zur Auswertung und eine Beschreibung der Vorgehensweise.

Die erhobenen Daten werden in Kapitel 5 präsentiert und ihre verschiedene Implikationen in Kapitel 6 besprochen sowie die gestellten Forschungsfragen beantwortet. Abschließend folgen in Kapitel 7 die Einschränkungen zur Validität sowie

in Kapitel 8 eine Zusammenfassung der Arbeit und ein Ausblick für zukünftige Forschungen, welche an diese hier vorliegende Arbeit anschließen.

Relevanz

Programmieren zu erlernen erfordert eine Vielzahl von Fähigkeiten. Gerade für Einsteiger*innen ist es schwer, die linguistischen Feinheiten von Programmiersprachen wie Java zu verstehen. Dies führt zu einer Vielzahl von Fehlern im Programmcode. Dazu kommt, dass Fehlermeldungen eines Compilers von Lernenden als nicht verständlich angesehen werden. Das erhöht die Schwierigkeit, die den Meldungen zugrunde liegenden Fehler zu beheben.[14]

Für Instruktor*innen von Programmierkursen ist es deswegen wichtig, vorherzusagen welche Arten von Fehlern den Lernenden Probleme bereiten. So können sie Einblicke in das Verhalten der Lernenden erhalten. Dies ermöglicht es, den Unterricht entsprechend anzupassen. Weiterhin gibt es Auskunft darüber, welche Themengebiete entsprechend vertieft unterrichtet werden sollten. So kann die Effizienz des Unterrichts erhöht werden und den Lernenden der Einstieg in die Programmierung erleichtert werden. [6, 20]

Ein Problem, dem Instuktor*innen begegnen, ist, dass sich die Ergebnisse in Programmierkursen häufig bimodal darstellen. Das bedeutet, dass nur wenige Studierende durchschnittliche Ergebnisse erzielen, während viele entweder mit hervorragenden Leistungen bestehen oder durchfallen. Dies erhöht die vorher dargestellte Dringlichkeit von Lehrenden, das Verhalten und die Fehler der Lernenden zu kennen.[24]

Interessant ist, wie konkret sich der Unterschied – welcher sich in bimodalen Ergebnissen niederschlägt – in den Programmierfähigkeiten von starken und schwachen Programmieranfänger*innen ausdrückt. Bisherige Forschung, die in Abschnitt 2.3 und Abschnitt 3.1 vorgestellt wird, beschäftigt sich vornehmlich mit Untersuchungen zum Lernverhalten und zu Vorkenntnissen. Dabei fehlt u.a. die Betrachtung, in welchem Ausmaß starke und schwache Programmierende Fehler in der Programmierung machen. Dieses Wissen ist aber notwendig für Lernende, um den Unterricht entsprechend anzupassen.

In der vorliegenden Arbeit wird spezifisch zu den Programmierfehlern von starken und schwachen Programmieranfänger*innen im Kurs Datenstrukturen geforscht werden, um Lehrpersonen ein genaues Bild über diese zu geben.

2. Theoretischer Hintergrund

Zur Lehre in der Programmierung gibt es verschiedene wissenschaftliche Betrachtungen. Zunächst wird in Abschnitt 2.1 der Begriff Programmieranfänger*in umrissen. Darauf folgt in Abschnitt 2.2 eine Übersicht über das Forschungsfeld der Programmierlehre. Anschließend werden verschiedene wissenschaftliche Untersuchung zu Unterschieden zwischen schwache und starke Programmieranfänger*innen in Abschnitt 2.3 vorgestellt.

2.1. Programmieranfänger*innen

Programmieranfänger*innen sind Personen, deren Kenntnisse in der Programmierung noch nicht ausgebaut sind. Sie befinden sich am Anfang des Lernprozesses der Programmierung. Im Durchschnitt benötigen Programmieranfänger*innen zehn Jahre, um zu Programmierexpert*innen zu werden. Programmieranfänger*innen zeichnen sich durch verschiedene Defizite im Wissen um die Programmierung aus. Sie haben ein lückenhaftes Verständnis u.a. von Variablen, Schleifen, Arrays und Rekursion. Weiterhin fällt ihnen das Planen von Programmen sowie das Testen dieser schwer. Ihr Wissen ist insgesamt oberflächlich und wenig strukturiert. Anstatt wie Programmierexpert*innen in Codeblöcken oder -strukturen zu denken, beschränken sich ihre Programmierfähigkeiten auf das Programmieren von aufeinanderfolgenden Codezeilen. [24]

2.2. Programmierlehre

Ebenso wie natürliche Sprachen folgen Programmiersprachen grammatikalischen Regeln. Sie bestehen aus einer überschaubaren Anzahl von Elementen, die auf unendlich viele Arten kombiniert werden können. Im Gegensatz zu natürlichen Sprachen sind Programmiersprachen künstliche Konstrukte. Treten in der menschlichen Kommunikation über Sprache Lücken auf, können diese mit dem Kontext der Unterhaltung gefüllt werden. Damit Programmiersprachen erfolgreich vom Compiler übersetzt werden können, müssen diese bis zu einem bestimmten Grad funktional, eindeutig und fehlerfrei sein. [24]

In der Informatik stellt das Erlernen des Programmierens eine wichtige Fähigkeit dar, deren Erwerb schon am Anfang eines Studiums fokussiert wird. Bennedsen und Caspersen betrachteten 2007 insgesamt 63 Institutionen, welche Einstiegskurse in der Programmierung anbieten. Dabei stellten sie eine Durchfallquote von durchschnittlich 33 % fest. Bei einzelnen der betrachteten Institutionen lag die Durchfallquote bei bis zu 50 %. In der Forschung von Watson 2014 wurden ähnliche Durchfallquoten in Einstiegskursen festgestellt. Die Notenverteilung der Studierenden in Programmierkursen weist in der Regel eine bimodale Verteilung auf. Das bedeutet, dass viele Lernende mit sehr guten Ergebnissen abschließen, viele durchfallen und nur wenige im Mittelfeld abschließen. [24], [27]

Studierende, welche Programmierkurse bestehen, verfügen nicht automatisch ein gutes Verständnis der Programmierung. Im Gegenteil scheint es so, dass das Verständnis von Studierenden nach ein oder zwei Semestern lückenhaft ist. Viele Studierende können in Studien nicht das Ergebnis von Codeausschnitten oder kleinen Codefragmenten vorhersagen. Des Weiteren bereitet das Abstrahieren von Problemen, der erste Schritt zu einem einem erfolgreichen Programmierprozess, Studierenden Probleme. Programmieranfänger*innen sind oftmals in der Lage, ihre Zielsetzungen in natürlicher Sprache zu artikulieren, stoßen jedoch auf Schwierigkeiten, diese in Programmcode zu überführen. [25]

Grundlegende Fähigkeiten für den Erwerb der Programmierfähigkeit

Welches Wissen benötigt wird, um Programmieren zu erlernen, ist Gegenstand verschiedener Forschungen. Einigkeit besteht in der Erkenntnis, dass das Erlernen des Programmierens Verständnis von komplexen Konzepten benötigt. In anderen Naturwissenschaften ist es möglich, komplexe Konzepte aufzubauen. Das Wissen um die Programmierung wird bisher nicht in dieser Form aufgebaut. Ebenfalls ist fraglich, ob ein solcher Ansatz beim Erlernen des Programmierens möglich ist. [27]

Allgemein werden zum Programmieren Strategien zur Implementierung von Algorithmen benötigt sowie Wissen und Verständnis zu den Algorithmen, wie auch die Fähigkeiten, diese anzuwenden. Zudem ist es erforderlich, mit den entsprechenden Tools oder Programmierumgebungen umzugehen. [24]

Ein Modell, um Wissenserwerb zu beschreiben, ist die Bloomsche Taxonomie. Diese unterteilt sechs Ebenen, nach denen Wissen erworben und aufgebaut werden kann. Fähigkeiten in den unteren Ebenen sind einfacher zu erlangen, während das Erlangen von Wissen in den oberen Ebenen komplexer ist. Ein weiteres Modell ist SOLO, dieses ist in fünf Ebenen unterteilt, wobei auch hier Fähigkeiten in höheren

Ebenen komplexer zu erlangen sind. Beide Modelle können als ergänzend angesehen werden. Zwischen Forscher*innen herrscht Uneinigkeit darüber, auf welchen Ebenen genau spezielle Fähigkeiten in der Programmierung eingeordnet werden und welche Fähigkeiten überhaupt zwingend notwendig sind, um Programmieren zu können. Interessant ist jedoch, dass sie sich in einem Punkt einig sind: Dass Programmieren insgesamt eine Fähigkeit ist, die sich sowohl in der SOLO als auch der Bloomschen Taxonomie in hohen Ebenen einordnen lässt. Das bedeutet, dass Programmieren zu lernen komplex ist und auch komplexer als andere naturwissenschaftliche Kurse. [27, 24]

Alternativ formulierten Shinners-Kennedy und Fincher [26] das Prinzip von Schwellenkonzepten in der Programmierung. Diese sind grundlegende Ideen und Prinzipien in einem bestimmten Bereich der Programmierung, welche als individuelle Verständnishürden auftreten. Diese zu überwinden stellt eine Herausforderung dar. Werden diese Hürden erfolgreich überwunden, erlangen Programmieranfänger*innen ein tiefgründiges Verständnis in der Programmierung. Schwellenkonzepte zeichnen sich dadurch aus, dass sie transformativ, integrativ, unwiderruflich, begrenzt und schwer zu verstehen sind. Es gibt keine einheitliche Meinung in der Forschung darüber, was genau als Schwellenkonzepte in der Programmierung bezeichnet werden kann. Um Schwellenkonzepte zu identifizieren, wurde ein zweidimensionales Gitter mit der Bezeichnung CoRe entwickelt. Dieses soll es Lehrkräften ermöglichen, spezifische Schwellenkonzepte ihrer Lernenden zu identifizieren. Ahadi et al.[1] entwickelten ebenfalls eine Metrik, um den Lernfortschritt von Programmierenden zu verfolgen. Diese basiert auf einer Einordnung anhand von Compilerfehlern. [24, 26]

Eine der Fähigkeiten, die in ihrer Relevanz beim Erlernen des Programmierens betrachtet wurde, ist das Debugging. So untersuchten 2005 Ahmadzadeh et al. [2] das Debugging-Verhalten von Programmieranfänger*innen und fanden heraus, dass ein Großteil der Programmierer*innen mit guten Debugging Fähigkeiten zugleich gute Programmier*innen waren.

Schwierigkeiten beim Erlernen des Programmierens

Wie genau die Fehler aussehen, welche die meisten Anfänger*innen oder Studierenden machen, ist wissenschaftlich umstritten. Des Weiteren gibt es keine einheitlich anerkannten Lösungen für diese Fehler.

In einer Studie zur Evaluation von Schwierigkeiten beim Erlernen der Programmierung befragten Lahtinen et al. [18] über 500 Studierende an sechs verschiedenen Universitäten. Dabei wurde sich auf Teilnehmende an Programmierkursen beschränkt, in welchen die Programmiersprachen Java und C++ verwendet wurden.

Über die Hälfte der Studierenden gab an, bereits Erfahrung in der Programmierung zu besitzen.

Nach den Angaben der Studierenden erlebten sie die Arbeit alleine als effektive Strategie zum Wissenserwerb. So wurde das individuelle Lernen als effektiver eingeschätzt als die Teilnahme an Vorlesungen und die individuelle Arbeit an Aufgaben des Kurses effektiver als die Teilnahme an angebotenen praktischen Übungen. Programme, die als Beispiele zur Verfügung gestellt wurden, empfanden die Studierenden als die lehrreichsten Unterrichtsmaterialien. [18]

Lehrende hingegen gaben die Arbeit in Übungen als eine effektive Lernerfahrung der Studierenden an. Überein stimmten sie mit den Studierenden in der Bewertung der individuellen Bearbeitung von Aufgaben aus dem Kurs. [18]

Am schwierigsten gestaltete sich für die Studierenden, ein Programm für eine spezifische Aufgabe zu entwerfen, Funktionalität in Methoden aufzuteilen und Fehler in den eigenen Programmen zu finden. All diese Punkte eint, dass sie eine komplexe Herausforderung darstellen und die Studierenden die Funktionalität von Teilen des Programmiercodes verstehen müssen. Detailwissen in bestimmten Bereichen reicht für die Lösung der genannten Schwierigkeiten nicht aus. Programmierkonzepte welche, die Studierenden als schwierig einschätzten, waren Pointer und Referenzen, abstrakte Datentypen, der Umgang mit Fehlern und die Verwendung von Bibliotheken der Programmiersprachen. Hier handelt es sich um komplexe Themenbereiche, die zum Teil keine Entsprechung in der realen Welt aufweisen. [18]

Lahtinen et al. [18] definieren folgende Punkte als Kernkompetenzen, um das Programmieren zu Erlernen: Das Verstehen von Programmierstrukturen, das Erlernen der Programmsyntax, die Kenntnis ein Programm zu planen welches eine bestimmte Aufgabe löst sowie Funktionalität in Methoden und Klassen aufzuteilen. Diese Kompetenzen konnten von den Studierenden entweder sehr leicht erlernt werden oder bereiteten ihnen sehr große Probleme. Insgesamt wurde festgestellt, dass Lernende zwar wenig Probleme hatten, Konzepte zu verstehen, dafür allerdings, diese anzuwenden.

In der Schlussfolgerung wurde festgestellt, dass praktische Lernerfahrung in der Programmierlehre einen wichtigen Stellenwert einnimmt. Studierende haben seltener Probleme beim Erlernen einzelner Aspekte sondern eher mit komplexen Konzepten. Komplexe Konzepte werden dadurch definiert, dass sie ein Verständnis von großen Codeblöcken erfordern. Im Gegensatz dazu stellen einfache Konzepte jene dar, bei denen ein Verständnis von Details oder wenigen Zeilen Code ausreichen. [18]

Daraus abgeleitet wurde, dass das individuelle Schreiben von Programmcode und die individuelle Anwendung der erlernten Programmierkonzepte einen wichtigen Stellenwert in der Programmierlehre einnehmen sollte. Die Kombination von wenigen Konzepten zur gleichen Zeit sollte genutzt werden, um die Programmierfähigkeiten der Lernenden zu gestalten. Im Speziellen sollte die Entwicklung von Programmen, deren Modifikation und das Debugging in der Lehre spezifisch betrachtet und unterrichtet werden.[18]

Hinderlich beim Erlernen des Programmierens scheinen die vom Compiler erzeugten Fehlermeldungen zu sein. McCall und Kölling [21] stellen fest, dass diese von den Lernenden meist als wenig hilfreich empfunden werden. Brown und Altadmri [9] kommen zu dem Schluss, dass die Qualität von ausgegeben Fehlernachrichten ungeeignet ist, da diese eher einen flüssigen Programmierprozess verhindern.

2.3. Unterschiede im Programmier- und Lernverhalten von schwachen und starken Programmieranfänger*innen

Bereits 1986 führten Perkins et al. Grundlagenforschung zum Programmierverhalten von Studierenden durch und teilten diese in drei Gruppen: Stoppers, Movers und extreme Movers. Mover bezeichnet Studierende, die ein Problem im Code analysieren und systematisch zu einer Lösung hin arbeiten. Stopper geben, wie der Name andeutet, auf und beenden das Programmieren. Extreme Movers probieren unsystematisch verschiedene Lösungen aus, erreichen aber keine zufriedenstellende Lösung mit diesem sprunghaften Verhalten. Weitere Forschung zeigte, dass Studierende sich nicht klar in diese Kategorien einordnen lassen. [25]

Liao et al. [19] führten eine Studie zu Verhaltensunterschieden beim Lernen von high performing (HP) und low performing (LP) Studierenden, die einen CS1 Programmierkurs belegten, durch. Zusätzlich wurden die Studierenden danach unterteilt, ob sie bereits Erfahrung in der Programmierung besitzen. Die Studierenden wurden darum gebeten an einer Umfrage teilzunehmen.

HP und LP Studierende fragen in einem ähnlichen Ausmaß Freunde und Kommilitonen nach Hilfe, wenn sie nicht weiterkommen. Der einzige Unterschied besteht darin, dass HP Studierende den Adressaten auf ihrer Frage basierend auswählen, während LP Studierende die Auswahl nach Verfügbarkeit und Komfort treffen. Beide Gruppen der Studierende nutzen alle verfügbaren Quellen, um Wissen zu erlangen. Es wurden nur zwei Unterschiede im Verhalten der Studierenden festgestellt. Zum einen nehmen Studierende der Gruppe HP weniger oft das Feedback

in der Mitte des Semesters wahr. Zudem hören sie den Podcast, der zu diesem Feedback gehört, seltener. [19]

Was beide Gruppen unterscheidet, ist, dass HP Studierende mit mehr Sorgfalt dem selbst geschriebenen Code entwickeln. Das bedeutet konkret, dass sie sicherstellen, ihren Code vollständig zu verstehen, bevor sie diesen abgeben. Außerdem treffen sie Vorhersagen über das Verhalten des Codes, bevor sie ihn ausführen und, falls sie in der Vorbereitung zur Klausur Unklarheiten auftreten, ähnliche Aufgaben referenzieren. Weiterhin helfen sie anderen Kommilitonen häufiger bei Problemen. LP Studierende hingegen konzentrieren sich vorrangig darauf, einen Programmcode zu schreiben, der den Anforderungen der gestellten Aufgaben entspricht, ob sie die eigene Abgabe selbst nachvollziehen können, ist hingegen nicht ihr Hauptaugenmerk. Der Grund für dieses Verhalten kann nicht klar herausgestellt werden. Wichtig ist anzumerken, dass HPs insgesamt ein besseres Verständnis von von Code besitzen und es ihnen dadurch leichter fällt, diesen nachzuvollziehen. Die Unterschiede zwischen den beiden Gruppen sind zwar vorhanden, aber in der Ausprägung nicht signifikant. [19]

Insgesamt wurde festgestellt, dass Studierende mit Erfahrung in der Programmierung bessere Leistungen in den Kursen erzielen als Studierende ohne Programmiererfahrung. So wurden die Teilnehmenden in diese zwei Gruppen unterteilt. Die Gruppe der Studierenden mit Kenntnissen wurde SPE genannt und die Gruppe der Studierenden ohne diese SPNE. Im Verhalten der Gruppen wurde festgestellt, dass SPEs häufiger externe Quellen als SPNEs nutzen, wenn Fragen auftreten. SPNEs hingegen nutzen häufiger die im Studium angebotenen Lehrmaterialien. SPNEs tendierten ebenfalls dazu, Wissen zur Vorbereitung für eine Klausur auswendig zu lernen. SPEs hingegeben erarbeiteten sich häufiger eigene Fragen und helfen Kommilitonen öfter Inhalte zu verstehen als SPNEs. Die festgestellten Unterschieden sind ebenfalls nicht signifikant. [19]

Liao et al. [19] schlussfolgern aus ihren Betrachtungen, dass HPs allgemein ihren Lernprozess konsequenter gestalten und durchführen. Spannend ist, dass beide Gruppen in ähnlichen Ausmaße dazu tendieren, Abgaben zu prokrastinieren, dies hat aber nur bei LPs eine Auswirkung auf das Ergebnis der Abgaben. Außerdem stellt sich heraus, dass HP Studierende insgesamt häufiger Erfahrung in der Programmierung besitzen.

Chinn et al. [10] führten 2010 eine Umfrage mit Studierenden, die einen CS1 Programmiereinsteigerkurs belegten, durch, um ihr Lernverhalten zu analysieren. Hauptaugenmerk lag auf den Lernansätzen in der Freizeit der Studierenden. Der Untersuchung zufolge lernen Studierende 5,3 Stunden in der Woche. Dabei gibt jedoch große individuelle Unterschiede. Als Erfolgsfaktoren wurden die Teilnahme

an den Veranstaltungen und die Vorerfahrung in der Programmierung identifiziert. Eine negative Auswirkung auf die Ergebnisse im Kurs hatten die Hilfe von Personen, die kein Lehrpersonal sind, Arbeit in Gruppen sowie die Nutzung des Internets und von Tutorials.

Zusammenfassend wird festgestellt, dass das Erlernen des Programmierens eine wichtige Kernkompetenz ist, um sich den Wissensbereich der Informatik zu erschließen. Dabei ist es eine komplexe Fähigkeit. Wissenschaftlich gibt es noch keine einheitliche Meinung dazu, welche Kenntnisse das Erlernen erleichtern oder nötiges Grundlagenwissen darstellen.

Chinn et al. [10] und Liao et al. [19] stellten fest, dass es Unterschiede im Lernverhalten von schwachen und starken Programmieranfänger*innen gibt. Ein Faktor, den beide als positiven Einfluss auf den Erfolg identifizieren, ist Erfahrung in der Programmierung. Dabei stellen Liao et al. fest, dass mit bestehender Erfahrung in der Programmierung kein verändertes Lernverhalten einhergeht. In der Tendenz scheinen HP Studierende ihrem Lernprozess mit mehr Sorgfalt zu gestalten. Chinn et al. arbeiten weiterhin heraus, dass eine Teilnahme an Lehrveranstaltungen mit positiven Leistungen korreliert.

3. Stand der Forschung

Ob und welche unterschiedlichen Programmierfehler schwachen und starken Programmieranfänger*innen machen, untersuchten Rodrigo et al. [25]. Diese werden zu Anfang des Kapitels vorgestellt. Anschließend werden verschiedene Arten vorgestellt, Programmierfehler zu kategorisieren. Zunächst wird in Unterabschnitt 3.2.1 eine Übersicht über Studien erstellt, die Programmierfehler anhand von Fehlermeldungen des Compilers einordneten. Abschließend werden verschiedene Arten der manuellen Fehlerkategorisierung in Unterabschnitt 3.2.2, Unterabschnitt 3.2.3, Unterabschnitt 3.2.4 und Unterabschnitt 3.2.5 vorgestellt.

3.1. Schwache und starke Programmieranfänger*innen

Rodrigo et al. [25] führten 2013 eine Studie mit Studienanfänger*innen in Computer Science und Information Systems an der Ateneo de Manila University durch, welche einen CS21A Programmierkurs belegten. Im Kurs wurde die IDE BluJ genutzt. Ziel war der Vergleich von starken, durchschnittlichen und starken Programmierer*innen in Hinsicht auf Jadud's Error Quotient (EQ), Lee's confusion Metrik und Eigenaussagen über den Schwierigkeitsgrad des Lernprozesses aus. Jadud's EQ stellt dar, wie erfolgreich Programmierende mit Syntaxfehlern umgehen. Der Wert kann zwischen null und eins liegen, ein niedriger EQ stellt einen besseren Umgang mit Syntaxfehlern dar. Lee's Confusion Metrik ist ein Machine Learning-Modell, das die durchschnittliche Zeit zwischen Kompilieren und Lösen eines Programmierfehlers bewertet. Auch hier wird ein Wert zwischen null und eins angegeben, wobei eine null keinerlei Verwirrung beim Lösen des Programmierfehlers bedeutet.

Weder im EQ noch in der Confusion Metrik konnten signifikante Unterschiede in den verschiedenen Gruppen der Studierenden nachgewiesen werden, wobei starke und durchschnittliche Programmierer*innen Syntaxfehler effizienter auflösen können als schwache Programmierer*innen. Dabei ist zu beachten, dass Programmierer*innen mit einem niedrigen EQ weniger Zeit des Programmierprozesses damit verbringen, Syntaxfehler zu beheben oder aber sie gar nicht erst machen. Ein

hoher EQ deutet hingegen darauf hin, dass mehrere Fehler gemacht wurden und es daher schwierig für den oder die Programmierer*in ist, den ursprünglichen Fehler zu beheben. Daraus schlussfolgern Rodrigo et al. [25], dass schwache Programmierer*innen weniger Zeit des Programmierprozesses damit verbringen, an der gestellten Aufgabe zu arbeiten, sondern stattdessen damit, die Programmierfehler zu beheben. Durchschnittliche und starke Programmierer*innen haben dadurch auch mehr Zeit, ihren Code zu testen.

Die unterschiedlichen Aussagen zum Schwierigkeitsgrad innerhalb des Programmierkurs hingegen waren signifikant. Während durchschnittliche und starke Programmierende ähnliche Aussagen zu der Schwierigkeit der Inhalte tätigten, gaben schwache Programmierende an, wesentlich mehr Schwierigkeiten mit diesen zu haben. [25]

Im Vergleich der beiden Gruppen der Studierenden konnten Rodrigo et al. [25] keine Unterschiede in den Programmierfehlern feststellen. Die Studierenden machen grundlegende syntaktische und semantische Fehler, wie fehlende Semikolons, falsch geschriebene Variablennamen und fehlende geschweifte Klammern. Wie lang eine Variable besteht und wann auf sie zugegriffen werden kann, ist ein Konzept, das für die Studierenden schwer zu verstehen ist. Grundlegende Prinzipien wie das Deklarieren und Ändern von Variablen bereiten den Studierenden Probleme. Es braucht Zeit, den Unterschied zwischen `print`, `println` und `printf` zu verstehen. Der Zuweisungsoperator `=` und Vergleichsoperator `==` werden vertauscht. Ebenfalls verstehen die Studierenden nicht auf Anhieb, wie Methoden angelegt und aufgerufen werden. Es werden der Punkt oder einzelne Klammern vergessen oder eine Methode mit falschen Parametern aufgerufen. In der Objektorientierung hadern die Studierenden mit dem Erstellen von Konstruktoren und der Initialisierung der Felder von objekteigenen Attributen. Die Sichtbarkeitsangaben `public` und `private` sind für Studierende ebenfalls schwer zu verstehen.

3.2. Kategorisierung von Programmierfehlern

Zum Kategorisieren und Auswerten von Programmierfehlern gibt es verschiedene Ansätze. Häufige Verwendung findet die Kategorisierung nach den von Compilern erzeugten Fehlermeldungen. Wissenschaftliche Arbeiten zu diesem Thema sowie eine Diskussion dieser Art der Fehlerkategorisierung ist unter Unterabschnitt 3.2.1 zu finden. Eine weitere Art der Fehlerkategorisierung ist die Manuelle Fehlerkategorisierung. Es gibt verschiedene Ansichten dazu, wie genau Fehlerarten kategorisiert werden. Diese werden in Unterabschnitt 3.2.2, Unterabschnitt 3.2.3, Unterabschnitt 3.2.4 und Unterabschnitt 3.2.5 vorgestellt und diskutiert.

3.2.1. Compilerbasierte Fehlerkategorisierung

Verschiedene Studien kategorisieren Programmierfehler anhand einer Fehlernachricht, die von einem Compiler erzeugt wird. Einige Studien, die diese Methode benutzen, werden im Folgenden vorgestellt.

Jadud [16] untersuchte 2006, wie sich der Programmcode von 62 Studierenden im ersten Studienjahr zwischen verschiedenen Compiliervorgängen verändert und welche Fehler auftreten. Es wurde festgestellt, dass sich aus diesem keine zuverlässige Vorhersage zu den Leistungen der Studierenden schließen lässt.

Tabano et al. [28] werteten ebenfalls Unterschiede im Programmcode zwischen Compiliervorgängen von Studierenden aus. Es sollten Indikatoren gefunden werden, um Studierende zu identifizieren, die potenzielle Schwierigkeiten im Erlernen der Programmierung zeigen könnten. Die Analyse ergab, dass Studierende, die häufig bestimmte Fehlermeldungen beim Programmieren erhalten, eine Tendenz zu schlechteren Leistungen in der Bewertung aufwiesen. Spezifisch handelte es sich bei den relevanten Fehlermeldungen um `cannot find symbol { variable`, `class`, `interface`, `enum expected` und `cannot find symbol-class errors`.

Jackson et al. [15] entwickelten eine integrierte Entwicklungsumgebung (IDE) speziell für den Lehrbetrieb an der United States Military Academy. Diese setze er ein um Daten zum Verhalten beim Programmieren von Programmieranfänger*innen zu sammeln, die am obligatorischen Einführungskurs in die Programmierung teilnahmen. Diese Datenbasis diente der Identifikation der am häufigsten vorkommenden Programmierfehler. Es zeigte sich, dass die Fehler `cannot resolve symbol`, `; expected` und `illegal start of expression` am häufigsten auftraten.

Denny et al. [11] untersuchten die Syntaxfehler, die von Programmieranfänger*innen gemacht wurden. Ziel war es, die benötigte Zeit zur Behebung der am häufigsten vorkommenden Fehler zu ermitteln. Als die häufigsten Fehler wurden `Cannot resolve identifier`, `Type mismatch` und `Missing ;` identifiziert.

Dy und Rodrigo [12] entwickelten ein Programmierungstool, dass eine Vielzahl von nicht-literalen Programmierfehlern identifiziert. Der Schwerpunkt bei der Entwicklung des Tools lag darauf, Fehlermeldungen zu generieren, die insbesondere für Programmieranfänger*innen leicht verständlich sind. Die Evaluierung der vorrangig auftretenden Fehler erfolgte anhand der Datensätze von Studierenden, die in den Jahren 2007 bis 2008 an einem Programmieranfängerkurs an der Ateneo de Manila University teilgenommen hatten. Als am häufigsten registrierte Fehler wurden `unknown variable`, `; expected` und `[, ], (, ), ',' expected` ermittelt.

Alzahrani et al. [4] werteten automatisch 78 bewertete Programmieraufgaben ei-

nes C++ Programmiereinsteiger*innenkurses aus, um Programmierfehler zu finden, mit denen Studierende erhebliche Probleme haben. Diese waren verschachtelte Schleifen, Verwendung von `else - if`, Zufallsbereiche, Ein- und Ausgabe, Iteration eines Vektors mit der `for`-Schleife, Verwendung einer `for`-Schleife mit der `if`-Anweisung, Indexierung eines Vectors, negierter Schleifenausdruck und boolsche Ausdrücke.

Qian und Lehmann [23] werteten häufige Programmierfehler von chinesischen High School Schüler*innen und das Verhalten dieser beim Auftreten der Fehler aus, um häufig gemachte Programmierfehler zu identifizieren. Dabei wurden 15 häufig auftretende Compilerfehlermeldungen registriert. Die drei am häufigsten auftretenden waren: `; expected`, `program name error` und `cannot find symbol`.

In vier der vorgestellten Studien tauchte das Fehlen der Semikolons als am häufigsten auftretender Fehler auf. Die Fehlermeldung `cannot find symbol` taucht in zwei der vorgestellten Studien auf. Insgesamt ist ein Vergleich der Ergebnisse schwer, da unterschiedliche Compiler unterschiedliche Fehlermeldungen ausgeben können. Für einen detaillierten Vergleich müssten die Fehlermeldungen inhaltlich verglichen werden.

Vor und Nachteile compilerbasierten Fehlerkategorisierung

Eine Fehlerkategorisierung auf der Grundlage von Fehlermeldungen, die von Compilern erzeugt werden, bietet einen großen Vorteil: sie ist automatisiert durchführbar.[21]

Dabei ergeben sich auch Probleme. So kann ein Fehler in verschiedenen Kontexten zu unterschiedlichen Fehlermeldungen führen. Gleichzeitig kann dieselbe Fehlermeldung bei verschiedenem fehlerhaften Code angezeigt werden. Daher können Fehlermeldungen vom Compiler für Programmieranfänger*innen nicht immer eindeutig sein, da es Kenntnisse und Erfahrungen über den Kontext, in dem der Fehler auftritt braucht, um diesen einzuordnen. [21], [22] Fehlermeldungen die vom Compiler erzeugt werden fokussieren sich zudem auf den technischen Aspekt des aufgetretenen Fehlers. Eine manuelle Einordnung von Fehlern ist dahingehend präziser. [22]

Weiterhin sollte nicht außer Acht gelassen werden, dass fehlerhafter Code von Programmieranfänger*innen manchmal mehrere Fehler in einer Codezeile beinhalten kann. Die kann zu Problemen bei einer automatisierten Auswertung führen. Es gibt außerdem eine Vielzahl an Programmierfehlern, die zu keinen Fehlern beim Compilieren führen. Auch ein falsch ausgegebenes Ergebnis ist ein Programmierfehler, aber eines das eine compilerbasierte Fehlerkategorisierung nicht erfassen kann. McCall und Kölling weisen spezifisch darauf hin, dass es es sinnvoll ist, verschiedene

Arten der Fehlersuche für unterschiedliche Arten von Fehlern zu verwenden. [22]

3.2.2. Manuelle Fehlerkategorisierung nach Hristova et al.

Für ihre Arbeit an dem Tool Exprresso definierten Hristova, Misra, Rutter und Mercuri [14] Fehlertypen, die bei Einsteiger*innen beim Erlernen des Programmierens mit Java auftreten. Dazu wurde eine Umfrage unter College Professor*innen, Tutor*innen und Studierenden der Informatik durchgeführt, um aus den Antworten eine Fehlerkategorisierung von Programmierfehlern zu entwickeln. Ihre Umfrage zielte speziell auf die Programmiersprache Java ab. So wurde gefragt, welche Fehler sie für die fünf häufigsten Programmierfehler halten, die von Einsteiger*innen in Java gemacht werden. Weiterhin wurden die Partizipant*innen *innen gebeten, die drei Fehler zu benennen, die ihrer Erfahrung nach am schwierigsten zu finden und zu beheben sind. [14]

Aus der Analyse der Antworten ergaben sich drei Fehlerkategorien: semantische, syntaktische und logische Programmierfehler. Eine detaillierte Übersicht der von Hristova et al. [14] formulierten Fehlerkategorien findet sich unter Anhang A1.

Syntaktische Programmierfehler bezeichnen Fehler wie eine falsch geschriebene Variable, eine fehlerhafte Punktation oder auch eine F fehlerhafte Reihenfolge der einzelnen Wörter oder Terme einer Anweisung im Programmcode. Syntaktische Fehler werden in der Regel leicht von Lernenden gefunden, da sie dazu führen, dass der geschriebene Programmcode nicht ausgeführt werden kann. Dabei ist anzumerken, dass die Fehlermeldung von einem Compiler nicht unbedingt einen Lernenden unterstützt, einen solchen Fehler zu beheben. [14]

Semantische Programmierfehler hingegen beschreiben Fehler, die mit einer hohen Wahrscheinlichkeit auf einer falschen Annahme der Programmierenden basieren, wie bestimmte Anweisungen interpretiert werden. Diese Fehlerart ist wesentlich abstrakter als syntaktische Programmierfehler, da sie die Bedeutung des geschriebenen Codes betreffen. [14]

Die letzte identifizierte Fehlerkategorie von Hristova et al. [14] sind die logischen Programmierfehler. Diese finden ihren Ursprung ebenfalls in einer falschen Annahme des Programmierenden, stehen jedoch in keinen Zusammenhang mit der verwendeten Programmiersprache.

Auftretende Programmierfehler können nicht immer nur exakt einer der Fehlerkategorien zugeordnet werden, und sie können außerdem in mehr als eine Kategorie eingeordnet werden. Ziel war es, diese aus Sicht eines Lernenden einzuordnen, der oder die einen Einsteigerkurs in Java belegt. [14]

Vor- und Nachteile der manuellen Fehlerkategorisierung von Hristova et al.

