TECHNISCHE UNIVERSITÄT
CHEMNITZ

Die praktische Anwendung von Syntaxaufgaben bei Programmieranfängern

Masterarbeit

zur
Erlangung des akademischen Grades
M.Sc.

Fakultät für Informatik
Professur Softwaretechnik

Eingereicht von: Dominik Gorgosch
Matrikel Nr.: [redacted]
Einreichungsdatum: 29.11.2021
Betreuerinnen: Prof. Dr. Janet Siegmund
Elisa Madeleine Hartmann

Danksagung

Ich danke meinen beiden Betreuerinnen Janet Siegmund und Elisa Madeleine Hartmann für die gute Zusammenarbeit und die hilfreichen Kommentare während der Ausarbeitung dieser Studie.

Außerdem möchte ich mich bei den Teilnehmern des Kurses Datenstrukturen bedanken, die meine Probanden waren.

Ich danke Jana und meiner Mutter für das Lesen und Kommentieren meiner Arbeit.

Schließlich danke ich auch Jonas für die Hilfe bei der Auswertung der Ergebnisse.

Abstract

Background: A variety of different articles have examined problems faced by novice programmers. Programming language syntax has been identified as a significant barrier to learning programming. Although mastering syntax is not cognitively demanding, it prevents novice programmers from learning more cognitively demanding skills, such as problem-solving.
Objective: To improve the teaching of introductory programming courses, we investigated the use of syntax exercises in data structures. We examined effects on program comprehension and programming tasks, as well as exam results.
Method: We conducted a pretest-posttest experiment with students and used syntax exercises as treatment. For the evaluation, we used program comprehension and programming tasks and analyze the exam results. For program comprehension tasks, we apply a remote eye tracker to measure visual attention. A total of 21 students participated in the experiment.
Results: On the one hand, a significant effect on the programming tasks and exam results was identified. On the other hand, there were no effects in the correctness of the program comprehension tasks, but in the response time and the number of clicks in Areas of Interest (AOI).
Conclusion: It can be summarized that syntax exercises have an effect on improving programming skills, but not on program comprehension. Although the visual attention changed and the response time improved, the subject group did not answer the tasks more correctly.
Future Work: Further studies are necessary to generalize the results. These could consist of optimized syntax exercises and contain more complex programs. In addition, the cognitive load could be measured with a stationary eye tracker (e.g. pupil dilation) or think-aloud protocols (at which points do students experience problems).

Keywords: syntax, cognitive load, novice programmer, program comprehension

Abstract

Hintergrund: Es gibt eine Vielfalt verschiedener Artikel, in denen Probleme von Programmieranfängern untersucht wurden. In einigen dieser wurde die Syntax von Programmiersprachen als signifikante Barriere für das Programmierenlernen identifiziert. Obwohl die Beherrschung der Syntax kognitiv wenig anspruchsvoll ist, hindert sie Programmieranfänger daran, kognitiv anspruchsvollere Fähigkeiten wie etwa die Problemlösung zu erlernen.
Forschungsziel: Zur Verbesserung der Lehre von Programmiereinführungskursen soll der Einsatz von Syntaxaufgaben im Kurs Datenstrukturen untersucht werden. Dabei werden Auswirkungen bei Programmverständnis- und Programmieraufgaben sowie den Klausurergebnissen betrachtet.
Forschungsmethode: Es wurde eine Pretest-Posttest-Studie bei Studierenden durchgeführt. Als Treatment wurden Syntaxaufgaben genutzt. Zur Überprüfung wurden Programmverständnis- und Programmieraufgaben eingesetzt sowie die Klausurergebnisse betrachtet. Bei den Programmverständnisaufgaben wurde ein Remote Eye Tracker zum Messen der visuellen Aufmerksamkeit angewendet. Insgesamt haben 21 Studierende an dem Experiment teilgenommen.
Ergebnisse: Es wurde ein signifikanter Effekt bei den Programmieraufgaben und Klausurergebnissen herausgearbeitet. Hingegen gab es bei den Programmverständnisaufgaben in der Korrektheit keine Auswirkungen, jedoch in der Beantwortungszeit und der Klickanzahl.
Schlussfolgerung: Es lässt sich zusammenfassen, dass Syntaxaufgaben einen Effekt bei der Verbesserung der Programmierfähigkeit, jedoch keinen beim Programmverständnis haben. Obwohl sich die visuelle Aufmerksamkeit verändert und die Beantwortungszeit verbessert hat, haben die Probanden die Aufgaben nicht korrekter beantwortet.
Zukünftige Arbeiten: Zur Generalisierung der Ergebnisse sind weitere Studien notwendig. Diese könnten aus optimierten Syntaxaufgaben bestehen und komplexere Programme enthalten. Außerdem könnte mit einem stationären Eye Tracker (z.B. Vergrößerung der Pupillen) oder Think-Aloud-Protokollen (an welchen Stellen treten Probleme bei Studierenden auf) die kognitive Belastung gemessen werden.

Keywords: Syntax, kognitive Belastung, Programmieranfänger, Programmverständnis

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AOI Areas of Interest

AuP Algorithmen und Programmierung

CEdR Computing Education Research

CLT Die Theorie der kognitiven Belastung

CS1 Computer Science 1

CS2 Computer Science 2

PVL Prüfungsvorleistungen

OPAL Online-Plattform für Akademisches Lehren und Lernen

PA Programmieranfänger

TUC Technische Universität Chemnitz

1 Einleitung

> "Suppose I assigned you to write a fifteen-page paper, due two months from now, on Napoleon's invasion of Russia. [...] Now suppose I told you that the paper must be written in Swedish, using a quill pen. Now what would you need to know? [...] And if I expected you to learn all those things, from scratch, in one semester, you'd think I was nuts. That's what we'll be doing in Computer Science 1 (CS1), trying to learn half a dozen different levels of knowledge at once." [32, S. 286]

Programmierenlernen stellt viele Programmieranfänger (PA)[1] vor große Herausforderungen. Anhand des Zitats von Stephen Bloch wird ersichtlich, welche Anforderungen Programmiereinführungskurse an ihre Teilnehmer stellen. Sie müssen Wissen in einer neuen Problemdomäne erlangen. Dazu müssen sie eine neue Sprache erlernen, die nicht einfach in einem Texteditor oder auf einem Blatt Papier ausgeführt werden kann, sondern die Benutzung einer Entwicklungsumgebung erfordert. Außerdem besitzen Programmiersprachen wenig bis keine Gemeinsamkeiten mit einer natürlichen Sprache. Im Laufe der vorliegenden Arbeit soll herausgefunden werden, ob gezieltes Lernen von einem dieser Aspekte - der Syntax einer Programmiersprache - eine Auswirkung auf den komplexen Vorgang des Programmierenlernens hat.

1.1 Problemstellung

Seit mehr als 40 Jahren werden Programmiereinführungskurse in der Computing Education Research (CEdR)-Literatur als schwierig beschrieben, wodurch in diesen hohe Abbruch- sowie Durchfallquoten auftreten [54, S. 330]. Bis zur Untersuchung von Bennedsen und Caspersen im Jahre 2007 bestanden diese Annahmen jedoch ausschließlich aus anekdotischer Evidenz oder Untersuchungen an wenigen Universitäten [6, S. 32]. In ihrer weltweiten Untersuchung an 63 Universitäten [6, S. 33] lag die Ausfallrate bei 33 % [6, S. 35]. Anhand der Studie von Luxton-Reilly et al. kann verglichen werden, ob dies einer hohe Ausfallrate entspricht. Jedoch sei darauf hinzuweisen, dass dies keine weltweite Studie war, da sie nur an neuseeländischen Universitäten durchgeführt worden ist. 2016 untersuchten sie,

[1] Für eine bessere Lesbarkeit dient ein Verzicht auf die gleichzeitige Verwendung weiblicher und männlicher Sprachformen in dem gesamten Forschungsbeleg. Somit gelten sämtlich neutral verwendete Personenbezeichnungen gleichermaßen für beiderlei Geschlecht. Im Falle, dass eine Personenbezeichnung ausschließlich auf ein Geschlecht verweist, wird nur die entsprechende Genderform angewandt.

wie hoch die Bestehensrate in verschiedenen Einführungskursen an neuseeländischen Universitäten war [32]. In den letzten 30 Jahren lag die Bestehensrate bei ca. 67 % in Programmiereinführungskursen, im Gegensatz zu 82 % bei anderen Einführungskursen. Diesem Unterschied kann entnommen werden, dass Programmiereinführungskurse als schwieriger empfunden werden als Einführungskurse in natürlichen Sprachen oder anderen Fachgebieten [32, S. 285]. Bennedsen und Caspersen wiederholten ihre Studie 2019 und stellten mit 28 % eine um 5 % verbesserte Fehlerrate in Programmiereinführungskursen fest [7, S. 30]. Es sei jedoch darauf hinzuweisen, dass nicht die gleichen Universitäten teilgenommen haben und die Zahl mit 170 Universitäten deutlich höher lag als 2007 [7, S. 32]. Die herausgearbeiteten Bestehensraten können zur Annahme führen, dass Programmierenlernen für viele PA eine Herausforderung darstellt. Es sei darauf hinzuweisen, dass in der vorliegenden Arbeit kein klassischer Programmiereinführungskurs, sondern ein zweiter Programmierkurs (in der CEdR als Computer Science 2 (CS2) bezeichnet) untersucht wird. Für diese gibt es keine größer angelegten Studien, in denen die Bestehensraten verglichen worden sind. Jedoch hat Raigoza die Daten von Studierenden einer amerikanischen Universität in drei aufeinander folgenden Jahren verglichen und eine Bestehensrate von 75% in CS2 herausgearbeitet [52].

Es gibt verschiedene Gründe, die in der CEdR herausgearbeitet worden sind, welche als Ursache für die Schwierigkeit des Programmierens gesehen werden. Zur Programmierung wird eine Vielzahl verschiedener Fähigkeiten benötigt. 2018 haben Medeiros et al. eine systematische Übersichtsarbeit über Lehren und Lernen in Programmiereinführungskursen herausgegeben [42]. Denn bis zu diesem Zeitpunkt gab es keinen Konsens über die grundlegenden Herausforderungen von PA und eine Einteilung dieser Herausforderungen in verschiedene Kategorien. Ähnliche Werke haben sich auf die Lehre und nicht die Herausforderungen von PA konzentriert, wie etwa Pears et al. [49]. Deswegen haben Medeiros et al. versucht die Herausforderungen für PA zu kategorisieren [42, S. 6]. Jedoch sei auf du Boulays fünf Themengebiete [15] oder auch auf Robins *programming framework* [54] hinzuweisen, welche auch verschiedene Anforderungen an PA aufzählen. Da beide nicht auf einer systematischen Übersichtsarbeit basieren und weniger detailliert aufgeschlüsselt sind, werden Medeiros Herausforderungen als Grundlage genommen.

Ein Programmiereinführungskurs enthält eine Vielzahl an Themen. Darunter fallen Problemlösung, grundlegende Programmierkonzepte, die Syntax und Semantik einer Programmiersprache sowie die Nutzung dieser Sprache zum Schreiben von Lösungen [42, S. 1]. Diese Themen können in Problemformulierung, Lösungsausdruck sowie Lösungsausführung und -evaluierung unterteilt werden [42, S. 5]. Problemformulierung besteht aus Problemlösung, der Abstraktheit der Programmierung und algorithmischem Denken [42, S. 4]. Problemlösung beinhaltet das Verständnis des Problems, Identifizieren der Schlüsselinformationen und die Erstellung eines Plans zur Lösung [42, S. 4]. Nach der Problemformulierung können PA mit dem Schreiben der Lösung beginnen [42, S. 6]. Dafür benötigen sie syntaktisches Wissen über eine Programmiersprache. Aber auch Wissen über die Anwendung von Kontroll- und Datenstrukturen. Kontrollstrukturen sind etwa Bedingungen, Rekur-

sion oder Schleifen. Hingegen gehören zu Datenstrukturen bspw. Variablen oder Arrays [42, S. 6]. Daneben müssen PA auch die Struktur eines Programms verstehen, um Interaktionen zwischen verschiedenen Objekten oder Klassen zu erkennen [42, S. 6]. Nachdem PA ihre Problemlösung in einer Programmiersprache geschrieben haben, sollten sie ihre Lösung testen. Dadurch können etwaige Fehler erkannt werden. Für das Verstehen der Lösungsausführung und -evaluierung benötigen PA Tracing- und Debuggingfähigkeiten [42, S. 6].

Beim ersten Kontakt zwischen dem Lernenden und einer Programmieraufgabe erleiden viele PA einen *shock*, der auf die Verknüpfung der verschiedenen Herausforderungen und Bestandteile des Programmierens zurückzuführen ist [15, S. 58]. Neben den Fähigkeiten müssen PA auch noch mit einer komplexen Entwicklungsumgebung arbeiten [16, S. 65]. Es gibt zwar auch Entwicklungsumgebungen, die besonders für die Lehre geeignet sind, wie etwa BlueJ für Java, aber häufig werden eher in der Industrie geläufige Lösungen genutzt wie etwa Eclipse.

Programmieren ist nicht nur eine komplexe Fähigkeit und die Bestehensraten von Programmiereinführungskursen sind gering, sondern Studierende weisen auch in ihren Programmierfähigkeiten Defizite auf. Jedoch gibt es keine genauere Definition, was Programmierfähigkeit bedeutet und wie diese gemessen wird. Für die Bewertung der Leistungen von Studierenden werden oft die sogenannten McCracken Studien genannt, in denen die Leistung der Studierenden der Einschätzung der Lehrenden gegenübergestellt worden sind [40,71]. Hingegen stellte Robins bimodale Ergebnisse in Programmiereinführungskursen fest [54, S. 332f.]. In diesen gibt es besonders viele sehr gute Studierende, aber auch besonders viele sehr schlechte. Bisher konnte noch nicht festgestellt werden, welche Unterschiede es zwischen diesen beiden Gruppen gibt. Demnach sollten leistungsstarke Studierende genauer untersucht werden, um Unterschiede zu leistungsschwachen Studierenden zu erkennen. Dadurch könnte ein Lehrplan mit einem Schwerpunkt auf der Verbesserung leistungsschwacher Studierender entwickelt werden, welcher aber auch nicht leistungsstarke Studierende unterfordert. Trotz dessen in der CEdR-Literatur verschiedene Lehrmethoden herausgearbeitet wurden, gibt es keine Blaupause für die perfekte Lehre und Lehrende können selbstständig entscheiden, mit welchen Methoden und Tools sie Programmieren vermitteln. Nach der Erläuterung des Problems folgt nun die Nennung der Forschungsziele dieser Arbeit.

1.2 Forschungslücke und Ziel

Programmieren stellt wohl die wichtigste Fähigkeit dar, die Informatikstudierende in ihrem Studium lernen sollen [53, S. 163]. Aufgrund der hohen Komplexität und vieler Bestandteile bzw. Elemente, die untereinander agieren, stellt Programmierenlernen für viele PA eine Hürde dar [10, S. 75]. In traditionellen Lehrveranstaltungen werden Modellierungsaspekte und die theoretische Repräsentation verschiedener Programmierkonstrukte und -konzepte an einem Beispiel gezeigt [56, S. 77]. Von den Studierenden wird erwartet, dass sie eine ausreichende Kompetenz in Lese-

und Verständnisfähigkeiten besitzen. Dennoch stellt eine Beherrschung dieser häufig ein Problem dar. Demgegenüber steht das Lernen einer natürlichen Sprache, wobei zuerst lesen und danach schreiben beigebracht und geübt wird [45, S. 2]. Dadurch besitzen die meisten Menschen in einer Fremdsprache bessere Lese- als Schreibfähigkeiten [45, S. 2]. In der vorliegenden Arbeit soll das Programmieren unterteilt werden in Syntax und Problemlösung als auch in Schreiben und Lesen von Programmen. Demnach ist das Forschungsziel der vorliegenden Arbeit die Evaluierung eines Ansatzes zur Verbesserung der Programmierfähigkeiten von PA. Im vorherigen Abschnitt wurde bereits herausgearbeitet, dass Programmieren nicht nur aus dem Lernen einer Programmiersprache besteht. Anhand der Problemstellung bieten folgende Aspekte einen Nährboden für die vorliegende Forschung: (1) hohe Abbruchquoten von PA in Programmiereinführungskursen, (2) geringe Programmierfähigkeiten von PA und (3) eine Überforderung der kognitiven Kapazitäten von PA durch die Vielfalt und Interaktivität an Themen und Fähigkeiten, die in einem Programmiereinführungskurs gelehrt werden. Das Ziel der vorliegenden Arbeit ist somit, die Rolle der Syntax und ein primäres Lernen dieser durch den gezielten Einsatz von Aufgaben, aufzuzeigen und dies empirisch zu analysieren, weil in verschiedenen Studien die Rolle der Syntax als extrinsische kognitive Belastung festgestellt wurde [17, 18, 21, 67]. Dafür wurden im Kurs Datenstrukturen, einem zweiten Programmierkurs, an der Technische Universität Chemnitz (TUC), Syntaxaufgaben eingesetzt. Diese bestanden aus Aufgaben mit Fehlern, die identifiziert und verbessert werden mussten, damit ein Programm kompilierbar wurde. Zur Evaluierung dieses Experiments wurde ein Pretest-Posttest- und Within-Subject-Design gewählt. Insgesamt haben 21 Studierende an dem Experiment im Sommersemester 2021 teilgenommen. Die vorliegende Arbeit legt den Fokus auf die Programmierfähigkeit, das Programmverständnis und die Klausurergebnisse. Diese Eingrenzung beruht auf der Tatsache, dass es einen Unterschied zwischen Programmieren und Programmverständnis gibt, jedoch beide Aspekte relevant für die Programmierausbildung sind. Professionelle Programmierer verbringen etwa die Hälfte ihrer Arbeitszeit mit dem Lesen und Verstehen von Programmiertexten, die bspw. von anderen geschrieben worden sind [61, S. 1]. Für dieses Experiment wurden im Pre- und Posttest jeweils zehn Programmverständnisaufgaben und zwei Programmieraufgaben eingesetzt. Für die Überprüfung des Programmverständnisses wurde unter anderem mit REyeker [43] ein Remote Eye Tracker eingesetzt, der die Augenbewegungen von Studierenden nachempfinden sollte. Dadurch konnte neben der Korrektheit und Beantwortungszeit bei Programmverständnisaufgaben auch die visuelle Aufmerksamkeit der Probanden betrachtet werden. Überdies wurde im Pretest ein Fragebogen eingesetzt, der die demografischen Daten der Probanden abgefragt hat. Zwischen dem Pre- und Posttest wurden als Treatment vier verschiedene Tests mit Syntaxaufgaben, zu den Konstrukten if-else-Bedingungen, for-, while- und do-while-Schleifen eingesetzt. Die vorliegende Ausarbeitung soll eine empirische Untersuchung über den Einsatz von Syntaxaufgaben bei PA darstellen.

Die Forschungslücke der vorliegenden Arbeit stellt den Einsatz in einem zweiten Programmierkurs und die Betrachtung des Programmverständnisses dar. Denn

bisher fanden alle Studien in CS1, dem ersten universitären Programmierkurs, statt, hingegen stellt Datenstrukturen, gemäß CEdR den zweiten Programmierkurs CS2 dar. Jedoch müssen die Probanden in Datenstrukturen mit Java eine neue Programmiersprache lernen, die andere syntaktische Merkmale besitzt als C. Somit führt das Lernen einer neuen Programmiersprache zur wieder auftretenden extrinsischen kognitiven Belastung von PA durch die neue Syntax. Zur Untersuchung dieses Ansatzes sollen folgende Forschungsfragen beantwortet werden:

Forschungsfrage 1: Wie beeinflussen Syntaxaufgaben die Beantwortungszeit, Korrektheit und visuelle Aufmerksamkeit von PA in Programmverständnisaufgaben?

Forschungsfrage 2: Wie beeinflussen Syntaxaufgaben die Beantwortungszeit und Korrektheit von PA in Programmieraufgaben?

Forschungsfrage 3: Wie beeinflussen Syntaxaufgaben die Ergebnisse von PA in der Klausur?

Somit bietet diese Ausarbeitung eine Hilfestellung für die Lehre und stellt die praktische Umsetzung eines Konzeptes dar und zeigt, ob dieses zur Verbesserung der Kompetenz von PA beigetragen hat.

1.3 Struktur

Die vorliegende Arbeit ist folgendermaßen gegliedert:

Kapitel 2 Im zweiten Kapitel wird der theoretische Rahmen der Arbeit vorgestellt. Dieser beinhaltet die kognitive Architektur des Menschen und eine Einführung in die CLT. Daneben wird der Begriff PA definiert und was in der Lehre von diesen gefordert wird. Zudem werden für die vorliegende Arbeit relevante Begrifflichkeiten, wie die Programmierfähigkeit, das Programmverständnis und Syntaxaufgaben definiert. Neben dem Einsatz von Syntaxaufgaben werden weitere Interventionsmöglichkeiten dargestellt, welche die Kompetenz von PA steigern sollen. Schlussendlich werden verschiedene Messmöglichkeiten für das Programmverständnis und die Programmierfähigkeit genannt.

Kapitel 3 In Kapitel 3 wird der Forschungsstand über den Einsatz von Syntaxaufgaben bei PA dargestellt.

Kapitel 4 Auf den vorherigen Kapiteln aufbauend wird in Kapitel 4 die Methodik der Arbeit und damit der Aufbau des Experiments beschrieben. Dazu zählt eine Konkretisierung der Operationalisierung, eine Darstellung der Probanden sowie des Materials und der Aufgaben. Zum Schluss wird das experimentelle Design präsentiert.

Kapitel 5 Aufbauend auf dem vierten Kapitel wird im fünften Kapitel der Ablauf des Experiments beschrieben.

Kapitel 6 Nach der Erläuterung des Experiments folgt eine Darstellung der Ergebnisse.

Kapitel 7 Im siebten Kapitel werden die Ergebnisse diskutiert, die Forschungsfragen beantwortet und das Experiment reflektiert. Außerdem werden Einschränkungen der Validität genannt und der Bezug zu Related Work beschrieben.

Kapitel 8 Schlussendlich werden im achten Kapitel die Ergebnisse der Arbeit zusammengefasst und etwaige zukünftige Arbeiten erläutert.

2 Theoretischer Rahmen

Im folgenden Kapitel wird der theoretische Rahmen dargestellt, der für das Verständnis der restlichen Arbeit benötigt wird. Zur Untersuchung der Effektivität von Syntaxaufgaben bei PA wird zuerst in Abschnitt 2.1 die kognitive Architektur des Menschen beschrieben. Daraufhin stellt Abschnitt 2.2 die CLT dar, welche eine zentrale Rolle bei der Unterrichtsplanung spielt. Im Abschnitt 2.3 werden Programmieranfänger beschrieben. Dabei wird auf die Eigenschaften von diesen eingegangen, welche Funktion die Lehre bei der Ausbildung hat, die Rolle der CLT in der Programmierung und wie durch eine zu hohe kognitive Belastung Probleme entstehen können. Anschließend folgen im Abschnitt 2.4 verschiedene Interventionsmöglichkeiten, die zu einer Verbesserung der Ergebnisse in Programmiereinführungskursen führen können. Der theoretische Hintergrund endet mit Messmöglichkeiten zur Effektivität der Interventionsmöglichkeiten, welche in der vorliegenden Arbeit genutzt werden.

2.1 Kognitive Architektur des Menschen

Alle Lernaktivitäten von Menschen finden in zwei Gedächtnistypen statt: dem Kurzzeit- und dem Langzeitgedächtnis. Dabei stellt das Kurzzeitgedächtnis den sogenannten aktiven Part dar. In den folgenden Unterabschnitten wird nun zuerst die Rolle des Arbeitsgedächtnisses präsentiert. Danach folgt eine Erläuterung des Langzeitgedächtnisses. Schlussendlich werden fünf basale Prinzipien genannt, die entscheidend für die Bildung von Wissen sind.

2.1.1 Das Arbeitsgedächtnis

Im Arbeitsgedächtnis werden Informationen nur für einen kurzen Zeitraum gespeichert. Das Arbeitsgedächtnis ist, im Gegensatz zum Langzeitgedächtnis, limitiert, sowohl in der Kapazität als auch in der Dauer, in welcher es Informationen halten kann [3, S. 115]. In der Literatur wurde früher von sieben ± zwei Informationsblöcken ausgegangen, die im Arbeitsgedächtnis bearbeitet werden können [4, S. 126]. Heutzutage wird von vier ± zwei Elementen ausgegangen [4, S. 126]. Jedoch kann ein Element aus *chunks* - einer Anzahl anderer Elemente - bestehen. Als Beispiel eines chunks kann eine mehrstellige Zahl genannt werden, welche genauso verarbeitet werden kann, wie eine einstellige Zahl. Für das Arbeitsgedächtnis ist sowohl eine einstellige, als auch eine mehrstellige Zahl eine Informationseinheit bzw. ein chunk [55, S. 236]. Damit spielt Chunking eine zentrale Rolle für einen effektiven

Einsatz des Arbeitsgedächtnisses [55, S. 236]. Denn durch die Limitationen der Kapazität müssen Informationen zu Wissensstrukturen gebündelt werden, da ansonsten Aufgaben nicht gelöst werden können. Bei der Betrachtung von Programmieraufgaben wird die Bedeutung von chunks ersichtlich, denn diese Aufgaben beinhalten viele Informationen über den aktuellen Zustand, ausgeführte Prozesse, den Gesamtaufbau und Ziele des Programms, Programmiersprachen und Werkzeuge. Um das alles zu handhaben, benötigen Programmierer Schemata, die als chunks in das Arbeitsgedächtnis geladen werden. Außerdem können nur Teile des Programms im Arbeitsgedächtnis gehalten werden. [55, S. 236]

2.1.2 Das Langzeitgedächtnis

Im Kontrast zu den geringen Kapazitäten des Arbeitsgedächtnisses steht das Langzeitgedächtnis, welches eine quasi unbegrenzte Kapazität besitzt. Dieses Fassungsvermögen wird durch die Bildung von Schemata ermöglicht. Ein Schema besteht aus einer Vielzahl an Informationen. Die Entstehung komplexerer Schemata ermöglicht eine Automatisierung der Bildung dieser. Neben der Tatsache, dass Schemakonstruktion und Automatisierung zentrale Bestandteile des Lernens sind, haben sie auch einen positiven Effekt auf die kognitive Belastung. Je komplexer die Schemata eines Lernenden sind, desto geringer ist die kognitive Belastung beim Lernen. Im Abschnitt 2.2 wird auf die effiziente Nutzung von Arbeits- und Langzeitgedächtnis eingegangen, welche zu Veränderungen im Langzeitgedächtnis führt und damit ein zentraler Bestandteil der CLT ist. [4, S. 126]