Allgemein spricht gegen eine manuelle Auswertung von Programmierfehler der erhöhte Aufwand beim Auswerten. Eine Auswertung basierend auf Fehlermeldungen lässt sich einfach automatisieren, was es ermöglicht, eine größere Datenmenge auszuwerten. [21]

Die von Hristova et al. [14] vorgestellte Fehlerkategorisierung wurde in einer Forschungsarbeit formuliert, deren Ziel es war, ein Tool zu entwickeln, welches Lernende unterstützen soll, die gemachten Fehler zu identifizieren und zu beheben. Aufgrund dessen wurde entschieden, einige Programmierfehler nicht in die Kategorisierung mit einzubeziehen. Zum einen wurden Fehler nicht einbezogen, deren Fehlermeldung vom verwendeten Compiler bereits für die Lernenden als verständlich identifiziert wurden. Weiterhin wurden Fehler, welche die Autor*innen nicht verstanden oder in ihrem Projekt nicht implementieren konnten, exkludiert. Außerdem wurden einige Laufzeitfehler oder Fehler, die eine Compilierbarkeit verhindern, nicht in ihrer Fehlerkategorisierung berücksichtigt. Das Ziel der vorliegenden Arbeit stellt nicht die Entwicklung eines Tools dar, sondern die detaillierten Analyse von Programmierfehlern zur Verbesserung der Lehre. Die von Hristova et al. nicht beachteten Fehler könnten somit relevant sein. Weiterhin stellt die von Altadmri und Brown dargestellte Fehlerkategorisierung eine Weiterentwicklung der von Hristova et al. formulierten Fehlerkategorien dar. Aus diesen Gründen wird die Fehlerkategorisierung von Hristova et al. in der vorliegenden Arbeit nicht verwendet. [14, 7, 8, 6]

3.2.3. Manuelle Fehlerkategorisierung nach Altadmri und Brown

Altadmri und Brown nutzten die Fehlerkategorisierung von Hristova et al. [14] als Grundlage für verschiedene wissenschaftliche Betrachtungen von Programmierungsfehlern von Programmieranfänger*innen und entwickelten diese weiter.[7, 8, 6] Zunächst überprüften sie die von Hristova et al. genannten am häufigsten auftretenden Prorgammierfehler. Dazu analysierten sie die Daten des Blackbox Datensets, welches zu der Zeit Daten von über 100 000 Studierenden umfasste. [7] Bei dem Datenset handelt es sich um Daten von Nutzer*innen des Programmiertools BlueJ. Nutzer*innen werden in der IDE gebeten, ihren Prozess des Programmierens zu dem Datensatz hinzuzufügen. [21]

2015 führten Altadmri und Brown eine Untersuchung von Programmierfehlern von Programmieranfänger*innen durch, in Hinsicht auf die Auftretenshäufigkeit,

die Zeit, die benötigt wurde, um die Fehler zu beheben, und wie verbreitet sie unter Nutzer*innen waren. Sie bezogen die Entwicklung der Lernenden über ein Jahr hinweg mit in die Auswertung ein. Als Grundlage für die Untersuchung dienten die über das Blackbox Datenset gesammelten Daten von mittlerweile 250 000 Studierenden. Mit dem auf 10 Millionen Programmierungssitzungen angewachsenen Blackbox-Datenset haben Altadmri und Brown 2017 die Auftretenshäufigkeit von Programmierfehlern und die Zeit, die es brauchte, diese zu beheben, erforscht. [8], [6]

Die Fehlerkategorien von Hristova et al. [14] wurden von Altadmri und Brown bereits 2014 von 20 auf 18 Fehlerkategorien reduziert. Bei der Analyse mit dem beschriebenen Datenset stellte sich heraus, dass der von Hristova et al. genannte semantische Fehler „Aufruf einer statischen Klassenfunktion auf einem Klassenobjekt“ keine Abbildung in dem Datensatz findet. Dafür wurde die Liste um den Fehler „Aufruf einer Instanzmethode auf einer Klasse“ ergänzt. Ein Fehler aus der Kategorie der semantischen Programmierfehler und ein weiterer aus der Liste der logischen Programmierfehler wurden von Altadmri und Brown nicht übernommen.[7]

Damit enthält die Fehlerkategorisierung von Altadmri und Brown [7] 20 Arten von Fehlern. Entgegen Hristova et al. [14] ordneten Altadmri und Brown die Fehler nicht in in die Oberkategorien semantische, syntaktische und logische Programmierfehler ein, sondern sortieren die Fehler von A bis R.

- C Ungleiche Anzahl von geschweiften oder eckigen Klammern und Anführungszeichen oder eine austauschbare Verwendung dieser Symbole.
- I Methodenaufruf mit falschen Argumenten (z.B. falsche Typen).
- O Der Kontrollfluss kann das Ende einer Nicht-void-Methode erreichen, ohne etwas zurückzugeben.
- N Eine Methode mit einem Nicht-void-Rückgabetyp wird aufgerufen, und ihr Rückgabewert wird ignoriert/verworfen.
- A Verwechseln des Zuweisungsoperators `=` mit dem Vergleichsoperator `==`.
- B Verwendung von `==` anstelle von `.equals` zum Vergleichen von Strings.
- M Versuch, eine nicht-statische Methode so aufzurufen, als wäre sie statisch.
- R Eine Klasse behauptet, ein Interface zu implementieren, implementiert jedoch nicht alle erforderlichen Methoden.
- P Beim Aufruf einer Methode werden die Typen der Parameter genutzt.
- E Semikolon nach oder vor der `if`-Anweisung, nach der `for`- oder `while`-Schleife oder vor der jeweiligen `for`- oder `while`-Schleife.
- K Semikolon am Ende eines Methodenkopfes.
- Q Typ, der von der Methode zurückgegeben wird und Typ der Variable, welcher der Wert zugewiesen wird, sind inkompatibel

D Verwechslung von „Short-Circuit“-Evaluatoren ((`&&` und | |) mit herkömmlichen logischen Operatoren (`&` und |).
J Fehlende Klammern nach einem Methodenaufruf.
L Falsches Verständnis von größer als oder gleich/kleiner als oder gleich, d.h. Verwendung von `=>` oder `=<` anstelle von `>=` und `<=`.
F Falsche Trennzeichen in `for`-Schleifen (Verwendung von Kommas statt Semikolons).
H Verwendung von Schlüsselwörtern als Methoden- oder Variablennamen.
G Für die Bedingung einer `if`-Anweisung werden geschweifte statt einfache Klammern verwendet.

Die genaue Auftretenshäufikeit von Programmierfehlern in den Arbeiten von Altadmri und Brown finden sich unter A2 und A3. Der am häufigsten aufgetretene Fehler in der Auswertung von 2014 war, dass Klammern vergessen oder in einer falschen Kombination genutzt wurden. Diesem folgte in der Häufigkeit, dass Methodenaufrufe mit falschen Argumenten aufgerufen wurden und in einer nicht `void`-Funktion das `return` Statement nicht immer erreichbar war. In der Erhebung von Altadmri und Brown 2015 fallen die meisten registrierten Fehler ebenfalls in diese Kategorien. In der Auflistung von 2014 ist die am vierthäufigste auftretende Fehlerart, dass eine Methode mit einem Nicht-void-Rückgabetyp aufgerufen wird, ihr Rückgabewert aber nicht verwendet wird. In weniger betrachteten Datensätzen fand sich eine Inkompatibilität eines Rückgabewerts einer Methode und des Wertes, dem dieser zugewiesen werden soll. Die weitere Reihenfolge der Auftretenshäufigkeit blieb bei beiden Studien gleich.

Vor- und Nachteile der Fehlerkategorisierung nach Altadmri und Brown

Die Arbeiten zu Fehlerkategorien von Altadmri und Brown basieren auf dem Blackbox Datensatz. Diese Daten wurde über die Java IDE BlueJ gesammelt und für Forschende veröffentlicht. BlueJ ist ein Tool, das für den Einsatz in der Lehre, spezifisch für das erste Jahr, entwickelt wurde. Der Aufbau reflektiert einen objektorientierten Ansatz der Entwicklung in Java. Es verwendet eine grafische Oberfläche, die sich von den meisten herkömmlichen IDEs unterscheidet. Die Entwicklung eines Programms startet mit dem Konzipieren eines UMLs, in dem Objekte und Klassen angelegt werden. Funktionen werden in der Interaktion mit den grafisch dargestellten Objekten hinzugefügt und können so ebenfalls getestet werden. Dies macht einen herkömmlichen Testingprozess zumindest am Anfang überflüssig. Für eine organisierte Testingumgebung wurde in BlueJ zusätzlich JUnit integriert. [17]

Es ist anzunehmen, dass diese spezialisierte Programmierumgebung die Arten von

Fehlern, die von Lernenden gemacht werden, beeinflusst. Durch die in einer grafischen Benutzeroberfläche angebotenen Möglichkeiten des Testings erübrigt sich die Nutzung einer main Funktion. Ein automatisches Anlegen der Klassen beugt Fehlern bei dieser vor.

Das Tool kann eine gewisse Distanz zwischen Programmierenden und dem geschriebenen Code erzeugen und so beim Lernprozess hinderlich sein, indem eine Abhängigkeit zum Tool entsteht. Grundlegende Fähigkeiten wie das Anlegen von Funktionen in einer herkömmlichen Programmierumgebung müssen später erlernt werden. [14]

Der Fokus von BlueJ auf der objektorientierten Programmierung kann dazu führen, dass Lernende andere Bereiche wie Datenstrukturen und Algorithmen weniger stark verinnerlichen. Dies kann auch ein Resultat der erfolgten Lehre mit dem Tool sein. Der zu betrachtende Kurs fokussiert sich auf das Erlernen von Datenstrukturen.[17]

Es ist möglich, dass die verwendete Fehlerkategorie aus den beschriebenen Gründen nicht umfänglich für die Anwendung im Kurs geeignet ist und bei der Arbeit erweitert wird.

3.2.4. Manuelle Fehlerkategorisierung nach McCall und Kölling

Eine Einteilung in semantische, syntaktische und logische Programmierfehler haben ebenfalls McCall und Kölling [21] vorgenommen. Sie betrachteten spezifisch Java-Code.

Um eine präzise Definition des zu untersuchenden Feldes zu gewährleisten haben sie folgende Terminologie definiert: Ein Fehler ist die Grundlage des Problems im Programmcode, welcher schlussendlich dazu führt, dass das Programm nicht kompiliert oder andere falsche Resultate liefert. Eine Fehlermeldung ist eine von Programmier*innen geschriebene Nachricht, welche für Menschen lesbar ist und bei Auftritt des Fehlers angezeigt wird. Eine Fehlannahme eines Programmierenden ist die Ursache hinter dem fehlerhaft geschriebenen Code. [21]

Im Unterschied zu Hristova et al. [14] haben McCall und Kölling [21] ihre Fehlerkategorien aus zwei Datensätzen, die über die BlueJ IDE gesammelt wurden, entwickelt. Grundlage der von McCall und Kölling 2014 vorgeschlagenen Fehlerkategorien ist eine Analyse von Programmcode von Anfänger*innen. Kategorien wurden formuliert, während fehlerhafte Stellen im Code analysiert wurden. Fehlerkategorien konnten hierarchisch geordnet werden. Allgemeine Fehler bilden Überkategorien und werden genutzt, wenn ein Fehler nicht weiter definiert werden kann. Für Fehler, die präziser zu bestimmen sind, wurden Unterkategorien beschrieben. Der Ansatz war, einen Fehler so präzise wie möglich einzuordnen. Dabei

ist es möglich, dass eine Fehlermeldung in mehreren Fehlern begründet liegt.

2019 setzten McCall und Kölling [22] gefundene Fehlerarten in Kontext zur Häufigkeit des Auftretens und der Zeit, die die Programmierenden benötigten, diese zu beheben. Als Datengrundlage diente ein randomisierter Auszug aus dem Blackbox Datensatz. Bei dieser Arbeit wurden die 2014 formulierten Fehlerkategorien weiterentwickelt. Dabei wurde die hierarchische Struktur der Fehlerkategorien beibehalten. Ergab sich eine thematische Überschneidung, wurde eine Elternkategorie für die Fehlerarten gebildet. In dem Datenbasierten Ansatz kamen McCall und Kölling zu dem Schluss, dass sich Fehler in die drei Oberkategorien syntaktisch, semantisch und logisch unterteilen lassen.

In dem Prozess wurden 80 Fehlerkategorien definiert. Zwischen den syntaktischen Programmierfehlern und Fehlermeldungen von Compilern konnte kein großer Zusammenhang festgestellt werden. Semantische Programmierfehler konnten eher bestimmten Fehlermeldungen von Compilern zugeordnet werden. Gegen eine Einteilung in semantische, syntaktische und logische Fehler spricht, dass eine feinere Aufteilung selbst bei Oberkategorien sinnvoller ist, um Programmierfehler von Programmieranfänger*innen einzuordnen. [21]

Um die Übersichtlichkeit zu gewährleisten, wurde im Rahmen dieser Arbeit die Fehlerkategorieren nummeriert. McCall und Kölling beschreiben 2019 folgende Fehlerkategorien: [22]

- 1.0 Variablen: Falsche Nutzung von Variablen.
  - 1.1 Bei einer Objektinitialisierung wird ein bereits vorhandener Variablenname genutzt, anstatt auf eine bestehende Variable zuzugreifen.
  - 1.2 Aus statischem Kontext wird auf eine nicht statische Variable zugegriffen.
  - 1.3 Variable einer anderen Klasse wird verwendet.
  - 1.4 Variable wurde nicht deklariert.
  - 1.5 Es wurde die falsche Variable verwendet.
- 2.0 Variable: Falsche Variablendeklaration.
  - 2.1 Lokale Variable wird mit nicht zulässigem Zugriffsmodifikator deklariert.
  - 2.2 Variablendeklaration mit Name und Typ in der falschen Reihenfolge.
  - 2.3 Variablendeklaration, bei der kein Name angegeben wurde.
  - 2.4 Variablendeklaration mit Leerzeichen im Namen der Variable.
  - 2.5 Variablendeklaration benötigt einen einzigartigen Namen.
- 3.0 Methoden: Nicht zulässiger Methodenaufruf.
  - 3.1 Methodenaufruf wird mit einem Variablennamen durchgeführt.
  - 3.2 Methodenaufruf mit dem falschen Datentyp.

- 3.3 Methodenaufruf: Parameter dient gleichzeitig als Deklaration.
- 3.4 Methodenaufruf: mit falscher Parameteranzahl.
- 3.5 Methodenaufruf: falsche Datentypen der übergebenen Parameter.
- 3.6 Methodenaufruf: Typen der übergebenen Parameter werden angegeben.
- 3.7 Methodenname wird bei Methodenaufruf falsch geschrieben.
- 3.8 Beim Methodenaufruf wurden keine runden Klammern genutzt(`()`).
- 3.9 Eine nicht statische Methode wird in einem nicht statischen Kontext aufgerufen.

4.0 Methoden: Falsche Methodendeklaration.

- 4.1 Methodendeklaration: Komma fehlt zwischen den Parameterangaben.
- 4.2 Methodendeklaration: Methodenkörper fehlt.
- 4.3 Methodendeklaration: Rückgabewert fehlt.
- 4.4 Methodendeklaration: Parameter ohne spezifische Typzuweisung.
- 4.5 Methodendeklaration: Rückgabewert sollte void sein.
- 4.6 Methodendeklaration: Semikolon am Ende des Methodenkopfes.
- 4.7 Methodendeklaration: Semikolon anstelle eines Kommas.
- 4.8 Methodendeklaration: Falsche Anzahl von Parametern verwendet.
- 4.9 Exaktes Duplikat einer Methode wurde verwendet.
- 4.10 Geschweifte Klammer nach dem Methodenkopf nicht verwendet.
- 4.11 Falscher Rückgabewert wurde verwendet.

5.0 Konstruktor: Falscher Konstruktorenaufruf.

- 5.1 Konstruktor wurde versucht, mit Variablennamen aufzurufen.
- 5.2 Aufruf eines nicht existierenden Kopierkonstruktors.
- 5.3 Konstruktorenaufruf ohne die Verwendung von `new`.
- 5.4 Konstruktorenaufruf mit einer falschen Anzahl von Parametern.
- 5.5 Konstruktorenaufruf mit Parametern, die falschen Datentyp haben.
- 5.6 Fehlende abschließende runde Klammer nach dem Konstruktorenaufruf.
- 5.7 Keine runden Klammern beim Aufruf eines Konstruktors genutzt.
- 5.8 Es gibt kein Leerzeichen nach `new`.
- 5.9 Bei der Objektinitialisierung wird ein Variablenname verwendet.

6.0 Konsturktor: Falsche Deklaration des Konstruktors.

- 6.1 Der Aufruf `super` ist nicht der erste Term im Konstruktor.
- 6.2 Ein Konstruktor bekommt den Rückgabewert `void`.

7.0 Falsche Verwendung einer Klasse oder eines Datentyps.

- 7.1 Klasse einer Bibliothek wurde verwendet ohne sie zu importieren.
- 7.2 Klasse wurde nicht definiert.
- 7.3 Klasse oder Datentyp wurde falsch geschrieben.
- 7.4 Eine Variable wurde in einem Kontext genutzt, in der ein Rückgabewert nötig gewesen wäre.

8.0 Semantische Fehler.

- 8.1 Zuweisung zu einem Element einer Sammlung.
- 8.2 Klasse implementiert eine erforderliche Funktion nicht.
- 8.3 Klasse sollte so deklariert werden, dass sie ein Interface implementiert.
- 8.4 Es fehlt das `return` am Ende der Funktion.
- 8.5 Typenfehler:
  - 8.5.1 Array wurde statt eines Elementes des Arrays verwendet.
  - 8.5.2 Sammlung wurde anstatt eines Elements genutzt.
  - 8.5.3 Falsches Casting von Typen.
  - 8.5.4 Typinkompatibilität bei der Zuweisung.
  - 8.5.5 Typeninkompatibilität, indem Arithmetik auf ein String Datenelement angewendet wird oder die `.length` Funktion angewendet wird, wenn diese Funktion bei Datentyp oder Klasse nicht vorhanden ist.
  - 8.5.6 Wert aus einer untypisierten Sammlung muss umgewandelt werden.
- 8.6 Nicht aufgelöste Exception.
- 8.7 Variablenzugriff als Anweisung verwendet.

- 9.0 Einfache syntaktische Fehler.
  - 9.1 `=` wurde anstatt eines Vergleichs `==` verwendet.
  - 9.2 `:` wurde anstatt eines Semikolons `;` verwendet.
  - 9.3 Abschlusszeichen `;` wurde vergessen.
  - 9.4 Kommentar wurde falsch deklariert.
  - 9.5 Es gibt eine schließende runde Klammer zu viel.
  - 9.6 Es gibt eine öffnende geschweifte Klammer zu viel.
  - 9.7 Öffnende runde Klammer zu viel zwischen `if-` und `else` - Anweisung.
  - 9.8 Eine extra geschweifte schließende Klammer.
  - 9.9 Eine extra oder falsch gesetzte einfache schließende runde Klammer.
  - 9.10 Ein Schlüsselwort wurde falsch geschrieben.
  - 9.11 Nicht zusammenpassende Klammern um einen Ausdruck.
  - 9.12 Fehlender Punkt `.` zwischen Namen und Mitglied einer zugreifenden Klasse/Paket.
  - 9.13 Fehlende schließende geschweifte Klammer.
    - 9.13.1 Fehlende schließende geschweifte Klammer nach Körper einer Kontrollanweisung.
    - 9.13.2 Fehlende schließende geschweifte Klammer am Ende einer Klasse.
    - 9.13.3 Fehlende schließende geschweifte Klammer am Ende einer Methode.
  - 9.14 Operator fehlt zwischen zwei Ausdrücken.
- 10.0 Code ist außerhalb einer Methode oder eines anderweitigen Blocks.
- 11.0 Unkategorisiert. [22]

Das Formulieren der Fehlerkategorien erfolgte 2014 in zwei Etappen. In der ersten Etappe wurde der Programmcode von etwa 240 Studierenden ausgewertetet,

die zwei Java Einstiegsprogrammierkurse belegten. Einer der Kurse wurde an der University of Kent in Großbritannien abgehalten, der andere an der University of Puget Sound in Washington in den USA. Im zweiten Schritt wurden, wie bereits erwähnt, Daten des Blackbox Datensets verwendet. Aus dem Datenset wurden 23 Programmiersessions im Zeitraum vom 11.06.2013 bis 01.01.2014 zufällig ausgewählt. Zusätzlich zum Erstellen der Fehlerkategorisierung wurden beide Datensets ausgewertet. [21]

Tabelle 3.1.: Auftretenshäufigkeit der Programmierfehler in der Studie von McCall und Kölling 2014 [21]

Category	**Frequency**
Variable not declared	11,1 %
; missing	10,3 %
Variable name written incorrectly	8,4 %
Invalid Syntax	7,9 %
Method name written incorrectly	4,9 %
Missing parentheses for constructor call	4,1 %
Unhandled exception	3,0 %
Class name written incorrectly	2,7 %
Method call: parameter type mismatch	2,4 %
Type mismatch in assignment	2,4 %

Die am häufigsten auftretende Fehlerkategorie bei der Erhebung von McCall und Kölling 2014 ist, dass eine Variable nicht deklariert wurde. Interessant ist hierbei, dass dies nicht mit den Ergebnissen von Altadmri und Brown übereinstimmt. Im Gegenteil taucht diese Fehlerkategorie in der Arbeit von Brown und Altadmri nicht auf. Nur zwei der von McCall und Kölling formulierten Fehlerkategorien haben überhaupt eine Entsprechung in der Fehlerkategorisierung von Altadmri und Brown. Diese sind „Missing parentheses for constructor call“ welche lose in die Fehlerkategorie C von Altadmri und Brown passt und die Kategorie „Metod call: parameter type mismatches“, was der Fehlerkategorie I entspricht.

Tabelle 3.2.: Auftretenshäufigkeit der Programmierfehler in der Studie von Kölling und McCall 2019 [22]

Error Category	**Frequency**
Variable not declared	8,4 %
Variable name written incorrectly	7,4 %
; missing	7,3 %
Simple syntactical error	6,5 %
Semantic error	4,8 %
Variable: Incorrect variable declaration	4,7 %
Variable: Incorrect attempt to use variable	4,6 %
Method name written incorrectly	4,6 %
Method: Incorrect method declaration	4,5 %
Class or type name written incorrectly	4,4 %

Um den Schweregrad verschiedener Fehler zu bestimmen, nutzen McCall und Kölling eine randomisierte Auswahl aus dem Blackbox Datensatz. Auftretende Fehler wurden wieder in die von ihnen entwickelten Fehlerkategorien eingeordnet. Eine Übersicht der häufigsten Fehler findet sich in Tabelle 3.2.

Bei der erneuten Kategorisierung von Fehlern änderte sich die Auftretenshäufigkeit einigen Kategorien. So ist die Kategorie: „: missing“ an der dritten statt wie 2014 an zweiter Stelle. Folgende Kategorien gehörten 2014 nicht zu den zehn am häufigsten aufgetretenen: „Simple syntactical error“, „Semantic error“, „Variable: Incorrect attempt to use variable“, „Variable: Incorrect attempt to use variable“, „Method: Incorrect method declaration“ und „Class or type name written incorrectly“. Sechs der zehn Fehlerkategorien, in die die häufigsten Fehler einsortiert waren gehörten somit 2014 nicht zu den am häufigsten auftretenden Fehlerkategorien. Die Fehlerkategorie „Method name written incorrectly“ war 2014 die am fünfthäufigsten auftretende Kategorie und ist 2019 die siebthäufigste Kategorie.

Vor- und Nachteile der manuellen Fehlerkategorisierung nach McCall und Kölling

Wie die Arbeit von Altadmri und Brown [6], basiert die Fehlerkategorisierung nach McCall und Kölling [22] auf dem Blackbox Datensatz. Demnach treffen hier dieselben Aussagen zu, die bereits bei Unterabschnitt 3.2.3 getroffen wurden.

Auffällig bei der Fehlerkategorisierung von McCall und Kölling ist der Detailgrad der Fehlerkategorien. Durch die Strukturierung der Fehlerkategorien ist es ebenfalls

einfach, neue Fehlerkategorien zu den übergeordneten Kategorien hinzuzufügen.

3.2.5. Weitere manuelle Fehlerkategorisierungen

Albrecht und Grabowski [3] untersuchten im Jahr 2020 Daten eines Einsteigerkurses in der Programmiersprache C, um häufige Fehler der Teilnehmenden zu identifizieren. Als Grundlage für die Einordnung wurde die Kategorisierung von Programmierwissen von Bayman und Mayer genutzt. Diese definierten drei Bereiche von Wissen, die für das Programmieren nötig sind. Diese sind syntaktisches Wissen, konzeptionelles Wissen und Wissen um Strategien.

1.0 Syntaktisches Wissen bezeichnet Kenntnisse des theoretischen Aufbaus der Programmiersprache.
2.0 Konzeptionelles Wissen beinhaltet semantische und logische Programmierfehler und das Wissen darum, wie Konstrukt in der Programmierung funktioniert.
3.0 Wissen, um Strategien beinhaltet Kenntnisse, wie Programmierkonstrukte angewendet werden um ein bestimmtes Ziel zu erreichen. [3]

Bei der Analyse der Daten wurden weitere Fehlerkategorien formuliert.[3]

4.0 Schludrigkeit bezieht sich auf Fehler, die von Programmierenden nicht intentional getroffen wurden, wie zum Beispiel ein vergessenes Semikolon.
5.0 Missinterpretation beinhaltet Fehler, die daraus resultieren, dass der Aufgabentext falsch verstanden wurde.
6.0 Domain beinhaltet Fehler, die gemacht werden, da das Wissen des Programmierenden für die gestellte Aufgabe unzureichend ist. Dies kann beispielsweise eine falsche Berechnung einer Variable durch fehlendes mathematisches Wissen sein. [3]

Ettles et al. [13] fokussierten sich darauf, häufig auftretende logische Fehler von Programmieranfänger*innen zu kategorisieren. Dazu werteten sie 15.000 Codefragmente in der Programmiersprache C, die mit dem Tool CodeWrite gesammelt wurden, von Studierenden aus. Dieses Tool präsentierte den Studierenden kurze Programmieraufgaben, die gelöst werden sollten.

Logische Programmierfehler werden von Ettles et al. [13] als Fehler definiert, bei denen das Programm ausführbar, also compilierbar ist, aber nicht das vom Programmierenden gewünschte Verhalten aufweist. Logische Fehler können auch als semantische Programmierfehler bezeichnet werden.

Diese sind für Studierende die am härtesten zu lösende Fehlerart, da es schwer ist, sie zu finden. Gerade Programmieranfänger*innen haben defizitäres Wissen

um Debuggingtechniken, welche zum Auffinden von logischen Fehlern angewendet werden müssen. Um logische Programmierfehler zu finden, muss meist viel Programmcode betrachtet werden, wobei nicht klar ist, wo genau der Fehler sich befindet. Der Programmcode der Studierenden wurde manuell ausgewertet. Dabei wurden folgende Fehlerkategorisierungen erarbeitet: [13]

1. **Algorithmische Fehler:** Diese Art Fehler basiert nicht auf fehlerhaftem Wissen der Programmiersprache, sondern tritt auf, wenn der Algorithmus von dem Programmierenden für etwas angewendet wird, wofür er nicht geeignet ist.
2. **Fehlinterpretation:** Diese Fehlerart bezeichnet falsche Annahmen oder lückenhaftes Verständnis der Aufgabenstellung und resultiert zum Beispiel in falschen Rückgabewerten. Auch dieser Fehler basiert nicht auf lückenhaftem Programmierwissen.
3. **Fehlannahme:** Diese Fehlerkategorie resultiert aus Unwissen in der Programmierung oder der Anwendung der Programmiersprache. Sie findet sich zum Beispiel in einer fehlerhaften Indexierung eines Arrays oder dem nicht Initialisieren von Variablen. [13]

Vor- und Nachteile der weiteren manuellen Fehlerkategorien

Albrecht und Grabowski [3] wählten die Fehlereinteilung nach Bayman und Mayer, da eine Einteilung nach semantischen, syntaktischen und logischen Programmierfehlern nicht die Misskonzepte hinter den Programmierfehlern reflektiert. In der Arbeit von Mayer und Bayman [5] wurden diese Kategorien entwickelt, um Wissen abzubilden, das beim Programmieren wichtig ist und nicht, um Programmierfehler zu kategorisieren. Das schafft eine schwere Grundlage dafür Programmierfehler exakt einzuordnen, da ohne ein Interview nicht immmer unterscheidbar ist, aus welchen Gründen Programmierfehler von den Programmierenden gemacht wurden. Als anschauliches Beispiel kann eine von einer*m Programmieranfänger*in geschriebene Funktion genommen werden, die den Umfang eines Kreises errechnen soll. Im hypothetischen Beispiel wird zwar kein Compilerfehler erzeugt, aber ein falsches Rechenergebnis zurückgegeben, da die Formel zur Berechnung des Umfangs falsch implementiert wurde. Ursprung des Fehlers kann fehlendes Wissen zur Implementation der Berechnung sein, was der Kategorie des strategischen Wissens entsprechen würde. Der Fehler könnte jedoch auch durch Schludrigkeit beim Schreiben der Methode verursacht werden. Ebenfalls könnte Domainwissen zum Berechnen des Umfangs eines Kreises fehlen. Ohne Rücksprache mit dem oder der betreffenden Studierenden kann keine klare Aussage darüber getroffen werden, aus welchem Grund der Programmierfehler gemacht wurde.

Bei der Arbeit von Ettles et al. [13] fällt ein Punkt gleich zu Beginn der Veröffentlichung auf. In der Arbeit wurde sich darauf fokussiert, logische Programmierfehler zu kategorisieren. Die Definition von logischen Programmierfehlern ist mit den Definitionen in anderen wissenschaftlichen Arbeiten nicht übereinstimmend.

Andere bereits vorgestellte Fehlerkategorisierungen trennen logische und semantische Programmierfehler. Hristova et al. [14] unterscheiden syntaktische, semantische und logische Programmierfehler als drei Arten von Programmierfehlern. McCall und Kölling [21] unterscheiden ebenfalls zwischen semantischen und logischen Programmierfehlern und nutzen die Begriffe nicht für dieselbe Art von Fehler. In der Arbeit von Ettles et al. [13] fehlt somit die Trennung von logischen und semantischen Programmierfehlern.

4. Methodik

Im Kapitel 4 erfolgt zunächst eine Übersicht der verwendeten Begriffe im Abschnitt 4.1. Anschließend werden die Forschungsfragen der Arbeit vorgestellt und Annahmen dazu formuliert, dargestellt im Abschnitt 4.2. Eine kurze Übersicht über den Kurs bietet Abschnitt 4.3, gefolgt von Informationen zu den Teilnehmenden, die die ASL-Abgaben eingereicht haben, im Abschnitt Abschnitt 4.4. Im Anschluss daran werden die zu bearbeitenden Aufgaben in Abschnitt 4.5 erörtert. Nach einer Diskussion der unter Abschnitt 3.2 eingeführten Fehlerkategorisierung und der Entscheidung für eine bestimmte Kategorisierung, schließt Abschnitt 4.7 mit einer Beschreibung der Auswertungsmethodik und der Vorstellung von Beispielen zur transparenten Einordnung ab.

4.1. Terminologie

Dieses Kapitel führt zentrale Begriffe und Klassifikationen ein, die für das Verständnis der vorliegenden Arbeit grundlegend sind. Zunächst werden die Begriffe „Programmierfehler“ und „Fehlerkategorie“ präzisiert und im Kontext dieser Arbeit definiert, um eine einheitliche Grundlage für die folgenden Ausführungen zu schaffen. Weiterhin wird eine Unterscheidung zwischen „komplexen“ und „einfachen“ Programmierkonzepten vorgenommen, welche die Basis für die anschließende Analyse bilden. Anschließend erfolgt eine Kategorisierung der Studierenden in „schwache“ und „starke“ Programmieranfänger*innen, basierend auf definierten Kriterien, die eine differenzierte Betrachtung ihrer Leistungen ermöglichen. Diese terminologischen Abgrenzungen sind essenziell, um die Auswertungsmethodik und die Interpretation der Ergebnisse konsistent zu gestalten.

Ein Programmierfehler bezeichnet in der vorliegenden Arbeit ein fehlerhaftes Codefragment oder einen fehlerhaften Abschnitt im Code. Signifikant ist, dass sich das geschriebene Programm nicht wie vom Programmierenden erwartet verhält. Ein Verhalten gilt als nicht erwartungskonform, wenn der Code die gestellte Aufgabe nicht erfüllt oder nicht compilierbar ist. Ein Programmierfehler kann von einem falsch gesetzten Zeichen über eine falsch implementierte Datenstruktur bis hin zu einer endlosen `for`- oder `while`-Anweisung reichen. Ein festgestellter Programmier-

fehler gibt noch keinen Hinweis über die Art des Fehlers. Es ist es möglich, dass ein Codefragment oder ein Codeabschnitt mehrere Fehler enthält, die zu einem nicht erwartungskonformen Programmablauf führen.

Wichtig ist zu beachten, dass ein Fehler im Programmcode nicht immer eine Fehlermeldung des Compilers hervorruft. Fehler können ebenfalls dadurch auffallen, dass sie unerwünschtes Verhalten des geschriebenen Programms verursachen. [22]

Eine Fehlerkategorisierung beschreibt ein Konstrukt, das ähnliche Programmierfehler zusammenfasst oder näher beschreibt. Eine Fehlerkategorie kann ein Spektrum ähnlicher Fehler umfassen oder sich auf eine spezifische Fehlerart konzentrieren. Die Fehlerkategorisierung gibt vor, welche Arten von Fehlern zusammengefasst werden.

Es wird in Kapitel 5 zwischen komplexen und einfachen Programmierkonzepten unterschieden. Die Definition dieser basiert auf der Arbeit von Lahtinen et al. [18]. Diese definieren komplexe Programmierkonzepte darüber, dass sie ein Verständnis von zusammenhängenden Codeblöcken erfordern. Dementsprechend sind einfache Programmierkonzepte jene, bei denen das Verständnis von wenigen Zeilen an Programmiercode ausreicht.

Ein Beispiel zur Veranschaulichung sind syntaktische Programmierfehler. Diese treten meist in einer Zeile von Code auf. Semantische und logische Fehler hingegen liegen im Ursprung oft an falschen Annahmen der Programmierenden und zeigen sich erst im Zusammenspiel von vielen Zeilen von Programmiercode.

In der Bezeichnung und Einordnung der Studierenden in die Gruppen der schwachen und starken Programmieranfänger*innen wird sich an der Terminologie von Liao et al. [19] orientiert. Schwache Programmieranfänger*innen werden im Folgenden als LPS – Low Performing Students – und starke Programmieranfänger*innen als HPS – High Performing Students – abgekürzt. Die Einordnung in die Gruppen erfolgt über die erreichte Note im Kurs Datenstrukturen.

4.2. Forschungsfragen und Annahmen

Im Vorfeld der Auswertung der ASLs wurden zwei Forschungsfragen aufgestellt. Anschließend werden Annahmen präsentiert, welche Ergebnisse zu erwarten sind.

> **RQ1:** Welche Unterschiede gibt es in der Anzahl und der Art der gemachten Programmierfehler zwischen LPS und HPS?
>
> **Annahme 1:** LPS machen mehr unterschiedliche Fehler als HPS.

Annahme 2: LPS machen im Umfang mehr Fehler als HPS.

Annahme 3: In der Entwicklung der Fehler in Umfang und Art gibt es über des Semesters hinweg Unterschiede zwischen HPS und LPS. LPS werden am Ende des Semesters noch mit grundlegenden Programmierkonzepten Probleme haben.

RQ2: Welche Unterschiede gibt es beim Umfang der logischen Fehler im Umgang mit neu eingeführten Programmierkonzepten zwischen LPS und HPS?

Annahme4: LPS machen im Umfang mehr Fehler in Bezug auf neue Datenstrukturen.

Grundlegende Programmierkonzepte umfassen grammatikalische oder syntaktische Konstrukte in der Programmierung. Ein Beispiel ist das Anlegen und Verwenden von Variablen und Methoden.