2.1.3 Fünf basale Prinzipien

Für Sweller et al. besteht die kognitive Architektur des Menschen aus fünf basalen Prinzipien, die ein Informationsverarbeitungssystem darstellen: das Ansammeln von Informationen durch das Informationsspeicher-Prinzip, das Erwerben von Informationen mithilfe des Prinzips des Entlehnens und Reorganisierens sowie durch das Prinzip der zufälligen Generierung, die Interaktion mit der externen Umwelt durch das Prinzip der begrenzten Veränderung sowie durch das Prinzip der Organisation der Umwelt und der Verknüpfung. Diese fünf Prinzipien stellen die Grundlage für die Aneignung von Wissen dar. Durch die Nutzung der genannten Prinzipien soll mehr Wissen im Langzeitgedächtnis gespeichert werden. [4, S. 124]

Das Informationsspeicher-Prinzip besagt, dass ein "massiver Informationsspeicher" benötigt wird, um die Vielzahl komplexer Informationen handhaben zu können. Aufgrund der Kapazität des Speichers ist ein Auseinandersetzen und Interagieren in unterschiedlichen Umweltbedingungen möglich. [4, S. 125]

Das Langzeitgedächtnis gilt als zentrales Element der Steuerung der meisten menschlichen Aktivitäten. Die Informationen des Langzeitgedächtnisses werden durch das Prinzip des Entlehnens und Reorganisierens erworben, wie etwa durch Hören und Lesen. Damit besteht die Grundlage des Wissens fast vollständig aus dem Langzeitgedächtnis anderer Menschen. Die so gewonnenen Informationen werden

jedoch verändert und neu konstruiert. Dieser Prozess der Repräsentationsbildung besteht in der Erstellung von Schemata. Ein Schema besteht aus einer Menge von Informationen, welche jedoch als einzelnes Element gespeichert werden. Der Unterschied zwischen Anfängern und Experten besteht in der Fülle domänenspezifischer Schemata, die diese besitzen. Demnach hängt die Kompetenz in einem Bereich von der Anzahl und Komplexität der Schemata ab. [68, S. 33f.]

Jedoch muss auch neues Wissen generiert werden, dies geschieht bei der Problemlösung oder eine "Trial-and-Error"-Vorgehensweise. Bei der Problemlösung werden Informationen aus dem Langzeitgedächtnis genutzt, die durch Entlehnen und Reorganisieren entstanden sind. Dennoch treffen Lernende auf Probleme, bei denen es zwei oder mehr mögliche Lösungen gibt. Hier greift das Prinzip der zufälligen Generierung. Die unterschiedlichen Schritte werden dann ausgeführt und der wirksamste davon wird ausgewählt. Da eine Vorhersage der Wirksamkeit der möglichen Schritte nicht gegeben ist, wird der Schritt zufällig ausgewählt. Veränderungen der Informationen des Langzeitgedächtnisses durch das Prinzip der zufälligen Generierung sind inkrementell und langsam. Denn große, schnelle und zufällige Änderungen könnten Schemata im Langzeitgedächtnis beschädigen. Hingegen belasten kleine Änderungen einen Großteil der gespeicherten Informationen nicht. Somit kann das Arbeitsgedächtnis als Barriere gesehen werden, die sicherstellt, dass nur eine begrenzte Anzahl von Veränderungen im Langzeitgedächtnis stattfinden. Ansonsten kann es zu einer Überlastung des Arbeitsgedächtnisses kommen, wodurch Schemata nicht mehr effektiv konstruiert werden und in das Langzeitgedächtnis übertragen werden[1]. Anhand des Prinzips der zufälligen Generierung wurde ersichtlich, dass die Generierung neuer Informationen nicht effektiv ist. Demnach sollte ein Unterrichtskonzept darauf beruhen, dass Informationen vom Dozenten entlehnt und reorganisiert werden. Außerdem sollte unnötige kognitive Belastung so weit wie möglich entfernt werden. Nach der CLT ist die Anweisung effektiver, je mehr Schemata entlehnt werden und nicht neu erstellt werden müssen. Jedoch sei darauf hinzuweisen, dass dieses Prinzip nur für Anfänger gilt, die mit Informationen umgehen, mit denen sie nicht vertraut sind. Informationen, die als Schemata im Langzeitgedächtnis gespeichert sind, unterliegen diesen Einschränkungen nicht. [68, S. 35-38]

Anders als beim Umgang mit neuen Informationen gibt es beim Arbeitsgedächtnis keine Begrenzung für den Umgang mit komplexen Schemata. Das Prinzip der Organisation der Umwelt und der Verknüpfung erklärt, wie große Mengen organisierter Informationen vom Langzeit- in das Arbeitsgedächtnis übertragen werden können. Das Prinzip der Organisation der Umwelt und der Verknüpfung kann als Rechtfertigung für die menschliche Kognition dienen: die Fähigkeit, sich in einer komplexen äußeren Umgebung zurechtzufinden. Durch das Prinzip der begrenzten Veränderung funktionieren die vorherigen vier Lernprinzipien ohne eine Zerstörung des Informationsspeichers. Sofern Informationen gespeichert sind, ermöglicht das

[1] Es gibt eine Reihe von CLT-Effekten, die zur Senkung der intrinsischen Belastung von Programmieraufgaben eingesetzt werden. Eine Aufzählung dieser würde den Rahmen der vorliegenden Arbeit sprengen. Für weitere Informationen, siehe [37]

Prinzip, dass diese Informationen für geeignete Maßnahmen genutzt werden. [68, S. 38f.]

Anhand der genannten Prinzipien wurde mit der Konstruktion von Schemata ein wichtiger Aspekt der Lehre herausgearbeitet. Daneben muss die Nutzung dieser Schemata automatisiert werden. Automatisierung geschieht, wenn Wissen unbewusst, anstatt bewusst im Arbeitsgedächtnis, verarbeitet wird. Problemlösung durch die Nutzung automatisierten Wissens ist deutlich einfacher und schneller als ein bewusstes Nutzen. Die Automatisierung von niedrigeren Schemata ist essenziell für die Bildung höherer Schemata. Ohne die automatische Verarbeitung der Buchstaben des Alphabets wäre es schwierig, diese Buchstaben zu Wörtern und Sätzen zu kombinieren, um das Lesen oder Schreiben zu ermöglichen. Beim Erlernen neuer Schemata werden oft neue Informationen mit Informationen, die im Langzeitgedächtnis gespeichert werden, kombiniert. Bei der Entwicklung von Schemata werden diese grundlegend durch das Prinzip des Entlehnens und Reorganisierens des Dozenten aufgebaut. Im Unterabschnitt 2.3.6 wird dargestellt, wieso die Generierung von Informationen ineffektiv ist. Daher beschäftigt sich CLT viel mehr mit Techniken der Präsentation für Lernende, als mit der Generierung neuer Informationen durch Lernende. [68, S. 34f.]

2.2 Kategorien der kognitiven Belastung

In den vorherigen Abschnitten wurde die Rolle der kognitiven Architektur für die Lehre herausgearbeitet. Maßgebliches Ziel der Lehre sollte sein, Wissen in Schemata zu untergliedern, welche im Langzeitgedächtnis gespeichert werden. Dafür müssen jedoch neue Informationen im Arbeitsgedächtnis verarbeitet werden. Bei der Verarbeitung von neuen Informationen ist das Arbeitsgedächtnis sowohl in seinen Kapazitäten limitiert als auch in der Dauer der Verarbeitung. Aufgrund dessen wurde die CLT von John Sweller und Paul Chandler entwickelt, welche die Relevanz des Arbeitsgedächtnisses beim Lernen beschreibt [69]. Die CLT ist ein neues Konzept innerhalb der CEdR, dessen Ursprünge in der Kognitionswissenschaft liegen [54, S. 343]. Als kognitive Belastung wird der mentale Aufwand beschrieben, welcher für das Bearbeiten einer Aufgabe benötigt wird [54, S. 343]. Demnach besitzen alle Lehrmaterialien eine kognitive Belastung. Diese können in intrinsische und extrinsische Belastungen unterteilt werden. Außerdem gibt es mit lernförderlichen Ressourcen eine dritte Kategorie, welche von intrinsischer und extrinsischer Belastung abhängt. Aus allen drei Belastungen ergibt sich die gesamte kognitive Belastung. Bei einer zu hohen kognitiven Belastung kann diese die Kapazität des Arbeitsgedächtnisses übertreffen. Dadurch wird die Informationsverarbeitung und damit auch die Effektivität des Lernens beeinträchtigt. Wenn die Gesamtbelastung des Arbeitsgedächtnisses zu hoch ist, ist die Wahrscheinlichkeit, nützliche Veränderungen des Langzeitgedächtnisses zu erreichen, verringert. [68, S. 40]

2.2.1 Intrinsische Belastung

Die intrinsische kognitive Belastung beschreibt den Aufwand, der durch die Art der Aufgabenstellung gegeben ist. Diese besteht aus einer Kombination der Schwierigkeit der Aufgabe und den Charakteristiken des Lerners [3, S. 115f.; 54, S. 343f.; **?**, S. 49]. Verschiedene Aufgaben beinhalten verschiedene intrinsische Belastungen, aber eine Reduzierung dieser Belastung ist für eine spezifische Aufgabe nicht direkt möglich. Eine der Hauptdeterminanten der intrinsischen Belastung ist die Elementinteraktivität [3, S. 115]. Diese beschreibt das Ausmaß interagierender Elemente einer Aufgabe, welche gleichzeitig im Arbeitsgedächtnis gehalten werden müssen [54, S. 344]. Programmieraufgaben haben eine hohe Elementinteraktivität und damit auch eine hohe intrinsische Belastung. Ein Beispiel für eine geringe kognitive Belastung ist das Lernen des Grundwortschatzes einer Fremdsprache, da jedes Wort unabhängig gelernt wird. Denn es gibt keine Interaktivität zwischen diesen [19, S. 3]. Der effektivste Weg zur Bewältigung dieser hohen Belastung in der Programmierung besteht darin, eine bekannte Eigenschaft des Arbeitsgedächtnisses auszunutzen, nämlich die Fähigkeit, Elemente zu einem sinnvollen größeren Teil zusammenzufügen [54, S. 344]. Dafür werden Schemata aus dem Langzeit- in das Arbeitsgedächtnis geladen. Nach dem Prinzip der Organisation der Umwelt und der Verknüpfung gibt es im Langzeitgedächtnis keine Grenzen für die Verarbeitung von Schemata und damit organisierter Informationen [68, S. 42]. Je größer deren Repertoire für eine Domäne ist und je besser organisiert diese sind, desto geringer ist die intrinsische Belastung für den Lernenden [68, S. 42]. Beruhend auf dem Prinzip der zufälligen Generierung sowie der begrenzten Veränderungen gibt es Einschränkungen, wie viele Informationen im Langzeitgedächtnis verändert werden können. Aufgrund dessen kann nur ein Teil neuer Informationen so organisiert werden, dass es zu einer Änderung des Langzeitgedächtnisses kommt [68, S. 42].

2.2.2 Extrinsische Belastung

Die extrinsische Belastung wird durch das verwendete Material im Lehrentwurf beeinflusst, bspw. sei hier eine ungünstige Aufgabenbeschreibung zu nennen [3, S. 116; 19, S. 3]. Ein Haupttreiber der Entwicklung der CLT lag in einer Reduzierung der extrinsischen Belastung [68, S. 42]. Eine hohe intrinsische und extrinsische Belastung können kombiniert zu einer Überlastung des Arbeitsgedächtnisses führen, dies tritt häufig bei PA auf [19, S. 3]. Wenn die Aufgabe eine hohe intrinsische Belastung verursacht, muss das Material so gestaltet werden, dass die extrinsische Belastung so gering wie möglich gehalten wird [19, S. 3]. Es gibt verschiedene Studien zur extrinsischen Belastung, welche unterschiedliche Methoden zur Reduzierung entwickelt haben. Bspw. haben das Zeigen ausgearbeiteter Lösungsbeispiele oder auch der Einsatz von Syntaxaufgaben [21] die extrinsische Belastung verringert [19, S.3; 3, S. 116].

Wenn ein Lehrentwurf dazu führt, dass Schemata nicht verändert werden, dann ist die extrinsische Belastung zu hoch. Dies kann bei der Lösungsfindung anhand

des Prinzips der zufälligen Generierung anstatt durch Entlehnen und Reorganisieren geschehen oder wenn der Fokus des Unterrichts auf der zufälligen Generierung liegt. Außerdem kann auch ein Ignorieren des Prinzips der begrenzten Veränderungen ein Problem darstellen. Daneben kann auch ein Nichtbeachten der Kapazitätsgrenze des Arbeitsgedächtnisses zu einer fehlenden Veränderung des Langzeitgedächtnisses führen. Wenn von Lernenden verlangt wird, dass sie eine Regel selbst entdecken müssen, benötigt dies sehr viel Aufwand. Hierfür kann als Beispiel ein Entdecken von syntaktischen Regeln einer Programmiersprache genannt werden. Dabei wird das Prinzip der begrenzten Veränderungen verletzt. Die erworbenen Kenntnisse hängen nur minimal von der Wissensvermittlung ab, wodurch auch das Prinzip des Entlehnens und Reorganisierens verletzt wird. Das nicht verfügbare Wissen muss durch das Prinzip der zufälligen Generierung erworben werden, welches besonders langsam und ineffektiv ist. Denn bei der Entdeckung neuen Wissens werden viele interagierende Elemente betrachtet, die keine Relation zum Lernziel haben. Aufgrund dessen ist der Aufbau von Schemata wenig effektiv, obwohl dies das Hauptziel vom Unterricht sein sollte. Es gibt wenig Evidenz für Lehrentwürfe, die auf Entdeckung beruhen. Hingegen gibt es aber Evidenz für die Vermittlung von Informationen. Als Beispiel können ausgearbeitete Lösungsbeispiele genannt werden [4]. Bei einer Ignorierung all dieser Prinzipien und Eigenschaften entstehen unnötig interagierende Elemente, die nicht vorkommen sollten, wodurch die Lehre ineffektiv wird. [68, S. 42f.]

2.2.3 Die lernförderlichen Ressourcen

1998 wurde die CLT von Sweller durch die lernförderlichen Ressourcen erweitert. Diese werden auch durch die Aufgabenbeschreibung beeinflusst. Im Gegensatz zur extrinsischen Belastung ist eine Erhöhung lernförderlicher Ressourcen effektiv für das Lernen [3, S. 116]. Bei einem Lehrentwurf mit einer reduzierten extrinsischen und einer zu geringen intrinsischen Belastung wird ein Teil der Kapazitäten des Arbeitsgedächtnisses nicht genutzt, welcher für die lernförderlichen Ressourcen genutzt werden sollte [19, S. 3]. Denn eine Reduzierung der extrinsischen Belastung wäre wenig effektiv, wenn die freien Ressourcen des Arbeitsgedächtnisses nicht für das Lernen genutzt werden würden [68, S. 43]. Durch eine bewusste Verarbeitung des Gelernten können Schemata entwickelt werden [19, S. 4]. Bspw. sei hier die Bearbeitung eines Beispiels zu nennen, welches vervollständigt werden muss, wobei die Studierenden die Schemata aus dem Beispiel abstrahieren müssen, um es verstehen zu können [19, S. 4]. Unterricht sollte so gestaltet sein, dass ein Großteil der Ressourcen für die Schemakonstruktion und -automatisierung genutzt wird. Damit sollte das Material eine hohe intrinsische Belastung haben, die der Kapazität des Arbeitsgedächtnisses entspricht sowie eine möglichst geringe extrinsische Belastung [68, S. 43]. Generell wird in der Forschung davon ausgegangen, dass eine Senkung der extrinsischen Belastung zu einer automatischen Steigerung der lernförderlichen Ressourcen führt [68, S. 44]. Bei einer intrinsischen Belastung, welche die Kapazitäten des Arbeitsgedächtnisses übertrifft, würde ein Anstieg lernförderlicher Ressourcen diese in

eine extrinsische Belastung umwandeln [68, S. 44].

Nach der Erläuterung der kognitiven Architektur des Menschen und der kognitiven Belastung, folgt nun eine Definition des Begriffs Programmieranfänger, welcher in der vorliegenden Arbeit genutzt wird.

2.3 Programmieranfänger

In der vorliegenden Arbeit werden Teilnehmende des Kurses Datenstrukturen als PA definiert. Der Kurs findet in der Regel als zweiter Programmierkurs an der TUC statt. In den folgenden Unterabschnitten soll erläutert werden, welche Eigenschaften PA besitzen und auf welche Herausforderungen sie während ihrer universitären Ausbildung treffen.

In der Abbildung 2.1 werden vier Herausforderungen von PA dargestellt: die Schwierigkeit des Programmierens, die Lehre, die Programmierumgebung und individuelle Schwächen. Für das Programmierenlernen werden eine Vielzahl verschiedener Kenntnisse und Strategien benötigt, die PA in der Regel fehlen oder welche diese nur rudimentär beherrschen. In Programmiereinführungskursen werden PA Problemlösungsfähigkeiten, grundlegende Programmierkonzepte, die Syntax und Semantik einer Programmiersprache sowie die Nutzung dieser Programmiersprache zum Schreiben von Lösungen vermittelt [42, S. 1]. Anhand des Beispiels einer einfachen Programmieraufgabe, welche die Einbindung einer Bedingung erfordert, soll aufgezeigt werden, welche Fähigkeiten PA beherrschen müssen [16, S. 18]: (1) das Verstehen und Zerlegen einer Aufgabenstellung, (2) das Programmierkonzept der Bedingung sowie dessen abstrakte Übertragung auf eine neue Aufgabe, (3) die Syntax des Programmierkonstruktes und (4) das Nutzen einer Entwicklungsumgebung zur Ausführung des Programms.

Programmierexperten besitzen gut organisierte und komplexe Schemata, die sie im Laufe ihrer Programmierausbildung erlernt und modifiziert haben. Diese Schemata sortieren sie anhand ihrer Funktionen, zu welchen Algorithmen zählen. Außerdem wenden sie für das Programmverständnis top-down Modelle an. Für die Problemlösung wenden sie sowohl generelle Strategien wie etwa divide-and-conquer als auch spezialisierte Strategien an. Außerdem verfügen sie über ein Repertoire an Test- und Debuggingstrategien [53, S. 139f.]. PA fehlt Kompetenz in diesen Themengebieten, da ihnen Schemata fehlen. Im Laufe dieses Abschnitts sollen verschiedene Eigenschaften dargestellt werden, die PA beherrschen müssen und wie diese ihnen in einem Programmierkurs beigebracht werden. Das primäre Ziel von Einführungsveranstaltungen besteht darin, dass PA nach einem erfolgreichen Abschluss Programme schreiben können. Im Unterabschnitt 2.3.1 wird ein allgemeiner Überblick über die Programmierlehre gegeben. Aus welchen verschiedenen Aspekten und Fähigkeiten das Programmieren besteht wird im Unterabschnitt 2.3.2 dargestellt. In der CEdR wurde die Syntax von Programmiersprachen als Problem bei PA identifiziert. Die Definition der Syntax einer Programmiersprache sowie die Darstellung syntaktischer Merkmale der Programmiersprache Java erfolgt im Unterabschnitt 2.3.3. Im

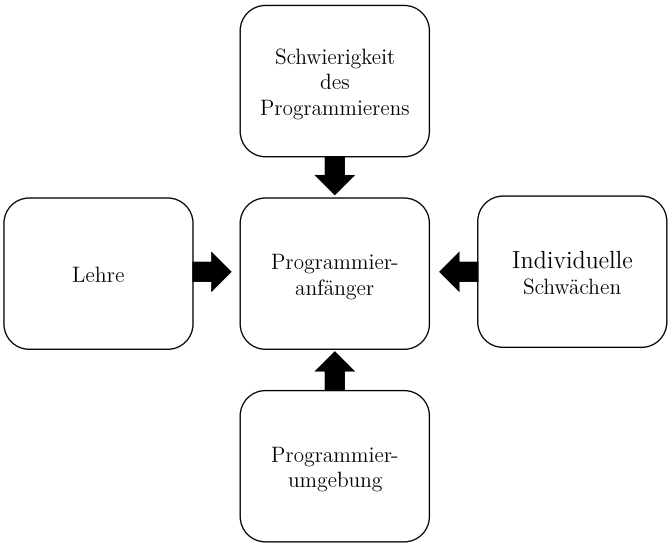


Abbildung 2.1: Herausforderungen von Programmieranfängern angelehnt an [56]

traditionellen Lehransatz werden den Studierenden zur Einführung von Programmierkonzepten Beispiele gezeigt. Das Lesen dieser Beispiele soll zum Verständnis der Konzepte führen. Jedoch wird das Lesen von Programmen nicht explizit in einem Programmierkurs gelehrt. Welche Rolle das Lesen von Programmen für das Programmverständnis hat wird im Unterabschnitt 2.3.4 präsentiert. Der Zusammenhang von Programmieren und Programmverständnis wird im Unterabschnitt 2.3.5 erläutert. Als Nächstes folgt im Unterabschnitt 2.3.6 eine Beschreibung der CLT in der Programmierung und im Unterabschnitt 2.3.7 werden mögliche Probleme durch eine zu hohe kognitive Belastung präsentiert.

2.3.1 Lehre

Ein traditioneller Programmierkurs besteht aus Vorlesungen und Übungen und baut auf einem Curriculum, das auf Wissen basiert, welches in einer Programmiersprache genutzt wird [53, S. 157]. PA erhalten eine Einführung in die Syntax und Semantik eines Programmierkonstruktes, infolgedessen sie mit diesem Programmierkonstrukt einige Programmieraufgaben lösen sollen. Danach wird ihnen das nächste Konstrukt beigebracht [46, S. 42]. Dieser traditionelle Ansatz entspricht etwa einem *spiral syntax approach* [60]. Bei dieser Art der Wissensvermittlung haben einige PA Probleme, denn die Erwartungen der Lehrkräfte entsprechen nicht dem, wozu Studierende in der Lage sind. Im Abschnitt 1.1 wurde die Vielfalt an Fähigkeiten aufgezeigt, die PA beherrschen müssen. Außerdem besitzen Programme eine hohe Elementinteraktivität: Wenn PA zu viele Fähigkeiten gleichzeitig lernen müssen, können diese überfordert sein, wodurch es zu einer Überlastung der kognitiven Kapazitäten kommt [10, S. 75]. Nach Winslow besteht eine gute Pädagogik darin, dass der Lehrende das zu Lernende simpel genug hält und die Komplexität erst steigert,

wenn PA Erfahrung sammeln und somit Fortschritte machen [75, S. 21].

Somit ist für die Lehre des Programmierens der Lehrplan, die Pädagogik, die Programmiersprache und auch der Einsatz bestimmter Tools von besonderer Bedeutung. Der Lehrplan gibt vor, was unterrichtet werden soll und welche Qualifikationsziele PA nach einem Programmierkurs erreicht haben sollten. Nach dem erfolgreichen Abschluss von Datenstrukturen an der TUC sollten PA bspw. in der Lage sein, Listen, Bäume sowie Sortier- und Suchverfahren zu nutzen [23, S. 848]. Die Pädagogik gibt vor, wie die Ziele des Lehrplans umgesetzt werden. Dabei stellt es sich als schwierig heraus, Lehren und Lernen des Programmierens zu definieren. Einige Forscher behaupten, dass die mathematische Basis des Programmierens am wichtigsten ist. Im Gegensatz dazu wird in Lehrbüchern meistens die Syntax und Sprachmerkmale einer spezifischen Sprache dargestellt. Ein anderer Ansatz greift als grundlegende Fähigkeit die Problemlösung auf, welche aus Problemformulierung, Anforderungsanalyse und Problemlösung besteht. [49, S. 206]

In den Curriculum Guidelines des ACM aus dem Jahre 2001 wurden, im Gegensatz zu der neuesten Fassung aus dem Jahre 2013, verschiedene Ansätze zur Gestaltung von Programmiereinführungskursen dargestellt. Zu diesen zählen etwa *imperative first* oder *objective first* [12, S. 18]. Bisher wurde in der Forschung keine Blaupause für die perfekte pädagogische Gestaltung eines Programmierkurses entwickelt, obwohl das Problem des Programmierens seit mehr als 40 Jahren untersucht wird.

Neben den verschiedenen pädagogischen Ansätzen spielt auch die Programmiersprache eine wichtige Rolle. Häufig werden industrierelevante Programmiersprachen wie Java, C oder C++ eingesetzt. Die syntaktische Komplexität dieser Programmiersprachen wird von Forschern kritisiert, weswegen im Unterabschnitt 2.3.3 anhand der Syntax von Java genauer auf diese Problematik eingegangen wird. Aufgrund dessen sind einfache oder visuelle Programmiersprachen, wie etwa Python, Alice oder Scratch, entwickelt worden. Der Einsatz dieser Programmiersprachen sowie auch der Einsatz eines neuen Curriculums, welches auf vier grundlegenden Fähigkeiten der Programmierung basiert, wird im Abschnitt 2.4 erläutert.

2.3.2 Programmierfähigkeit

Im vorherigen Unterabschnitt wurden verschiedene Fähigkeiten und Wissensgebiete präsentiert, die PA nach einem Programmiereinführungskurs beherrschen sollten. Aus dieser Vielzahl an Fähigkeiten wird nun zuerst die Programmierfähigkeit beschrieben. Als zentraler Prozess des Programmierenlernens nennen Robins et al. das Schreiben von Programmen [53, S. 163]. Dies stellt auch das Hauptziel der meisten Programmiereinführungskurse dar [40, S. 125]. Doch aus welchen Bestandteilen besteht die Programmierfähigkeit?