4.3. Lehrveranstaltung Datenstrukturen

Der Kurs „Datenstrukturen“ ist ein Lehrangebot der Professur Softwaretechnik, welches sich mit der Lehre von Datenstrukturen und Algorithmen in der Programmierung sowie der Umsetzung in Programmcode befasst und jeweils im Sommersemester angeboten wird. Betrachtet wird hier der Kurs aus dem Sommersemester 2023. Alle teilnehmenden Studierenden werden angehalten, sich in dem Kurs auf der Online-Plattform für Akademisches Lehren und Lernen OPAL anzumelden. Dort können sie sich für die angebotenen Übungen einschreiben und erhalten die Aufgabenstellungen für die zu erbringenden Leistungen. Weiterhin finden Studierende dort die Übungsmaterialien, die Vorlesungsfolien sowie weiteres für den Kurs relevantes Material.

Der Kurs ist als ein weiterbildender Programmierkurs konzipiert. Dies bedeutet, dass erste Vorkenntnisse in der Programmierung vorausgesetzt werden. Trotzdem kann der Kurs als ein Basisprogrammierkurs bezeichnet werden.

Es war den Studierenden freigestellt in der Programmiersprache Python oder Java zu arbeiten. In der vorliegenden Arbeit wird sich auf die Programmiersprache Java konzentriert. In der nächsten angebotenen Iteration des Kurses wird die Wahl auf die Programmiersprache Java beschränkt.

Der Kurs umfasste zwei Vorlesungstermine pro Woche. Es wurde eine theoretische Vorlesung angeboten, die sich auf die Vorstellung der Datenstrukturen und Algorithmen sowie ihrer grundlegenden Implementierung fokussiert. In der zweiten,

praktischen Vorlesung liegt der Fokus auf der Implementierung der behandelten Algorithmen und Datenstrukturen.

Zusätzlich wurden jede Woche sechs verschiedene Übungstermine angeboten. In zwei dieser Veranstaltungen wurde die Programmiersprache Python verwendet. Drei der Übungstermine fokussierten sich auf die Programmiersprache Java. Eine Übung wurde explizit für Studierende des Bachelorstudiengangs Wirtschaftsinformatik angeboten.

Im Kurs Datenstrukturen gibt es zwei Arten der Leistungserbringung für Studierende. Welche Art die Studierenden ablegen müssen, ist in deren jeweiligen Studienordnungen definiert. Eine Leistung ist die Prüfungsvorleistung (PVL), welche als Zulassungsvoraussetzung für die Prüfung am Ende des Kurses dient. Die andere Leistung ist die anrechenbare Studienleistung (ASL). Diese stellt eine Ersatzleistung zur Klausur dar. Die Bewertung der ASLs entspricht damit der erreichten Note im Kurs. Insgesamt mussten die Studierenden sieben PVLs oder ASLs bewältigen. Zusatzpunkte konnten durch die Teilnahme an fünf verschiedenen Testaufgaben erworben werden. Im Schnitt hatten die Studierenden für die Bearbeitung einer PVL oder ASL zwei Wochen Zeit. Die Bearbeitung erfolgte ohne Aufsicht außerhalb der Vorlesung und Übung.

In den abgehaltenen Übungsterminen wurde eine Studie zum Lernverhalten im Einsatz mit ChatGPT durchgeführt. Diese nahm jeweils etwa eine halbe Stunde Zeit in Anspruch.

Folgende Themenbereiche wurden im Kurs Datenstrukturen im Sommersemester 2023 behandelt: Einführung in die Programmiersprache Java, Klassen, Sortieralgorithmen, verkettete Listen, Stack und Queue, Polymorphie, Clean Code, Bäume, Exceptions, Grapen, HeapSort, Hashen, HeapStack und Parametercloning. Die Implementierung der Datenstrukturen stellt den Inhalt der angebotenen Übungen sowie der PVLs und ASLs dar.

4.4. Teilnehmende und Datengrundlage

Studierende verschiedener Bachelor-, Master- und Diplomstudiengänge können an dem Kurs teilnehmen. Dabei beschränkt sich das Lehrangebot nicht ausschließlich auf Studierende aus dem Bereich der Informatik. So gibt es teilnehmende Studierende aus den Fachbereichen der Wirtschaft, Mathematik, Biomedizin und anderen technischen Fachbereichen. Insgesamt haben sich 157 Studierende für die Teilnahme des Kurses in der Online Lernplattform OPAL registriert. OPAL ist eine browserbasierte Webanwendung, die es Lehrenden ermöglicht, Gruppen für Ler-

nende zu erstellen und Unterlagen darüber in digitaler Form für die Studierenden bereitzustellen. Im Aufbau ist sie einem Forum nicht unähnlich.

Als Datengrundlage für diese Arbeit wurden alle Abgaben der Studierenden für die ASL gewählt. Grund für diese Entscheidung war der Inhalt der Aufgaben. Als Ersatzleistung für die Prüfung handelt es sich um komplexere Programmieraufgaben. Somit haben diese einen größeren Umfang als die PVLs und bieten mehr Material zur Auswertung. Vor der Auswertung wurden den Studierenden Nummern zugewiesen, um die Daten zu anonymisieren. Bei der Bearbeitung der Abgaben wurden die Studierenden angehalten, herkömmliche IDEs im Speziellen Visual Studio und Intellij, zu verwenden.

Im Kursverlauf erhielten vier Studierende die Note 5,0 und schlossen somit den Kurs nicht erfolgreich ab. Von diesen Studierenden wurde jeweils nur die erste ASL bearbeitet und eingereicht. Dies ermöglichte keine umfassende Auswertung der Programmierfehler. Die niedrigste Note unter den bestandenen Prüfungen war eine 2,7, welche von drei Studierenden erreicht wurde. In der vorliegenden Arbeit werden diese Studierenden der Gruppe der LPS zugeordnet. Die höchste in den ASL-Abgaben erreichte Bewertung betrug 1,3, welche drei Studierende erhielten. Diese Studierenden werden folglich in der vorliegenden Arbeit der Gruppe der HPS zugeordnet. Zur Erhöhung der Übersichtlichkeit und Vergleichbarkeit der Analyseergebnisse werden im Folgenden absolute Fehlerzahlen verwendet. Angesichts der identischen Anzahl von Studierenden in beiden Gruppen stellt dieses Verfahren eine methodisch adäquate Vorgehensweise dar. Die Tabelle 4.1 gibt eine Übersicht über die Nummern der Studierenden, welcher Gruppe sie zugeordnet wurden und welche ASLs von ihnen eingereicht wurden.

Tabelle 4.1.: Übersicht über untersuchte Studierende

Kennziffer	**HPS**	**LPS**	**abgegebene ASLs**
19		X	1, 2, 3, 4, 5, 6, 7
24		X	1, 2, 3, 4, 5, 6, 7
49		X	2, 3, 4, 5, 6, 7
26	X		1, 2, 3, 4, 5, 6, 7
27	X		1, 2, 3, 4, 5, 6, 7
42	X		1, 2, 3, 4, 5, 6, 7

4.5. Aufgaben der einzelnen anrechenbaren Prüfungsleistungen

Im Folgenden werden die von den Studierenden zu bearbeitenden ASLs inhaltlich vorgestellt. Insgesamt konnten über den Verlauf des Semesters sieben dieser Aufgaben von den Studierenden bearbeitet werden. Für jede eingereichte ASL vergaben die Prüfenden null bis zehn Punkte. Die Summe der erreichten Punkte wurde am Ende des Semesters addiert und über eine prozentuale Verteilung in eine Note übersetzt. Somit war es möglich, dass nicht jede ASL von den Studierenden bearbeitet werden musste, um erfolgreich den Kurs abzuschließen. User Wie in Tabelle 4.1 dargestellt, wurde die erste ASL von eine*r Student*in der LPS nicht eingereicht. Die Prüfungsleistung wird als bestanden gewertet, sofern von den Studierenden eine Bewertung mit der Note 4,0 oder besser erzielt wird.

Zugriff auf die Aufgabenblätter und weitere Hilfsmittel der zu bearbeitenden Aufgaben erhielten die Studierenden über die Lernplattform OPAL, sobald die Bearbeitungszeit begann. Über dasselbe Portal konnten die Studierenden ihre Abgaben einreichen. Lösungen konnten bis zu zwei Tage nach dem Abgabedatum per E-Mail an die Übungsleiter gesendet werden. Für diese nachträglichen Einreichungen erfolgte jedoch ein Punktabzug.

Aufgabe der ersten ASL war die Arbeit mit der Datenstruktur Matrix. Als Material für die Lösung der ASL 1 stand den Studierenden eine Datei der Klasse `Forecast` zur Verfügung. Diese enthielt eine Matrix. Die Studierenden sollten die Daten in der Matrix auswerten und manipulieren. Elf Studierende reichten Lösungen für diese ALS ein (siehe Anhang A6).

In der ASL 2 sollten die Studierenden mit der Datenstruktur Klassen und Interfaces arbeiten. Es wurden zehn Javadokumente zur Verfügung gestellt: fünf Interfaces und fünf Dokumente, welche die entsprechenden Klassen enthielten: `Student`, `Docent`, `Course`, `University` und `UniversityNetwork`. Die Klassen besaßen verschiedene Eigenschaften und sollten zusammen ein Universitätsnetzwerk simulieren (siehe A7).

Schwerpunkt der ASL 3 war ebenfalls die Datenstruktur Klasse in Kombination mit doppelt verketteten Listen. Gegeben waren vier Javadokumente. Zwei enthielten entsprechende Interfaces und zwei die Klassen `Konto` und `Bank`. Die Studierenden sollten selbst eine doppelt verkette Liste implementieren. Dafür mussten sie sowohl die Logik der Knotenklasse als auch der Listenklasse programmieren. Zehn Studierende reichten Abgaben für diese ASL ein (siehe A8).

Die ASL 4 beinhaltete die Datenstrukturen Klassen, Interfaces und eine doppelt verkettete Liste und kombiniert diese mit einer Prioritätswarteschlange. Den Stu-

dierenden wurden acht Javadomkumente zur Verfügung gestellt. Vier enthielten Interfaces und vier die Klassen `Node`, `Order`, `Restaurant` und `List`. Aufgabe der Studiernden war es, eine Sheduling-Anwendung zu schreiben, die den Ablauf in einem Restaurant simuliert. Dazu mussten sie wieder selbst eine doppelt verkettete Liste implementieren, und diese als Prioritätswarteschlange verwenden. Neun Studierende reichten Lösungen für diese ASL ein (siehe A9).

Die Datenstruktur eines binären Suchbaums sollte neben den Datenstrukturen Klasse in der ASL 5 implementiert werden. Gegeben waren vier Javadokumente, je zwei Interfaces und die beiden Klassen `Process` und `Processor`. Die Aufgabe der Studierenden bestand darin, die Arbeit eines CPUs zu simulieren. Diese ASL wurde von acht Studierenden abgegeben (siehe A10).

Schwerpunkt der ASL 6 war die Implementierung eines Graphs. Gegeben waren zwei Javadokumente, ein Interface und eines, welches die Klasse `Graph` enthielt. Die Studierenden sollten den längsten Pfad zwischen zwei Knoten ermitteln und zurückgeben sowie analysieren, ob der Graph schneidende Kanten besitzt. Neun Studierende haben Lösungen für die ASL eingereicht (siehe A11).

Die in der ASL 7 zu implementierende Datenstruktur war ebenfalls ein Graph. Gegeben waren zwei Javadokumente. Diese enthielten ein Interface und die Klasse `EnergyPlanner`. Die Studierenden sollten die Planung der Energieversorgung auf einer zweidimensionalen Karte simulieren. Jedes Feld der Karte sollte verschiedene Informationen zur Energiegewinnung enthalten. Weiterhin sollte beachtet werden, dass Verbindungen zwischen Feldern einen Energieverlust darstellt. Für diese ASL wurden von neun Studierenden Lösungen abgegeben (siehe A12).

4.6. Verwendete Fehlerkategorisierung

Die unter Unterabschnitt 3.2.2 vorgestellte Fehlerkategorisierung wurde von Altadmri und Brown wie in Unterabschnitt 3.2.3 weiterentwickelt. McCall und Kölling [22] erarbeiteten eine Fehlerkategorisierung mit verschiedenem Programmiercode u.a. aus dem Blackbox Datenset. Vorgestellt wurde diese unter Unterabschnitt 3.2.4. Albrecht und Grabowski nutzen als Grundlage für ihre Fehlerkategoisierung eine Kategorisierung des Programmierwissens und Ettles et al. [13] kategorisierten logische Programmierfehler in Unterabschnitt 3.2.5.

Es wurde nur die Fehlerkategorisierung von Altadmir und Brown [6] für die Verwendung im Kurs betrachtet, da diese eine Weiterentwicklung der von Hristova et al.[14] erstellten darstellt. Wie in Unterabschnitt 3.2.5 dargestellt, ist eine Einteilung von Fehlern nach der Kategorisierung von Albrecht und Grabowski [3] ohne

zusätzliches Material wie Interviews nur schwer umzusetzen. Deshalb wurde sich gegen den Einsatz dieser Fehlerkategorisierung entschieden. Der Fokus der von Ettles et al. [13] vorgestellten Fehlerkategorisierung liegt auf der Einordnung von logischen Fehlern. In der vorliegenden Arbeit soll eine umfangreichere Analyse der Fehler in den Abgaben der Studierenden erfolgen. Deshalb wird von dem Einsatz dieser Fehlerkategorisierung ebenfalls abgesehen.

Im Vergleich der Fehlerkategorisierungen von Altadmri und Brown [6] und McCall und Kölling [22] fällt auf, dass die erstere deutlich weniger umfangreich ist. Um die wesentlichen Unterschiede herauszuarbeiten, wurde überprüft, inwiefern sich die Fehlerkategorisierung nach Altadmri und Brown durch die Fehlerkategorien von McCall und Kölling darstellen lassen und umgekehrt. Eine tabellarische Zuordnung von McCall und Kölling zu Altadmri und Brown befindet sich im Anfang unter: A13. Eine entsprechende Übersicht für die Zuordnung Altadmri und Brown zu McCall und Kölling befindet sich im Anhang unter: A14. Wie in der A14 ersichtlich, kann nicht jede Fehlerkategorie von Altadmri und Brown einer spezifischen Unterkategorie von McCall und Kölling zugeordnet werden. Die von McCall und Kölling formulierten Oberkategorien können diese Fehler jedoch abbilden. Sollten diese Fehler mehrfach auftreten, könnte die Fehlerkategorisierung von McCall und Kölling erweitert werden.

In der Übersicht in Anhang A13 besitzen ebenfalls nicht alle Fehlerkategorien von McCall und Kölling Entsprechungen in der Kategorisierung von Altadmri und Brown. Sollten Fehler in den nicht abgebildeten Kategorien auftreten, müssten diese ergänzt werden. Dabei ist der Aufbau der Fehlerkategorisierung von Altadmri und Brown weniger dazu geeignet, diese zu erweitern.

Einige Fehler werden in der Fehlerkategorisierung von McCall und Kölling wesentlich detaillierter dargestellt als in der von Altadmri und Brown. In letzterer gibt es nur eine Fehlerkategorie, die das Fehlen einer Klammer oder eine falsche Kombination von Fehlern auffängt. In der Fehlerkategorisierung von McCall und Kölling werden 13 verschiedene Fälle von fehlenden Klammern unterschieden, wie in der Übersicht in Anhang A13 ersichtlich wird. Diese werden unterschieden durch den Kontext, in dem der Fehler auftritt. Gleiches gilt für einen Methodenaufruf, bei dem die falschen Argumente verwendet werden. Je nach Kontext kann dieser Fehler in fünf verschiedenen Fehlerkategorien dargestellt werden.

Während sieben der von Altadmri und Brown formulierten Fehlerkategorien keine genaue Entsprechung in der Fehlerkategorisierung von McCall und Kölling besitzen, finden sich 54 der Fehlerkategorien von McCall und Kölling nicht in der Fehlerkategorisierung von Altadmri und Brown wieder. Durch den Aufbau der Fehlerkategorisierung von McCall und Kölling könnten die fehlenden Fehlerkategorien

in Überkategorien fallen oder einfach als Fehlerkategorien in den entsprechenden Kategorien ergänzt werden.

Durch den höheren Detailgrad und der besseren Erweiterbarkeit wird in der vorliegenden Arbeit die Fehlerkategorisierung von McCall und Kölling verwendet. Weiterhin wurden die fehlenden Kategorien von Altadmri und Brown nicht in der Fehlerkategorisierung von McCall und Kölling ergänzt. Stattdessen wird in dieser Arbeit ein ähnlicher Ansatz verfolgt wie von McCall und Kölling [22]. Sollten vermehrt auftretende Fehler in den Abgaben der Studierenden registriert werden, wird die Fehlerkategorisierung entsprechend erweitert.

Bei der Arbeit wurde die gewählte Fehlerkategorisierung um die Kategorie logische Fehler erweitert. Bei der Abgrenzung zu anderen Programmierfehlern wurde die Definition von Hristova et al. [14] verwendet. Nach dieser sind semantische und logische Fehler mit einer hohen Wahrscheinlichkeit das Resultat falscher Annahmen von Programmierenden. Semantische Programmierfehler stehen aber in Zusammenhang mit der Programmiersprache, während logische Fehler dies nicht tun.

4.7. Auswertung

Insgesamt wurden 67 Lösungen von den Studierenden abgegeben, von denen 66 ausgewertet wurden. Eine Abgabe für die ALS 5 wurde für die vorliegende Arbeit nicht ausgewertet. Durch viele veränderte Rückgabewerte und schwerwiegende Fehler konnte die Funktionalität der Methoden nicht mehr nachvollzogen werden. Die Abgabe konnte deshalb nicht im selben Maß wie die anderen Abgaben der Studierenden ausgewertet werden und bei einer Auswertung hätte eine Vergleichbarkeit nicht mehr gewährleistet werden können. Es handelt sich um keine Abgabe eine*r Student*in aus der Gruppe der LPS oder HPS.

Für die anrechenbaren Studienleistungen sind Tests vorhanden, die der Bewertung dienen. Mit deren Einsatz konnte die Funktionalität des Programmcodes in Hinsicht auf die Aufgabenstellung getestet werden. Diese Tests dienten für die Fehlersuche als Ausgangspunkt. Darüber hinaus wurde der abgegebene Code gelesen, um alle Fehler, die die Tests nicht abgefangen hatten, zu notieren. Die Fehler der Studierenden wurden zunächst separat erfasst. Eine Einordnung in die gewählte Fehlerkategorisierung erfolgte später in einem weiteren Arbeitsschritt.

Die Kategorisierung der aufgetretenen Fehler in den Abgaben der Studierenden erfolgte in einer Tabelle. Für die Zuordnung der zuvor notierten Fehler wurde z.T. das Sprachlernmodell ChatGPT genutzt. Traten Fehler auf, die sich keiner Unterkate-

gorie zuordnen ließen, fand eine Zuordnung zu der entsprechenden Oberkategorie statt. Zusätzlich wurden diese Fehler markiert. Trat ein Fehler ohne Zuordnung zu einer Unterkategorie mehr als einmal auf, wurde eine neue Unterkategorie für diese Art von Fehler ergänzt. So sollte eine zu spezifizierte Liste an Fehlerkategorien vermieden werden, die eine Auswertung erschwert, und gleichzeitig dennoch alle Fehler so konkret wie möglich benannt werden. Im Folgenden werden einige Fehler in den Abgaben der Studierenden gezeigt, anhand dieser die Vorgehensweise verdeutlicht wird.

Abbildung 4.1.: Fehler Gruppe LPS in der ASL2

Der gezeigte Fehler wurde in der ASL 2 von einem Studierenden der Gruppe der LPS verursacht. Dieser Fehler verdeutlicht das fehlende Verständnis des Studierenden im Umgang mit Datentypen. Die Variable `AvailableDocents` ist vom Typ `ArrayList`. Bei einem durchgeführten Vergleich wird untersucht, ob der Inhalt dieser Variable `null` ist. Dieser Vergleich wird als Ergebnis immer `false` zurückgeben, da die Variable vorher im Code initialisiert wurde. Der Ursprung des Fehlers liegt vermutlich in einem mangelnden Verständnis über die Datentypen. Das Programm wird wegen dieser Zeile keine Compilerfehler erzeugen, es verhält sich aber nicht so, wie der Programmierende es vorgesehen hat. Da der Fehler in Zusammenhang mit der Programmiersprache auftritt, ist es ein semantischer Fehler. Für den Vergleich eines komplexen Datentyps mit den Vergleichsoperatoren wurde keine Unterkategorie erstellt, der Programmierfehler wurde also in die Fehlerkategorie *8.0* semantische Fehler eingeordnet. Für die korrekte Implementierung der Abfrage müsste die Methode `.equals` aufgerufen werden.

```
public boolean setDocent(Docent docent){
    if(docents.isEmpty()) {
        for (int i = 0; i < docent.getCourse_name().size(); i++) {
            if (Objects.equals(name, docent.getCourse_name().get(i))) {
                docents.add(docent);
                return true;
            }
        }
    }
    return false;
}
```

Abbildung 4.2.: Fehlergruppe HPS in der ASL2

Die Abbildung 4.2 zeigt eine Methode aus der ASL 2 eines Studierenden der Gruppe der HPS. Die implementierte Logik des Studierenden erhält ein Objekt des Types `Docent`. Wenn die Liste `docents` leer ist, wird geprüft, ob das übergebene Objekt angehängt werden kann. Der Fehler des Studierenden besteht darin, dass immer nur ein Objekt dem Attribut `docents` zugewiesen werden kann. In der Aufgabenstellung der ASL 2 wird aber spezifiziert, dass diese Liste mehrere Einträge enthalten können soll. Dieser Fehler verhindert nicht das Compilieren des Programmcodes und steht nicht in Zusammenhang mit der Programmiersprache Java. Daher wurde er als ein logischer Fehler eingeordnet. Er wurde spezifisch der Kategorie *12.4* unvollständige Logik zugewiesen. Eine Überprüfung zur Erweiterung der Liste ist nicht implementiert worden.

```
for(int i = 0; i < Anz;i++)
{
    if(swap1index==i)
    {
        swap1 = help;
        break;
    }
    help = help.next;
}
```

Abbildung 4.3.: Fehler Gruppe LPS in der ASL3

Im oben stehenden Beispiel ist eine Zeile Code zu sehen, in der gleich zwei Fehler auf einmal auftreten. Die Variable swap1index wurde in einer `for` - Schleife initialisiert. Diese Schleife wurde bereits beendet und die Variable existiert somit beim Ausführen dieser Codezeile nicht mehr. Somit wird auf eine Variable zugegriffen, die nicht existiert, was der Fehlerkategorie *1.4* entspricht. Dass die Variable nicht mehr existiert, liegt aber nicht daran, dass die Variable nie initialisiert wurde, sondern daran, dass sie an dem Zeitpunkt nicht mehr existiert. Dies ist ein Konzept, das nicht nur spezifisch in der Programmiersprache Java auftritt, sondern ebenfalls in anderen Programmiersprachen. Deshalb wurde entschieden, den Fehler ebenfalls in die Kategorie *12.0* logische Fehler einzusortieren. Dieser Fehler trat in der Abgabe des Studiernden vier Mal auf.

```
public int maxRootToLeafPath() {

    int maxRootToLeafPath = maxRoot(root);

    return maxRootToLeafPath;
}
```

Abbildung 4.4.: Fehler Gruppe HPS in der ASL5

In dem in Abbildung 4.4 abgebildeten Code ist eine Funktion zu sehen, in der zwei Fehler auftreten. Der Rückgabewert der Methode `maxRootToLeafPath` wurde vom Studierenden von `List<Process>` zu `int` geändert. Dieser Fehler wurde in der Kategorie *7.5.3* Rückgabewert eine vererbten Funktion wurde verändert eingeordnet. Aufgabe war es, den längsten Weg von der Wurzel zum Blatt zurückzugeben. In der implementierten Version wird aber die Länge des längsten Pfades zurückgegeben. Das ist ein logischer Fehler, da er nicht in Zusammenhang mit einer Programmiersprache steht. Da der Fehler konkret in der Verwendung mit der Datenstruktur eines binären Baums auftrat, wurde er in die Kategorie *12.2.1* eingeordnet.

5. Ergebnisse

Die Präsentation der Ergebnisse beginnt mit einer umfassenden Übersicht der Leistungen aller Studierenden in Abschnitt 5.1. Darauf folgend liegt in Abschnitt 5.2 der Schwerpunkt auf der Analyse der Leistungsunterschiede zwischen den Gruppen der LPS und HPS über alle Aufgaben hinweg. Abschließend werden die Unterschiede in Abgaben zu einzelnen Aufgaben der ASLs betrachtet.

5.1. Programmierfehler in den ASLs allgemein

Insgesamt wurden in 66 Abgaben von 13 Studierenden betrachtet und 710 Fehler registriert. Wie in Kapitel 4 beschrieben wurde bei der Arbeit die Liste der Fehlerkategorien von McCall und Kölling erweitert. Die hinzugefügten lauten:

- 1.6 Variablenname wird bei Variablenaufruf falsch geschrieben,
- 3.10 Aufruf einer nicht implementierten Funktion,
- 7.5 Interface- und Vererbungsimplementierungsfehler,
- 7.5.1 Fehlverwendung der `@Override`-Annotation,
- 7.5.2 Klasse statt Interface wird übergeben,
- 7.5.3 Rückgabewert einer vererbten Funktion wurde verändert,
- 8.8 Im Vergleich von zwei Strings wird `==` oder `!=` statt equals verwendet,
- 8.9 Unzureichende Behandlung von Spezialfällen,
- 8.9.1 Nicht-Abfangen negativer Indexwerte,
- 8.10 Fehlerhafte Indexverwendung,
- 12.0 Logische Fehler,
- 12.1 Logische Fehler bei der Arbeit mit Klassen,
- 12.2 Logische Fehler bei der Arbeit mit Graphen,
- 12.2.1 Logische Fehler bei der Arbeit mit einem Binärbaum,
- 12.3 Logische Fehler bei der Arbeit mit selbst implementierten verketteten Listen,
- 12.4 Unvollständige Logik in einer Funktion,

12.5 Logische Fehler bei der Arbeit mit einer Queue oder einem Stack.

Die am häufigsten auftretende Fehlerkategorie lautet „*12.3* Logische Fehler bei der Arbeit mit selbst implementierten verketteten Listen“. Die vier Fehlerkategorien, in welchen die meisten Fehler der Abgaben der Studierenden eingeordnet wurden, machen 34,87 % aller Fehler aus. Eine detailierte Übersicht der registrierten Fehler in den Abgaben der Studierenden findet sich unter Anhang A4.

Weiterhin wurden in folgenden Fehlerkategorien jeweils zwei Programmierfehler der Studierenden eingeordnet, was jeweils 0,28 % aller Fehler bedeutet:

- 3.8 Beim Methodenaufruf wurden keine runden Klammern genutzt`(())`,
- 8.4 Es fehlt das `return` am Ende der Funktion,
- 9.3 Abschlusszeichen `;` wurde vergessen,
- 9.8 Eine extra geschweifte schließende Klammer,
- 9.10 Ein Schlüsselwort wurde falsch geschrieben,
- 9.13.1 Fehlende schließende geschweifte Klammer nach Körper einer Kontrollanweisung,
- 9.15 Eine schließende runde Klammer zu wenig,
- 9.16 Fehlende öffnende geschweifte Klammer,
- 10.0 Code ist außerhalb der Methode oder eines anderweitigen Blocks,
- 12.5 Logische Fehler bei der Arbeit mit Queue oder Stack.

Jeweils ein Fehler, was 0,14 % aller Fehler entspricht, wurde in den Abgaben der Studierenden folgenden Fehlerkategorien zugeordnet:

- 1.3 Variable einer anderen Klasse wird verwendet,
- 3.0 Methoden: Nicht zulässiger Methodenaufruf,
- 3.5 Methodenaufruf: Datentypen der übergebenen Parameter sind falsch,
- 4.3 Methodendeklaration: Rückgabewert fehlt,
- 9.2 `:` wurde anstatt eines Semikolons `;` verwendet,
- 9.5 Schließende runde Klammer zu viel,
- 9.6 Öffnende geschweifte Klammer zu viel,
- 9.9 Extra oder falsch gesetzte einfache schließende runde Klammer.

Es traten in 51 der Fehlerkategorien keine Fehler in den Abgaben der Studierenden auf. Eine Übersicht der Fehlerkategorien findet sich im Anhang A5. Anschließend wurden die Fehler aus den Unter-und Oberkategorien addiert. So kann eine Übersicht erlangt werden, welche Arten von Fehlern am häufigsten gemacht wurden.

Tabelle 5.1.: Auftretenshäufigkeit in den Überkategorien

	Fehlerkategorie	**Absolut**	**Relativ**
12.0	Logische Fehler	500	42,81%
8.0	Semantische Fehler	450	35,79%
7.0	Falsche Verwendung von Klasse oder Datentyp	178	15,24%
1.0	**Variable:** Falsche Nutzung	96	8,22%
3.0	**Methoden:** Nicht zulässiger Methodenaufruf	96	8,22%
9.0	Einfache syntaktische Fehler	48	4,11%
4.0	**Methoden:** Falsche Methodendeklaration	40	3,42%
11.0	Unkategorisiert	10	0,86%
5.0	**Konstruktor:** Falscher Konstruktorenaufruf	8	0,68%
2.0	**Variable:** Falsche Deklaration	6	0,51%
10.0	Code ist außerhalb Methode oder anderweitigen Blocks	4	0,34%
6.0	**Konsturktor:** Falsche Deklaration	0	0%

Analog zur detailierten Fehlerauflistung sind die Kategorien *10.0* und *8.0* die beiden am häufigsten auftretenden Fehlerkategorien in den Abgaben der Studierenden. Die Unterkategorien der Fehlerkategorien *7.0* tauchen nicht so häufig auf, wenn man die Unterkategorien einzeln betrachtet. Rechnet man alle Fehler der Unterkategorien zusammen, machen die Studierenden Fehler, die dieser Kategorie zuzuordnen sind, am dritthäufigsten. Die Unterkategorien *1.0* tauchen ebenfalls individuell nicht so häufig auf, wenn die Fehlerkategorien einzeln aufgelistet werden, aber es werden in den einzelnen Unterkategorien so viele Fehler registriert, dass dies die Fehlerkategorie mit den vierthäufigsten zugeordneten Fehlern darstellt.

Bei der Auswertung fiel auf, dass die auftretenden Fehlerarten stark zwischen den einzelnen ASLs variierten. Deshalb werden nachfolgend die Fehlerkategorien nach den ASLs getrennt betrachtet.

5.2. Unterschiede zwischen schwachen und starken Programmierer*innen

Von den insgesamt 710 registrierten Fehlern wurden 263 von schwachen Programmieranfänger*innen und 112 von starken Programmieranfänger*innen verursacht. Der Anteil der Fehler von den schwachen Programmieranfänger*innen betrug damit 37,04 % aller Fehler, während die Fehler der starken Programmieranfänger*innen 15,77 % entsprechen. Schwache Programmierer*innen hatten also einen mehr als doppelt so hohen Anteil an den gesamten registrierten Programmierfehlern.

Eine vollständige Grafik, die die Auftretenshäufigkeit der Fehlerkategorien auflistet, ist im Anhang A4 zu finden. Auffällig ist bei der Übersicht, dass in der Gruppe der LPS mehr Fehler in verschiedenen Fehlerkategorien eingeordnet wurden. Von der Gruppe, die als HPS definiert wurde, wurden Fehler in 15 unterschiedlichen Fehlerkategorien gemacht. Die Fehler der Gruppe der LPS hingegen weist hingegen Fehler in 34 verschiedenen Fehlerkategorien auf.

In der Tabelle 5.2 werden die am häufigsten registrierten Fehlerkategorien in der Gruppe der LPS über alle Abgaben hinweg vorgestellt. Dabei traten drei Fehlerkategorien gleich häufig auf.

Tabelle 5.2.: Häufigste Fehlerkategorien der LPS

	Fehlerkategorie	**Absolut**
8.8	Im Vergleich von zwei Strings wird == oder ! = statt equals verwendet	23
12.0	Logische Fehler	22
12.3	Logische Fehler bei der Arbeit mit selbst implementierten verketteten Listen	22
12.4	Unvollständige Logik in einer Funktion	22
8.0	Semantische Fehler	21

Auffällig ist dabei, dass semantische und logische Fehler den Studierenden, die in die Gruppe der LPS eingeordnet wurden, am meisten Schwierigkeiten bereitet haben. Die einzige Fehlerkategorie, in der ähnlich viele Fehler registriert werden, ist die Kategorie 3.10. Der Aufruf einer nicht implementierten Funktion wurde 20 Mal in den Abgaben festgestellt.

Die häufigsten Fehlerkategorien ,in denen Fehler in den Abgaben der HPS über das Semester hinweg festgestellt wurden, werden in Tabelle 5.3 dargestellt.

Tabelle 5.3.: Häufigste Fehlerkategorien der HPS

	Fehlerkategorie	**Absolut**
8.10	Fehlerhafte Indexverwendung	24
12.0	Logische Fehler	18
12.3	Logische Fehler bei der Arbeit mit selbst implementierten verketteten Listen	15
12.4	Unvollständige Logik in einer Funktion	15
8.0	Semantische Fehler	10

In der Gruppe der HPS ist eine Unterkategorie der semantischen Fehler an erster Stelle der Auftretenshäufigkeit. Dabei ist anzumerken, dass alle 24 registrierten Fehler von einem/einer Studierenden der Gruppe in der ASL 1 verursacht wurden. Daher ist der Wert an und für sich wenig aussagekräftig, da er nur das Unverständnis von Indexen eines*r Studierenden aus der Gruppe widerspiegelt. Um ein vollständiges Bild zu erhalten, wurde daher die am vierthäufigsten auftretende Fehlerkategorie der Gruppe hinzugefügt.

Interessant beim Vergleich der am häufigsten auftretenden Fehler beider Gruppen ist, dass sowohl semantische als auch logische Fehler über alle Abgaben hinweg am häufigsten aufgetreten sind. Auch in den spezifischen Kategorien gibt es Überschneidungen. So finden sich die Kategorien *8.0*, *12.0*, *12.3* sowie die *12.4* in der Übersicht der am häufigsten aufgetretenen Fehlerkategorien. Bis auf die am häufigsten registrierte Fehlerkategorie ähnelt sich auch die Reihenfolge der aufgezählten Fehlerkategorien.

Ebenfalls gibt es eine Ähnlichkeit mit den am häufigsten auftretenden Fehlern in den ASLs allgemein. Auch dort tauchten unter dem am häufigsten auftretenden Fehlerkategorien die *12.3*, die *8.10*, die *8.0* und die *12.0* auf.

Der wohl größte Unterschied zwischen der Gruppe der LPS und HPS stellt die Anzahl der registrierten Fehler in den Kategorien dar. Bis auf die Fehlerkategorie *8.10* wurde in keiner anderen Fehlerkategorien eine Anzahl von insgesamt über 20 Fehlern von den HPS erreicht. Im Gegensatz dazu wurden in allen der am häufigsten auftretenden Fehlerkategorien der LPS über 20 aufgetretene Fehler registriert.

Klarer wird der Unterschied der gemachten Fehler, wenn man eine Übersicht der Fehler, die in den Überkategorien registriert wurden, ansieht. Wie zuvor wurden die Fehler der Unterkategorien hier den Oberkategorien hinzugefügt. Fehlerkategorien, in denen bei beiden Gruppen keine Fehler registriert wurden, werden der Übersicht nicht hinzugefügt.