In der CEdR-Literatur werden häufig du Boulays fünf Bereiche genannt [15, S. 57f.]: (1) eine allgemeine Orientierung, wofür Programme benötigt werden und wofür diese eingesetzt werden können; (2) *the notional machine* – die Darstellung eines mentalen Modells, welches den physischen Computer abbildet; (3) Kenntnisse über die Notation einer Programmiersprache, welche aus Syntax und Semantik besteht;

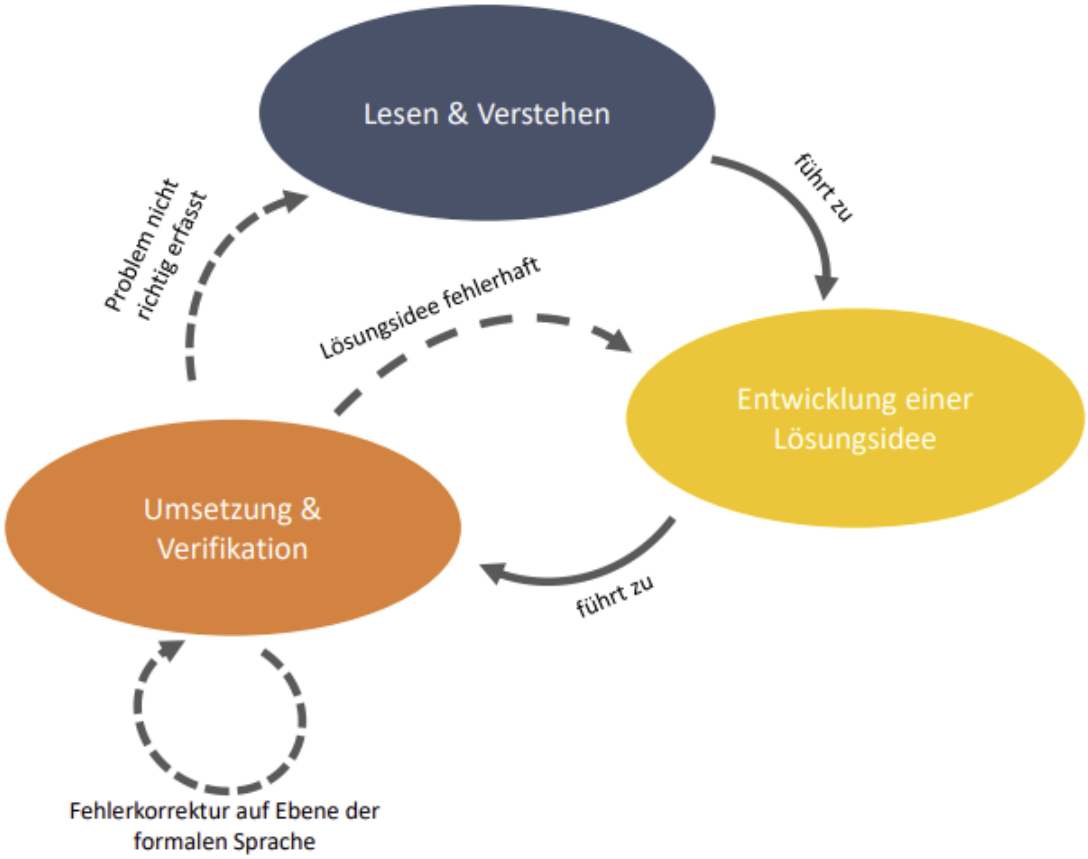


Abbildung 2.2: Phasen des Programmiervorgangs [16, S. 33]

(4) Strukturen bzw. Schemata, die zur Erreichung der Ziele bestimmter Pläne benötigt werden und wiederkehrend sind; (5) Pragmatiken, die Möglichkeit alle Stufen eines Programms zu entwerfen - planen, entwickeln, testen und debuggen - und dabei alle zur Verfügung stehenden Werkzeuge nutzen zu können. Im Gegensatz dazu gliedern Robins et al. Programmieren in die Fähigkeiten Entwurf, Erstellung und Evaluierung sowie in die Attribute Wissen, Strategien und Modelle [54, S. 335]. Ebert hat in seiner Arbeit verschiedene Definitionen des Programmiervorgangs verglichen und eine eigene Definition entwickelt. Diese fokussiert sich auf verschiedene Teilprozesse, die sowohl beim Programmieren als auch beim Programmierenlernen von Bedeutung sind [16, S. 29]. Er unterteilt den Programmiervorgang mit einem Fokus auf die algorithmische Umsetzung in drei Phasen [16, S. 31]. Aufgrund der Entstehung dieser Definition im Rahmen des Forschungsprojekts "EVELIN (Experimentelle Verbesserung der Lehre des Software Engineerings)", wird seine Definition als Grundlage für die Fähigkeit Programmieren genommen. Anhand der Abbildung 2.2 soll herausgearbeitet werden, welche Fähigkeiten PA beherrschen müssen, damit sie ein Problem in einer Programmiersprache lösen können. Der Programmiervorgang beginnt mit dem Lesen und Verstehen einer Aufgabe [16, S. 31]. Wenn dieser

Schritt erfolgt ist, müssen PA eine Lösungsidee entwickeln, welche nur abstrakt und unabhängig von einer bestimmten Programmiersprache ist [16, S. 31]. Dafür wird das Problem in Teilprobleme zerlegt. Für das Lösen dieser Teilprobleme werden Teile von Algorithmen oder ganze Algorithmen benötigt. Als letzter Schritt erfolgt die Umsetzung und Verifikation des Algorithmus. Dafür wird der Algorithmus in eine Programmiersprache übersetzt, damit ein Computer diese ausführen kann. Bei der Verifikation können unterschiedliche Fehler auftreten, deren Behebung auf verschiedenen Ebenen stattfinden muss. Dazu zählen Syntaxfehler, Laufzeitfehler und semantische Fehler. Syntaktische und Laufzeitfehler können in der Phase der Umsetzung und Verifikation behoben werden. Hingegen muss bei semantischen Fehlern der Algorithmus korrigiert werden. Dafür müssen PA in der Phase der Entwicklung einer Lösungsidee, ihren Algorithmus verbessern. [16, S. 31f.]

Anhand der Phasen des Programmiervorgangs, die in Abbildung 2.2 dargestellt sind, werden neben dem eigentlichen Programmieren zwei Bestandteile, die eine besondere Relevanz für die vorliegende Arbeit besitzen, herausgearbeitet: Die Syntax einer Programmiersprache und Programmverständnisfähigkeiten. Anhand von Medeiros Herausforderungen in der Programmierung entspricht die Syntax einer Programmiersprache der Syntax des Lösungsausdruckes [42, S. 6]. Für das Lesen fremder Programme oder zur Ausführung eines eigenen Programms werden unter anderem Tracingfähigkeiten benötigt, die einem PA ermöglichen Programme zu verstehen [42, S. 6]. Damit gehören beide Fähigkeiten zur Phase der Umsetzung und Verifikation.

2.3.3 Syntax einer Programmiersprache

Eine Programmiersprache definiert syntaktische und semantische Anforderungen an ein Programm [5, S. 183]. Die Syntax besteht aus Symbolen und Wörtern sowie der grammatischen Struktur eines Programms [33, S. 50]. Darüber hinaus besitzt jede Programmiersprache eine einzigartige Syntax, wodurch die Syntax als programmiersprachenabhängig zu bezeichnen gilt [33, S. 50]. Die Syntax beschreibt Merkmale, Fakten und Regeln einer Programmiersprache. Ein Programmiereinführungskurs beginnt mit der Lehre der Syntax, wodurch das Schreiben syntaktisch korrekter Programme häufig die erste Fähigkeit ist, die PA Schwierigkeiten bereitet [20, S. 105]. Denn Anweisungen in einer Programmiersprache müssen vollkommen fehlerfrei und korrekt sein, damit der Computer sie ausführen kann. Dies steht im Gegensatz zu einer natürlichen Sprache, bei welcher der Gesprächspartner in der Regel auch die Aussage versteht, wenn diese Fehler enthält. Daneben besitzt eine Programmiersprache viele verschiedene Konstrukte, wodurch die Syntax immer wieder neu vermittelt werden muss, wenn PA neue Konstrukte gezeigt werden. Außerdem hängen die Konstrukte zusammen und in vielen Klausuren werden mehr als zehn Konstrukte verlangt, damit diese überhaupt bestanden werden können [33, S. 51].

Zur Darstellung eines Überblicks der Komplexität der Syntax einer Programmiersprache werden anhand eines Beispiels verschiedene Konstrukte der Programmiersprache Java dargestellt. Im Quellcode 1 wird ein simples Java-Programm

```java
public class Example {
    public static void main(String[] args) {
        int[] array = {1, 5, 3, 2, 4};

        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }
    }
}
```

Quellcode 1: Array in Java

dargestellt, welches bereits nach wenigen Stunden Programmiererfahrung den Studierenden gezeigt werden könnte. Die Studierenden müssen in der Klassen- und Methodendefinition in den Zeilen eins und zwei auf die Groß-Kleinschreibung der jeweiligen Schlüsselwörter (public, class, static, void und main) achten. Außerdem startet der Klassenblock mit einer geöffneten schweifenden Klammer. Der Parameter, welcher der main-Methode übergeben wird, besteht aus einem String-Array. In Zeile zwei müssen PA auch noch die Groß-Kleinschreibung des Schlüsselwortes Strings sowie die Darstellung eines Arrays als Parameter einer Methode betrachten. Die zweite Zeile endet wieder mit einer öffnenden Klammer, die den Beginn des Methodenblocks darstellt. Bei der Initialisierung eines Integer-Arrays in Zeile drei müssen die Studierenden beachten, dass das Schlüsselwort int, im Gegensatz zum Schlüsselwort String kleingeschrieben wird. Zudem müssen zwei eckige Klammern gesetzt werden, um ein Array zu initialisieren. Für das Zuweisen der Elemente muss ein Gleichheitszeichen genutzt werden und die Initialisierung beginnt mit einer geöffneten geschweiften Klammer. Darauf folgen die einzelnen Elemente, welche Ganzzahlen sein müssen. Diese Elemente werden mit Kommata getrennt und der Abschluss des Arrays wird durch eine geschlossene geschweifte Klammer signalisiert. Die Anweisung wird mit einem Semikolon abgeschlossen. In der fünften Zeile folgt der Schleifenkopf einer for-Schleife. Diese startet mit dem Schlüsselwort for, bei dem auf die Groß-Kleinschreibung geachtet werden muss. Die Schritte des Schleifenkopfes starten mit einer öffnenden runden Klammer. Der erste Schritt beginnt mit der Definition der Zählvariable, die ein Integer sein muss. Dieser Variable muss ein Wert zugewiesen werden. Danach folgt ein Semikolon als Trennzeichen zwischen der Start- und End-Bedingung. Die Endbedingung prüft, ob die Zählvariable einen Wert, der das Ende darstellt, überschritten hat. Im Quellcode 1 läuft die Schleife, bis der Wert der Länge des Arrays minus eins entspricht. Zur Bestimmung der Länge des Arrays wird die Variable length genutzt. Jedoch ist diese nur für Arrays abrufbar. Bei der Bestimmung der Länge eines Strings, wird die String-Methode length() aufgerufen. Nach der Endbedingung folgt als Trennzeichen wieder ein Semikolon. Danach wird die Anweisung für das Hochzählen der Zählvariable definiert, der Schritt beträgt in

diesem Fall eins. Die Schleifenbedingung endet mit einer schließenden runden Klammer. Die öffnende geschweifte Klammer ist optional. Zur Ausgabe der Elemente des Arrays wird in Zeile sechs die Methode println() der System-Klasse aufgerufen. In dieser Zeile muss wieder auf die Groß-Kleinschreibung der Klasse, der Instanz des PrintStream-Typen *out* sowie des Methodenaufrufs geachtet werden. Der Methodenaufruf startet mit einer runden Klammer und erhält als Parameter das Element, welches im Array an Stelle i steht. Der Zugriff auf die Stelle i erfolgt durch eckige Klammern. Der Methodenaufruf endet mit einer schließenden runden Klammer und die gesamte Anweisung endet mit einem Semikolon. Danach muss von PA noch darauf geachtet werden, dass jede sich öffnende geschweifte Klammer eine geschlossene benötigt.

An diesem kurzen Beispiel wird die Komplexität der Java-Syntax ersichtlich. Diese Komplexität kann bereits in der ersten Stunde einer Einführungsveranstaltung zu Problemen führen. Denn in der Regel wird PA ein Programm gezeigt, welches "Hello World!" ausgibt. Für das Lesen und Verstehen dieses Programms benötigen sie bereits Informationen über die Syntax einer Klasse, einer Methode sowie eines Methodenaufrufs. Wenn Studierende danach ein eigenes Programm schreiben sollen, welches nur eine Ausgabe tätigt, müssen sie auf alle syntaktischen Merkmale achten. Denn ein einziger syntaktischer Fehler, wie ein Fehler in der Groß-Kleinschreibung, ein falscher Klammertyp, eine falsche Anzahl an öffnenden und schließenden Klammern oder das Vergessen eines Semikolons führt zu einem nicht kompilierbaren Programm. Im folgenden Unterabschnitt wird das Programmverständnis dargestellt.

2.3.4 Programmverständnis

Im Unterabschnitt 2.3.2 wurde dargestellt, aus welchen Bestandteilen Programmieren besteht. Das Hauptziel von Programmierkursen ist es, dass PA Programmieren und somit Programme schreiben können. Die Fähigkeit Programme lesen zu können, wird häufig nicht explizit gelehrt, obwohl in der Wissenschaft herausgearbeitet wurde, dass es einen Unterschied zwischen Quellcode lesen und schreiben gibt [58, 70]. Folglich wird in diesem Unterabschnitt die Relevanz der Lesefähigkeit von Programmen für das Programmverständnis und damit auch für die Programmierung dargestellt.

Das Lesen von Programmen besteht aus zwei Schritten: Tracing und Programmverständnis. Der erste Schritt ist das Tracing. Wenn ein PA in der Lage ist ein Programm zu tracen, dann weiß er wie dieses ausgeführt wird und kann ein Programm, dem eine Eingabe übergeben wird, mental ausführen und die Ausgabe des Programms erläutern. Dabei entwickelt ein PA eine mentale Repräsentation des Programms, aus dem sich ein mentales Modell entwickelt: die notional machine [45, S. 105]. Damit stellen mentale Modelle die Fähigkeit dar, sich die Problemdomäne oder den Computer als mentale Repräsentation vorstellen zu können [53, S. 149]. Mentale Modelle entwickeln sich im Langzeitgedächtnis, wodurch bei Bedarf auch neue entwickelt werden [65, S. 54]. Tracing stellt damit eine Strategie für das Programmverständnis dar und führt dazu, dass ein PA das Verhalten des Programms

versteht [45, S. 12]. Programmverständnis ist ein interner, kognitiver und Hypothesen getriebener Problemlösungsprozess, der folgend definiert werden kann:

> "Comprehension is usually conceptualized as a process in which an individual constructs his or her own mental representation of the program". [58, S. 66]

Programmverständnis kann in drei verschiedene Lesearten unterschieden werden, die je nach Wissensstand vom Nutzer angewendet werden – Top-down-Modelle, Bottom-up-Modelle, als auch Integrated-Modelle [57, S. 151] [62, S. 14].

2.3.4.1 Top-down-Modelle

Top-down Modelle werden von Programmierern mit Domänenwissen genutzt [38, S. 80]. Das Lesen und Verstehen werden durch eine initiale Hypothese über das Ziel und die Funktionalität des Programms geleitet. Um diese Hypothese aufstellen zu können und diese mit anderen Programmen zu vergleichen, wird ihr vorheriges Wissen benötigt [62, S. 14]. Durch das Fokussieren auf *beacons* - „sets of features that typically indicate the occurence of certain structures or operations in the code" [8, S. 548] - wie bspw. eine Schleife, werden Details ausgeblendet.

2.3.4.2 Bottom-Up-Modelle

Bottom-up Modelle werden von PA genutzt, da ihnen Schemata und Kenntnisse von beacons fehlen. PA lesen Programme Zeile für Zeile, wodurch sie die Bedeutung jedes einzelnen Programmierkonstrukts versuchen zu verstehen. Durch chunking, einem rekursiven Prozess, wird aus diesen einzelnen Informationen die Bedeutung des gesamten Programms zusammengestellt. Bei diesem Modell werden für das Programmverständnis viele kognitive Ressourcen benötigt, weswegen Programmierexperten ein Programm top-down lesen. [62, S. 14]

2.3.4.3 Integrated Modelle

In größeren Programmen werden sowohl das Top-down- als auch das Bottom-Up-Modell genutzt. Selbst erfahrene Programmierer verfügen nicht über Wissen in allen Problemdomänen und allen Bereichen eines Programms. Teile des Quellcodes, von denen sie Domänenwissen besitzen, lesen sie top-down und stellen direkt eine Hypothese des Zwecks auf. Bei den anderen, unbekannten, Teilen des Programms wenden sie Bottom-up-Modelle an. Aufgrund der Effizienz des Top-down-Modells versuchen erfahrene Programmierer diese Variante bevorzugt zu nutzen. [62, S. 15]

2.3.5 Programmieren und Programmverständnis

Programmieren und Programmverständnis sind unterschiedliche Fähigkeiten, die laut Winslow nicht in direktem Zusammenhang stehen:

> "Studies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught along with some basic test and debugging strategies" [75, S. 21].

Jedoch stellten Lister und Teague beim Vergleich zwischen Tracing- und Programmierfähigkeiten fest, dass PA Programme zu 50 % korrekt tracen mussten, bevor sie sicher eigene Programme schreiben konnten [31, S. 165]. Deswegen sollten PA in der Lage sein, Programme zu tracen und zu erklären, was bei deren Ausführung geschieht, bevor sie Quellcode schreiben [31, S. 165]. Aber es wurde festgestellt, dass PA nach dem Abschluss eines Programmiereinführungskurses grundlegende Tracingfähigkeiten nicht beherrschten [30]. Diese Schwierigkeiten bestehen selbst bei Studierenden nach dem dritten und vierten Studienjahr und hängen somit auch mit den schlechten Ergebnissen während eines Informatikstudiums zusammen [45, S. 1]. Für Programmieren und Programmverständnis benötigen PA unter anderem eine umfassende Sammlung an Schemata, weil Programmierenlernen sequentiell und aufeinander aufbauend ist. Deswegen sollten PA zur Entwicklung von Schemata kurze Aufgaben mit wenigen Elementen bearbeiten [70, S. 290]. Mögliche Strategien, die dies in der Lehre ermöglichen, werden im Abschnitt 2.4 beschrieben. Nach der Erläuterung des Programmiervorgangs, der Syntax und dem Programmverständnis folgt eine Beschreibung der CLT beim Bearbeiten einer Programmieraufgabe.

2.3.6 CLT beim Programmieren

In diesem Unterabschnitt erfolgt die beispielhafte Darstellung der CLT bei einer Programmieraufgabe anhand der Abbildungen 2.2 und 2.3 sowie des Quellcodes 2. Bei der Auseinandersetzung mit einer Programmieraufgabe werden im Arbeitsgedächtnis die notwendigen Informationen verarbeitet. Die intrinsische kognitive Belastung einer Programmieraufgabe ist in der Regel hoch, weil sie aus vielen interagierenden Elementen besteht. Aufgrund dessen soll die extrinsische Belastung geringgehalten werden. Eine mögliche Aufgabenstellung für den Quellcode 2 könnte lauten: Schreiben Sie eine Methode, welche die Summe der Elemente eines Arrays berechnet. Das Array ist bereits in der main-Methode vorgegeben. Die Methode soll die Summe als Integer zurückgeben. In der main-Methode soll dieser Rückgabewert ausgegeben werden.

Ein PA muss die Aufgabenbeschreibung lesen und verstehen. Anhand des Beispiels 2 muss ein PA in der Lage sein zu verstehen, welche Aspekte der Aufgabenstellung für die Lösungsentwicklung relevant sind. In dieser Aufgabenstellung zählen dazu das Array, die Elemente des Arrays sowie die Summe dieser Elemente. Die notwendigen Informationen hierzu werden im Arbeitsgedächtnis verarbeitet. Wegen dessen geringer Kapazität müssen Schemata aus dem Langzeitgedächtnis geladen werden. Bei Programmieraufgaben ist die hohe intrinsische Belastung durch die Anzahl interagierender Elemente zu beachten. Bereits bei einer simplen Aufgabe, wie im Beispiel, interagieren die genannten Elemente untereinander. Dadurch ist die intrinsische Belastung hoch. Die extrinsische Belastung sollte durch eine gute

Aufgabenbeschreibung und das notwendige Vorwissen geringgehalten werden, da es ansonsten zu einer Überlastung des Arbeitsgedächtnisses kommen kann. Eine Überlastung reduziert die lernförderlichen Ressourcen und verhindert eine effektive Bildung oder Modifizierung von Schemata.

Aus den relevanten Elementen der Aufgabenstellung kann ein PA eine Lösung entwickeln, wobei er dafür Wissen über ein Array und eine Schleife benötigt. Außerdem muss er erkennen, dass die Summe gespeichert werden muss. Wenn ein PA die notwendigen Informationen gelesen und verstanden hat, muss er für die Lösungsentwicklung aus dem Langzeitgedächtnis verschiedene Schemata in das Arbeitsgedächtnis laden. In der Phase der Entwicklung einer Lösungsidee benötigt er dafür Problemlösungskonzepte. Im letzten Schritt des Programmiervorgangs muss er seine Lösung in eine Programmiersprache umwandeln. Dafür werden sowohl syntaktische als auch semantische Merkmale einer Programmiersprache und deren Konzepte benötigt. Ein PA lädt für den letzten Schritt des Programmiervorgangs syntaktische Schemata aus dem Langzeitgedächtnis in das Arbeitsgedächtnis. Anhand der Lösung der Beispielaufgabe im Quellcode 2 wird die Vielzahl syntaktischer Konstrukte ersichtlich, welche bereits im Unterabschnitt 2.3.3 beschrieben worden sind.

Falls dies nun die erste Programmieraufgabe wäre, die ein PA lösen muss, wäre sein Arbeitsgedächtnis überfordert. Wahrscheinlich hätte er die Aufgabe lesen und verstehen können und auch eine Lösungsidee entwickeln können. Für die Umsetzung würde er aber wenigstens Kenntnisse über eine Variable, der Ausgabe, eine Schleife, ein Array, das Schreiben einer Methode und das Durchlaufen eines Arrays besitzen müssen. In einem guten Lehrdesign werden PA verschiedene Probleme und für die Lösung notwendige Programmierkonstrukte schrittweise gezeigt und erläutert, um das Prinzip der begrenzten Veränderung nicht zu verletzen. Anhand der Beispielaufgabe wäre eine mögliche Reihenfolge: (1) Darstellen der Ausgabe am Beispiel Hello World; (2) Initialisierung einer Variablen; (3) Ausgabe der Variable; (4) Einführung einer Schleife; (4) Einführung Array; (5) Durchlaufen eines Arrays.

Im ersten Schritt entwickelt ein PA ein Schema, welches die Syntax und Semantik einer Ausgabe enthält. Darauf folgt die Initialisierung einer Variablen. Diese muss einem primitiven Datentypen entsprechen. In der Beispielaufgabe ist die Variable ein Integer. Diese wird mit einem Wert initialisiert. Dadurch entwickelt ein PA ein Schema für die Verwendung von Variablen sowie deren Syntax. Darauf folgt die Einführung einer Schleife, welche aus einer Startbedingung, einer Endbedingung und einem Inkrementationsschritt besteht. Danach könnte die Einführung eines Arrays und deren Relevanz bei Programmen beschrieben werden. Schlussendlich könnte noch das Durchlaufen eines Arrays explizit dargestellt werden.

Dadurch müsste ein PA in der Beispielaufgabe nur das Problem der Summierung lösen. Er würde sowohl für Variablen, Schleifen, Arrays und der Ausgabe Schemata besitzen. Die intrinsische Belastung der Programmieraufgabe wäre zwar immer noch hoch, aber die extrinsische Belastung wäre niedrig. Denn ein PA hätte die Syntax und Semantik der verschiedenen Programmierkonstrukte verstanden. Dies würde zu einer Senkung der intrinsischen Belastung führen, denn es kann ein neues

```java
public class Example {
    public static void main(String[] args) {
        int[] array = {1, 5, 3, 2, 4};

        System.out.println(sumArray(array));
    }

    public static int sumArray(int[] array) {
        int sum = 0;

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

        return sum;
    }
}
```

Quellcode 2: Summe eines Arrays

Schema gebildet werden, welches die Summierung der Elemente eines Arrays enthält. Dadurch würde die Expertise des PAs steigen, denn komplexere Schemata und deren Automatisierung sind ein Anzeichen für die Kompetenz auf einem Wissensgebiet.

Anhand der CLT kann aber auch evaluiert werden, wieso einige PA die Ziele von Programmiereinführungskursen nicht erreichen und nicht in der Lage sind, Probleme in einer Programmiersprache zu lösen. Deswegen folgt als Nächstes die Entstehung von Schwierigkeiten durch eine zu hohe kognitive Belastung. Dabei wird besonderer Fokus auf syntaktische Schwierigkeiten gelegt, da in dieser Arbeit versucht wird, syntaktische Schwierigkeiten zu minimieren. Dadurch sollen PA mehr kognitive Ressourcen für die Programmerstellung zur Verfügung stehen [17, S. 217].

2.3.7 Entstehung von Schwierigkeiten durch eine zu hohe kognitive Belastung

In der CEdR ist eine Reihe von Faktoren bekannt, die zu Missverständnissen und Schwierigkeiten bei PA führen. Dazu zählen die Aufgabenkomplexität und die damit einhergehende kognitive Belastung, fehlerhafte mentale Modelle, unangemessene Muster und Strategien, natürliche Sprachen, Umwelteinflüsse und die Unterrichtsgestaltung sowie das Wissen der Lehrenden [51, S. 1:7]. Aufgrund des Umfangs und des Themenschwerpunkts der vorliegenden Arbeit wird die Rolle der extrinsischen kognitiven Belastung durch die Syntax einer Programmiersprache beschrieben.