Tabelle 5.4.: Auftretenshäufigkeit Fehler der HPS und LPS in den Überkategorien

	Fehlerkategorie	**LPS Absolut**	**HPS Absolut**
12.0	Logische Fehler	79	52
8.0	Semantische Fehler.	67	42
7.0	Falsche Verwendung von Klasse oder Datentyp.	50	18
3.0	**Methoden:** Nicht zulässiger Methodenaufruf	24	0
1.0	**Variable:** Falsche Nutzung von Variablen.	23	0
4.0	**Methoden:** Falsche Methodendeklaration	11	0
9.0	Einfache syntaktische Fehler.	5	0
5.0	**Konstruktor:** Falscher Konstruktorenaufruf	4	0

Einige Fehlerkategorien, die in der Fehlerübersicht aller Studierenden auftauchen, sind in der Übersicht der HPS und LPS nicht vorhanden. So wurden in den Abgaben der Gruppen keine falschen Methodendeklarationen, was der Fehlerkategorie *4.0* entspricht, registriert. Genauso tauchen keine Fehler in der Kategorie unkategorisierte Fehler (11) und falsche Variablendeklarationen (2.0) auf. In Bezug zu den anderen Fehlerkategorien wurden weniger falsche Nutzungen von Variablen festgestellt.

Im Gegensatz zu den Vergleich der beiden Gruppen der Studierenden fällt auf, dass LPS wesentlich mehr Fehler der Kategorie *7.0* machten als die Gruppe der HPS. Der Unterschied in der Häufigkeit des Auftretens der Fehler beträgt in fünf Fehlerkategorien 20 Fehler.

Wie in der allgemeinen Betrachtung der Fehler in den ASLs gab es auch bei den HPS und LPS Unterschiede zwischen den verschiedenen ASLs. Diese werden im nächsten Unterabschnitt betrachtet.

5.2.1. Fehler der HPS und LPS in der ASL 1

Auffällig ist, dass in der Fehlerkategorie 8.10 mit Abstand am meisten Fehler in der ASL 1 in der Gruppe der HPS registriert wurden. Dabei ist anzumerken, dass alle Fehler in dieser Kategorie von einem einzigen Studierenden gemacht wurden. Die ASL 1 ist die einzige Abgabe, bei der dadurch mehr Fehler von den HPS als von den LPS auftraten. Was sich in den registrierten Fehlern zeigt, ist, dass die Fehlerkategorie 8.8 in der Gruppe der LPS am häufigsten auftaucht. Diese machten in den gesamten ASLs die am häufigsten auftretende Fehlerkategorie in der Gruppe der LPS aus.

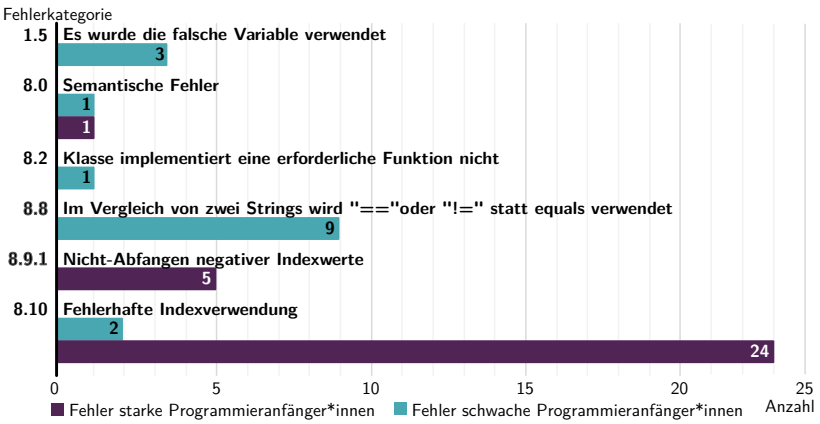


Abbildung 5.1.: Fehlerkategorien HPS und LPS ASL1

Während in der Gruppe der LPS der Fehler, dass negative Indexwerte nicht abgefangen werden, nicht auftaucht, wurde der Fehler in der Gruppe der HPS fünf Mal registriert. Ebenfalls ist bereits in den Abgaben der ASL 1 ersichtlich, dass die Gruppe der LPS mehr Fehler in verschiedenen Fehlerkategorien macht. In der Gruppe der LPS hingegen traten Fehler in fünf verschiedenen Fehlerkategorien auf, in der Gruppe der HPS nur Fehler in drei Kategorien.

5.2.2. Fehler der HPS und LPS in der ASL 2

Wie bereits in der Betrachtung der Fehler in der ASL 1 beschrieben, zeichnet sich in der ASL 2 der Trend ab, dass die Gruppe der LPS auch in den einzelnen Fehlerkategorien mehr Fehler bei der Programmierung macht. Ebenfalls wird erneut deutlich, dass von der Gruppe der LPS wesentlich mehr Fehler in verschiedenen Fehlerkategorien registriert wurden als in der Gruppe der HPS. So wurden für die Gruppe der HPS nur Fehler in acht verschiedenen Fehlerkategorien gefunden, für die Gruppe der LPS wurden in der ASL 2 Fehler in 23 verschiedenen Kategorien registriert. Auffällig ist auch, dass in der ASL 2 wesentlich weniger Fehler in der Gruppe der HPS gefunden wurden als in der Gruppe der LPS.

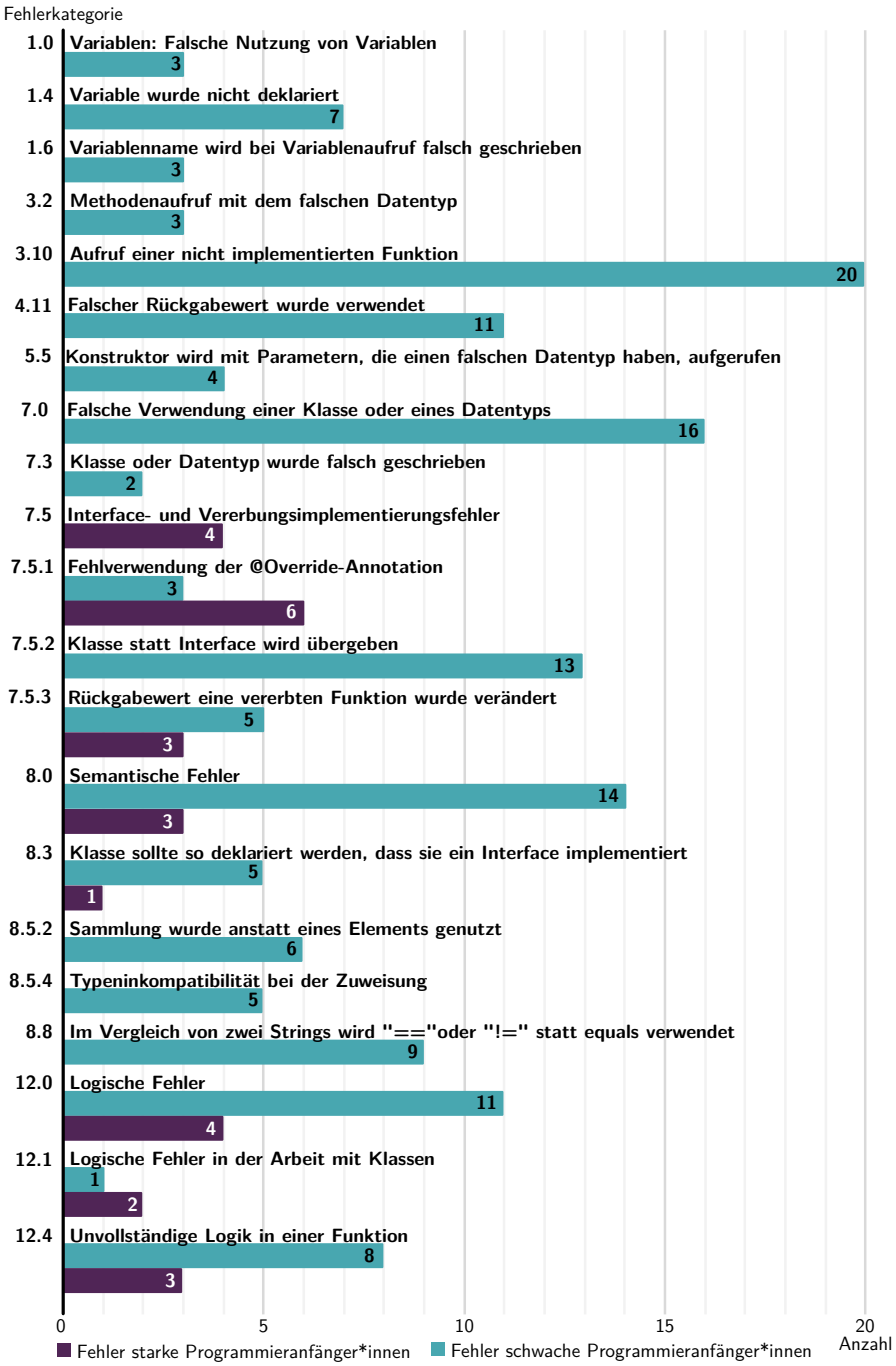


Abbildung 5.2.: Fehlerkategorien HPS und LPS ASL2

In folgenden Kategorien wurde von den LPS nur ein Fehler registriert:

1.3 Variable einer anderen Klasse verwendet,
3.7 Methodenname bei Methodenaufruf falsch geschrieben und
8.2 Klasse implementiert erforderliche Funktion nicht.

Die mit Abstand am häufigsten auftretende Fehlerart in der ASL 2 in der Gruppe der LPS war der Aufruf einer nicht implementierten Funktion. Hier sollte jedoch angemerkt werden, dass alle Fehler von einem oder einer einzigen Studierenden gemacht wurden. Dies trifft ebenso auf folgende Kategorien zu:

1.4 Variable nicht deklariert,
4.11 Falscher Rückgabewert verwendet,
7.0 Falsche Verwendung von Klasse oder Datentyp,
7.5.2 Klasse statt Interface übergeben,
8.5.2 Sammlung anstatt eines Elements genutzt,
8.5.4 Typinkompatibilität bei der Zuweisung und
8.8 Im Vergleich von zwei Strings wird `==` oder `!=` statt equals verwendet.

5.2.3. Fehler der HPS und LPS in der ASL 3

In der ASL 3 zeichnet sich erneut ab, dass von der Gruppe der LPS Fehler in wesentlich mehr Fehlerkategorien als der Gruppe der HPS gemacht wurden. So wurden in den Abgaben der LPS Fehler gefunden, die sich zehn verschiedenen Fehlerkategorien zuordnen lassen. In den Abgaben der Studierenden, die zur Gruppe der HPS gehören, wurden nur Fehler gefunden, die sich fünf verschiedenen Fehlerkategorien zuordnen ließen. Auffällig ist, dass in einer Unterkategorie der logischen Programmierfehler die meisten Fehler in beiden Gruppen registriert wurden. Während die Fehler der LPS nun jedoch in weniger Fehlerkategorien registriert wurden, wurden in der Gruppe der HPS mehr Fehler in verschiedenen Kategorien notiert.

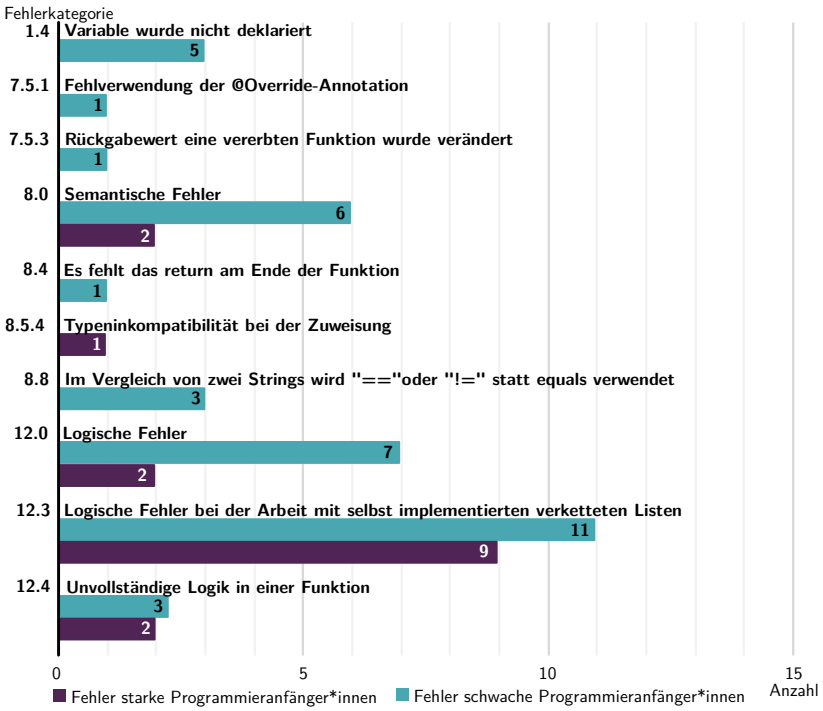


Abbildung 5.3.: Fehlerkategorien HPS und LPS ASL3

In der Oberkategorie *8.0* wurden insgesamt zehn Fehler in der Gruppe der LPS registriert. In der Gruppe der LPS wurden insgesamt drei semantische Fehler gefunden. Insgesamt wurden in der ASL 3 von beiden Gruppen weniger Fehler in den Abgaben der ASLs registriert.

5.2.4. Fehler der HPS und LPS in der ASL 4

In der ASL 4 waren die meisten Fehler der Studierenden logischer Art. Hier wurden jedoch von der Gruppe der HPS Fehler in mehr Fehlerkategorien als der Gruppe der LPS registriert. HPS machten Fehler in sieben verschiedenen Fehlerkategorien, LPS nur in sechs verschiedenen Kategorien. In beiden Gruppen steigen die Fehler in der Kategorie *12.4* an, was bedeutet, dass öfter die geforderte Funktionalität der Aufgabe nicht komplett umgesetzt wurde. In beiden Gruppen wurde fünf Mal festgestellt, dass Funktionen nicht komplett oder gar nicht umgesetzt wurden. Auch die Anzahl der gemachten Fehler insgesamt unterscheidet sich in den Gruppen nicht so deutlich voneinander wie in anderen ASLs. So wurden in den Abgaben der LPS 26 Fehler gefunden, in den Abgaben der HPS 19.

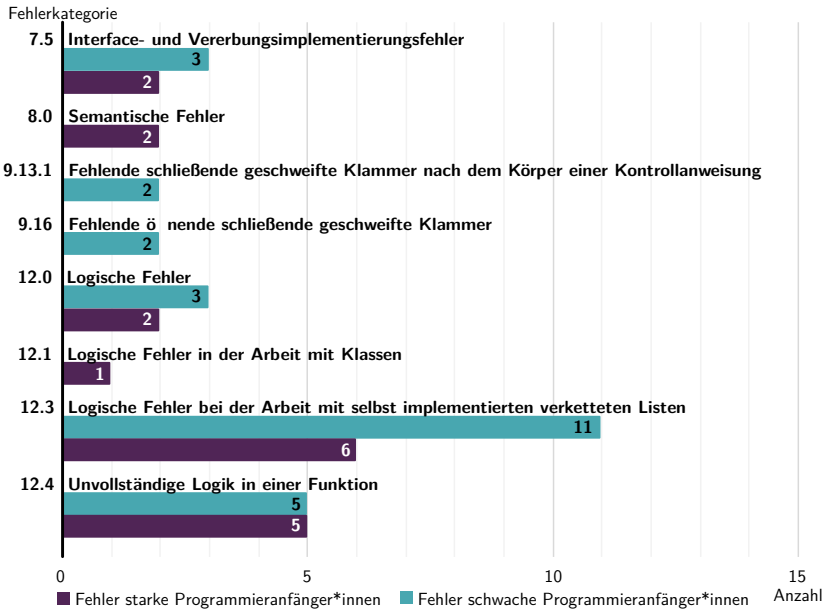


Abbildung 5.4.: Fehlerkategorien HPS und LPS ASL4

Auffallend ist, dass, obwohl die logischen Fehler von beiden Gruppen am häufigsten gemacht wurden, diese zumindest in der Kategorie *12.3* von den LPS häufiger registriert wurden als in der Gruppe der HPS. Ebenfalls interessant ist, dass von beiden Gruppen erneut Fehler in Bezug auf Interfaces und Vererbung gemacht wurden. Diese Fehlerkategorie wurde in der ALS 3 für beide Gruppen nicht notiert, in der ASL 2 nur für die Gruppe der HPS.

5.2.5. Fehler der HPS und LPS in der ASL 5

In der ASL 5 wurden wieder Fehler in jeweils fünf Fehlerkategorien von beiden Gruppen registriert. Damit wurden für beide Gruppen Fehler in weniger Fehlerkategorien als in der vorangegangenen ASL registriert. Der am häufigsten auftretende Fehler war für beide Gruppen wieder eine Fehlerkategorie der logischen Fehler. Interessant ist, dass in der letzten ASL viele Fehler in den Unterkategorien der semantischen Fehler auftraten. In dieser ASL gab es jedoch keine Fehler dieser Kategorie. Ebenfalls ist die Anzahl der gemachten Fehler in beiden Gruppen wie zuvor kaum unterschiedlich. LPS machten insgesamt zwölf Fehler in den Abgaben, in den Abgaben der Studierenden der Gruppe der LPS wurden elf Fehler notiert.

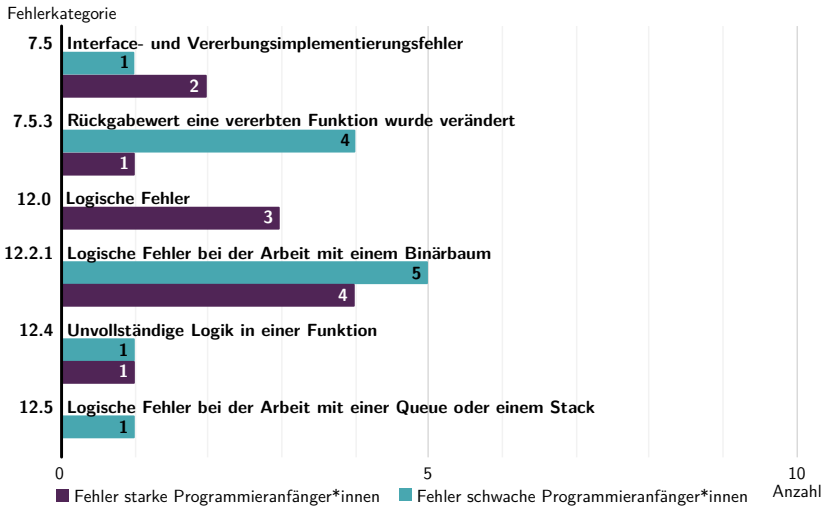


Abbildung 5.5.: Fehlerkategorien HPS und LPS ASL5

Fehler, die im Zusammenhang mit Interfaces und Vererbung stehen, wurden in den Abgaben der Studierenden auch hier wieder festgestellt. Allerdings veränderte die Gruppe der LPS häufiger Rückgabewerte von Funktionen, während in der Gruppe der HPS allgemeine Fehler in Bezug auf Vererbung und Verwendung mit Interfaces festgestellt wurden.

5.2.6. Fehler der HPS und LPS in der ASL 6

In der ASL 6 wurden insgesamt acht Fehler in der Gruppe der LPS und vier Fehler der HPS festgestellt. In den Abgaben der LPS tauchten zwar insgesamt mehr Fehler auf, dafür tauchten in den Abgaben der Gruppe der HPS Fehler in mehr verschiedenen Unterkategorien auf. So wurden in den Abgaben der LPS Fehler in drei verschiedenen Fehlerkategorien notiert, in der der HPS Fehler in vier verschiedenen Fehlerkategorien. Während logische Fehler in der Gruppe der LPS die häufigsten Fehler ausmachten, gab es keinen klaren Trend für die Gruppe der HPS.

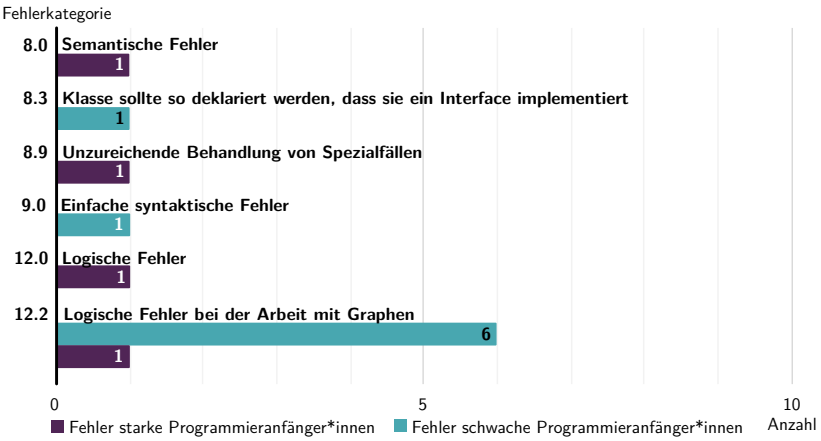


Abbildung 5.6.: Fehlerkategorien HPS und LPS ASL6

Auffällig ist, dass nur in der Gruppe der LPS vermehrt Fehler in einer bestimmten Kategorie auftauchten.

5.2.7. Fehler der HPS und LPS in der ASL 7

In der ASL 7 wurden wieder die meisten Fehler beider Gruppen in Unterkategorien der logischen Fehler gefunden. Diese traten dadurch auf, dass die Studierenden die geforderte Logik aus der ASL nicht vollständig umsetzten. In der Gruppe der LPS wurden Fehler in acht verschiedenen Fehlerkategorien registriert, in den Abgaben der HPS wurden Fehler in drei verschiedenen Fehlerkategorien registriert. Damit wurden von der Gruppe der LPS Fehler in mehr verschiedenen Fehlerkategorien registriert als in den Abgaben der ASL 6.

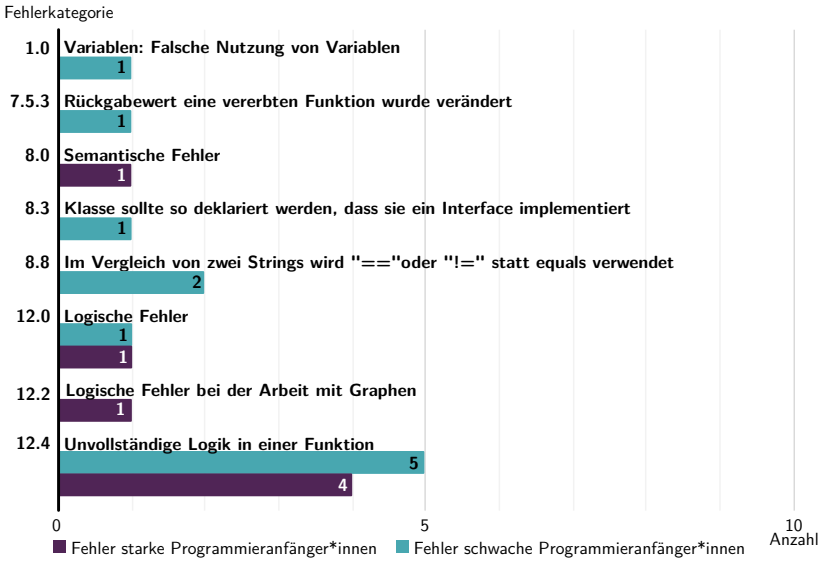


Abbildung 5.7.: Fehlerkategorien HPS und LPS ASL7

Während die gemachten Fehler der HPS ausschließlich logische Fehler sind, wurden von der Gruppe der LPS auch Fehler in den Oberkategorien *1.0*, *7.0* und *8.0* registriert. Ebenfalls wurden von der Gruppe der LPS wieder mehr Fehler registriert, zwölf, während in den Abgaben der LPS nur sechs Fehler gefunden wurden.

Insgesamt waren logische Fehler in den Abgaben der Studierenden allgemein und in den Abgaben der Studierenden der LPS und HPS die am häufigsten auftretende Fehlerart. Es wurden Unterschiede in den Arten wie auch deren Umfang der Fehler zwischen den Gruppen der LPS und HPS festgestellt. In den Abgaben zu den spezifischen ASLs gab es eine große Varianz, sowohl in der Art als auch dem Umfang der registrierten Fehler.

6. Diskussion

Zu Anfang werden die in Abschnitt 4.2 definierten Forschungsfragen in diesem Kapitel mit den in Kapitel 5 präsentierten Daten beantwortet. Anschließend soll in Abschnitt 6.3 eine Synthese aus der Beantwortung der Forschungsfragen gezogen werden, diese in den Kontext der vorgestellten Forschung gestellt werden und eine Auswertung bezüglich des in Abschnitt 4.3 vorgestellten Kurses Datenstrukturen erfolgen.

6.1. Unterschiede der Fehler von HPS und LPS

Die RQ1 und die dazugehörigen Annahmen lauten:

> **RQ1:** Welche Unterschiede gibt es in der Anzahl und der Art der gemachten Programmierfehler zwischen LPS und HPS?
> **Annahme 1:** LPS machen mehr unterschiedliche Fehler als HPS.
> **Annahme 2:** LPS machen im Umfang mehr Fehler als HPS.
> **Annahme 3:** In der Entwicklung der Fehler in Umfang und Art gibt es über des Semesters hinweg Unterschiede zwischen HPS und LPS. LPS werden am Ende des Semesters noch mit grundlegenden Programmierkonzepten Probleme haben.

Dazu werden zunächst die verschiedenen Annahmen in Unterabschnitt 6.1.1, Unterabschnitt 6.1.2 und Unterabschnitt 6.1.3 bewertet. Anschließend wird mit Hilfe der Erkenntnisse zu den Unterschieden der HPS und LPS in Art und Umfang der Fehler sowie deren Entwicklung die RQ1 in Unterabschnitt 6.1.4 beantwortet.

6.1.1. Unterschiede in der Art der Fehler von LPS und HPS

Im Folgenden sollen die Ergebnisse der Analyse der Programmierfehler in Kontext der ersten Annahme ausgewertet werden. Dazu werden zunächst die Arten der registrierten Fehler der HPS und LPS betrachtet.

Fehlerarten der HPS

Aus den Daten, die in Kapitel 5 vorgestellt wurden, geht hervor, dass HPS weniger unterschiedliche Fehler als LPS machen. Zunächst sollen die Fehlerarten der HPS betrachtet werden. In folgenden Fehlerkategorien wurden keine Fehler in den Abgaben der HPS registriert:

1.0 Variabeln: Falsche Nutzung von Variabeln,
2.0 Variable: Falsche Variabeldeklaration,
3.0 Methoden: Nicht zulässiger Methodenaufruf,
4.0 Methoden: Falsche Methodendeklaration,
5.0 Konstruktor: Falscher Konstruktorenaufruf,
6.0 Konsturktor: Falsche Deklaration des Konsturktors,
9.0 Einfache syntaktische Fehler,
10.0 Code ist außerhalb einer Methode oder eines anderweitigen Blocks und
11.0 Unkategorisiert.

Eine Schlussfolgerung, die daraus gezogen werden kann, dass keine Fehler in den Kategorien *9.0* und *10.0* registriert wurden, ist, dass sich HPS durch eine sichere Kenntnis der Programmsyntax auszeichnen. Ebenfalls weisen sie ein klares Verständnis vom Umgang mit Variablen, Methoden und Konstruktoren auf. Dies könnte seinen Ursprung darin haben, dass HPS bereits Erfahrung mit dem Programmieren besitzen. Das unterstützen auch die Ergebnisse von Rodrigo et al. [25], Liao et al. citeLiao2021 und Chinn et al. [10] die in Abschnitt 2.3 und Abschnitt 3.1 vorgestellt wurden.

In den Abgaben der HPS wurde eine Vielzahl an Fehlern beobachtet, speziell logische und semantische Fehler sowie Fehler in der Handhabung von Klassen und Datentypen. In der Arbeit mit Klassen und Datentypen traten Fehler im Zusammenhang mit der `@Override`-Annotation, allgemeine Interface und Vererbungsfehler auf. Außerdem veränderten HPS den Rückgabewert einer Funktion. Dies kann darauf schließen lassen, dass HPS ein lückenhaftes Verständnis in der Verwendung von Klassen und Interfaces besitzen.

Die spezifischen aufgetretenen semantischen Fehler betrafen Inkompatibilität von Typen bei der Zuweisung, unzureichende Behandlung von in den Aufgabenstellungen dargestellten Spezialfällen, speziell das Abfangen von negativen Indexwerten und fehlerhafte Indexverwendung. Es wurden auch weitere semantische Fehler re-

gistriert, die in keiner speziellen Unterkategorie eingeordnet wurden.

Neben allgemeinen logischen Fehlern machten HPS auch logische Fehler in der Arbeit mit Klassen, mit Graphen, speziell auch mit Binärbäumen und in der Arbeit mit selbst implementierten verketteten Listen. Ebenfalls beendeten HPS die Funktionalität von Methoden nicht. Dies könnte unter anderem daran liegen, dass sie nicht wussten, wie die entsprechende Logik implementiert wird oder, dass sie die Aufgabe nicht in der gegebenen Zeit lösen konnten.

Zusammenfassend lässt sich festhalten, dass HPS dazu neigen, Fehler in den komplexen Themenbereichen der Programmierung zu verursachen. Die Verwendung von Klassen erfordert ein Verständnis des Zusammenwirkens dieser. Auch das Auftreten von logischen Fehlern, die meist nicht in einer einzelnen Codezeile auftreten, spricht dafür. Die Ursache der registrierten semantischen Fehler könnte mangelndes detailliertes Wissen in der Anwendung der Programmiersprache sein. In grundlegenden Programmierkonzepten wie der Syntax und der Verwendung von Variablen, Methoden und Konstruktoren operieren HPS jedoch vergleichsweise fehlerfrei. Sie weisen Probleme bei der Arbeit mit Klassen auf. Weiterhin setzen sie logische Konstrukte unsicher in Programmcode um. Speziell sind die Datenstrukturen Graphen und verkettete Listen zu nennen. In Anbetracht der von Lahtinen et al. [18] in Abschnitt 2.2 vorgestellten Forschung ist vermutlich eher die Umsetzung als das theoretische Verständnis dieser die Ursache.

Fehlerarten der LPS

Im Gegensatz zur Gruppe der HPS weisen LPS eine größere Vielfalt an Programmierfehlern auf. Wie auch bei den HPS wurden bei der Gruppe der LPS Fehler in Zusammenhang mit komplexen Programmierbereichen festgestellt. Zusätzlich wurden auch Fehler in grundlegenden Aspekten der Programmierung festgestellt. Keine Fehler hingegen wurden in folgenden Kategorien in den Abgaben der LPS registriert:

- 2.0 Variable: Falsche Variabeldeklaration,
- 6.0 Konstruktor: Falsche Deklaration des Konstruktors,
- 10.0 Code ist außerhalb von Methode oder anderweitigen Blocks und
- 11.0 Unkategorisiert.

Im Gegensatz zu den HPS weisen LPS also Fehler in der Arbeit mit Methoden und Variablen auf. Auch einfache syntaktische Fehler wurden in den Abgaben der LPS

festgestellt. Daraus kann geschlossen werden, dass sie keine sicheren Kenntnisse in der Syntax der Programmiersprache besitzen. Interessant ist, dass keine Fehler bei der Deklaration von Variablen registriert wurden, dafür aber in der Arbeit mit diesen. Die Forschungsergebnisse von Rodrigo et al, Liao et al. und Chinn et al., die in den Kapiteln 2.3 und 3.1 vorgestellt wurden, legen nahe, dass LPS keine Vorkenntnisse in der Programmierung aufweisen. Dies könnte den Unterschied in den Arten der registrierten Fehler erklären. So wurden nicht nur allgemeine Fehler bei der Nutzung von Variablen gemacht, es wurden auch Variablen anderer Klassen verwendet sowie Variablen, die vorher nicht deklariert wurden.

Nachfolgend werden die erfassten Fehler der Unterkategorien ausgewertet und Annahmen zu den Ursachen ihres Auftretens formuliert. Das Auftreten von Fehlern wie der Verwendung von nicht zuvor deklarierten Variablen (1.4) und dem Aufruf nicht implementierter Funktionen (3.10) kann darauf hinweisen, dass LPS möglicherweise grundlegende Lücken im Verständnis der Programmierung im Allgemeinen und der Programmiersprache Java im Besonderen aufweisen.

Ebenfalls scheinen LPS Probleme in der Arbeit mit Datentypen aufzuweisen. Dafür spricht, dass bei Methodenaufrufen falsche Rückgabewerte verwendet wurden(4.11), Konstruktoren mit Parametern mit falschen Datentypen aufgerufen wurden (5.5), eine Klasse statt einem Interface übergeben wurde (7.5.2), Methoden mit dem falschen Datentyp aufgerufen wurden (3.2) und Klassen oder Datentypen falsch verwendet wurden(7.3).

Das Prinzip der Kapselung scheint LPS ebenfalls Probleme in der Umsetzung zu bereiten. Das wird daraus abgeleitet, dass Variablen aus anderen Klassen verwendet wurden (1.3), Fehler bei der Verwendung mit Interfaces und Vererbung registriert wurden (7.5) und die `@Override`- Annotation falsch verwendet wurde (7.5.1).

Andere Fehler könnten hingegen ein Hinweis auf eine allgemeine Überforderung sein: Dass falsche Variablen in den Abgaben verwendet wurden (1.5), Variablen bei Aufruf falsch geschrieben wurden (1.6) und Methodennamen bei Aufruf falsch geschrieben wurden (3.7) könnte dafür sprechen. Speziell sind das alles Fehler, die bei gründlicher und ruhiger Arbeitsweise und einem Verständnis für Syntax vermieden werden könnten.

Resümee zur Annahme 1

Die Annahme 1 kann somit zum Teil bestätigt und spezifiziert werden. In den Abgaben der HPS wurden vor allem komplexe Programmierfehler registriert. In den Abgaben der LPS wurden darüber hinaus einfache Programmierfehler gefunden.

Ursprung dieser Fehler könnten fehlende Vorkenntnisse in der Programmierung, aber auch Überforderung beim Erlernen der Programmierung und dem Implementieren von neuen Datenstrukturen sein.

6.1.2. Unterschiede im Umfang der Fehler von LPS und HPS

Allgemein lässt sich feststellen, dass in den Abgaben der HPS wesentlich weniger Fehler als in den Abgaben der LPS identifiziert wurden. Insgesamt wurden in den Abgaben der HPS 112 Fehler erfasst, was einem relativen Anteil von 15,77 % aller in den ASLs festgestellten Fehler entspricht. In den Abgaben der LPS wurden insgesamt 263 Fehler verzeichnet, was einem relativen Anteil von 37,04 % aller festgestellten Fehler in den ASLs entspricht.

Fehlerumfang der HPS

Wie bereits ausgeführt wurden in den Abgaben der Gruppe der HPS weniger als halb so viele Fehler wie in denen der LPS identifiziert. Abgesehen von der Fehlerkategorie 8.10 wurden in keiner Fehlerkatgorie von den HPS mehr als zwanzig Fehler notiert.

Insgesamt wurden von HPS am häufigsten Fehler in der Kategorie *12.0* registriert. Die logischen Fehler machen somit den größten Anteil an Fehlern in den Abgaben der HPS aus. Danach folgen die semantischen Fehler und fehlerhafte Verwendungen von Klassen oder Datentypen. Daraus kann abgeleitet werden, dass die HPS in der Bearbeitung der ASLs die größten Probleme mit logischen Fehlern hatten.

Einen Sonderfall stellen die Fehler der unvollständigen Logik in der Kategorie *12.4* dar. Diese kennzeichnen, dass eine Aufgabe nicht vollständig umgesetzt wurde, was verschiedene Gründe haben kann. Vermutet werden kann, dass die Studierenden die Umsetzung der geforderten Logik nicht leisten konnten oder ihnen die Zeit zur Umsetzung fehlte. Diese Fehler wurden 15 Mal in den Abgaben der Studierenden der Gruppe der HPS festgestellt.