Die meisten Menschen sind mit der Grammatik ihrer Muttersprache vertraut und

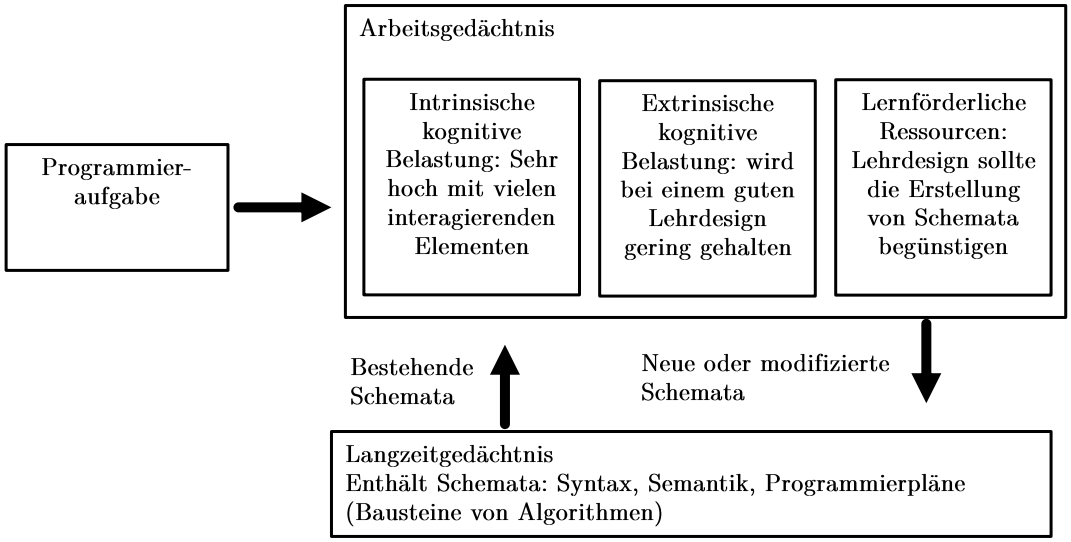


Abbildung 2.3: Beziehungen der kognitiven Belastungen bei der Programmierung, angelehnt an [19]

auch beim Erlernen einer Fremdsprache entdecken sie vergleichbare Muster [20, S. 105]. Aus diesem Grund erweisen sich Programmiersprachen als intuitiver für PA, wenn sie Gemeinsamkeiten mit der englischen Sprache aufweisen [66, S. 19:23]. Im Gegensatz zu natürlichen Sprachen steht die Syntax einer Programmiersprache. Denn in natürlichen Sprachen sind Fehler nicht so relevant wie in Programmiersprachen, da durch den Kontext und einen gesunden Menschenverstand Fehler ignoriert werden können und dennoch die Aussage verstanden werden kann. Hingegen muss bei Programmiersprachen das Produkt vollständig, eindeutig und fehlerfrei sein [20, S. 105]. Ansonsten kann das Programm nicht kompiliert werden [20, S. 105]. Dadurch ist es herausfordernd für PA syntaktisch korrekte Programme zu schreiben [14, S. 208]. Wenn leistungsschwache Studierende immer wieder nicht kompilierbaren Quellcode abgeben, führt dies zu einer immer höheren Frustration [18, S. 1]. Außerdem wenden PA oft viel Zeit bei der Behebung häufig auftretender Fehler auf [29, S. 84f.]. Auch Kummerfeld und Kay bestätigen, dass Studierende dadurch entmutigt werden weiterhin zu programmieren [27, S. 1]. Eine Bekämpfung der Syntaxfehler von PA ist wichtig und wird in der Lehre immer noch vernachlässigt, obwohl eine Reduzierung der kognitiven Belastung den Studierenden ermöglichen würde, schneller und effizienter zu lernen [27, S. 1].

Trotzdem konzentrieren sich Programmiereinführungskurse von Beginn an auf Algorithmen und Problemlösung und erwarten von den Studierenden, dass sie selbstständig die Syntax lernen [14, S. 208]. Meistens wird diese nur kurz anhand eines neuen Programmierkonstruktes erläutert [56, S. 77]. Falls hierbei Probleme auftreten, sind diese auf sich allein gestellt [14, S. 208]. Denn es gibt in klassischen Lehrplänen keine Aufgaben, deren primäres Ziel eine Verbesserung der Beherrschung der Syntax ist. Dies kann bereits zu Beginn Schwierigkeiten verursachen,

denn Syntaxfehler zählen zu den ersten Problemen, mit denen PA konfrontiert werden [20, S. 107]. Durch den fehlenden Syntaxfokus könnte die Lernkurve bereits zu Beginn eines Programmiereinführungskurses zu steil sein [35, S. 9]. Durch die Einführung neuer Programmierkonstrukte, welche für das Beantworten komplexerer Aufgaben unerlässlich sind, besteht die Syntaxbarriere während eines gesamten Programmierkurses, denn neue Programmierkonstrukte führen zu neuen syntaktischen Fehlern [20, S. 107].

Die Syntaxbarriere entsteht durch ein gleichzeitiges Lernen der Syntax, die als Werkzeug dient, und der Problemlösung, welche die Anwendung des Werkzeugs darstellt. Dies führt bei PA dazu, dass diese nicht aus der Problemlösung lernen können, da sie nicht mit der Benutzung des Werkzeugs vertraut sind [17, S. 217]. Demnach benötigen PA mehr kognitive Ressourcen für die Benutzung des Werkzeugs als für den Einsatz des Werkzeugs. Aus diesem Grund kommt es zu einer höheren kognitiven Belastung und die Syntax einer Programmiersprache, welche kognitiv nicht so anstrengend ist, benötigt mehr Kapazitäten, als dies der Fall sein könnte und sollte [17, S. 217]. Dadurch fallen Studierende, die bereits zu Beginn Probleme haben, zurück „they fall behind and stay behind“ [20, S. 107]. Hinzu kommt, dass obwohl die Syntax einer Programmiersprache wenig kognitive Anstrengung benötigt [48, S. 354], stellt sich in der Forschung die Frage, warum diese dennoch so frustrierend ist [17, S. 216]. Sie sollte der leichteste Teil des Programmierens sein und könnte dies auch sein, wenn die Beherrschung der Syntax einer Programmiersprache als Fundament für das Erlernen des Problemlösens genutzt wird [17]. Denn zu jedem Element der Syntax gehört ein zugehöriges Konzept, welches von einem PA verstanden werden muss. Bspw. erfordert bereits eine sehr kurze Zeile Quellcode: *Int x = 3* - Kenntnisse, über eine Variable: x, eine Zuweisung: = und einen literalen Integer Wert [34, S. 27]. Dementsprechend sollte der erste Schwerpunkt eines Programmiereinführungskurses auf einer Beherrschung der Syntax liegen, wodurch PA mehr Kapazitäten für die Problemlösung zur Verfügung stehen [29, S. 85].

Im Gegensatz zu Anfängern haben Experten ihr Wissen in ihrem Langzeitgedächtnis als Schemata gespeichert, wodurch das Arbeitsgedächtnis entlastet und die Problemlösung im Langzeit- und nicht im Arbeitsgedächtnis durchgeführt wird [33, S. 49; 19, S. 2f.]. Sofern es keine abgespeicherten Beispiele im Langzeitgedächtnis gibt, sind Menschen sehr schlecht im komplexen Denken [19, S. 3]. Das Arbeitsgedächtnis ist nicht in der Lage hochkomplexe Interaktionen mit neuartigen Elementen auszuführen [19, S. 3]. Bei PA wird dieses häufiger überfordert, da sie komplexe Argumentationsketten aufrufen müssen, wodurch eine hohe kognitive Belastung auftritt [19, S. 3]. Falls diese größer als die Kapazität des Arbeitsgedächtnisses ist, kann das Lernen behindert werden [33, S. 49]. Deswegen ist eine Reduzierung dieser Last besonders wichtig und kann durch die Bildung von Schemata erreicht werden [19, S. 3]. Experten können die für ihren Bereich relevanten Informationen automatisch verarbeiten, jedoch müssen Anfänger diese bewusst gebrauchen [19, S. 3]. Bspw. muss ein erfahrener Autofahrer beim Bedienen und Fahren des Autos nicht nachdenken, ein Anfänger muss die Kupplung, den Gangwechsel und die Straße beachten sowie zusätzlich noch das Lenkrad bewegen [19, S. 3]. Jeder Übungsen-

twurf sollte nicht nur die Konstruktion von Schemata unterstützen, sondern auch die Verarbeitung dieser [19, S. 3]. Durch die begrenzte Kapazität des Arbeitsgedächtnisses können nur wenige Schemata parallel verarbeitet werden [19, S. 3]. Aufgrund dessen ist der bedachte Einsatz von chunking eine Möglichkeit viele Elemente, als Teile komplexerer Elemente, im Arbeitsgedächtnis, zu speichern. Für Programmierexperten kann ein Element ein ganzer Algorithmus sein, für PA nur ein Teil des Algorithmus oder eine Regel der Syntax [41, S. ii]. Experten können neue Informationen, durch den Einsatz ihrer Schemata, aufgreifen und werden hierbei nicht durch deren kognitive Belastung behindert. Im Gegensatz dazu benutzen PA einen Großteil ihres Arbeitsgedächtnisses, um Informationen zu verarbeiten, denn ihre entwickelten Schemata sind nicht in der Lage, die zur Erledigung der Aufgabe erforderlichen Informationen zu erfassen. Dadurch besitzen sie nur noch wenig Kapazitäten für die kognitive Anstrengung des Lernens im Arbeitsgedächtnis. Dies führt dazu, dass Aufgaben zu schwierig für Anfänger sein können. Zudem können einmal erworbene Schemata neu kombiniert und modifiziert werden, um eine Aufgabe mit hoher intrinsischer kognitiver Belastung zu lösen, ohne den Lernenden übermäßig zu belasten [54, S. 343].

2.4 Interventionsmöglichkeiten in Programmiereinführungskursen

Die Hauptfunktion des Unterrichts besteht darin, die im Langzeitgedächtnis gespeicherten Informationen zu verändern. Dabei müssen die fünf genannten Prinzipien der Grundlage der menschlichen kognitiven Architektur beachtet werden. Für die CLT ist die Ansammlung von Wissen im Langzeitgedächtnis besonders wichtig. Mithilfe des Prinzips des Entlehnens und Reorganisierens wird ein Großteil des menschlichen Wissens von anderen gelernt und nicht durch Problemlösung erlangt. Damit sich die Informationen im Langzeitgedächtnis von Studierenden ändern, ist besonders das Prinzip der begrenzten Veränderung relevant. Wie bereits im Unterabschnitt 2.1.1 erläutert ist die Menge an Informationen, mit denen das Arbeitsgedächtnis umgehen kann, begrenzt. Der Hauptzweck der CLT ist es, die Darstellung neuartiger Informationen zu strukturieren. Dabei wird besonders darauf geachtet, das Arbeitsgedächtnis nicht zu überlasten und die Bildung von Schemata im Langzeit-gedächtnis zu ermöglichen.

In der CEdR-Literatur wurden verschiedene Möglichkeiten untersucht, um Schwierigkeiten von PA zu verringern, wodurch die Noten und Bestehensraten in Programmiereinführungskursen verbessert werden als auch die Abbruchquoten des Informatikstudiums reduziert werden sollten. Vihavainen et al. untersuchten in ihrem Artikel verschiedene Ansätze zur Intervention in Programmiereinführungskursen und ihre Auswirkungen [72]. Die drei meistgenannten Interventionen waren *content change*, *peer support* und *collaboration*. Im Gegensatz dazu wurde bei *media computation*, *group work* und *CS0* der größte Effekt gemessen, anhand der Bestehen-

squote von Programmiereinführungskursen [72, S. 22]. Content change beschreibt eine Änderung oder Aktualisierung des Kursinhalts [72, S.21]. Peer support stellt die Unterstützung von PA durch andere Kursteilnehmer oder Tutoren dar. Collaboration ähnelt peer support und beschreibt die Zusammenarbeit in Klassenzimmern oder Laboren [72, S. 21]. Den stärksten Effekt zeigte media computation. Hierbei verändern PA bspw. Pixel eines Bildes, um dieses zu bearbeiten [72, S. 21]. Group work sind Gruppenarbeiten, wie teambasiertes oder kooperatives Lernen [72, S. 21]. CS0 beschreibt die Einführung eines Vorkurses vor dem ersten Programmiereinführungskurs CS1 [72, S. 21].

Nach der Nennung verschiedener Interventionsmöglichkeiten folgen vier Beispiele aus diesem Bereich, die alle einen besonderen Fokus auf die Syntax sowie die Lese- und Schreibfähigkeiten von Studierenden legen: die Einführung eines neuen Lehrplans, bei dem Fähigkeiten inkrementell gelernt werden, der Einsatz vereinfachter Programmiersprachen, die Nutzung visueller Programmiersprachen sowie der Einsatz von Syntaxaufgaben. Alle Konzepte stellen den Ansatz content change dar. Die Einführung eines grundlegend neuen Lehrplans ist dabei ein aufwendigerer Eingriff als die Einführung von Syntaxaufgaben oder die Nutzung anderer Programmiersprachen.