In der Kategorie der semantischen Fehler wurden 42 erfasst. Da aber allein 24 der Fehler von einem Studierenden in einer ASL stammen, ist es schwer zu sagen, dass sich HPS dadurch auszeichnen, dass sie vor allem Probleme mit logischen und semantischen Aspekten der Programmierung aufweisen. Präziser wäre die Aussage, dass HPS vor allem logische Fehler in der Programmierung machen.

Ein weiteres Problem für die Gruppe der HPS scheint in der Programmierung der Umgang mit Klassen und Datentypen zu sein. Die meisten registrierten Fehler in einer Unterkategorie Falsche Verwendung einer Klasse oder eines Datentyps (7.0)

wurden in der Fehlerkategorie *7.5* Interface- und Vererbungsimplementierungsfehler verzeichnet. Acht Mal traten in der Gruppe der HPS Interface- und Vererbungsimplementierungsfehler auf. Sechs Mal wurde in den Abgaben der Gruppe der HPS die `@Override`-Annotation falsch verwendet und vier Mal der Rückgabewert verändert. Das lässt darauf schließen, dass HPS Probleme in der Arbeit mit Interfaces haben.

Die Anzahl der erfassten Fehler ist in der Gesamtheit aber bei weitem nicht so hoch wie die der logischen Fehler. Die logischen Fehler dominieren in ihrer Anzahl die registrierten Fehler der HPS. Daraus lässt sich schlussfolgern, dass sich HPS dadurch auszeichnen, dass sie vor allem Fehler in der Kategorie der logischen Fehler machen, aber auch mit dem Konzept von Interfaces und weiteren semantischen Eigenheiten Probleme haben.

Neben der Arbeit mit selbst implementierten verketteten Listen sticht die nicht vollständig implementierte Logik ins Auge. Es wurden bereits Gründe aufgezählt, warum das Beenden von implementierter Logik nicht stattfindet. Diese sollten verifiziert werden, um den Studierenden die Implementierung von in den Aufgabenstellungen gestellten Aufgaben leichter zu gestalten oder sie darin zu unterstützen. Die anderen Fehlerkategorien werden von ihnen in der Regel nur in einem kleinen Maße verursacht, was dafür spricht, dass diese Konzepte den HPS weniger Probleme bereiten.

Fehlerumfang der LPS

Zusätzlich dazu, dass LPS Fehler in vielen verschiedenen Fehlerkategorien in ihren Abgaben haben, ist die Verteilung der Anzahl der Fehler ebenfalls weniger eindeutig als bei der Gruppe der HPS. Allgemein ist die Kategorie mit den meisten registrierten Fehlern ebenfalls die der logischen Fehler. Die Anzahl der der Kategorie zugeordneten Fehler ist aber höher als bei der Gruppe der HPS. Insgesamt traten 79 logische Fehler in den Abgaben der LPS auf. Danach folgt die Kategorie der semantischen Fehler, in denen 67 Fehler in den Abgaben der LPS registriert wurden. Dies sind ebenfalls wesentlich mehr als in den Abgaben der HPS.

Der größte Unterschied in den registrierten Fehlern findet sich in der falschen Verwendung einer Klasse oder eines Datentyps *7.0*. In den Abgaben der LPS wurden 50 Mal eine fehlerhafte Verwendung einer Klasse oder eines Datentypen registriert, im Gegensatz zu 18 Fehlern in den Abgaben der HPS. Dass eine Klasse statt eines Interface übergeben wurde (Kategorie 7.5.2), trat 13 Mal in den Abgaben der Studierenden auf, der Rückgabewert einer vererbten Funktion wurde elf Mal verändert (Kategorie 7.5.2). Offensichtlich haben LPS Probleme mit dem Konzept und der Umsetzung von Klassen und weisen Wissenslücken in diesem Bereich auf.

Die Verwendung von Methoden und Variablen schien den Studierenden in der Gruppe der LPS ebenfalls mehr Problemen als denen in der Gruppe der HPS zu verursachen. Fehler bei der Methodendeklaration, einfache syntaktische Fehler und falsche Konstruktorenaufrufe machten hingegen nur einen kleinen Anteil der registrierten Fehler in der Gruppe der LPS aus.

Betrachtet man sich die Schlussfolgerungen aus den registrierten Fehlerkategorien an, müssen einige Aussagen neu bewertet werden. In den Kategorien *1.4* und *3.10* wurden insgesamt 32 Fehler in den Abgaben der LPS registriert. Das spricht dafür, dass Studierende in der Gruppe der LPS Lücken im Verständnis der Programmiersprache haben, diese sind aber nicht so deutlich ausgeprägt wie zuvor angenommen.

Die Annahme, dass LPS insgesamt mit dem Konzept von Datentypen Probleme haben, kann bei der Betrachtung, wie häufig diese Fehler aufgetaucht sind, ebenfalls nicht ohne Einschränkungen bestehen bleiben. In den dafür als Beweis genannten Kategorien *4.11*, *5.5* und *7.5.2* wurden insgesamt 30 Fehler bei der Auswertung der Abgaben registriert. Dabei ist anzumerken, dass die Fehler in den Kategorien *4.11*, in der ein falscher Rückgabewert verwendet wird, und *7.5.2* dass eine Klasse statt einem Interfaces übergeben wird, wesentlich mehr Fehler auftraten als in den anderen genannten Fehlerkategorien. LPS haben Probleme mit dem Konzept der Datentypen, dies ist trotzdem nicht so ausgeprägt wie zuvor angenommen. Spezifischer wäre es korrekt zu sagen, dass LPS verschiedene Probleme mit Rückgabewerten von Methoden aufweisen.

Die Annahme, dass LPS allgemein Probleme mit dem Prinzip der Kapselung haben, kann keinen Bestand haben. In den genannten Kategorien wurden lediglich sechs Fehler insgesamt registriert.

Die Fehler, die aus einer Überforderung resultieren könnten, sind ebenfalls nur in geringer Ausprägung vorhanden. Nur fünf Fehler wurden insgesamt in den Kategorien *1.5*, *1.6* und *3.7* registriert. Diese Fehlerarten verursachen also ebenfalls keine schwerwiegenden Probleme.

Abschließend kann die Aussage getroffen werden, dass logische und semantische Fehler von LPS häufig und im Vergleich zu der Gruppe der HPS häufiger registriert wurden. Aber auch die Arbeit mit Konstruktoren und Variablen bereitet den Studierenden der Gruppe der LPS Probleme. Die Durchmischung im Umfang der eingeordneten Fehler zeigt, dass LPS partiell Probleme mit grundlegenden Konzepten der Programmierung haben.

Ebenfalls häufig wurde der Fehler registriert, dass versucht wurde, einen String mit einem Vergleichsoperator statt der Methode `.equals` zu vergleichen (Katego-

rie 8.8). Hier muss angemerkt werden, dass diese über alle ASLs hinweg von einem einzigen*r Studierenden verursacht wurden. Das spricht dafür, dass dieser Studierende ein fehlerhaftes Verständnis von komplexen Datentypen besitzt. In den restlichen Unterkategorien der semantischen Fehler lagen alle registrierten Fehler in den jeweiligen Gruppen in ihrer Anzahl unter zehn.

Logische Fehler traten wie bei der Gruppe der HPS vermehrt in der Gruppe der LPS auf. Dort gab es keine Abweichung im Vergleich, außer, dass in den Kategorien mehr Fehler registriert wurden.

Resümee zur Annahme 2

Die zweite getroffene Annahme war folgende: LPS machen im Umfang mehr Fehler als HPS.

Auch, wenn die Annahme mit den vorliegenden Daten bestätigt wird, ist es wichtig, an dieser Stelle eine Anmerkung hinzuzufügen: 110 der registrierten Fehler der Gruppe der LPS stammen von einem einzigen Studierenden aus einer einzigen Abgabe. Bei einer Gruppe, die nur aus drei Studierenden besteht, verzerrt dies die gesammelten Daten erheblich. Die Studierenden der beiden anderen Gruppen machten lediglich 126 Fehler. Wenn man davon ausgeht, dass dies einen Durchschnitt von 63 Fehlern ergibt, würden drei solcher Studierenden 189 Fehler machen. Die gemachten Fehler der HPS sind zwar immer noch in größerer Anzahl vorhanden, der Unterschied in der Fehleranzahl wäre jedoch wesentlich kleiner. Auch, wenn die Annahme immer noch Bestand hätte, wäre das Ausmaß im Unterschied ein anderes.

Insgesamt konnte die Annahme jedoch bestätigt werden. LPS machen generell mehr Fehler als HPS. Dabei ist interessant, dass die Verteilung der registrierten Fehler in den beiden am häufigsten auftretenden Fehlerkategorien ähnlich sind. Insgesamt bereiten logische Fehler und semantische Fehler LPS mehr Probleme als HPS. Zusätzlich machen LPS noch Fehler in verschiedenen anderen Fehlerkategorien.

6.1.3. Entwicklung der Art der Fehler und des Umfangs der LPS und HPS

Um diese Annahme zu betrachten, wird zunächst die Entwicklung der Art der Fehler von LPS betrachtet. Im Anschluss werden die Veränderungen in der Fehleranzahl über das Semester hinweg analysiert.

Entwicklung der Art der Fehler der LPS und HPS

Wie aus der Datenbeschreibung in Kapitel 5 hervorgeht, gibt es eine große Variation in den verschiedenen ASLs, sowohl in den Fehlerkategorien als auch in der Anzahl der registrierten Fehler. Zur Veranschaulichung der Unterschiede der aufgetretenen Fehlerkategorien dient die folgende Grafik.

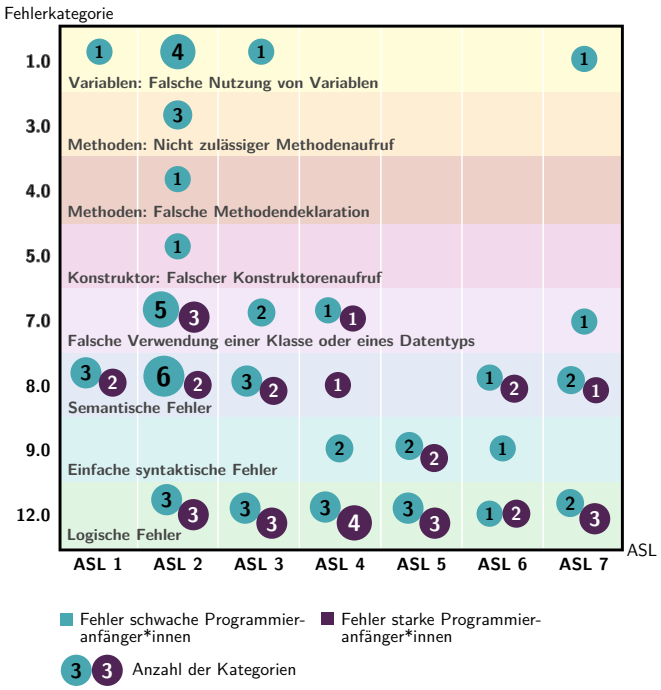


Abbildung 6.1.: Fehlerkategorien HPS und LPS ASLs

Interessant ist zu sehen, dass die große Varianz der Fehlerkategorien der LPS, in denen Fehler registriert wurden, allem voran in den ersten beiden ASLs auftritt. Ebenfalls wird in den ersten zwei ASLs eine größere Varianz an Fehlern in den Kategorien der Fehler registriert, in denen auch in der Gruppe der HPS Fehler notiert wurden. In der dritten und vierten ASL nimmt der Unterschied im Umfang der gemachten Fehler ab. In den fünften, sechsten und siebten ASLs gleicht sich der Umfang der gemachten Fehler beinahe an. Die wesentlichen Unterscheide in der Art der auftretenden Fehler bestehen somit vor allem am Anfang des Semesters. Dass immer wieder logische Fehler gemacht werden, könnte daran liegen, dass im Verlauf des Kurses neue Datenstrukturen vorgestellt werden und folglich in den ASLs von den Studierenden implementiert werden müssen.

Entwicklung der Umfang der Fehler LPS und HPS

Analog zu dem Umfang der Fehlerkategorien wurden auch bei der Anzahl der Fehler in der Entwicklung der ASLs große Unterschiede festgestellt. Zur Übersichtlickeit wird auch an dieser Stelle eine Grafik bereitgestellt.

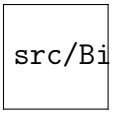


Abbildung 6.2.: Fehlerkategorien HPS und LPS ASLs

Auffällig ist, dass in den Abgaben der ersten ASL mehr Fehler in der Gruppe der HPS als in der der LPS registriert wurden. Dabei ist wieder anzumerken, dass ein Großteil davon von einem*r Studierenden verursacht wurde. Es handelt sich um die 24 Fehler in der Kategorie *8.10*.

In der zweiten ASL wurden die meisten Fehler in den Abgaben der Gruppe LPS registriert. Auch hier stammen 110 der registrierten Fehler wieder von einem*r Studierenden. Dies würde bedeuten, dass ohne diese Fehler die Anzahl der Fehler von LPS und HPS wesentlich näher beieinander lägen.

Im Gesamtbild sinken die registrierten Fehler der Gruppe der LPS über den Verlauf der Abgaben der ASLs von der zweiten bis zur sechsten ASL. Lediglich bei der Anzahl der gemachten Fehler in der ALS 7 gibt es einen leichten Anstieg.

Die notierten Fehler der HPS hingegen sinken von der ersten bis zur dritten ASL. In der ASL 4 wurden wieder etwas mehr Fehler registriert. Im Verlauf zu der ASL 6 sinkt die Gesamtzahl der Fehler in den Abgaben wieder. Wie auch bei der Gruppe der LPS steigt die Anzahl der Fehler in der ASL 7 wieder leicht an. Dass in beiden Gruppen ein leichter Anstieg der Anzahl der Fehler registriert wurde, kann darauf hindeuten, dass die Aufgabe als schwerer als die vorhergehenden von den Studierenden wahrgenommen wurde.

LPS machen mehr unterschiedliche Fehler als HPS zu Beginn des Kurses Datenstrukturen. Dabei ist in Hinsicht auf die Lehre im Kurs Datenstrukturen anzumerken, dass die Varianz der registrierten Fehler über den Verlauf des Semesters stark abnimmt und sich am Ende in etwa angleicht.

Resümee zur Annahme 3

In der Entwicklung der Fehler in Umfang und Art gibt es über die Entwicklung des Semesters Unterschiede zwischen HPS und LPS. LPS werden auch am Ende

des Semesters noch mit grundlegenden Programmierkonzepten Probleme haben.

Die dritte Annahme kann nicht vollständig bestätigt werden. Der erste Teil der Annahme kann bestehen bleiben. Insbesondere in den ersten beiden Abgaben verursachten Studierende der Gruppe der LPS wesentlich mehr Fehler in mehr Fehlerkategorien als HPS. In der Entwicklung kann ab der zweiten ASL eine Verbesserung der Kenntnis in der Programmierung gedeutet werden. Insbesonders in den weniger komplexen Bereichen der Programmierung machten LPS im Verlauf der ASLs weniger bis gar keine Fehler. In der Gruppe der HPS hingegen ist eine weniger deutliche Entwicklung zu betrachten, wobei Studierende der Gruppe der HPS im Verlauf des Semesters ebenfalls weniger Fehler in weniger Fehlerarten verursachen.

Der zweite Teil der Annahme muss als widerlegt betrachtet werden. Bei den Abgaben der sechsten und siebten ASL wurden auch bei der Gruppe der LPS vor allem logische und semantische Fehler festgestellt. Dies sind eher komplexe Programmierkonzepte.

6.1.4. RQ1: Welche Unterschiede gibt es in der Anzahl und der Art der gemachten Programmierfehler zwischen LPS und HPS?

Nachdem die drei Annahmen diskutiert wurden, kann im Anschluss die RQ1 beantwortet werden. Sie lautet:

Welche Unterschiede gibt es in der Anzahl und der Art der gemachten Programmierfehler zwischen LPS und HPS?

Insgesamt traten in den Abgaben der HPS weniger Fehler in einem kleineren Umfang als bei den LPS auf. Logische Fehler stellen den großen Anteil der Fehler der LPS dar. Semantische Fehler bereiten manchen der HPS Probleme. Besonders die Indexverwendung in der Arbeit mit einer Matrix ist hervorzuheben.

LPS hingegen verursachten vergleichsweise viele Fehler in vielen verschiedenen Fehlerkategorien. Auch bei ihnen stellt der Themenbereich der logischen Fehler den großen Anteil der verursachten Fehler. Im Umgang mit Klassen oder Datentypen und Rückgabewerten sowie der Arbeit mit Interfaces weisen LPS ebenfalls Wissenslücken auf. Weitere Fehlerarten treten in den Abgaben der LPS zwar ebenfalls auf, jedoch in einem kleinen Ausmaß. Zu erwähnen ist, dass LPS ebenfalls ein lückenhaftes Verständnis der verwendeten Programmiersprache aufweisen. Diese sind aber nicht so schwerwiegend wie die Probleme mit Semantik, Logik und Datentypen.

Anzumerken ist, dass die Unterschiede am Anfang des Semesters deutlich auftreten und sich im Verlauf der Abgaben annähernd angleichen. Auch, wenn am Ende immer noch mehr Fehler in mehr Fehlerkategorien verzeichnet werden, ist der Unterschied nicht so gravierend wie in den ersten Abgaben. LPS werden im Verlauf des Kurses also wesentlich sicherer im Umgang mit Semantik, Logik und Datentypen. Auch HPS werden im Verlauf des Semesters sicherer in der Verwendung von diesen Konzepten.

6.2. Unterschiede im Umfang der logischen Fehler bei neu eingeführten Programmierkonzepten

Im Folgenden werden die Forschungsfrage RQ2 mit der dazugehörenden Annahme 4 wiederholt. Diese lauten:

> **RQ2:** Welche Unterschiede gibt es beim Umfang der logischen Fehler im Umgang mit neu eingeführten Programmierkonzepten zwischen LPS und HPS?
> **Annahme 4:** LPS machen im Umfang mehr logische Fehler in Bezug auf neue Datenstrukturen.

Um die Forschungsfrage zu beantworten, wird sich zunächst der Überprüfung der Annahme zugewendet. Zur Verdeutlichung folgt eine Tabelle mit einer Übersicht zu den aufgetretenen Fehlern in den Abgaben der ASLs im Kontext mit den Datenstrukturen, die in diese implementiert wurden und in einer vorhergehenden ALS noch nicht implementiert werden mussten. Dabei gibt es zwei Einschränkungen bei der ASL 1 und ASL 7.

Dazu folgt in Tabelle 6.1 eine Übersicht, welche Datenstrukturen die Aufgaben der ASLs beinhalteten. Eine ausführliche Beschreibung der Aufgaben findet sich unter Abschnitt 4.5. Dabei ist anzumerken, dass die Datenstruktur Klasse nur genannt wurde, wenn mehr als eine Klasse implementiert werden sollte.

Tabelle 6.1.: Beinhaltete Datenstrukturen der ASLs

ASL	Datenstrukturen
1	Matrix
2	Klassen
3	Klassen, doppelt verkettete Liste
4	Klassen, doppelt verkettete Liste, Prioritätswarteschlange
5	Klassen, Binärer Suchbaum
6	Klassen, Graph
7	Graph

Bei der ASL 1 traten keine logischen Fehler im Umgang mit einer Matrix auf. Die Fehler, die auftraten, betrafen falsche Indexverwendungen beim Zugriff auf den Inhalt der Matrix. Diese wurden in der Fehlerkategorie *8.10* festgehalten. Deshalb wurde diese Kategorie in die Tabelle aufgenommen, auch, wenn es sich nicht um logische, sondern um semantische Fehler handelt.

In der ASL 7 wurde keine Fehlerkatgorie für speziell die Arbeit mit gerichteten Graphen in der Übersicht ergänzt. Wie in der Methodik beschrieben, wurde eine neue Fehlerkategorie nur ergänzt, wenn in dieser mindestens zwei Fehler auftauchten. Alle Fehler in der ASL 7 wurden allgemeinen Fehlern in der Arbeit mit Graphen zugeordnet und nicht speziell der Arbeit mit gerichteten Graphen.

Tabelle 6.2.: Fehler in neuen Datenstrukturen

ASL	Datenstruktur	Fehler-kategorie	Absolut LPS	Absolut HPS
1	Matrix	8.10	2	24
2	Klassen	12.1	1	2
3	Listen	12.3	11	9
4	Prioritätswarteschlange	12.5	0	0
5	Binärer Suchbaum	12.2.1	5	4
6	Graph	12.2	6	1
7	Gewichteter Graph	12.2	2	0

Auffällig für die Gruppe der HPS ist, dass die meisten Fehler in der Indexierung in der ASL 1 auftraten und in der Arbeit mit selbst implementierten verketteten Listen in der ASL 3. Dabei wurde bereits angemerkt, dass alle Fehler in der Kategorie 8.10 von einem Studierenden der Gruppe der HPs verursacht wurden.

Spannend ist, dass obwohl HPS sich dadurch auszeichnen, dass, sie vor allem logische Fehler machen und im Umgang mit Klassen und Datentypen ebenfalls Probleme haben, bei der ersten Arbeit mit Klassen logische Fehler nur wenig auftauchen. Das spricht dafür, dass HPS spezifisch Probleme bei der Verwendung mit Klassen haben, nicht jedoch mit der Logik dieser. Daraus könnte geschlossen werden, dass das Verständnis von Klassen nicht lückenhaft ist, jedoch das Wissen um die Nutzung dieser.

Anders sieht es bei selbst implementierten verketteten Listen aus. Neun der insgesamt 15 registrierten Fehler im Umgang mit verketteten Listen stammen aus den Abgaben der ALS, in denen diese Datenstruktur zuerst auftaucht. Das spricht dafür, dass das theoretische Wissen zu diesem Thema bei der Gruppe der HPS lückenhaft sein könnte.

Andere Datenstrukturen scheinen HPS keine Probleme zu bereiten, wie die wenigen Fehler in diesen Kategorien anzeigen. Das lässt darauf schließen, dass diese Datenstrukturen von den Studierenden verstanden wurden und das Wissen für die Implementierung dieser vorhanden ist.

Interessant bei der Betrachtung der LPS ist, dass sich bis auf die ASL 1 die Fehlerhäufigkeit der Logikfehler bei der Arbeit mit neuen Datenstrukturen ähnlich darstellt wie bei der Gruppe der HPS. Dies könnte dafür sprechen, dass es sich bei den Problemen nicht um Probleme handelt, die spezifisch HPS und LPS betreffen, sondern alle Studierende. Wobei die Anzahl der Fehler bei den LPS wieder höher ist als die bei den HPS.

Wie bei der Gruppe der HPS sind die Fehler im Umgang mit Klassen und Datentypen keine logischen Fehler. Dies könnte darauf hindeuten, dass theoretische Grundlagen zu den Klassen verstanden wurden, die Implementierung dieser den Studierenden jedoch Probleme bereiten.

Bei der Arbeit mit verketteten Listen traten die meisten logischen Fehler bei Nutzung einer neuen Datenstruktur auf. Elf der insgesamt 22 registrierten logischen Fehler, die bei der Arbeit mit dieser Datenstruktur festgestellt wurden, tauchten somit bei der ersten Abgabe, die eine Implementation der Datenstruktur fordert, auf. Auch hier kann davon ausgegangen werden, dass es sich um ein allgemeines Problem handelt, welches aber Studierende der Gruppe der LPS stärker betrifft.

Auch beim Auftreten der anderen neuen Datenstrukturen machten LPS mehr Fehler als HPS. Das spricht dafür, dass sie mehr Probleme in der Implementierung dieser hatten und ihr Wissen in diesen Bereichen mehr Lücken aufweist als das der HPS. Dies ist ein Problem, das speziell die Gruppe der LPS zu betreffen scheint.

Zusammenfassend kann ausgesagt werden, dass die Studierenden mit dem Konzept

der verketteten Liste im Besonderen Probleme bei den Abgaben der ASLs hatten. Vereinzelt scheint es ebenfalls Probleme bei der Indexierung von Matrizen zu geben. Andere Datenstrukturen wurden allgemein relativ sicher implementiert und es wurden in Bezug auf diese wenige logische Fehler registriert. LPS unterschieden sich von den HPS darin, dass sie mehr logische Fehler bei der Implementierung einer neuen Datenstruktur verursachen.

Logische Fehler bei der Implementierung der Datenstrukturen im Verlauf des Semesters

Es soll eine kurze Verfolgung des Verlaufs der logischen Fehler in Bezug auf die Datenstrukturen erfolgen. Wie bereits aufgezeigt, variieren die Fehler der Studierenden über den Verlauf der verschiedenen ASLs sehr. Weiterhin wurden nur einige der registrierten Fehler bei der ersten Nutzung der Datenstrukturen nachgewiesen. Dazu folgt eine Grafik, die den Verlauf der Entwicklung der logischen Fehler über alle ASLs darstellt.

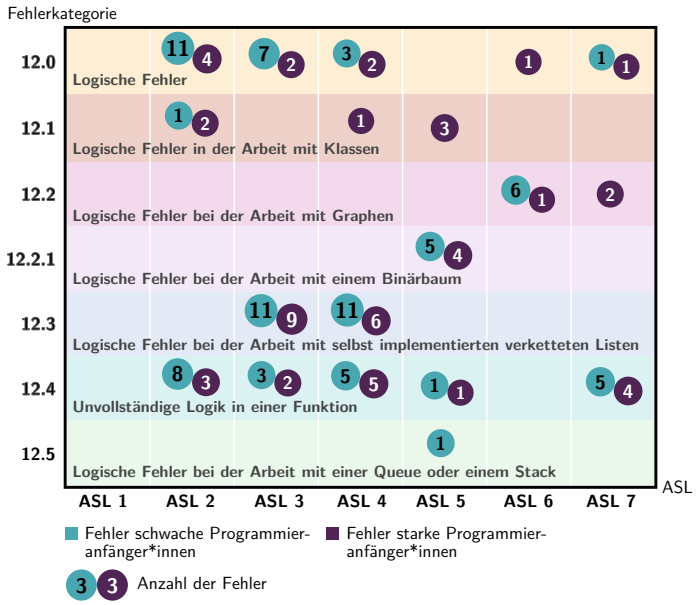


Abbildung 6.3.: Anzahl der logischen Fehler in den ASLs

Interessant bei der Übersicht der aufgetretenen Fehler in den Kategorien der logischen Fehler ist, wie diese sich über die verschiedenen ASLs hinweg verändern. Die Fehlerkategorie 8.10 wurde bei dieser Übersicht nicht mit einbezogen, da Fehler

in dieser Kategorie nur in der ersten ASL auftraten und der Ursprung der Fehler bereits diskutiert wurde.

Wie in Tabelle 6.1 dargestellt, wurde in der ASL 3 zum ersten Mal die Datenstruktur der verketteten Liste von den Studierenden implementiert. Während die Gruppe der HPS aber bereits in der ASL 4 mit dieser Datenstruktur weniger Probleme hat, wurden für die Gruppe der LPS gleich viele Fehler bei der Verwendung der Datenstruktur registriert. Unklar ist, ob ein Lerneffekt bei der Implementierung von verketteten Listen bei der Gruppe der LPS eingetreten ist, da die Datenstruktur in der ASL 5 nicht mehr implementiert werden musste.

Eine Datenstruktur, die ebenfalls mehrfach in den ASLs gefordert wurde, waren Graphen. Wie in Tabelle 6.1 dargestellt, wurde in der ASL 5 die Implementierung eines binären Suchbaums gefordert, was die Spezialform eines Graphen darstellt. In der ASL 6 musste dann ein einfacher und in der ASL 7 ein gewichteter Graph implementiert werden. Diese Datenstrukturen weisen genügend Ähnlichkeit auf, um die Fehler, die in der Arbeit mit der Datenstruktur auftraten, vergleichen zu können. So wurden in der ASL 5 ähnlich viele Fehler in der Arbeit mit Graphen von LPS und HPS registriert. Beide Gruppen an Studierenden hatten also Probleme bei der Implementierung der Datenstruktur. Interessant ist, dass bei der Implementierung eines Graphens in der ASL 6 ein Fehler mehr in den Abgaben der LPS auftrat. Parallel dazu sinkt die Fehleranzahl der logischen Fehler bei der Arbeit mit einem Graphen in der Gruppe der HPS. In der ASL 7 wurden in der Gruppe der LPS keine Fehler mehr in der Arbeit mit Graphen registriert. Die vergleichsweise hohe Anzahl der nicht vollständig implementierten Logik kann darauf hindeuten, dass die gewichteten Graphen nur zum Teil implementiert wurden.

Es kann also geschlussfolgert werden, dass die Gruppe der HPS bei der Umsetzung der Graphen sicherer wurde, was durch das Auftreten von weniger logischen Fehlern bewiesen werden kann. Bei der Gruppe der LPS kann diese Aussage nicht mit Sicherheit getroffen werden. Anhand der logischen Fehler könnte geschlussfolgert werden, dass die Studierenden der LPS mit den Prinzipien der Graphen generell Schwierigkeiten in der Programmierung haben.

Resümee Annahme 4

Die Annahme, dass LPS mehr absolute Fehler in Bezug auf neue Datenstrukturen machen, kann zumindest teilweise bestätigt werden.

Dazu muss angemerkt werden, dass wie in Tabelle 6.2 dargestellt wurde, sich die Anzahl der Fehler nicht so stark unterscheidet wie angenommen. Die Unterschiede sind meist eher klein. Da sie so groß ist wie angenommen, könnte es sein, dass

die Ergebnisse eher widerspiegeln, dass Studierende bei bestimmten Datenstrukturen allgemein Probleme mit deren Implementation haben. Besonders sticht die Datenstruktur der verketteten Listen heraus, bei der beide Gruppen die meisten spezifischen logischen Fehler machten. Interessant ist, dass, obwohl die logischen Fehler den Großteil der aufgetretenen Fehler ausmachen, diese nicht spezifisch bei der Implementierung einer neuen Datenstruktur auftreten.

6.2.1. RQ2: Welche Unterschiede gibt es beim Umfang der logischen Fehler im Umgang mit neu eingeführten Programmierkonzepten zwischen LPS und HPS?

Die Frage, welche Unterschiede es im Umfang der logischen Fehler bei neu eingeführten Programmierkonzepten zwischen den LPS und HPS gibt, ist nicht so einfach zu beantworten. Wie zuvor beschrieben, unterscheiden sich beide Gruppen nur minimal in der Anzahl der gemachten Fehler bei der Nutzung neuer Programmierkonzepte. Deutlicher fallen dafür die Fehler in der Nutzung der Datenstrukturen in den ASLs aus.

Dabei konnte festgestellt werden, dass die Gruppe der HPS in einer zweiten ASL mit derselben genutzten Datenstruktur weniger Fehler machen als LPS. Tatsächlich blieben die verzeichneten Fehler der LPS in der Verwendung von verketteten Listen und Graphen ähnlich, wobei sich bei der Nutzung von Graphen die Art der Graphen änderte. Daraus kann der Schluss gezogen werden, dass HPS sicherer bei der Implementierung der Datenstrukturen werden, während LPS weiterhin Probleme haben.

6.3. Fazit

In diesem Abschnitt sollen die in der Diskussion hervorgehobenen Erkenntnisse zusammengefasst und in Relation zum Kurs Datenstrukturen gesetzt werden. Folgend sollen Vorschläge zur Verbesserung der Lehre vorgenommen werden.

Zusammenfassung der Ergebnisse

Durch die Auswertung der Ergebnisse wurde in Unterabschnitt 6.1.4 ergeben hat, zeichnen sich HPS dadurch aus, dass der Hauptteil der von ihnen verursachten Fehler logische Fehler sind. Syntaktische oder logische Fehler treten in von ihnen geschriebenem Code seltener auf. Insgesamt machen sie eine kleinere Bandbreite an Fehlern als Studierende der Gruppe LPS. Fehler, die von HPS verursacht werden,

tendieren dazu, Fehler in komplexen Themenbereichen der Programmierung zu sein.

Die Gruppe der LPS zeichnet sich ebenfalls dadurch aus, dass ein Großteil der auftretenden Fehler logische Fehler sind. Anders als die HPS verursachen sie jedoch auch semantische Fehler, als auch Fehler in Umgang mit Klassen und Datentypen in einem relevanten Ausmaß. Allgemein kann man beide Gruppen daran unterscheiden, dass LPS wesentlich mehr Fehler als HPS in mehr Kategorien machen.

Wie bereits erwähnt macht die Anzahl der logischen Fehler in beiden Gruppen die am häufigsten auftretende Fehlerart aus. Beide Gruppen haben ebenfalls Probleme beim Umgang mit Klassen und Datentypen, wobei der Umfang dieser Fehler in der Gruppe der LPS wesentlich größer ist als der der HPS.

Beide Gruppen unterscheiden sich ebenfalls in der Entwicklung der Arten der Fehler und der Umfang der Fehler über den Verlauf des Semesters. LPS in der vorliegenden Studie machten vor allem in den ersten ausgewerteten Abgaben Fehler in weniger komplexen Bereichen der Programmierung. In den letzten drei Abgaben konnte eine Annäherung sowohl in der Anzahl der Arten der Fehler als auch bei der Anzahl der Fehler beobachtet werden.

Es kann angenommen werden, dass HPS Erfahrung in der Programmierung besitzen. Im Verlauf des Kurses Datenstrukturen scheint es, als könnte die Gruppe der LPS ihr Wissensdefizit grundsätzlich verringern.

Fraglich ist, ob dies auch spezifisch für die Kategorie der logischen Fehler zutrifft. Der Umfang der logischen Fehler für eine spezifische Datenstruktur, die in einer ASL neu auftaucht, ähneln sich in der Betrachtung. Sieht man sich Datenstrukturen wie verkettete Listen und Graphen an, die in mehreren ASLs gefordert werden, bleibt die Anzahl der Fehler in den entsprechenden Fehlerkategorien in der Gruppe der LPS ähnlich. Die Anzahl der Fehler der HPS hingegen nimmt in den entsprechenden logischen Fehlerkategorien ab.

Offen bleibt die Frage, warum die Logik in einigen Abgaben nicht beendet wurde. Annahmen dazu sind, dass die Zeit zur Bearbeitung nicht ausreichte oder das dafür nötige Wissen nicht vorhanden war.

Vergleich zum Forschungsstand

Ein Vergleich mit den Ergebnissen von McCall und Kölling [22] ist für die vorliegende Arbeit nur bedingt von Interesse und soll nur kurz erfolgen. Grund dafür ist, dass sich die Forschung von McCall und Kölling mit der Auswertung der aufgetretenen Fehler in Arbeiten von Programmieranfänger*innen allgemein und nicht

spezifisch mit dem Vergleich von LPS und HPS beschäftigten. Trotzdem können große Unterschiede in den Ergebnissen der vorliegenden Arbeit erkannt werden.

Kölling und McCall [22] identifizierten vorrangig Fehler in der Verwendung und Initialisierung von Variablen sowie Methoden, wie in 3.2 ersichtlich. Des Weiteren wurden einfache syntaktische und semantische Fehler als die häufig auftretenden Fehlerarten benannt. Die Studie erstellte allerdings keine separate Kategorie für logische Fehler. Dieser Aspekt ist besonders relevant, da in der vorliegenden Arbeit semantische und logische Fehler am häufigsten in den Gruppen der LPS und HPS festgestellt wurden.

Eine mögliche Erklärung für diese Diskrepanz könnte die Verwendung der IDE BlueJ sein. Wie in 3.2.3 beschrieben, unterscheidet sich diese spezialisierte IDE in ihrem Aufbau von traditionellen Entwicklungsumgebungen, was zu unterschiedlichen Ergebnissen führen könnte. Der Einsatz von BlueJ mag dazu beigetragen haben, dass die Studierenden Konzepte außerhalb der Objektorientierung weniger intensiv erlernten. Zudem könnten Unterschiede im Lehrplan im Vergleich zu den Kursen an der Technischen Universität eine Rolle spielen. Auch die Qualität der Lehre und die Vorkenntnisse der Studierenden sind als mögliche Faktoren zu berücksichtigen. Dennoch lässt sich festhalten, dass die teilnehmenden Studierenden einfache Programmierkonzepte offensichtlich besser verstanden und umgesetzt haben als dies in den Untersuchungen von McCall und Kölling der Fall war.

Dass semantische Fehler in der vorliegenden als auch in der Arbeit von McCall und Kölling als häufig auftretende Fehler identifiziert wurden, kann dafür sprechen, dass semantische Programmierkonzepte Studierenden insgesamt Probleme bereiten. Diese können als komplexe Programmierkonzepte bezeichnet werden. Wie Lathinen et al. [18] ausgeführt haben, stellen diese komplexen Themenbereiche eine Herausforderung im Erlernen des Programmierens dar.

Ebenfalls auffällig ist, dass in bestimmten Fehlerkategorien keine Fehler in den Aufgaben der Studierenden registriert wurden. Eine Ursache dafür kann sein, dass der Kurs wie in Abschnitt 4.3 dargestellt erst als zweiter Programmierkurs empfohlen wird. Es kann also davon ausgegangen werden, dass die Teilnehmenden etwas Erfahrung in der Programmierung aufweisen. Eine bestimmte Aussage zu der Ursache kann an der Stelle nicht getroffen werden.

Weiterhin soll ein Vergleich zu den Ergebnissen von Rodrigo et al. [25] vorgenommen werden. Diese Studie eignet sich für den Vergleich im Besonderen, da dort auch Vergleiche von Programmierfehlern zwischen LPS und HPS angestellt wurden. Auffällig ist, dass Rodrigo et al. keine Unterschiede in den auftretenden Programmierfehlern der Gruppen der Studierenden feststellen konnten. Alle Gruppen der Studierenden vergaßen Semikolons, schrieben Variablennamen falsch und

vergaßen geschweifte Klammern. Das Verständnis, wie lang eine Variable besteht, wie diese deklariert und verändert wird, wie Methoden angelegt und verändert werden bereitete den Studierenden ebenfalls Probleme. Auch in Bereichen der Objektorientierung machten Studierende Fehler, wie zum Beispiel beim Erstellen von Konstruktoren oder dem Initialisieren von objekteigenen Attributen.

Die von Rodrigo et al. [25] registrierten syntaktischen Fehlern, finden sich in dieser Ausprägung nicht unter den Fehlern die Studierende der Gruppe der HPS verursachen. Wie unter Kapitel 5 aufgeführt, traten diese Fehler vor allem in den Abgaben der Gruppe der LPS auf. Diese fanden sich vermehrt in den ersten Abgaben der ASLs auf und in einem geringen Umfang als semantische oder logische Fehler.

In der Studie von Rodrigo et al. [25] wurde nicht erfasst, ob Studierende Erfahrung in der Programmierung besitzen. Es ist anzunehmen, dass Studierende am Kurs teilnehmen, die diese bereits besitzen. Damit ist die Ausgangslage eine ähnliche wie in der vorliegenden Arbeit. Ein Grund für den Unterschied der Fehler könnte sein, dass mehr teilnehmende Studierende Erfahrung in der Programmierung besitzen und grundlegende Programmierkonzepte beherrschen. Diese Annahme wird u.a. davon gestützt, dass Liao et al. Erfahrung als einen Erfolgsfaktor beim Bestehen eines Programmierkurses ausmachen. Ebenfalls könnten Unterschiede in der Art der Lehre sowie deren Inhalte Grund für die abweichenden Ergebnisse sein. Ein Faktor, der ebenfalls beachtet werden sollte, ist, dass auch in den von Rodrigo et al. betrachteten Kursen die IDE BlueJ eingesetzt wird.

Eine Erkenntnis aus der Arbeit von Rodrigo et al. [25], die für die vorliegende Arbeit interessant ist, lautet, dass Studierende aus der Gruppe der starken Programmieranfänger*innen Syntaxfehler effizienter auflösen können oder diese gar nicht erst machen. Dies könnte eine Erklärung für die festgestellten Unterschiede gerade in den ersten Abgaben der ASLs erklären. Eine Annahme, die im Kontext des vorliegenden Kurses getroffen werden kann, ist, dass Studierende der Gruppe der LPS diese Fähigkeit im Verlauf des Semesters erwerben. Das könnte erklären, warum diese Fehler am Ende des Semesters nicht mehr in ihren Abgaben auftauchen.

Bedeutung für den Kurs Datenstrukturen

Wie in Unterabschnitt 6.1.3 dargestellt wird, gibt es über den Verlauf des Semesters eine Veränderung in den von den Studierenden eingereichten Abgaben. Besonders in der Gruppe der LPS konnte eine positive Entwicklung in der Art der Programmierfehler sowie in ihrem Umfang festgestellt werden. Ebenfalls wurden in der Gruppe der HPS im Verlauf des Semesters weniger Fehlerarten in einem kleineren Umfang registriert. Diese Entwicklung kann als ein Erfolg der Lehre im

Kurs Datenstrukturen betrachtet werden. Bei der Auswertung wurden auch Problemfelder festgestellt, die im Kurs berücksichtigt werden sollten, um die Qualität der Lehre weiterhin zu verbessern.

Eine Feststellung betrifft die Wiederholung von Fehlern in den Abgaben der Studierenden. So verwendete zum Beispiel eine Person der Gruppe der LPS in den Abgaben der ASLs den Vergleichsoperator, um Variablen des Typs `String` miteinander zu vergleichen. Damit verursachte dieser den Großteil der in Kategorie *8.8* erfassten Fehler. Auch in den einzelnen Abgaben traten verschiedene Fehler wiederholt auf. Wie in Abschnitt 6.1.3 erwähnt, stammen alle in der Fehlerkategorie *8.10* registrierten Fehler der Gruppe der HPS von einem Studierenden. Lehrpersonal könnte Studierenden, die Fehler in einem großen Umfang oder wiederholt verursachen, Informationen zur Korrektur dieser individuell zur Verfügung stellen. Die Erkennung der Fehler sowie Notizen zu diesen könnten bereits bei der Korrektur und Benotung der Abgaben erstellt werden, um einen erheblichen Mehraufwand an Arbeit zu vermeiden. Durch eine personalisierte Ansprache der Studierenden können diese ihre falschen Annahmen zeitnah korrigieren.

Insgesamt wird empfohlen, die Abgaben in einer ähnlichen Art wie der hier vorliegenden Arbeit über den Verlauf des Semesters zu überwachen. Dabei kann die Einteilung der Fehler weniger formell oder in einer weniger detaillierten Form als in der vorliegenden Arbeit erfolgen. Eine Einteilung der Fehler in die Überkategorien könnte dafür ausreichen. Mit dem Erfassen der Fehler der Studierenden könnten Studierende der Gruppe der LPS frühzeitig erkannt und in ihrem Lernprozess entsprechend unterstützt werden. Wünschenswert wäre es, wenn damit auch die Studierenden angesprochen werden, die dazu tendieren, den Kurs abzubrechen. Die Ergebnisse der Forschung legen die Annahme nahe, dass auch Studierende, die kein gutes Verständnis von komplexen Programmierkonzepten besitzen, diese im Verlauf des Kurses erlernen. Daraus könnte geschlussfolgert werden, dass auch Studierende, die den Kurs abbrechen, einen Lernfortschritt erzielen würden, wenn sie die Bearbeitung der Aufgaben fortsetzten.

Ob ein Lernfortschritt in der Gruppe der LPS in der logischen Handhabung der entsprechenden Datenstrukturen vorhanden ist oder nicht, kann aus den hier vorliegenden Daten nicht ohne Zweifel beantwortet werden. Dazu werden weiterführende Daten der Studierenden benötigt. Insgesamt wird angeraten, eine Auswertung der Abgaben der Studierenden mit Interviews über die wahrgenommene Schwierigkeit der Abgaben zu führen, um einen besseren Einblick in das Verständnis der Studierenden über den Inhalt im Kurs zu erhalten.

Ein besonderes Augenmerk bei der Gestaltung der Lehre sollte ebenfalls auf die Datenstrukturen Klassen, verkettete Listen und Graphen gelegt werden. Beide

Gruppen der Studierenden wiesen Probleme in der Implementation dieser auf. Im Besonderen fällt auf, dass die erste ASL, welche eine Arbeit der Datenstruktur Klasse erfordert, eine Vielzahl von diesen beinhaltet. Ein Projekt mit weniger Klassen könnte es den Studierenden der Gruppe der LPS erleichtern, ein Verständnis für diese Datenstruktur zu entwickeln. Wenn die Ergebnisse der Arbeit von Lathinen et al. [18] in Betracht gezogen werden, ergeben sich weitere Verbesserungsvorschläge bei der Lehre der als problematisch identifizierten Datenstrukturen. In der Arbeit wurde festgestellt, dass das individuelle Schreiben von Programmiercode am besten dazu geeignet ist, um auch komplexe Programmkonzepte zu verstehen. Eine Verbesserung könnte demnach darin bestehen, mehr Übungsaufgaben im Umgang mit den Datenstrukturen anzubieten. Parallel dazu könnten den Studierenden Lösungen zur Einsicht angeboten werden. Programmcode wurde von Lathinen et al. als das am hilfreichsten empfundene Arbeitsmaterial herausgestellt. Diese Maßnahmen könnten somit eine Verbesserung des Verständnis der zuvor genannten Datenstrukturen erwirken.

7. Einschränkung der Validität

Um die Ergebnisse der Arbeit möglichst genau einzuordnen, werden im Folgenden Einschränkungen und Bedenken zu deren Validität erörtert. Dabei wird im Abschnitt 7.1 damit begonnen, einzuordnen, wie geeignet das Vorgehen in der Arbeit für die Beantwortung der Forschungsfragen ist. Anschließend wird im Abschnitt 7.2 diskutiert, inwiefern die vorliegenden Daten eine valide Aussage über die gestellte Forschungsfrage geben können. Darauffolgend werden im Abschnitt 7.3 Einschränkungen zur Allgemeingültigkeit der Ergebnisse getroffen.

7.1. Konstruktvalidität

In dieser Arbeit werden unter Abschnitt 3.2 diverse Fehlerkategorisierungen präsentiert, welche anschließend im Abschnitt 4.6 hinsichtlich ihrer Anwendbarkeit für die vorliegenden Daten sorgfältig verglichen werden. Die darauf aufbauende Auswahl der verwendeten Fehlerkategorisierung ist für die Auswertung der verwendeten Daten geeignet. Wie in Abschnitt 3.2 erläutert, dient die Fehlerkategorisierung in der einschlägigen Forschung häufig dazu, Missverständnisse und Wissenslücken der Studierenden zu identifizieren. Zukünftig könnten Interviews zur Ergänzung dieser Auswertungen durchgeführt werden, um weitere Einblicke in die Ursachen der Programmierfehler in den von den Studierenden abgegebenen Programmcode zu gewinnen.

Die Einordnung der Studierenden in die Gruppe der LPS und HPS erfolgte auf Basis der erlangten Benotung bei der Abgabe der ASLs. Analog zur Studie von Rodrigo et al. [25] wurde die Einteilung der Studierenden basierend auf deren Leistung im Kurs vorgenommen. Dabei ist anzumerken, dass die Bewertung nicht nach klaren wissenschaftlichen Regeln während des Verlaufs des Kurses erfolgte. Insofern ist nicht klar, ob diese die Konzepte der schwachen und starken Programmieranfänger*innen widerspiegelt. In zukünftigen Arbeiten könnte ein eigener wissenschaftlich basierter Bewertungsmaßstab im Vorfeld entwickelt werden. Dieser sollte dann als Grundlage zur Einordnung der Studierenden in LPS und HPS unabhängig von der erreichten Note im Kurs genutzt werden.

Der Ausschluss von Studierenden mit der Note 5.0 aus der LPS-Gruppe, wie in Abschnitt 4.4beschrieben, lenkt die Aufmerksamkeit auf die Notwendigkeit einer präzisen Bewertungsmethodik. Grund für den Ausschluss ist, dass für die Studierenden, die die Bearbeitung der Abgaben abgebrochen haben, nicht ausreichend Daten für eine Auswertung vorliegen. Es könnte argumentiert werden, dass die Gruppe an Studierenden, die als LPS eingeordnet wurden, auch als mittelmäßige Programmierer*innen bezeichnet werden könnten. Aus den registrierten Unterschieden zwischen den betrachteten Gruppen lässt sich schlussfolgern, dass es zwischen ihnen große Unterschiede gab. In diesem Kontext eignen sich die als LPS kategorisierten Studierenden, um schwache Programmieranfänger*innen zu repräsentieren. Eine vergleichende Auswertung mit den starken Programmieranfänger*innen ist somit eine gültige Analysemethode.

7.2. Interne Validität

Um einer Beeinflussung durch Vorurteile vorzubeugen, wurden die Abgaben der Studierenden anonymisiert. So wurde verhindert, dass bestimmte Abgaben aufgrund einer angenommen Herkunft oder eines Geschlechts anders ausgewertet werden.

Eine Einschränkung der internen Validität muss für die Zuordnung von Fehlern in Fehlerkategorien getroffenen werden. Bei der unter Unterabschnitt 3.2.4 vorgestellten Fehlerkategorisierung arbeiteten mehrere Forscher*innen an dem Projekt und es erfolgte eine Überprüfung der Einordnung der Fehler in Kategorien. Dies konnte in dieser Arbeit nicht reproduziert werden. Daher ist die Einschätzung über die Zuordnung von konkreten Fehlern in eine Fehlerkategorie lediglich die Meinung eines Programmierenden dar.

Im Kurs Datenstrukturen 2023 wurde erstmalig den Studierenden aktiv die Nutzung des Sprachmodells ChatGPT nahegelegt. Dieses wurde als ein Werkzeug vorgestellt, welches beim Erlernen des Programmierens helfen kann. Daraus können gleich mehrere Probleme in der internen Validität entstehen. Zum einen kann nicht zweifelsfrei bewertet werden, ob Teile im Code der Studierenden nicht von dem Sprachmodell generiert wurden. So könnte eine unbekannte Anzahl an Fehlern von diesem generiert worden sein. Zum anderen kann die Nutzung des Sprachmodells den Lernprozess der Studierenden verändern. Es ist möglich, dass sich LPS und HPS auch darin unterscheiden, wie effektiv sie das Tool nutzen konnten.

Dem anschließend ist das Sprachmodell nicht der einzige externe Faktor, der die Daten verzerren könnte. Ebenfalls kann nicht nachvollzogen werden, ob die Abgaben wirklich von den Studierenden programmiert wurden. Es ist zwar verboten,

Programmcode abzugeben, welcher von anderen Personen geschrieben wurde, jedoch ist eine umfassende Prüfung an dieser Stelle nicht möglich. Teile der Aufgaben oder ganze Abgaben der ASLs könnten von externen Personen geschrieben worden sein.

7.3. Externe Validität

Die größte Einschränkung für die externe Validität stellt die Menge an ausgewerteten Daten dar. Den Gruppen der LPS und HPS konnten jeweils nur drei Studierende aus dem Kurs zugeordnet werden. Weiterhin stellt der Kurs Datenstrukturen einen sehr spezifischen Kontext zum Erlernen des Programmierens dar. Die Inhalte sind im Kurs eingeschränkt, wie in Abschnitt 4.3 dargestellt wurde. Damit können die in der Arbeit gewonnen Erkenntnisse ohne Einschränkung nur auf den Kurs, in dem die Daten generiert wurden, angewandt werden. Vor der Anwendung der Ergebnisse auf andere Kurse an der Universität sollte eine Prüfung bezüglich der Vergleichbarkeit angestellt werden.

Weiterhin zeigt die Teilnahme an dem Kurs nur einen spezifischen Ausschnitt aus dem Lernprozess der Studierenden. Ebenfalls kann die bereits vorhandene Erfahrung mit Programmierung der teilnehmenden Studierenden variieren.

Die Ergebnisse der vorliegenden Arbeit können auch nur für Studierende aussagekräftig sein, welche die Programmiersprache Java verwenden. Auf Betrachtungen anderer Programmiersprachen sind die vorliegenden Ergebnisse nur bedingt anwendbar.

8. Zusammenfassung und Ausblick

Es folgt nun eine Zusammenfassung der Forschungsarbeit in Abschnitt 8.1 sowie eine Aussicht der Forschung in Abschnitt 8.2, die dieser Arbeit angeschlossen werden könnte.

8.1. Zusammenfassung

Ziel der Forschungsarbeit war es, Unterschiede in den Fehlern von LPS und HPS in den Abgaben der ASLs des Kurses Datenstrukturen zu untersuchen. Die zwei gestellten Forschungsfragen lauteten: RQ1: Welche Unterschiede gibt es in der Anzahl und der Art der gemachten Programmierfehler zwischen LPS und HPS? RQ2: Welche Unterschiede gibt es beim Umfang der Fehler im Umgang mit neu eingeführten Programmierkonzepten zwischen LPS und HPS?

Auf Grundlage der vorhandenen Abgaben und erreichten Noten im Kurs wurden Studierende den Gruppen LPS und HPS zugeordnet. Für die Auswertung der Fehler wurden verschiedene Arten der Fehlerkategorien betrachtet und verglichen. Es wurde sich für die Fehlerkategorisierung von McCall und Kölling [22] entschieden. Grundlage für die Entscheidung war u.a. die feine Gliederung der Fehlerkategorien und deren einfache Erweiterbarkeit, die es erlaubt, die Kategorien für die vorliegende Arbeit anzupassen.

Die Auswertung der Abgaben der Studierenden erfolgte nach der Anonymisierung derselben. Zunächst wurden diese unter Zuhilfenahme der Tests, die zum Zwecke der Bewertung erstellt wurden, ausgewertet und alle aufgetretenen Fehler gesammelt. Anschließend wurden diese den Fehlerkategorien zugeordnet. Sofern nötig wurde die Fehlerkategorisierung erweitert.

Die übergeordnete Fehlerkategorie, in der am meisten Fehler inklusive der Unterkategorie notiert wurden, war sowohl bei den LPS als auch den HPS die der logischen Fehler. Weiterhin war auffällig, dass die Anzahl wie auch die Arten der Fehler sich in den einzelnen ASLs stark unterschieden.

Bei der Auswertung der Daten wurde festgestellt, dass LPS wesentlich mehr unterschiedliche Fehler machen als HPS. Dabei waren die Arten der von den HPS

registrierten Fehler vor allem logische und semantische Fehler. Weiterhin wurden Fehler bei der Arbeit mit Klassen und Datentypen festgestellt. Die von den LPS verursachten Fehler waren hingegen wesentlich vielfältiger und umfassten auch simplere Bereiche der Programmierung wie z.B. die Arbeit mit Methoden und Variablen.

In den Abgaben der Studierenden der Gruppe der LPS wurden 263 Fehler registriert, in den Abgaben der Studierenden der Gruppe der HPS hingeben nur 112. Trotz der großen Varianz an Fehlern sind semantische und logische Fehler sowie die Arbeit mit Datentypen und Klassen die Fehlerkategorien, die am häufigsten bei LPS auftreten. Auch in der Gruppe der HPS dominierten die logischen Fehler. Sowohl semantische Fehler als auch Fehler in der Arbeit mit Datentypen und Klassen, kamen, im Vergleich zur Gruppe der LPS, weniger häufig vor.

In der Entwicklung der Fehler konnte festgestellt werden, dass die Arten sowie der Umfang der Fehler sich über den Verlauf der Abgaben angleichen. Auch wenn LPS mehr Fehler verursachen als HPS und eine höhere Varianz an Fehlern verursachen, war dieser Unterschied bei der letzten Abgabe nicht mehr so ausgeprägt wie in der zweiten Abgabe. Die erste Abgabe stellt eine Ausnahme dar.

Im Bezug auf den Kurs Datenstrukturen wurde festgestellt, dass die Erkenntnisse als ein Lernerfolg der Kurses interpretiert werden können. Der Umgang mit Klassen und Datentypen sollte dennoch eine Vertiefung im Kurs erhalten, da beide Gruppen, wenn auch in anderem Ausmaß, damit Probleme aufwiesen. Es wird ebenfalls eine Überwachung der Fehler während der Abgaben empfohlen, um Studierende der Gruppe der LPS zu registrieren, spezifisch zu unterstützen und ein Abbrechen des Kurses zu vermeiden.

8.2. Ausblick

An diese Arbeit anschließend sollte eine Betrachtung erfolgen, wie die vorgeschlagenen Verbesserungen in den Kurs Datenstrukturen implementiert werden können. Besonders die Vermittlung der Programmierkonzepte, die häufig fehlerhaft umgesetzt werden, sollte detaillierter untersucht werden.

Allgemein wird eine Wiederholung der Betrachtung der Programmierfehler der Studierenden sowie die Auswertung der Unterschiede von schwachen und starken Programmieranfänger*innen als sinnvoll erachtet. Im Speziellen sollte evaluiert werden, ob die implementierten Änderungen den erwünschten Lernerfolg erzielen.

Bei einer erneuten Evaluation des Kurses könnten Interviews mit den Studierenden angeschlossen werden. Dabei kann sich an den Interviews von Rodrigo et al. [25]

und Lahtinen et al. [18] orientiert werden. So könnten mögliche zugrundeliegende Verständnisprobleme der Studierenden ermittelt werden, um den Kurs weiter zu verbessern und die Fehlerquote zu senken.

Literaturverzeichnis

[1] Ahadi, A., Lister, R., Mathieson, L.: Syntax error based quantification of the learning progress of the novice programmer. In: ITiCSE 2018: Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (2018)

[2] Ahmadzadeh, M., Elliman, D.G., Higgins, C.A.: An analysis of patterns of debugging among novice computer science students. In: ACM SIGCSE Bulletin. vol. 37 (2005)

[3] Albrecht, E., Grabowski, J.: Sometimes it's just sloppiness - studying students' programming errors and misconceptions. In: SIGCSE '20: Proceedings of the 51st ACM Technical Symposium on Computer Science Education (2020)

[4] Alzahrani, N., Vahid, F., Edgcomb, A.D., Lysecky, R., Lysecky, S.: An analysis of common errors leading to excessive student struggle on homework problems in an introductory programming course. In: ASEE Annual Conference and Exposition, Conference Proceedings (2018)

[5] Bayman, P., Mayer, R.E.: Using conceptual models to teach basic computer programming. Journal of Educational Psychology (1988)

[6] Brown, N.C.C., Altadmri, A.: Novice java programming mistakes: Large-scale data vs. educator beliefs. ACM Transactions on Computing Education 17(2) (2017)

[7] Brown, N.C., Altadmri, A.: Investigating novice programming mistakes: educator beliefs vs. student data. In: ICER '14: Proceedings of the tenth annual conference on International computing education research (2014)

[8] Brown, N.C., Altadmri, A.: 37 million compilations: Investigating novice programming mistakes in large-scale student data. In: SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education (2015)

[9] Brown, P.J.C.: Error messages: the neglected area of the man/machine interface. Communications of the ACM 26 (1983)

[10] Chinn, D.D., Sheard, J., Carbone, A., Laakso, M.J.: Study habits of cs1 students: what do they do outside the classroom? In: ACE '10: Proceedings of the Twelfth Australasian Conference on Computing Education. vol. 103 (2010)

[11] Denny, P., Luxton-Reilly, A., Tempero, E.: All syntax errors are not equal. In: ITiCSE '12: Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (2012)

[12] Dy, T., Rodrigo, M.M.: A detector for non-literal java errors. In: Koli Calling '10: Proceedings of the 10th Koli Calling International Conference on Computing Education Research (2010)

[13] Ettles, A., Luxton-Reilly, A., Denny, P.: Common logic errors made by novice programmers. In: ACE '18: Proceedings of the 20th Australasian Computing Education Conference (2018)

[14] Hristova, M., Misra, A., Rutter, M., Mercuri, R.T.: Identifying and correcting java programming errors for introductory computer science students. In: SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education (2003)

[15] Jackson, J., Cobb, M., Carver, C.: Identifying top java errors for novice programme. In: Proceedings Frontiers in Education 35th Annual Conference (2005)

[16] Jadud, M.C.: Methods and tools for exploring novice compilation behaviour. In: ICER '06: Proceedings of the second international workshop on Computing education research (2006)

[17] Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: The bluej system and its pedagogy. Computer Science Education 13(4) (2003)

[18] Lahtinen, E., Ala-Mutka, K., Järvinen, H.M.: A study of the difficulties of novice programmers. In: ACM SIGCSE Bulletin. vol. 37 (2005)

[19] Liao, S.N., Shah, K., Griswold, W.G., Porter, L.: A quantitative analysis of study habits among lower- and higher-performing students in cs1. In: ITiCSE '21: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (2021)

[20] Mase, M.B., Nel, L.: Common code writing errors made by novice programmers: Implications for the teaching of introductory programming. In: Communications in Computer and Information Science. vol. 1461 (2022)

[21] McCall, D., Kölling, M.: Meaningful categorisation of novice programmer errors. In: Conference: Frontiers In Education Conference (FIE) (2014)

[22] McCall, D., Kölling, M.: A new look at novice programmer errors. Special Section on ML Education and Regular Articles (2019)

[23] Qian, Y., Lehman, J.: An investigation of high school students' errors in introductory programming: A data-driven approach. Journal of Educational Computing Research 58 (2020)

[24] Robins, A.V.: Novice Programmers and Introductory Programming. Cambridge University Press (2019)

[25] Rodrigo, M.M.T., Andallaza, T.C.S., Castro, F.E.V.G., Marc Lester V. Armenta, T.T.D., Jadud, M.C.: An analysis of java programming behaviors, affect, perceptions, and syntax errors among low-achieving, average, and high-achieving novice programmers. Journal of Educational Computing Research (2013)

[26] Shinners-Kennedy, D., Fincher, S.A.: Identifying threshold concepts: from dead end to a new direction. In: ICER '13: Proceedings of the ninth annual international ACM conference on International computing education research (2013)

[27] Sorva, J.: Visual Program Simulation in Introductory Programming Education. Ph.D. thesis, Aalto University Schools of Technology (2012)

[28] Tabanao, E.S., Rodrigo, M.M.T., Jadud, M.C.: Predicting at-risk novice java programmers through the analysis of online protocols. In: ICER '11: Proceedings of the seventh international workshop on Computing education research (2011)

A. Anhang

Anhang: Fehlerkategorisierung nach Hristova et al.

Syntaxfehler

1. Verwechseln des Zuweisungs-Operators (=) mit dem Vergleichsoperator (==).
2. Vergleich von zwei Stringvariablen mit dem Vergleichsoperator (==) anstatt der Stringfunktion (`.equals`). Der Vergleichsoperator vergleicht die Speicheradresse der beiden Variabeln anstatt deren Inhalt.
3. Falsche Verwendung von verschiedenen Klammern, den einfachen und doppelten Anführungszeichen. Diese Fehlerkategorie beinhaltet Fehlen von schließenden Klammern, sowie falsche Verwendung von Zitierungszeichen oder falscher Einsatz von Klammern. Zum Beispiel wenn eine einfache Klammer zum öffnen einer Funktion anstatt den geschweiften Klammern genutzt wird.
4. Verwechslung von Kurzschluss-Evaluierung (`&&`, ||) mit logischen Operatoren (`&`, |).
5. Verwendung eines Semikolon nach einer if-Anweisung oder einer Schleifenanweisung. Wodurch Code nach der ursprünglichen Anweisung nicht ausgeführt wird.
6. Verwendung von falschen Separatoren in einer for- Schleife.
7. Verwendung von geschweiften Klammern bei einer if-Anweisung anstatt von einfachen Klammern.
8. Nutzung eines Java keywords als Name einer Variable oder Methode.
9. Funktionsaufruf mit der falschen Anzahl an übergebenen Argumenten.
10. Parameterklammern nach einem Funktionskopf werden vergessen.
11. Setzen eines Semikolons nach einem Methodenkopf.
12. Verwendung eines Leerzeichens zwischen dem Objekt und einer aufgerufenen Funktion.

13. Falsche Zeichenreihenfolge bei einem Vergleichsoperator (<=, >=).

Semantische Fehler

1. Aufruf einer statischen Klassenfunktion auf einem Klassenobjekt.

Logische Fehler

1. Fehler bei der Typenkonvertierung. Dieser Fehler kann sich dadurch auszeichnen, dass die einfachen Klammern bei einer Typenkonvertierung nicht genutzt wurden und so einfach eine neue Variable deklariert wird. Es kann aber auch sein, dass einfach eine integer Division stattfindet. Ein Fehler beim Lernen des Programmierens besteht darin, die Unterschiede von verschiedenen Zahlentypen nicht zu verstehen.
2. Ergebnis einer Methode wird nicht aufgefangen. Dies entsteht wenn Funktionen aufgerufen werden, der Rückgabewert aber nicht in einer Variable aufgefangen wird.
3. In einer nicht void-Methode erfolgt keine Rückgabe.
4. Missverständnis zwischen dem Deklarieren von Variablen und der Übergabe von Variabel an eine Funktion. Während in einem Funktionskopf Variablen, die erwartet werden deklariert werden müssen, muss das bei der Übergabe der Variablen beim Funktionsaufruf nicht getan werden. Wenn dieser Unterschied nicht verstanden wird, kann es zu Fehlern führen.
5. Variable, die den Rückgabewert einer Funktion auffangen soll, ist inkompatibel zum Rückgabewert der Funktion.
6. Es werden nicht alle Funktionen, die ein Interface vorgibt implementiert.

[14]

Anhang: Häufigkeit der Programmierfehler von Brown und Altadmri 2014

Tabelle A.1.: Auftretenshäufigkeit der Programmierfehler in der Studie von Brown und Altadmri 2014 [7].

	Mistake	**Frequency**	**Error Type**
C	Unbalanced parentheses, curly or square brackets and quotation marks, or using these different symbols interchangeably.	404560	Syntax
I	Invoking methods with wrong arguments (e.g. wrong types).	165832	Type
O	Control flow can reach end of non-void method without returning.	137230	Semantic
N	A method that has a non-void return type is called and its return value ignored/discarded.	86107	Semantic
A	Confusing the assignment operator (=) # with the comparison operator.	68254	Syntax
B	Use of == instead of .equals to compare strings.	45012	Semantic
M	Trying to invoke a non-static method as if it was static.	30754	Semantic
R	Class claims to implement an interface, but does not implement all the required methods.	24846	Semantic
P	Including the types of parameters when invoking a method.	21694	Syntax
E	Incorrect semicolon after an if selection structure before the if statement or after the for or while repetition structure before the respective for or while loop.	20264	Syntax
K	Incorrect semicolon at the end of a method header.	16156	Syntax
Q	Incompatible types between method return and type of variable that the value is assigned to.	14371	Type
D	Confusing 'short-circuit' evaluators (&& and ||) with conventional logical operators (& and |).	11212	Syntax
J	Forgetting parentheses after a method call.	8332	Syntax

Tabelle A.2.: Auftretenshäufigkeit der Programmierfehler in der Studie von Brown und Altadmri 2014, fortgesetzt[7].

	Mistake	**Frequency**	**Error Type**
L	Getting greater than or equal/less than or equal wrong, i.e. using => or =< instead of >= and <=.	1916	Syntax
F	Wrong separators in for loops (using commas instead of semicolons).	1171	Syntax
H	Using keywords as method or variable names.	415	Syntax
G	Inserting the condition of an if statement within curly brackets instead of parentheses.	63	Syntax

Anhang: Häufigkeit der Programmierfehler von Brown und Altadmri 2015

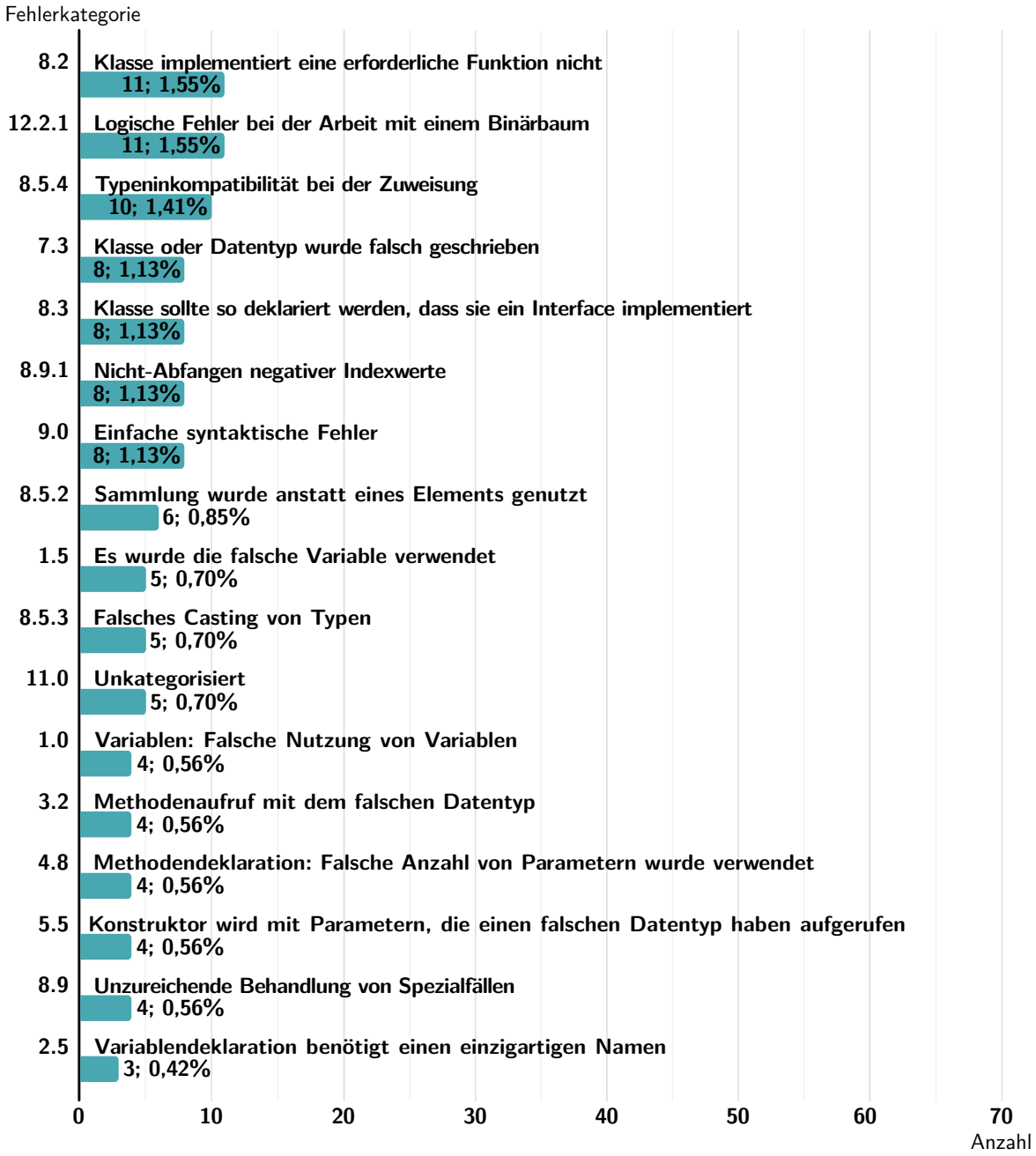
Tabelle A.3.: Auftretenshäufigkeit der Programmierfehler in der Studie von Altadmri und Brown 2015 [8].

	Mistake	**Frequency**	**Error Type**
C	Unbalanced parentheses, curly or square brackets and quotation marks, or using these different symbols interchangeably.	793232	Syntax
I	Invoking methods with wrong arguments (e.g. wrong types).	464075	Type
O	Control flow can reach end of non-void method without returning.	342891	Semantic
A	Confusing the assignment operator (=) # with the comparison operator (==).	173938	Syntax
N*	A method that has a non-void return type is called and its return value ignored/discarded.	121663	Semantic
B*	Use of == instead of .equals to compare strings.	121172	Semantic
M	Trying to invoke a non-static method as if it was static.	86625	Semantic
R	Class claims to implement an interface, but does not implement all the required methods.	79462	Semantic
P	Including the types of parameters when invoking a method.	52862	Syntax
E*	Incorrect semicolon after an if selection structure before the if-statement or after the for or while repetition structure before the respective for or while loop.	49375	Syntax
K	Incorrect semicolon at the end of a method header.	38001	Syntax
D*	Confusing 'short-circuit' evaluators (&& and ||) with conventional logical operators (& and |).	29605	Syntax
J	Forgetting parentheses after a method call.	18955	Syntax
Q	Incompatible types between method return and type of variable that the value is assigned to.	16996	Type

Tabelle A.4.: Auftretenshäufigkeit der Programmierfehler in der Studie von Altadmri und Brown 2015, fortgesetzt [8].

	Mistake	**Frequency**	**Error Type**
L	Getting greater than or equal/less than or equal wrong, i.e. using => or =< instead of >= and <=.	4214	Syntax
F	Wrong separators in for loops (using commas instead of semicolons).	1171	Syntax
H	Using keywords as method or variable names.	2719	Syntax
G	Inserting the condition of an if statement within curly brackets instead of parentheses.	118	Syntax

Anhang: Diagramm der Ergebnisse der Abgaben der Studierenden.


Fehlerkategorie
8.2
Klasse implementiert eine erforderliche Funktion nicht
11; 1,55%
12.2.1
Logische Fehler bei der Arbeit mit einem Binärbaum
11; 1,55%
8.5.4
Typeninkompatibilität bei der Zuweisung
10; 1,41%
7.3
Klasse oder Datentyp wurde falsch geschrieben
8; 1,13%
8.3
Klasse sollte so deklariert werden, dass sie ein Interface implementiert
8; 1,13%
8.9.1
Nicht-Abfangen negativer Indexwerte
8; 1,13%
9.0
Einfache syntaktische Fehler
8; 1,13%
8.5.2
Sammlung wurde anstatt eines Elements genutzt
6; 0,85%
1.5
Es wurde die falsche Variable verwendet
5; 0,70%
8.5.3
Falsches Casting von Typen
5; 0,70%
11.0
Unkategorisiert
5; 0,70%
1.0
Variablen: Falsche Nutzung von Variablen
4; 0,56%
3.2
Methodenaufruf mit dem falschen Datentyp
4; 0,56%
4.8
Methodendeklaration: Falsche Anzahl von Parametern wurde verwendet
4; 0,56%
5.5
Konstruktor wird mit Parametern, die einen falschen Datentyp haben aufgerufen
4; 0,56%
8.9
Unzureichende Behandlung von Spezialfällen
4; 0,56%
2.5
Variablendeklaration benötigt einen einzigartigen Namen
3; 0,42%
0
10
20
30
40
50
60
70
Anzahl

Anhang: Tabelle der Fehlerkategorien, die nicht in den Abgaben der Studierenden auftreten

1.1 Bei einer Objektinitialisierung wird ein schon vorhandener Variablenname genutzt, eigentlich sollte auf die bestehende Variable zugegriffen werden,
1.2 Aus statischen Kontext wird auf nicht statische Variable zugegriffen,
2.0 Variable: Falsche Variablendeklaration,
2.1 Lokale Variable wird mit nicht zulässigen Zugriffsmodifikator deklariert,
2.2 Variablendeklaration mit Name und Typ in falscher Reihenfolge,
2.3 Variablendeklaration bei der kein Name angegeben wurde,
2.4 Variablendeklaration mit Leerzeichen im Name der Variable,
3.1 Methodenaufruf wird mit Variablenname durchgeführt,
3.3 Methodenaufruf: Parameter dient gleichzeitig als Deklaration
3.4 Methodenaufruf: Methode wird mit falscher Parameteranzahl aufgerufen
3.6 Methodenaufruf: Typen der übergebenen Parameter werden angegeben,
3.9 Eine nicht statische Methode wird in einem nicht statischen Kontext aufgerufen,
4.0 Methoden: Falsche Methodendeklaration,
4.1 Methodendeklaration: Komma fehlt zwischen den Parameterangaben,
4.2 Methodendeklaration: Methodenkörper fehlt,
4.4 Methodendeklaration: Parameter wird kein spezifischer Typ zugewiesen,
4.5 Methodendeklaration: Rückgabewert sollte void sein,
4.6 Methodendeklaration: Semikolon am Ende des Methodenkopfes,
4.7 Methodendeklaration: Semikolon anstelle eines Kommas,
4.9 Exaktes Duplikat einer Methode wurde verwendet,
4.10 Geschweifte Klammer nach dem Methodenkopf nicht verwendet,
5.0 Konstruktor: Falscher Konstruktorenaufruf,
5.1 Konstruktor versucht mit Variablenname aufzurufen,
5.2 Aufruf eines nicht existierenden Kopierkonstruktors,
5.3 Konstruktor wird ohne Verwendung von `new` aufgerufen,
5.4 Konstruktor wird mit einer falschen Anzahl von Parametern aufgerufen,
5.6 Es fehlt die abschließende runde Klammer nach dem Konstruktorenaufruf,
5.7 Keine runden Klammern beim Aufruf eines Konstruktors genutzt,
5.8 Es gibt kein Leerzeichen nach `new`,
5.9 Bei der Objektinitialisierung wird ein Variablenname verwendet,

6.0 Konsturktor: Falsche Deklaration des Konsturktors,
6.1 Der Aurfur super ist nicht der erste Term im Konstruktor,
6.2 Ein Konstruktor bekommt den Rückgabewert `void`,
7.1 Klasse einer Bibliothek wurde verwendet ohne sie zu importieren,
7.2 Klasse wurde nicht definiert,
7.4 Variable in Kontext genutzt in der Rückgabewert nötig ist,
8.1 Zuweisung zu einem Element einer Sammlung,
8.5 Typenfehler,
8.5.1 Array wurde statt eines Elementes des Arrays verwendet,
8.5.5 Typeninkompatibilität indem Arithmetik auf String Datenelement angewendet oder `.length` Funktion angewendet wird, wenn Funktion bei Datentyp oder Klasse nicht vorhanden ist,
8.5.6 Wert aus untypisierten Sammlung muss umgewandelt werden,
8.6 Nicht aufgelöste Exception,
8.7 Variablenzugriff als Anweisung verwendet,
9.1 `=` anstatt eines Vergleichs `==` verwendet,
9.4 Kommentar falsch deklariert,
9.7 Öffnende runde Klammer zu viel zwischen `if-` und `else` - Anweisung,
9.11 Nicht zusammenpassende Klammern um einen Ausdruck,
9.12 Fehlender Punkt `.` zwischen Namen und Mitglied einer zuzugreifenden Klasse oder Paket,
9.13 Fehlende schließende geschweifte Klammer,
9.13.2 Fehlende schließende geschweifte Klammer am Ende einer Klasse,
9.13.3 Fehlende schließende geschweifte Klammer am Ende einer Methode,
9.14 Operator fehlt, zwischen zwei Ausdrücken,

Anhang: Aufgabenstellung ASL 1

Administratives

Abgabe Termin

24.04.2023 - 23:59

Abgabe

Ihre Abgabe darf entweder in Java oder Python geschehen. Nutzen Sie für die Abgabe das Code-Template aus dem OPAL-Bereich "Anrechenbare Studienleistung 1", dieses gibt eine Ordnerstruktur vor. Passen Sie lediglich den Platzhalter für Ihren Namen an. Abseits der Struktur ist es Ihnen erlaubt, weitere Dateien anzulegen. Sollten Sie sich nicht an diese Vorgaben halten, werden Ihnen Punkte abgezogen.
Laden Sie Ihre Lösung als ZIP-Archiv im OPAL-Bereich "Anrechenbare Studienleistung 1" hoch.

Erlaubte Klassen

Sie dürfen Klassen und Methoden aus der Standardbibliothek von Java bzw. Python nutzen. Bitte benutzen Sie keine Bibliotheken von Dritten.

Information zu Plagiaten

Es ist **erlaubt**, Lösungen aus dem Internet in Ihre Abgabe zu integrieren. Bitte **markieren** Sie jeglichen Code, der nicht von Ihnen selbst stammt, indem Sie einen Kommentar mit der Quelle des Codes (URL ist ausreichend) an die entsprechende Stelle setzen. Dies soll verhindern, dass Ihre Lösung fälschlicherweise als Plagiat erkannt wird.
Da es sich hierbei um eine Prüfungsleistung handelt, ist es notwendig, dass Sie die Aufgaben selbständig bearbeiten. Daher werden alle Abgaben untereinander auf Plagiate überprüft. Sollten Plagiate erkannt werden, gilt die Abgabe von **sämtlichen** beteiligten Parteien als ungültig.

Kompilierbarkeit und Clean Code

Es werden nur Abgaben gewertet, welche sich in einem ausführbaren Zustand befinden. Achten Sie außerdem auf die Leserlichkeit des Codes. Unverständliche oder unangemessene Bezeichner von Klassen, Variablen, etc. können zu Punkteabzügen führen.

Fragen

Bitte stellen Sie ihre Fragen im Forum im Thread ASL 1, damit alle Studierenden Zugang zu allen notwendigen Informationen erhalten.

ASL 1: Wetterbericht

Erstellen Sie eine Klasse **Forecast**, welche die Daten beliebig vieler Wetterstationen auswerten und daraus eine Wettervorhersage erstellen kann.

```
KONSTRUKTOR(int[][] rainfall, String[] descriptors)
```

Ein Konstruktor in der entsprechenden Programmiersprache, welchem die Daten der Wetterstationen in zwei Parametern übergeben werden. Zum einen die Niederschlagsdaten als zweidimensionales integer Array *rainfall* und zum anderen den allgemeinen Wetterbericht des Tages in Form eines beschreibenden Strings, im String-Array *descriptors*.

In dem Array *rainfall* entspricht jede Zeile einer Wetterstation und jede Spalte einem Tag.

In dem Array *descriptors* entspricht ebenfalls jede Spalte einem Tag. Die haben jeweils einen der Werte ”sunny”, ”rainy” oder ”thunderstorm”.

Beispiel:

Weather Stations	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Station 1	-10	22	33	19	45	75	20
Station 2	35	-6	57	8	10	-100	10
Station 3	15	20	29	39	30	75	20

Table 1: Niederschlagsdaten von 3 Wetterstationen über einen Zeitraum von 7 Tagen

Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
sunny	rainy	thunderstorm	sunny	sunny	thunderstorm	sunny

Table 2: Wetterberichte der entsprechenden 7 Tage

```
dataPreparation()
```

Die Niederschlagsdaten können fehlerhaft sein und negative Werte enthalten. Diese Methode soll die Niederschlagsdaten so bearbeiten, dass Sie später weiterverwendet werden können. Dazu müssen mögliche fehlerhafte (negative) Daten entsprechend der nachfolgenden Spezifikationen entfernt werden:

- Ist der descriptor des entsprechenden Tags ”sunny”, wird der negative Wert mit 0 ersetzt.
- Ist der descriptor des entsprechenden Tags ”rainy”, wird der negative Wert mit dem Durchschnitt aller (positiven) Werte der anderen Wetterstationen von diesem Tag ersetzt. Rechnen Sie mit ganzen Zahlen. Sollten die Daten aller Stationen an diesem Tag fehlerhaft sein, tragen Sie bei allen Stationen 0 ein.
- Ist der descriptor des entsprechenden Tags ”thunderstorm”, wird der negative Wert durch den entsprechenden absoluten Wert ersetzt.

Beispiel: So sollen die oben angegebenen Beispieldaten nach der Modifizierung durch die Methode dataPreparation aussehen. Veränderungen sind rot markiert.

Weather Stations	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Station 1	0	22	33	19	45	75	20
Station 2	35	21	57	8	10	100	10
Station 3	15	20	29	39	30	75	20

Table 3: Niederschlagsdaten nach dataPreparation()

`totalRainfall()`

Diese Methode berechnet den Gesamtniederschlag, den die Wetterstationen in Messzeitraum gemessen haben.

Beispiel:
Mit den gegebenen Beispieldaten beträgt der Gesamtniederschlag **683**.

```
1 getRainfall(int day_index, String mode)
```

Diese Methode bekommt den Index des Tages im Array/ Liste *rainfall* und einen *mode* übergeben. Falls ein fehlerhafter *day_index* übergeben wird, geben Sie -1 zurück. *mode* kann entweder ”biggest”, ”lowest” oder ”sum” lauten.

- Ist der mode des entsprechenden Tags ”lowest”, wird der geringste Wert des entsprechenden Tages zurückgegeben.
- Ist der mode des entsprechenden Tags ”biggest”, wird der höchste Wert des entsprechenden Tages zurückgegeben.
- Ist der mode des entsprechenden Tags ”sum”, wird die Summe der Niederschläge des entsprechenden Tages zurückgegeben.

Beispiel:

```
getRainfall(0, "lowest")
   → es wird 0 zurückgegeben

getRainfall(3, "biggest")
   → es wird 39 zurückgegeben

getRainfall(4, "sum")
   → es wird 85 zurückgegeben

getRainfall(7, "sum")
   → es wird -1 zurückgegeben
```

`calculateTemperature()`

Diese Methode bestimmt die Temperatur pro Tag pro Wetterstation und gibt diese als zweidimensionales Array/ Liste zurück. Die Temperatur wird dabei folgendermaßen berechnet:

- Ist der Wetterbericht am angegeben Tag *sunny*, dann verdreifache die Regenmenge, um die Temperatur zu erhalten.

- Ist der Wetterbericht am angegeben Tag *rainy*, reduziere die Regenmenge um die Hälfte, um die Temperatur zu erhalten.
- Ist der Wetterbericht am angegeben Tag *thunderstorm*, gebe den Restbetrag der Division der Regenmenge durch den aktuellen Tagesindex zurück. Wenn der Tagesindex 0 beträgt, dividieren Sie die Regenmenge durch die Anzahl der Wetterstationen.

Beispiel:

0	11	1	57	135	0	60
105	10	1	24	30	0	30
45	10	1	117	90	0	60

Table 4: Rückgabe von calculateTemperature()

`trend(int n)`

Diese Methode erstellt eine Vorhersage beruhend auf dem durchschnittlichen Niederschlag pro Station und Tag der letzten n Tage. Die Vorhersage wird in Form eines Strings, der einem der etablierten descriptors entspricht, zurückgegeben.

- Ist der durchschnittliche Niederschlag der letzten n Tage unter 50, gebe ”sunny” zurück.
- Ist der durchschnittliche Niederschlag der letzten n Tage 50 oder höher, gebe ”rainy” zurück.
- In dem besonderen Fall, dass der durchschnittliche Niederschlag der letzten n Tage *genau* 75 beträgt, gebe ”thunderstorm” zurück.

Beispiel:

```
n = 3
Durchschnitt der letzten 3 Tage = (45+75+20+10+100+10+30+75+20)/9 = 42
→ Es wird "sunny" zurückgegeben.

n = 2
Durchschnitt der letzten 2 Tage = (75+20+100+10+75+20)/6 = 50
→ Es wird "rainy" zurückgegeben.
```

Anhang: Aufgabenstellung ASL 2

Datenstrukturen: Anrechenbare Studienleistung 2

Administratives

Abgabe Termin

08.05.2023 - 23:59

Abgabe

Ihre Abgabe darf entweder in Java oder Python geschehen. Nutzen Sie für die Abgabe das Code-Template aus dem OPAL-Bereich "Anrechenbare Studienleistung 2" und halten Sie sich an die vorgegebene Struktur. Wenn nötig dürfen Sie weitere Dateien zur Lösung der Aufgabe erstellen. Laden Sie alle Dateien, die zum Ausführen des Programms notwendig sind, im OPAL-Bereich "Anrechenbare Studienleistung 2" hoch.

Erlaubte Klassen

Sie dürfen Klassen und Methoden aus der Standardbibliothek von Java bzw. Python nutzen. Bitte benutzen Sie keine Bibliotheken von Dritten.

Information zu Plagiaten

Es ist **erlaubt**, Lösungen aus dem Internet in Ihre Abgabe zu integrieren. Bitte **markieren** Sie jeglichen Code, der nicht von Ihnen selbst stammt, indem Sie einen Kommentar mit der Quelle des Codes (URL ist ausreichend) an die entsprechende Stelle setzen. Dies soll verhindern, dass Ihre Lösung fälschlicherweise als Plagiat erkannt wird.
Da es sich hierbei um eine Prüfungsleistung handelt, ist es notwendig, dass Sie die Aufgaben selbständig bearbeiten. Daher werden alle Abgaben untereinander auf Plagiate überprüft. Sollten Plagiate erkannt werden, gilt die Abgabe von **sämtlichen** beteiligten Parteien als ungültig.

Kompilierbarkeit und Clean Code

Es werden nur Abgaben gewertet, welche sich in einem ausführbaren Zustand befinden. Achten Sie außerdem auf die Leserlichkeit des Codes. Unverständliche oder unangemessene Bezeichner von Klassen, Variablen, etc. können zu Punkteabzügen führen.

Fragen

Bitte stellen Sie ihre Fragen im Forum im Thread ASL 2, damit alle Studierenden Zugang zu allen notwendigen Informationen erhalten.

Aufgabe

Schreiben Sie Klassen für die Verwaltung eines Universitätsnetzwerks.
Dafür sollen Klassen für Studierende, Lehrveranstaltungen, Dozierende und Universitäten selbst angelegt werden.

Ein Student wird dabei durch die Matrikelnummer (1-999) eindeutig gekennzeichnet, außerdem besitzt ein Student einen Vor- und Nachnamen und eine E-Mail. Weiterhin soll ein Student eine Sammlung von Lehrveranstaltungen besitzen. Die Lehrveranstaltungen sollen intern zusätzlich als "bestanden" oder "nicht bestanden" gespeichert werden. Weiterhin soll gespeichert werden, wie viele Fehlversuche es bereits in einem Kurs bei dem jeweiligen Studenten gab. Für die neuste Prüfung soll die Note gespeichert werden, eine 5.0 soll ein "nicht bestanden" sein.

Eine Lehrveranstaltung soll einen Titel, eine Anzahl an Leistungspunkten und einen Dozenten besitzen.

Ein Dozent soll einen Vor- und Nachnamen und eine E-Mail haben. Ebenso soll ein Dozent eine Liste an möglichen Lehrveranstaltungen besitzen.

Eine Universität soll eine Sammlung aus Studierenden, Dozierenden und Lehrveranstaltung sein. Zusätzlich soll eine Universität einen Namen besitzen.

Das Universitätsnetzwerk soll ein Zusammenschluss aus mehreren Universitäten sein.

Implementieren Sie folgende Klassen und Methoden:

Student

`Student(String first_name, String surname, String email, String studentID)`

Welche dem Konstruktor Ihrer gewählten Sprache entspricht und einen Studenten mit den erzeugten Werten erstellt.

`addCourse(Course course, String grade)`

Welche dem Studierenden den entsprechenden Kurs hinzufügt, sollte dieser noch nicht existieren. Sollte dieser bereits existieren, so soll die alte Note überschrieben werden. Sollte die neue Note *"5.0"* sein, so soll der Zähler für Fehlversuche erhöht werden.

`getFirstName()`

Welche den Vornamen zurückgibt.

`getSurname()`

Welche den Nachnamen zurückgibt.

`getEmail()`

Welche die E-Mail zurückgibt.

`getStudentID()`

Welche die Matrikelnummer zurückgibt.

`getCourses()`

Welche die Liste von belegten Kursen zurückgibt.

`getGradeInCourse(Course course)`

Welche die Note in der übergebenen Lehrveranstaltung zurückgibt. Sollte es diese Lehrveranstaltung nicht geben, geben Sie *None/ null* zurück.

`getNumberOfFailedAttempts(Course course)`

Welche eine den Wert zurückgibt, wie viele Fehlversuche in der jeweiligen Lehrveranstaltung getätigt wurden. Sollte es diese Lehrveranstaltung nicht geben, geben Sie *None/ null* zurück.

`getReachedCP()`

Welche die Anzahl der Leistungspunkte zurückgibt, die durch bereits bestandene Kurse erreicht wurden.

`getAverageGrade()`

Welche die Durchschnittsnote, *ohne "5.0"*, berechnet und zurückgibt. Die Noten sollen dabei die Gewichtung der Leistungspunkte der Kurse haben. Je mehr Creditpoints ein Kurs gibt, desto stärker fließt diese Note in die Durchschnittsnote ein.

Docent

`Docent(String first_name, String surname, String email)`

Welche dem Konstruktor Ihrer gewählten Sprache entspricht und einen Dozenten mit den erzeugten Werten erstellt.

`getFirstName()`

Welche den Vornamen zurückgibt.

`getSurname()`

Welche den Nachnamen zurückgibt.

`getEmail()`

Welche die E-Mail zurückgibt.

`addCourse(String course_name)`

Welche dem Dozenten die Möglichkeit gibt, Lehrveranstaltungen mit dem übergebenen Namen zu geben.

`getCourses()`

Welche eine Liste von Lehrveranstaltungsnamen, die der Dozierende halten kann, zurückgibt.

Course

`Course(String name, int cp)`

Welche dem Konstruktor Ihrer gewählten Sprache entspricht und eine Lehrveranstaltung mit den erzeugten Werten erstellt.

`getName()`

Welche den Namen der Lehrveranstaltung zurückgibt.

`getCP()`

Welche die Anzahl der Leistungspunkte zurückgibt, welche in diesem Kurs erreichbar sind.

`setDocent(Docent docent)`

Welche den Dozenten der Lehrveranstaltung setzt und damit den ehemaligen überschreibt, wenn vorhanden. Dies soll aber nur getan werden, wenn der Dozent diese Lehrveranstaltung geben kann. Sollte der Dozent gesetzt werden, geben sie *True* zurück, ansonsten *False*.

`removeDocent()`

Welche den aktuellen Dozenten von der Lehrveranstaltung abzieht.

`getDocent()`

Welche den Dozenten, wenn gesetzt, der Lehrveranstaltung zurückgibt. Ansonsten *None/ null*.

University

`University(String name)`

Welche dem Konstruktor Ihrer gewählten Sprache entspricht und eine Universität mit den erzeugten Werten erstellt.

`addDocent(Docent docent)`

Welche der Universität den übergebenen Dozenten hinzufügt. Sollte dieser Dozent bereits an der Universität sein, geben Sie den Wahrheitswert *False* zurück, ansonsten den Wahrheitswert *True*

`getDocent(String first_name, String surname)`

Welche den Dozierenden mit dem entsprechenden Namen zurückgibt. Sollte kein Dozent diesen Namen haben, geben Sie *None/ null* zurück.

`getListOfDocents()`

Welche eine Liste von Dozenten zurückgibt.

`addCourse(String course_name, int cp)`

Welche einen neuen Kurs an der Universität erstellt und diesen zurückgibt.

`getCourse(String course_name)`

Welche die Lehrveranstaltung mit dem entsprechenden Namen zurückgibt. Sollte keine Lehrveranstaltung diesen Namen haben, geben Sie *None/ null* zurück.

`getListOfCourses()`

Welche eine Liste von Lehrveranstaltungen zurückgibt.

`getListOfEmptyCourses()`

Welche eine Liste von Lehrveranstaltungen zurückgibt, die keinen Dozenten haben.

`addStudent(String first_name, String surname, String email)`

Welche der Universität einen Studenten hinzufügt. Dabei soll ein Objekt der Klasse Student erstellt werden. Die Matrikelnummer soll dabei durch eine konsekutive Zahl, startend bei *1*, gegeben sein. Die Matrikelnummer soll immer sechs Stellen besitzen. Sollte die Zahl nicht sechs Stellen haben, füllen Sie die Matrikelnummer mit vorangestellten Nullen.

`removeStudent(Student student)`

Welche den Studierenden exmatrikuliert. Sollte dieser existieren und exmatrikuliert wurden sein, geben Sie *True* zurück, ansonsten *False*

`getStudent(String first_name, String surname)`

Welche den Studierenden mit dem entsprechenden Namen zurückgibt. Sollte kein Student diesen Namen haben, geben Sie *None/ null* zurück.

`getListOfStudents()`

Welche eine Liste von Studenten zurückgibt.

`distributeCourses()`

Welcher den jeweiligen Lehrveranstaltungen Dozenten hinzufügt. Dabei soll beachtet werden, welche Lehrveranstaltungen von einem Dozenten gegeben werden können. Ebenso soll ein einzelner Dozent so wenig Lehrveranstaltungen wie möglich geben. Sollte es also 2 Dozenten geben, die einen Kurs geben können, so soll der Dozent ausgewählt werden, welcher weniger Kurse hat. Dafür sollen vor der Berechnung die Dozenten aller Kurse entfernt werden.

`getStudentsByGrade(List<Course> courses)`

Welche eine Liste von Studenten zurückgibt, welche abhängig von der Durchschnittsnote sortiert ist. Dabei sollen nur Studenten in Betracht gezogen werden, die eine Lehrveranstaltung aus *courses* belegt haben.

`getAverageGradeOfCourse(Course course)`

Welche die Durchschnittsnote des Kurses berechnet und zurückgibt. ”5.0” sollen dabei nicht mit in die Betrachtung gezogen werden.

`getCoursesByFailureRate()`

Welche die Liste von Kursen zurückgibt. Die Kurse sollen dabei absteigend in Bezug auf ihre Durchfallquote sortiert sein.

UniversityNetwork

`UniversityNetwork()`

Welche dem Konstruktor Ihrer gewählten Sprache entspricht und ein leeres Universitätsnetzwerk erstellt

`registerUniversity(University university)`

Welche eine Universität dem Netzwerk hinzufügt. Sollte diese Universität bereits im Netzwerk sein, geben Sie den Wahrheitswert *False* zurück, ansonsten den Wahrheitswert *True*

`getUniversityByStudentCount()`

Welche eine Liste an Universitäten zurückgibt, abhängig von der Anzahl der eingeschriebenen Studenten. Die Liste soll absteigend sortiert sein.

`getUniversityByCoveredCourses()`

Welche eine Liste an Universitäten zurückgibt, abhängig von der Anzahl der Lehrveranstaltungen, die einen Dozenten haben. Die Liste soll absteigend sortiert sein.

Anhang: Aufgabenstellung ASL 3

Datenstrukturen: Anrechenbare Studienleistung 3

Administratives

Abgabe Termin

22.05.2023 - 23:59

Abgabe

Ihre Abgabe darf entweder in Java oder Python geschehen. Nutzen Sie für die Abgabe das Code Template aus dem OPAL-Bereich "Anrechenbare Studienleistung 3". Sie dürfen zusätzliche Klassen und Methoden anlegen, um die Aufgabe zu lösen.
Laden Sie alle .java oder .py Dateien, die zum Ausführen Ihres Codes nötig sind, im OPAL-Bereich "Anrechenbare Studienleistung 3" hoch. Hinweis für Java: Von uns vorgegebene Interfaces brauchen Sie nicht mit abzugeben.

Erlaubte Klassen

Sie dürfen Klassen und Methoden aus der Standardbibliothek von Java bzw. Python nutzen. Bitte benutzen Sie keine Bibliotheken von Dritten.

Information zu Plagiaten

Es ist **erlaubt**, Lösungen aus dem Internet in Ihre Abgabe zu integrieren. Bitte **markieren** Sie jeglichen Code, der nicht von Ihnen selbst stammt, indem Sie einen Kommentar mit der Quelle des Codes (URL ist ausreichend) an die entsprechende Stelle setzen. Dies soll verhindern, dass Ihre Lösung fälschlicherweise als Plagiat erkannt wird.
Da es sich hierbei um eine Prüfungsleistung handelt, ist es notwendig, dass Sie die Aufgaben selbständig bearbeiten. Daher werden alle Abgaben untereinander auf Plagiate überprüft. Sollten Plagiate erkannt werden, gilt die Abgabe von **sämtlichen** beteiligten Parteien als ungültig.

Kompilierbarkeit und Clean Code

Es werden nur Abgaben gewertet, welche sich in einem ausführbaren Zustand befinden. **In den jeweiligen Templates sind Main-Methoden mit simplen Testfällen enthalten. Diese müssen funktionieren, ansonsten erhalten Sie 0 Punkte.** Achten Sie außerdem auf die Leserlichkeit des Codes. Unverständliche oder unangemessene Bezeichner von Klassen, Variablen, etc. können zu Punkteabzügen führen.

Fragen

Da es sich um eine Prüfungsleistung handelt, ist es wichtig, dass jeder Teilnehmer Zugang zu allen notwendigen Informationen hat. Deswegen bitten wir Sie, Fragen direkt ins Forum, in den Thread "ASL 3: Fragen" zu stellen.

Aufgabe

Erstellen Sie eine Double Linked List, die als Bank verwendet werden kann. Die Knoten der Liste sollen durch die Klasse **Konto** repräsentiert werden. Jeder Knoten in der Liste soll neben dem Vorgänger- und Nachfolgerknoten auch die Daten eines Kontos enthalten, einschließlich des Vor- und Nachnamens, der Telefonnummer und des aktuellen Kontostands.

Wichtig: Sie sollen hier eine eigene Datenstruktur manuell implementieren und *nicht* die bereits vorhandenen Listen der Standardbibliothek verwenden. Abgaben, welche einfach nur die bereits vorhandenen Listen 'neu verpacken' werden nicht gewertet.

Implementieren Sie für die Klasse **Konto** folgendes:

```
constructor(String firstName, String name, String phoneNumber, int balance)
```

Welche dem Konstruktor Ihrer genutzten Sprache entspricht und die Attribute des Knotens auf 'firstName', 'name', 'phoneNumber' und 'balance' setzt. Der Vorgänger und Nachfolger des Knotens sollen dabei mit 'null' oder äquivalent mit 'None' initialisiert werden.

```
getPrev()
```

Welche das vorherige Konto zurückgibt.

```
getNext()
```

Welche das nachfolgende Konto zurückgibt.

```
setPrev(Konto konto)
```

Welche das übergebene Konto als Vorgänger setzt. Hatte das ursprüngliche Konto bereits Vorgänger, sollen diese als Vorgänger des übergebenen Kontos gesetzt werden.

```
setNext(Konto konto)
```

Welche das übergebene Konto als Nachfolger setzt. Hatte das ursprüngliche Konto bereits Nachfolger, sollen diese als Nachfolger des übergebenen Kontos gesetzt werden.

Implementieren Sie für die Klasse **Bank** folgendes:

`constructor()`

Welche dem Konstruktor Ihrer genutzten Sprache entspricht und eine leere Liste erzeugt.

`getHead()`

Gibt den Kopf der Liste (das erste Element) zurück.

`append(String firstName, String name, String phoneNumber, int balance)`

Fügt ein neues Konto an der letzten Stelle zur Bank hinzu.

`get(index)`

Welche das Konto am übergebenen Index zurückgibt. Ist der Index nicht in der Liste vorhanden, geben Sie null bzw. None zurück.

`insert(int index, String firstName, String name, String phoneNumber, int balance)`

Welche ein neues Konto mit den übergebenen Werten erzeugt und dieses an dem gegebenen Index hinzufügt. Sollte der übergebene Index der aktuellen Größe der Liste entsprechen, wird das neue Konto am Ende der Liste hinzugefügt.
Falls das Hinzufügen erfolgreich ist, geben Sie den Wahrheitswert *True* zurück, ansonsten *False*.

`delete(int index)`

Welche das Konto an der Stelle des Indexes löscht. Konnte das Konto erfolgreich gelöscht werden, geben Sie den Wahrheitswert *True* zurück, ansonsten *False*.

`search(int mode, String value)`

Welche das Konto mit dem ersten Auftreten des übergebenen *value* je nach übergebenem *mode* in der Bank sucht und das Konto zurückgibt.

Der Parameter *mode* kann Werte von 0-3 annehmen, welche jeweils folgende Bedeutung haben:

- 0 = firstName
- 1 = name
- 2 = phoneNumber
- 3 = bankBalance

Hat 'mode' den Wert 0 bezieht sich 'value' also auf den Vornamen der Kontoinhaber. Es soll das erste Konto zurückgegeben werden, bei dem 'firstName' dem übergebenen 'value' entspricht.

`printAccounts()`

Welche alle Konten der Bank auf der Konsole ausgibt. Achten Sie auf eine leserliche Formatierung der Kontodaten.

`swap(int index1, int index2)`

Welche zwei Konten an den übergebenen Indizes in der Bank tauscht. Falls der Tausch erfolgreich ist, geben Sie den Wahrheitswert *True* zurück, ansonsten *False*.

`deleteSublist(int startIndex, int accountsToDelete)`

Welche eine Anzahl von 'accountsToDelete' Konten aus der Bank löscht, ausgehend von 'startIndex'. 'startIndex' inklusive. Geben Sie die Liste, die übrig bleibt, zurück.

`sort(int mode, int orderKind)`

Welche die Konten anhand der übergebenen Parameter sortiert. Es soll dabei anhand des *mode* ausgewählt werden, anhand von welchem Attribut des Kontos sortiert werden soll. Die Werte, welche *mode* annehmen kann, entsprechen denjenigen in der Methode 'search'. *orderKind* kann Werte von 0-1 annehmen und sagt aus, ob die Liste aufsteigend oder absteigend sortiert werden soll:

- 0 = aufsteigend
- 1 = absteigend

Beispiel: sort(1, 1)
→ Die Liste wird in alphabetisch absteigender Reihenfolge nach Nachnamen sortiert

`getMedian()`

Welche den Median der Kontostände bestimmt. Der Median ist der Wert, der genau in der Mitte einer sortierten Datenreihe liegt. Wenn die Anzahl der Konten ungerade ist, ist der Median also die 'balance' des Kontos in der Mitte der Liste. Wenn die Anzahl der Konten gerade ist, gibt es keinen eindeutigen mittleren Wert, und der Median ist der Durchschnitt der beiden mittleren Werte.

`deleteSublistSuccessive(int total)`

Löscht nacheinander alle Teillisten der Liste, deren Kontostände summiert dem übergebenen 'total' entsprechen.

Beispiel: [6, 7, 3, 4, 8, 9, 2]
deleteSublistSuccessive(10)
Nach 1. Iteration: [6, 4, 8, 9, 2] (7 und 3 wurden gelöscht)
Nach 2. Iteration [8, 9, 2] (6 und 4 wurden gelöscht)

`removeDuplicates(int mode, String value)`

Welche bis auf das erste Auftreten des Wertes *value* des übergebenen *mode* alle Konten, die einem Duplikat entsprechen, aus der Bank löscht. Die möglichen Werte von *mode* entsprechen denen in den 'search' und 'sort' Methoden.

`fuse(Bank secondBank)`

Welche 2 Banken zusammenfügt. Falls Vorname, Nachname und Telefonnummerzweiter Konten identisch sind, addiere den Kontostand beider Konten und entferne das Konto, das in der zweiten (also der übergebenen Bank) steht.

Merkhilfe

mode

- 0 = firstName
- 1 = name
- 2 = phoneNumber
- 3 = bankBalance

orderKind

- 0 = aufsteigend
- 1 = absteigend

Hinweis für Java

Wenn der übergebene **mode** 3 ist, dann können Sie für die Umwandlung des Strings in einen Integer - Integer.parseInt() - nutzen. Beispiel:

```
String value = "123";
int intValue = Integer.parseInt(value);
```

Anhang: Aufgabenstellung ASL 4

Datenstrukturen: Anrechenbare Studienleistung 4

Administratives

Abgabe Termin

05.06.2023 - 23:59

Abgabe

Ihre Abgabe darf entweder in Java oder Python geschehen. Nutzen Sie für die Abgabe das Code-Template aus dem OPAL-Bereich "Anrechenbare Studienleistung 4" und halten Sie sich an die vorgegebene Struktur. Wenn nötig, dürfen Sie weitere Dateien zur Lösung der Aufgabe erstellen. Laden Sie alle Dateien, die zum Ausführen des Programms notwendig sind, im OPAL-Bereich "Anrechenbare Studienleistung 4" hoch.

Information zu Plagiaten

Es ist **erlaubt**, Lösungen aus dem Internet in Ihre Abgabe zu integrieren. Bitte markieren Sie jeglichen Code, der nicht von Ihnen selbst stammt, indem Sie einen Kommentar mit der Quelle des Codes (URL ist ausreichend) an die entsprechende Stelle setzen. Dies soll verhindern, dass Ihre Lösung fälschlicherweise als Plagiat erkannt wird. Da es sich hierbei um eine Prüfungsleistung handelt, ist es notwendig, dass Sie die Aufgaben **selbstständig** bearbeiten. Daher werden alle Abgaben untereinander auf Plagiate überprüft. Sollten Plagiate erkannt werden, gilt die Abgabe von **sämtlichen** beteiligten Parteien als ungültig.

Kompilierbarkeit und Clean Code

Es werden nur Abgaben gewertet, welche sich in einem **ausführbaren** Zustand befinden. Achten Sie außerdem auf die Leserlichkeit des Codes. Unverständliche oder unangemessene Bezeichner von Klassen, Variablen, etc. können zu Punkteabzügen führen.

Fragen

Bitte stellen Sie Ihre Fragen im Forum im Thread ASL 4, damit alle Studierenden Zugang zu allen notwendigen Informationen erhalten.

Aufgabe

Prioritätswarteschlangen werden häufig in ”Scheduling-Anwendungen” benutzt. Ein Beispiel, wo solch eine Anwendung benutzt wird, wäre ein Restaurant, das die Zubereitung und Ausgabe seiner Bestellungen nach verschiedenen Parametern einschätzt und ausführt. Das Restaurant, welches in unserem Falle vereinfacht aus mehreren Ausführungseinheiten, den Köchen und Köchinnen, besteht, kann eine Vielzahl von Bestellungen annehmen und ausführen.
In dieser Aufgabe soll eine Bestellung aus einer eindeutigen ”ID”, einer ”description” des zugehörigen Gerichts und einer ”priority” bestehen. Zudem sollen zu jeder Bestellung ”arrivalTime” und ”executionTime” gespeichert werden.
Das Restaurant soll die Möglichkeit haben, aus mehreren ”executionUnits” zu bestehen (so können mehrere Bestellungen gleichzeitig bearbeitet werden). Die ”arrivalTime” beschreibt den diskreten Zeitpunkt, ab wann die Bearbeitung (Zubereitung) einer Bestellung gestartet werden kann. Die ”executionTime” beschreibt, wie viele diskrete Zeitschritte nötig sind, um die Bestellung fertigzustellen. Das Attribut ”priority” gibt an, welche Bestellung präferiert werden soll. Eine höhere Zahl bedeutet dabei, dass die Zubereitung dieser Bestellung Vorrang hat. Sollten mehrere ”executionUnits” bereit zur Ausführung sein, so soll die Bestellung mit der höchsten Priorität zubereitet werden.

Implementieren Sie bitte folgende Klassen und Methoden:

Order

Diese Klasse soll zur Repräsentation einer Bestellung dienen. Es werden folgende Methoden gefordert:

```
Order(int id, String description, int priority, int arrivalTime, int executionTime)
```

Implementieren Sie einen Konstruktor (Python: ___init___(...)), welcher die Attribute der Bestellung basierend auf den übergebenen Werten setzt.

```
int getID()
```

Welche die ID der Bestellung zurückgibt.

```
String getDescription()
```

Welche die ”description” der Bestellung zurückgibt. Die ”description” kann beispielsweise das zubereitete Gericht als String sein.

```
int getArrivalTime()
```

Welche die ”arrivalTime” der Bestellung zurückgibt.

```
int getExecutionTime()
```

Welche die ”executionTime” der Bestellung zurückgibt.

```
int getPriority()
```

Welche die ”priority” der Bestellung zurückgibt.

Node

Diese Klasse soll zum Speichern einer Bestellung in einer doppelt verketteten Liste dienen. Implementieren Sie bitte folgende Methoden:

`Node(Order order)`

Welche dem Konstruktor Ihrer gewählten Sprache entspricht. Dieser soll eine Bestellung entgegennehmen und das Attribut des Knotens entsprechend setzen.

`Node getPrev()`

Welche den Vorgänger des Knotens zurückgibt.

`Node getNext()`

Welche den Nachfolger des Knotens zurückgibt.

`void setPrev(Node prev)`

Welche den Vorgänger des Knotens auf ”prev” setzt.

`void setNext(Node next)`

Welche den Nachfolger des Knotens auf ”next” setzt.

List

Diese Klasse soll zum Speichern der verschiedenen Bestellungen in einer doppelt verketteten Liste dienen. Es werden folgende Methoden von Ihnen gefordert:

`List()`

Welche dem Konstruktor Ihrer gewählten Sprache entspricht und eine leere Liste anlegt.

```
boolean append(Node node)
```

Welche den Knoten ”node” an die Liste anhängt. Die Methode soll ”wahr” zurückgeben, wenn das Anhängen funktioniert hat. Andernfalls soll ”falsch” zurückgegeben werden.

```
boolean insert(int index, Node node)
```

Welche den Knoten ”node” am Index ”index” einfügt. Die Methode soll ”wahr” zurückgeben, wenn das Anhängen funktioniert hat. Andernfalls soll ”falsch” zurückgegeben werden.

```
boolean remove(int index)
```

Welche den Knoten am Index ”index” aus der Liste entfernt. Die Methode soll ”wahr” zurückgeben, wenn das Anhängen funktioniert hat. Andernfalls soll ”falsch” zurückgegeben werden.

```
boolean swap(int first_index, int second_index)
```

Welche den Knoten der Liste am Index ”first_index” mit dem Knoten am Index ”second_index” tauscht. Falls der Tausch funktioniert hat, soll ”wahr”, andernfalls ”falsch”, zurückgegeben werden.
Hinweis: Achten Sie besonders auf Sonderfälle, z.B., wenn sich zwei Knoten direkt hintereinander befinden.

```
void sort()
```

Welche die Liste **absteigend** nach den Prioritäten der Bestellungen sortiert. Den Sortieralgorithmus dürfen Sie selbst wählen.

`Order getMostUrgentOrder()`

Welche die Bestellung mit der größten Priorität zurückgibt und den jeweiligen Knoten aus der Liste löscht.

Restaurant

Diese Klasse soll zur Modellierung eines Restaurants dienen. Ein Restaurant soll "executionUnits" Köche und Köchinnen haben und eine doppelt verkettete Liste der Bestellungen speichern.
Implementieren Sie folgende Methoden:

```
Restaurant(int executionUnits)
```

Welche ein Restaurant mit "executionUnits" Köchen und Köchinnen und einer leeren Bestellliste initialisiert.

```
boolean addOrder(int id, String description, int priority, int arrivalTime,
int executionTime)
```

Welche eine Bestellung in die Bestellliste des Restaurant einfügt. Beachten Sie, dass die ID jeder Bestellung eindeutig ist. Falls also die übergebene ID schon in der Liste vorkommt, soll die neue Bestellung nicht eingefügt werden. Wenn das Einfügen funktioniert hat, soll "wahr", andernfalls "falsch", zurückgegeben werden.

```
boolean removeOrder(int id)
```

Sollte es in der Bestellliste eine Bestellung mit der ID "id" geben, soll diese gelöscht werden. Wenn die Bestellung gelöscht wurde, soll die Methode "wahr", ansonsten "Falsch" zurückgeben. Wenn das Löschen funktioniert hat, soll "wahr", andernfalls "falsch", zurückgegeben werden.

```
void getKMostUrgentOrders(int k)
```

Welche die Daten der "k" höchstpriorisiertesten Bestellungen des Restaurant auf dem Bildschirm ausgibt. Die Ausgabe soll absteigend nach den Prioritäten der Bestellungen erfolgen. Zudem soll pro Bestellung folgendes Ausgabeformat verwendet werden: *"id, description, priority"*.
Falls k größer als die Länge der Liste ist, soll eine Fehlermeldung ausgegeben werden.

```
List execute()
```

Mit Aufruf dieser Methode sollen die Zubereitung der Bestellungen beginnen (startend beim Zeitpunkt 0, positiv fortlaufend). Es sollen alle Köche und Köchinnen des Restaurant tätig werden, um **alle** Bestellungen abzuarbeiten. Zu jedem Zeitpunkt soll, nach Möglichkeit, immer die höchstpriosiertesten Bestellungen bearbeitet werden. Wenn während der Ausführung einer Bestellung höherpriorisierte Bestellungen eingehen, soll die Ausführung der ursprünglichen Bestellung jedoch nicht unterbrochen werden. Die Ausführung gilt als beendet, wenn alle Bestellungen finalisiert sind.
Um das korrekte Abarbeiten der Bestellungen zu tracken, soll die Methode am Ende eine Liste aus Listen zurückgeben. Die Liste soll folgendes Format haben: *"[[Bestellungs-description vom Koch 0 bearbeitet, Bestellungs-description vom Koch 1 bearbeitet, ...], ...]"*. Eine innere Liste gibt am Index i an, welche Bestellung gerade vom Koch/von der Köchin i ausgeführt wird. Eine Liste am Index j beschreibt den j-ten Zeitpunkt.
Zum Verständnis: Die Methode soll solange arbeiten, bis alle Bestellungen abgearbeitet sind. Erst danach können wieder neue Bestellungen hinzugefügt werden.
Hinweis: Beachten Sie, dass eine Bestellung erst ausgeführt werden kann, wenn Sie angekommen ist. Benutzen Sie zum Zurückgeben der Ergebnisse bitte die Listen-/Array-Klasse, die **bereits in Ihrer Sprache eingebaut** ist.

Beispiel

```
restaurant = Restaurant(2)

restaurant.addOrder(1, "Pommes", 5, 0, 3)           // True, ["Pommes"]
restaurant.addOrder(2, "Salat", 5, 3, 2)            // True, ["Pommes", "Salat"]
restaurant.addOrder(3, "Steak", 10, 0, 5)           // True, ["Pommes", "Salat", "Steak"]
restaurant.addorder(1, "Spaghetti", 10, 2, 7)       // False
restaurant.addorder(4, "Spaghetti", 10, 2, 7)       // True,
                                                    // ["Pommes", "Salat", "Steak",
                                                    //  "Spaghetti"]

restaurant.removeOrder(4)                           // True, ["Pommes", "Salat", "Steak"]

restaurant.getKMostUrgendOrders(2)                  // 3, "Steak", 10
                                                    // 1, "Pommes", 5
                                                    // 2, "Salat", 5

result = restaurant.execute()
// [
// ["Steak", "Pommes"]
// ["Steak", "Pommes"]
// ["Steak", "Pommes"]
// ["Steak", "Salat"]
// ["Steak", "Salat"]
// ]
```

Anhang: Aufgabenstellung ASL 5

Datenstrukturen: Anrechenbare Studienleistung

Administratives

Abgabe Termin

19.06.2023 - 23:59

Abgabe

Ihre Abgabe darf entweder in Java oder Python geschehen. Nutzen Sie für die Abgabe das Code-Template aus dem OPAL-Bereich "ASL 5".
Laden Sie Ihre Lösung im OPAL-Bereich "Anrechenbare Studienleistung 5" hoch.
Sollten Sie sich nicht an die angegebene Struktur halten, wirkt sich dies auf Ihre Bewertung aus.

Fragen

Da es sich um eine Prüfungsleistung handelt, ist es wichtig, dass jeder Teilnehmer Zugang zu allen notwendigen Informationen hat. Deswegen bitten wir Sie, Fragen direkt ins Forum, in den Thread "ASL 5: Fragen" zu stellen.

Aufgabe

Implementieren Sie einen Binären Suchbaum, der die CPU eines Computers darstellt. Die einzelnen Prozesse werden dabei als Knoten dargestellt. Jeder Prozess enthält eine id und eine Kategorie.

Implementieren Sie eine Klasse **Process**.

```
constructor(int id, String category)
```

Welche dem Konstruktor Ihrer genutzten Sprache entspricht und die ID des Prozesses mit 'id' sowie die Kategorie des Prozesses mit 'category' initialisiert.

```
getID()
```

Welche die 'id' des Prozesses zurückgibt.

```
setID(int id)
```

Welche die 'id' des Prozesses auf die 'id' setzt.

```
getCategory()
```

Welche die 'category' des Prozesses zurückgibt.

```
setCategory(String category)
```

Welche die 'category' des Prozesses auf die 'category' setzt.

`getLeft()`

Welche das linke Kind des Prozesses zurückgibt.

`setLeft(Process left)`

Welche das linke Kind des Prozesses auf den übergebenen Prozess 'left' setzt.

`getRight()`

Welche das rechte Kind des Prozesses zurückgibt.

`setRight(Process right)`

Welche das rechte Kind des Prozesses auf den übergebenen Prozess 'right' setzt.

Implementieren Sie eine Klasse **Processor**.

`constructor()`

Welche dem Konstruktor ihrer genutzten Sprache entspricht und einen leeren Binären Suchbaum erstellt.

`getProcess(int id)`

Welche den Knoten mit der übergebenen ID zurückgibt. Sollte die ID nicht im Baum enthalten sein, werfen Sie eine 'IndexOutOfBoundsException' in Java, oder eine 'IndexError' Exception in Python.

`getCategory(int id)`

Welche die Kategorie des Prozesses mit der übergebenen id zurückgibt. Sollte die id nicht im Baum enthalten sein, werfen Sie eine 'IndexOutOfBoundsException' in Java, oder eine 'IndexError' Exception in Python.

`insert(int id, String category)`

Welche einen neuen Prozess mit den übergebenen Parametern zum Prozessor hinzufügt. Sollte die id kleiner als 0 oder bereits im Baum enthalten sein, werfen Sie eine 'IndexOutOfBoundsException' in Java, oder eine 'IndexError' Exception in Python.

`delete(int id)`

Welche den übergebenen Prozess mit der gegebenen id löscht. Sollte die id kleiner als 0 oder nicht im Baum enthalten sein, werfen Sie eine 'IndexOutOfBoundsException' in Java, oder eine 'IndexError' Exception in Python.

`getProcessesByCategory(String category)`

Welche eine Liste aller Knoten der Prozesse der gegebenen Kategorie zurückgibt. Falls es keinen Prozess der übergebenen 'category' gibt, geben Sie eine leere Liste zurück.

`getProcessesById(int lowerBound, int upperBound)`

Welche eine Liste aller Prozesse zurückgibt, die eine id größer als 'lowerBound' und kleiner als 'upperBound' besitzen. Die Reihenfolge der Liste ist egal.

`toInorderList()`

Welche eine Liste zurückgibt, die die Knoten basierend auf der Inorder-Traversierung ausgibt.

`leftView()`

Welche eine Liste zurückgibt, die aus den 'id's der am weitesten links liegenden Knoten je Ebene besteht. Der Index des Wertes soll der Höhe des Knotens im Baum entsprechen. Am ersten Index steht die Wurzel.

`maxRootToLeafPath()`

Welche den längsten Weg von der Wurzel (Startpunkt) bis zu einem Blatt (Endpunkt) im Baum zurückgibt.

`height()`

Welche die Höhe des Prozessors zurückgibt. Die Wurzel entspricht dabei der Höhe 0.

`smallestGreaterKey(Process current, String category)`

Diese Methode sucht den kleinsten Prozess im Baum, der größer als der gegebene Prozess ist und der übergebenen 'category' entspricht. Wenn kein solcher Prozess vorhanden ist, wird 'Null' bzw. 'None' zurückgegeben.

`findKthSmallest(int k)`

Diese Methode findet den k-kleinsten Prozess im Prozessor und gibt ihn zurück. Wenn der k-kleinste Prozessor nicht existiert, soll die Methode 'null' bzw. 'None' zurückgeben.

`areIdentical(Processor otherProcessor)`

Diese Methode überprüft, ob der gegebene Prozessor otherProcessor mit dem aktuellen Prozessor identisch ist, d.h. ob sie die gleiche Struktur haben und alle Prozesse übereinstimmen. Sie gibt True zurück, wenn die Prozessoren identisch sind, andernfalls False.

`getCommonAncestor(Process firstProcess, Process secondProcess)`

Diese Methode gibt den gemeinsamen Vorfahren der Prozesse zurück. Der gemeinsame Vorfahre ist der Knoten, von dem aus beide Prozesse erreichbar sind.

`getLeafCount()`

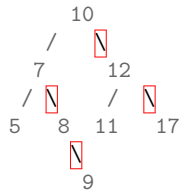
Diese Methode gibt die Anzahl der Blätter im Baum zurück, d.h. die Anzahl der Knoten ohne Kinder.

`invert()`

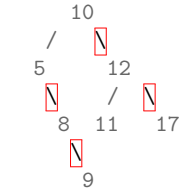
Welche den Prozessor invertiert, sodass kleinere Prozesse rechts statt links stehen. Neu hinzugefügte Prozesse sollen ebenfalls rechts statt links gespeichert werden. Bei einem erneuten Aufruf von invert() sollen die Prozessorstruktur und der Speichermodus wieder umgedreht werden, sodass kleinere Prozesse wieder links stehen. Erzeugen Sie hierbei keine neuen Prozesse oder eine neue Prozessorstruktur.

Beispiel

```
processor = Processor()

processor.insert(10, "Game")
processor.insert(12, "Game")
processor.insert(11, "Application")
processor.insert(7, "Game")
processor.insert(8, "Game")
processor.insert(9, "Application")
processor.insert(17, "Application")
processor.insert(5, "Browser")
```




```
processor.getNode(7)     // [7, "Game"]

processor.getCategory(7) // "Game"

processor.getProcessesByCategory("Application")
// [11, 9, 17]

processor.getProcessesById(9, 15)
// [10, 12, 11]

processor.leftView()
// [10, 7, 5, 9]

processor.toInorderList()
// [5, 7, 8, 9, 10, 11, 12, 17]

processor.height() // 3


processor.remove(7)
```



Anhang: Aufgabenstellung ASL 6

Datenstrukturen: ASL 6

Administratives

Abgabe Termin

03.07.2023 - 23:59

Abgabe

Ihre Abgabe darf entweder in Java oder Python geschehen. Nutzen Sie für die Abgabe das Code Template aus dem OPAL-Bereich "Anrechenbare Studienleistung 6". Laden Sie Ihre Lösung im OPAL-Bereich "Anrechenbare Studienleistung 6" hoch.
Sollten Sie sich nicht an die angegebene Struktur halten, wirkt sich dies auf Ihre Bewertung aus.

Fragen

Da es sich um eine Anrechenbare Studienleistung handelt, ist es wichtig, dass jeder Teilnehmer Zugang zu allen notwendigen Informationen hat. Deswegen bitten wir Sie, Fragen direkt ins Forum, in den Thread "ASL 6: Fragen" zu stellen.

Aufgabe

Implementieren Sie einen ungerichteten Graphen. Ein Graph sei dabei ein Tupel/Liste von Knoten V und Kanten E : G=(V,E). Jeder Knoten des Graphs soll dabei eine x und y Koordinate aus dem Zahlenbereich der Integer besitzen. Eine Kante soll eine beidseitig nutzbare Verbindung zwischen 2 Knoten sein.
Implementieren Sie folgende Methoden, um einen solchen Graphen zu realisieren:

```
newNode(int x, int y)
```

Welche einen neuen Knoten dem Graphen hinzufügt und eine eindeutige Nummer zurückgibt. Diese Nummer soll dabei durch aufsteigende Zahlen repräsentiert werden, beginnend bei 0. Fügen Sie den Knoten nicht hinzu, sollte bereits ein Knoten mit den gleichen x und y Werten existieren. Geben Sie in diesem Fall -1 zurück.

```
setEdge(int from, int to)
```

Welche eine Kante zwischen den Knoten FROM und TO erstellt. Geben Sie "true" zurück, sollte die Kante erfolgreich erstellt worden sein, andernfalls geben Sie "false" zurück.

```
getEdges()
```

Welche eine Liste von Listen aus Knoten (Integer) zurückgibt. Dabei soll es eine Liste aus Integern für jeden Knoten geben. Die i-te Integer-Liste soll dabei repräsentieren, welche Knoten vom Knoten i aus erreichbar sind.

```
getNGons(int n)
```

Die als Eingabe einen Parameter n erhält und eine Liste von Listen mit Knoten zurückgibt. Die Methode soll alle paarweise disjunkten n-Gons finden und speichern.
Ein n-Gon ist eine Struktur im Graphen, die entsteht, indem man von einem Startknoten ausgehend n-1 Knoten über vorhandene Kanten erreicht und schließlich wieder am Startknoten endet. Die Knoten in der Liste sollen so angeordnet sein, dass benachbarte Knoten in der Liste eine Kante haben, die für das n-Gon verwendet wurde. Gleiches gilt für das Paar aus dem ersten und letzten Knoten in der Liste.

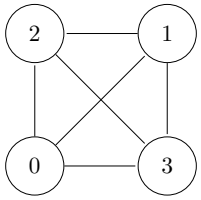
`getLongestPath(int from, int to)`

Welche eine Liste aus Knoten (Integer) zurückgibt. Die Knoten sollen dabei die Knoten auf dem längsten Weg von FROM nach TO sein. Sollte es keinen Weg geben, geben Sie eine leere Liste zurück. Die Länge des Weges entspricht dabei der Summe der einzelnen Kantenlängen. Jeder Knoten soll dabei nur einmalig besucht werden. Eine Kantenlänge zwischen zwei Punkte $(x_1, y_1), (x_2, y_2)$ entspricht dabei $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

`getNonCrossingGraph()`

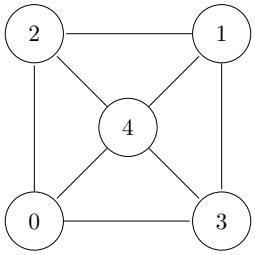
Welche einen Graphen zurückgibt, welcher keine sich schneidenden Kanten besitzt. kreieren Sie hierfür einen Graphen aus dem aufrufenden, bei welchem Schnittpunkte von Kanten durch einen Knoten ersetzt werden, welche an der nächstgelegenen Koordinate liegt. Löschen Sie die sich schneidenden Kanten aus dem Graphen und fügen Sie neue zum gerade erstellten Knoten hinzu.
Beispielsweise hat man die Punkte A(0,0), B(5,5), C(5,0), D(0,5) mit den Kanten A-B und C-D so schneiden die Kanten sich im Punkt (2.5, 2.5). Die nächstgelegene Koordinate wäre E(3,3). Also entstehen die Kanten A-E, B-E, C-E, D-E. Runden Sie ab .5 auf und darunter ab.

Beispiel




```
ASL_6 graph = new ASL6();
graph.newNode(0,0); // 0
graph.newNode(5,5); // 1
graph.newNode(5,0); // 2
graph.newNode(0,5); // 3
graph.setEdge(0,1); //true
graph.setEdge(0,1); //false
graph.setEdge(0,0); //false
graph.setEdge(0,2); //true
graph.setEdge(0,3); //true
graph.setEgde(0,4); //false
graph.setEdge(1,3); //true
graph.setEdge(1,2); //true
graph.setEdge(2,3); //true

graph.getEdges();
//[
//[1,2,3],
//[0,2,3],
//[0,1,3],
//[0,1,2]
//]
```

```
graph.getNGons(3);
//[
//[0,1,3],
//[0,1,2],
//[0,2,3],
//[1,2,3]
//]

graph.getLongestPath(2, 1); //[2,3,0,1]

graph.getNonCrossingGraph(); //graph below
```



Anhang: Aufgabenstellung ASL 7

Datenstrukturen: Anrechenbare Studienleistung

Administratives

Abgabe Termin

17.07.2023 - 23:59

Abgabe

Ihre Abgabe darf entweder in Java oder Python geschehen. Nutzen Sie für die Abgabe das Code-Template aus dem OPAL-Bereich "ASL 7".
Laden Sie Ihre Lösung im OPAL-Bereich "Anrechenbare Studienleistung 7" hoch.
Sollten Sie sich nicht an die angegebene Struktur halten, wirkt sich dies auf Ihre Bewertung aus.

Fragen

Da es sich um eine Anrechenbare Studienleistung handelt, ist es wichtig, dass jeder Teilnehmer Zugang zu allen notwendigen Informationen hat. Deswegen bitten wir Sie, Fragen direkt ins Forum, in den Thread "ASL 7: Fragen" zu stellen.

Aufgabe

Ein Netzbetreiber möchte eine neue Energieinfrastruktur, basierend auf erneuerbaren Energien, aufbauen. Doch gibt es hierfür einige Unklarheiten bezüglich der bestmöglichen Infrastruktur, Verteilung und Auslastung im gesetzlichen Rahmen. Der Netzbetreiber besitzt eine grobe 2D Karte seines Interessenbereichs. Dieser Bereich soll durch eine Breite und Höhe gegeben sein. Weiterhin soll dieser Bereich aus Planquadraten bestehen. Für jedes dieser Planquadrate gibt es verschiedenste Informationen. So gibt es die Information über den potenziellen Energiegewinn von Windkraftanlagen und Solaranlagen. Weiterhin wird pro Planquadrat genannt, ob dieses besiedelt ist. Sollte dieses besiedelt sein, so werden sowohl die Einwohnerzahl als auch der gesamte durchschnittliche Energieverbrauch angegeben. Weiterhin soll jedes Feld einen Kostenfaktor besitzen, welcher beschreibt, wie teuer es wäre, ein Kabel über dieses Feld zu legen. Die Kosten eines Kabels zwischen 2 benachbarten Feldern werden durch die Addition der Kosten beider Felder beschrieben.
Schreiben Sie ein Klasse 'EnergyPlanner', welches dieses Problem in Form einer Klasse abstrahiert. Implementieren Sie hierfür folgende Methoden:

```
constructor(int width, int height)
```

Welche dem Konstruktor Ihrer genutzten Sprache entspricht und ein Feld der Größe 'width'x'height' erzeugt. Dabei sollen alle Felder nicht besiedelt sein und einen potenziellen Energiegewinn von 0 für Wind- und Solarenergie besitzen.

```
setSettlement(int x, int y, int population, int
powerConsumption)
```

Welche auf des Feld [x][y] eine Siedlung mit der Einwohneranzahl 'population' und dem gesamten Energieverbrauch 'powerConsumption' setzt.

```
setEnergyPotential(int x, int y, String kind, int potential)
```

Welche das Energiepotential für das Feld [x][y] auf 'potential' für die Energieart 'kind' setzt. 'kind' kann dabei entweder 'solar' oder 'wind' sein.

`setCost(int x, int y, int cost)`

Welche die Baukosten für ein Kabel auf einem Feld [x][y] setzt.

`setWindTurbine(int x, int y)`

Welche auf ein Feld eine Windkraftanlage setzt, welche Energie entsprechend der potenziellen Energiegewinnung des Feldes produziert.

`setSolarPanel(int x, int y)`

Welche auf ein Feld eine Solaranlage setzt, welche Energie entsprechend der potenziellen Energiegewinnung des Feldes produziert.

`getCost(int x1, int y1, int x2, int y2)`

Welche die Kabelkosten zwischen den benachbarten Feldern [x1][y1] nach [x2][y2] zurückgibt.

`getMinimalCostPath(int x1, int y1, int x2, int y2, float maxFlow)`

Welche einen potenziellen Weg zwischen [x1][y1] und [x2][y2] zurückgibt. Dabei soll die Energie in [x1][y1] nach [x2][y2] übertragen werden. Die Energie soll der produzierten Energie in [x1][y1] entsprechen. Dabei soll die Energie die Summe der Windenergie und Solarenergie sein, sollten diese Anlagen auf dem Feld gesetzt sein. Weiterhin soll beachtet werden, dass je Schritt Kabelnutzung die transportierte Energie um '0.1' verringert wird. Finden Sie einen kostenminimalen Weg von [x1][y1] nach [x2][y2], bei welchem am Ende noch Energie übertragen wird. Geben Sie diesen Weg in Form einer Liste aus Listen aus Integern zurück. Dabei soll eine innere Liste ein Koordinatenpaar repräsentieren und die äußere Liste die Koordinaten von [x1][y1] nach [x2][y2]. Sollte es keinen Weg geben, bei welchem Energie am Ziel ankommt, geben Sie eine leere Liste zurück. Durch einen Kabelabschnitt können dabei maximal 'maxFlow' Stromeinheiten durchfließen.

`getMinimalCostCables(float maxFlow)`

Welche eine Liste von Kabelabschnitten zurückgibt. Finden Sie eine Menge an Wegen, die möglichst alle Siedlungen mit Strom abdeckt und dabei kostenminimal ist. Die Wege sollen wie in "getMinimalCostPath" aufgebaut sein und 'maxFlow' Stromeinheiten als maximalen Stromfluss pro Kabel nutzen. Sollte eine vollständige Abdeckung nicht möglich sein, streichen Sie sukzessive die Siedlungen mit den geringsten Einwohnern (wobei die Kabel weiterhin über diese Felder führen können). Geben Sie eine Liste von Kabelabschnitten zurück. Ein Kabelabschnitt soll dabei eine Liste von Strings sein. Die ersten zwei Einträge sollen den [x][y] Koordinaten des Ursprungs darstellen, der dritte und vierte Eintrag sollen die Koordinaten [x][y] des Ziels des Kabelabschnitts sein. Der 5. Eintrag soll angeben, wie viel Energie am Ursprungsfeld produziert wurde und durch dieses Kabel geleitet wird. Der 6. Eintrag soll darstellen, wie viel Energie dieses Kabels am Zielfeld genutzt wird (beachten Sie dabei die 0.1 Energieverlust pro Schritt). Der 7. Eintrag sei, wie viel Energie über dieses Kabel übertragen wird. Die Reihenfolge der Kabelabschnitte sei dabei egal.

Beispiel

```
grid = EnergyPlanner(5, 3)

for i in range(5):
    for j in range(3):
        grid.setCost(i, j, 10)
grid.setCost(0, 2, 1)
grid.setCost(0, 1, 1)
grid.setCost(0, 0, 1)
grid.setCost(1, 0, 1)
grid.setCost(2, 0, 1)
grid.setCost(2, 1, 1)
grid.setCost(2, 2, 1)
grid.setCost(3, 2, 1)
grid.setCost(4, 2, 1)
grid.setCost(4, 1, 1)
grid.setCost(4, 0, 1)

// [01, 01, 01, 10, 01]
// [01, 10, 01, 10, 01]
// [01, 10, 01, 01, 01]

grid.setSettlement(4, 0, 10, 5)
grid.setSettlement(4, 2, 5, 3)

grid.setEnergyPotential(0, 2, "wind", 15)
grid.setEnergyPotential(0, 2, "solar", 2)

setWindTurbine(0, 2)
setSolarPanel(0, 2)

grid.getCost(0, 2, 1, 2) // 11
grid.getCost(0, 2, 0, 1) // 2

grid.getMinimalCostPath(0, 2, 4, 0, 30)
// [
// [0,2],[0,1],[0,0],[1,0],[2,0],
// [2,1],[2,2],[3,2],[4,2],[4,1],[4,0]
// ]

getMinimalCostCables(30)
// [
// [0, 2, 0, 1, 16.0, 0.1, 16.0],
// [0, 1, 0, 0, 0.0, 0.1, 15.9],
// [0, 0, 1, 0, 0.0, 0.1, 15.8],
// [1, 0, 2, 0, 0.0, 0.1, 15.7],
// [2, 0, 2, 1, 0.0, 0.1, 15.6],
```

```
// [2, 1, 2, 2, 0.0, 0.1, 15.5],
// [2, 2, 3, 2, 0.0, 0.1, 15.4],
// [3, 2, 4, 2, 0.0, 5.1, 15.3],
// [4, 2, 4, 1, 0.0, 0.1, 10.1],
// [4, 1, 4, 0, 0.0, 10.1, 10.1],
// ]
```

Anhang: Vergleich der Fehlerkategorien McCall und Kölling zu Altadmri und Brown

Tabelle A.5.: Vergleich Fehlerkategorien McCall und Kölling zu Altadmri und Brown

Fehlerkategorien nach Altadmri und Brown	**Fehlerkategorien nach McCall und Kölling**
Falsch verstandene oder vergessene Syntax	
A	4.9
C	9.5, 9.6, 9.7, 9.8, 9.9, 9.11, 9.12, 9.13, 9.15, 9.16, 4.10, 5.6, 5.7
D	Keine konkrete Zuordnung, Oberkategorie: 9.0.
E	Keine konkrete Zuordnung, Oberkategorie: 9.0.
F	Keine konkrete Zuordnung, Abhängig vom spezifischen Fehler: 9.0, 9.2.
G	Keine konkrete Zuordnung, kann in die Kategorie 9.11 eingeordnet werden.
H	Keine konkrete Zuordnung, Oberkategorie: 2.0.
J	3.8
K	4.6
L	Keine konkrete Zuordnung, Oberkategorie: 9.0.
P	3.6
Typenfehler	
I	3.2, 3.4, 3.5, 3.6, 4.8
Q	Abhängig vom spezifischen Fehler: 4.5, 4.11.
Andere semantische Fehler	
B	Keine konkrete Zuordnung, Oberkategorie: 8.0.
M	3.9
N	Abhängig vom spezifischen Fehler: 4.3, 8.4.
O	Keine konkrete Zuordnung, Oberkategorie: 8.0.
R	Keine konkrete Zuordnung, Oberkategorie: 7.0.

Anhang: Vergleich Fehlerkategorien Altadmri und Brown zu McCall und Kölling

Tabelle A.6.: Vergleich Fehlerkategorien Altadmri und Brown

Fehlerkategorien nach Altadmri und Brown	**Fehlerkategorien nach McCall und Kölling**
1.0	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
1.1, 1.2, 1.3, 1.4, 1.5, 1.6	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
2.0	H
2.1, 2.2, 2.3, 2.4, 2.5	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
3.0	J
3.1	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
3.2	I
3.3	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
3.4; 3.5	I
3.6	P
3.7	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
3.8	J
3.9	M
4.0	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
4.1, 4.2	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
4.3	N
4.4	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
4.5	Q
4.6	K
4.7	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
4.8	I
4.9	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
4.10	C
4.11	Q
5.0	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
5.1, 5.2, 5.3, 5.4, 5.5	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
5.6, 5.7	C
5.8, 5.9	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
6.0	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
6.1, 6.2	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
7.0	R
7.1, 7.2, 7.3, 7.4	Keine Zuordnung, Typenfehler.
8.0	B
8.1, 8.2, 8.3	Keine Zuordnung, andere semantische Fehler.
8.4	N
8.5, 8.5.1, 8.5.2, 8.5.3, 8.5.4, 8.5.5, 8.5.6, 8.6, 8.7	Keine Zuordnung, andere semantische Fehler.

Tabelle A.7.: Vergleich Fehlerkategorien Altadmri und Brown, fortgesetzt

Fehlerkategorien nach Altadmri und Brown	**Fehlerkategorien nach McCall und Kölling**
9.0	Je nach Fehler: D, E, F, L
9.1	A
9.2	F
9.3, 9.4	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
9.5, 9.6, 9.7, 9.8, 9 9	C
9.10	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
9.11	Je nach Fehler: C, G
9.12, 9.13, 9.13.1, 9.13.2, 9.13.3	c
9.14	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
9.15, 9.16	C
10.0	Keine Zuordnung, falsch verstandene oder vergessene Syntax.
11.0	Keine Zuordnung.