2.4.1 Fähigkeiten inkrementell lernen

Xie et al. haben vier verschiedene Fähigkeiten herausgearbeitet, die aufeinander aufbauen und von Studierenden beherrscht werden müssen, damit diese einen Programmiereinführungskurs bestehen [76]. Diese Fähigkeiten sind Quellcode zu tracen, syntaktisch korrekte Programme zu schreiben, Vorlagen zu vergleichen, um dadurch wieder benutzbare Abstraktionen von Programmierwissen zu generieren sowie schlussendlich Programme mithilfe der Nutzung dieser Vorlagen zu schreiben. PA sollen zuerst Programme lesen und verstehen, bevor sie diese schreiben und zudem zuerst verstehen wie Quellcode funktioniert, bevor sie lernen Quellcode zu nutzen, um damit Probleme zu lösen. Den ersten Schritt beherrschen Studierende, wenn sie Quellcode verfolgen können und die Auswirkung syntaktischer Konstrukte auf das Programmverhalten bestimmen können [76, S. 214]. Tracing kann als Programmierablaufverfolgung aller Zuordnungen zwischen Syntax, Semantik und Zustand des Programms während der Kompilierung definiert werden [76, S. 214]. Jedoch führt die Beherrschung von tracing nicht unbedingt dazu, dass Studierende in der Lage sind, syntaktisch korrekten Quellcode oder Programme, die ein Problem lösen, zu schreiben [76, S. 214]. Dennoch ist tracing die Grundlage für alle weiteren Fähigkeiten, die ein PA erlernen muss und kann damit als Vorstufe zum Schreiben syntaktisch korrekter Programme und der Nutzung von Vorlagen gesehen werden [76, S. 214]. Jedoch sind PA durch Kontrollieren dieser Fähigkeit noch nicht in der Lage den Zweck eines Programms zu verstehen, denn sie verstehen nur den Zweck des Programmierkonstrukts [76, S. 214]. Die Bedeutung des gesamten Programms wird durch die Erlangung von Vorlagen und damit der dritten von Xie et al. genannten Fähigkeit erlernt [76, S. 214]. Da Studierende verstehen, wie ein

Programm abläuft und welchen Zweck verschiedene Programmierkonstrukte haben, sind sie in der Lage, die zweite Fähigkeit, das Schreiben syntaktisch korrekter Programme, zu erlernen.

Syntaktisch korrekten Quellcode zu schreiben bedeutet, dass natürlichsprachliche Beschreibungen von Konstrukten in einer Programmiersprache geschrieben werden können, sodass jenes Programm wie erwartet kompiliert und ausgeführt werden kann [76, S. 214]. Einige Ansätze in der CEdR ersetzen das Erlernen dieser Fähigkeit durch den Einsatz blockbasierter Sprachen [76, S. 215]. Jedoch ist die Effektivität dieses Einsatzes fraglich, denn es ist nicht möglich, das Wissen auf eine textbasierte Sprache zu transferieren[2]. Um syntaktisch korrekten Quellcode zu schreiben, muss ein PA wissen, wie Programmkonstrukte während der Kompilierung ausgeführt werden und damit die erste Fähigkeit beherrschen [76, S. 215]. Wenn eine einfache Aufgabenbeschreibung in ein lauffähiges Programm übersetzt werden kann, beherrscht ein Studierender diese Fähigkeit [76, S. 215]. Falls ein PA die Fähigkeit syntaktisch korrekten Quellcode beherrscht, aber nicht die erste, schreibt dieser zwar syntaktisch korrekten Quellcode, welcher aber nicht zum erwarteten Ergebnis führt [76, S. 215]. Demgegenüber steht eine Beherrschung der ersten Fähigkeit und eine Nichtbeherrschung der zweiten. Diese führt zu einem semantisch korrekten Programm, welches aber syntaktische Fehler aufweist und deswegen nicht ausgeführt werden kann [76, S. 216]. Nachdem die ersten beiden Fähigkeiten gelehrt und beherrscht werden, folgt das Lesen von Vorlagen.

Ein Studierender ist in der Lage Vorlagen zu lesen, wenn er aus seinem Programmierwissen wiederverwendbare Abstraktionen erstellen kann, die zur Lösung eines Problems benötigt werden [76, S. 216]. PA erwerben Kenntnisse über Vorlagen, indem sie diese in anderen Programmen erkennen, durch Anweisungen lernen oder neu erfinden [76, S. 216]. Beim Lesen von Quellcode können Teile von diesem als Vorlage, die zur Erreichung eines Ziels benötigt wird, erkannt und abstrahiert werden [76, S. 216]. Damit dieses Wissen erworben werden kann, müssen mehrere Zeilen Quellcode miteinander verbunden werden. Damit ein PA dazu in der Lage ist, muss er tracing-Kenntnisse besitzen [76, S. 216]. Falls dieses Wissen fehlt, können Missverständnisse bei der Codeausführung entstehen, wodurch eine Vorlage nicht erkannt werden kann [76, S. 216].

Die letzte Fähigkeit ist auch die schwierigste, da ein PA eine mehrdeutige Problembeschreibung lesen muss und für diese eine Vorlage auswählen muss, die zur Problemlösung genutzt werden kann. Denn letztendlich muss er auch noch diese Lösung in Quellcode übersetzen [76, S. 216]. Der erste Schritt zum Schreiben einer Vorlage besteht aus dem Extrahieren eines Ziels aus einer natürlichsprachlichen Problembeschreibung [76, S. 216]. Ein Studierender besitzt Kenntnisse in der Fähigkeit Vorlagen zu schreiben, wenn er beim Lesen einer Problembeschreibung die Notwendigkeit einer Vorlage erkennt, um mit dieser einen Plan zu erstellen, der zur Lösung eines Problems führt [76, S. 216]. Für das Erkennen der richtigen Programmierkonstrukte wird die zweite Fähigkeit, syntaktisch korrekten Quellcode zu

[2] siehe Unterabschnitt 2.4.3

schreiben, benötigt [76, S. 217]. Falls ein PA schlechte Vorlagen-Lese-Fähigkeiten besitzt, ist er nicht in der Lage Vorlagen abzurufen und mithilfe dieser ein Problem zu lösen [76, S. 217]. Falls ein Studierender in den ersten beiden Fähigkeiten Defizite aufweist, ist er nicht in der Lage, die Vorlage in ein syntaktisch und semantisch korrektes Programm zu übersetzen [76, S. 217].

2.4.2 Vereinfachte Programmiersprachen

Neben der Einführung eines neuen Lehrplans können PA die Syntaxbarriere auch durch vereinfachte Programmiersprachen durchbrechen. Bei der Untersuchung verschiedener syntaktischer Konstrukte stellten Stefik und Siebert fest, dass eine intuitive Syntax PA unterstützen kann [66, S. 19:35]. Durch ihre Forschung entwickelten sie mit Quorum eine neue Programmiersprache [66].

Demgegenüber entwickelten Mannila et al. keine eigene Programmiersprache, sondern untersuchten, ob PA in der Lage sind, Konzepte einer syntaktisch simplen Programmiersprache (Python) auf eine syntaktisch komplexe Sprache wie etwa Java zu übertragen [36]. PA verstanden die grundlegenden Konzepte und konnten diese auch in Java implementieren [36, S. 219]. Des Weiteren fühlten sich die Studierenden durch das Lernen in Python besser vorbereitet für das Programmieren in einer komplexeren Programmiersprache [36, S. 217]. Wenn PA Fehler in Java machten, lag dies an neuen Programmierkonstrukten, die noch nicht in Python unterrichtet worden waren [36, S. 219]. Außerdem erfüllten die eingereichten Programme auch spezifische syntaktische Anforderungen von Java, die nicht in Python benötigt werden, wie etwa Semikolons oder Klammern [36, S. 219].

2.4.3 Visuelle Programmiersprachen

Zur Verringerung der extrinsischen kognitiven Belastung können visuelle Programmiersprachen, wie etwa Alice und Scratch, eingesetzt werden. Visuelle Programmiersprachen bestehen aus Blöcken und PA müssen keine syntaktischen Konstrukte auswendig lernen. Die Blöcke können unterschiedlich angeordnet sowie die Werte von Variablen angepasst werden. Dadurch gibt es zwar zu Beginn eines Programmiereinführungskurses keine steile Lernkurve und PA können schnell Fortschritte machen. Demnach fanden Weintrop et al. beim Vergleich einer block- zu einer textbasierten Sprache an einer Highschool heraus, dass Schüler, die eine blockbasierte Programmiersprache genutzt haben, besser abschnitten, als Schüler, die eine textbasierte Programmiersprache genutzt haben [74, S. 122]. Jedoch tritt beim Übergang zu textbasierten Sprachen, wie Python oder Java, die extrinsische kognitive Belastung durch die Syntax wieder auf [50, S. 214]. Die Teilnehmenden berichteten davon, dass sie zwar theoretisch wüssten, was sie machen müssen, jedoch nicht wissen, wie sie dies syntaktisch umsetzen können [50, S. 214]. Außerdem werden visuelle Programmiersprachen nur in der Lehre und nicht in der Industrie genutzt. Aufgrund der Relevanz textbasierter Programmiersprachen in der Industrie

werden PA während ihres Informatikstudiums häufig mit diesen Sprachen in Kontakt kommen. In der Literatur wird der Nutzen von visuellen Programmiersprachen diskutiert und insbesondere die Tatsache, dass der Übergang zu einer textbasierten Sprache verbessert werden müsste [67, S. 783].

2.4.4 Syntaxaufgaben

Eine weitere Möglichkeit zur Senkung der CLT besteht im Einsatz von Syntaxaufgaben in einem Programmiereinführungskurs. In der Literatur gibt es drei verschiedene Arten von Syntaxaufgaben [67, S. 783]. Der erste Aufgabentyp umfasst Tippaufgaben. Bei diesen werden Studierenden kurze Programme gezeigt, welche sie abtippen müssen. Für das Bearbeiten von Tippaufgaben benötigen Studierende keine Logik, denn sie sollen nicht nur Programme lesen, sondern diese auch tippen. Durch dieses wiederholte Üben soll sich die Syntax im Gedächtnis verfestigen. Der zweite Aufgabentyp beschreibt Syntaxaufgaben, bei denen Programmierkonzepte in kleine Schritte aufgeteilt werden. Gonzales gliedert diese in *Creation Exercises*, *Debugging Exercises* und *Modification Exercises* [21, S. 13]. Bei Modification Exercises muss ein Programmierkonstrukt, wie etwa eine Schleife, minimal angepasst werden, sodass diese bspw. nicht mehr von null bis drei, sondern von eins bis drei zählt. Außerdem können Programme Lücken enthalten, welche Studierende ausfüllen müssen, damit das Programm korrekt kompilierbar ist. Im Gegensatz dazu enthalten bei Debugging Exercises Programme Fehler, die erkannt und behoben werden müssen. Schlussendlich erweitern Creation Exercises Modification Exercises und Studierende werden aufgefordert ein kurzes Programm mit dem, in den Creation und Debugging Exercises, gelernten Konstrukt zu schreiben. Hierbei müssen Studierende etwa eine Schleife schreiben, welche fünfmal "Hello World!" ausgibt. Mithilfe der Syntaxaufgaben erlernen PA Syntax Schritt für Schritt. Der letzte Aufgabentyp befasst sich mit Programmieraufgaben, hierbei benötigen PA für die Beantwortung neben der Syntax einer Programmiersprache auch ihre Problemlösungsfähigkeiten. [67, S. 783]

2.5 Messmöglichkeiten zur Effektivität der Interventionsmöglichkeiten

Um die Annahme zu belegen, dass die Syntax und die damit verbundene kognitive Belastung ein Grund für das schlechte Abschneiden von PA in Programmiereinführungskursen ist, werden Verfahren genannt, mit denen der Einsatz der Syntaxaufgaben gemessen werden kann. Für eine Beschreibung der Programmierfähigkeit und des Programmverständnisses, siehe Unterabschnitte 2.3.2 und 2.3.4

2.5.1 Programmverständnis

Zur Untersuchung der Effektivität einer Interventionsmöglichkeit in einem Programmiereinführungskurs, zur Verbesserung des Programmverständnisses von PA, kann deren Leseverhalten untersucht werden. Das Programmverständnis wird bereits seit mehr als 30 Jahren untersucht [62, S. 13]. In der Vergangenheit wurden verschiedene Methoden, wie etwa Think-Aloud-Protokolle, genutzt [3]. Bei Think-Aloud-Protokollen sprechen die Teilnehmer beim Bearbeiten einer Aufgabe ihre Gedanken aus. Diese werden aufgenommen und transkribiert [62, S. 13]. Bei der Analyse der Gespräche wird untersucht, ob ähnliche Muster auftreten, bei denen Teilnehmende auf Probleme treffen. Der Aufwand für diese Art der Messung des Programmverständnisses ist hoch, weswegen viele Forscher diesen meiden und auf andere Methoden setzen [62, S. 14].

Nachdem 1990 die erste Eyetracking-Studie in der Softwaretechnik durchgeführt worden ist, wuchs die Anzahl der Studien in den vergangenen 15 Jahren [59, S. 5]. Mithilfe eines Eyetrackers kann bspw. das Lesemuster beim Tracen oder Debugging von Programmen untersucht werden [59, S. 2]. Der Blick kann dabei Hinweise auf die Aufmerksamkeit, den Aufwand und die Länge, die für das Verständnis benötigt worden ist, geben [59, S. 7]. In der Softwaretechnik wurden bspw. das Programmverständnis, Diagrammverständnis und Codezusammenfassungen untersucht [59, S. 14-16]. Wegen der Relevanz des Programmverständnisses für die vorliegende Arbeit wird nur diese genauer erläutert. Die Nutzung eines Eyetrackers ermöglicht etwa den Vergleich verschiedener Programmiersprachen, zwischen Rekursion und Iteration oder zwischen verschiedenen Implementierungsstilen. Für die Messung der Unterschiede werden in Experimenten als abhängige Variablen vor allen Dingen die Zeit, Leistung, visuelle Aufmerksamkeit und Sichtmuster gewählt [47, S. 5:31]. Die Zeit stellt dabei entweder die Bearbeitungszeit für eine Aufgabe dar, die Fixierungszeit oder die Antwortzeit [47, S. 5:30]. Die Leistung kann auch als Genauigkeit betitelt werden und beschreibt die Anzahl korrekter Antworten bei einer Aufgabe [47, S. 5:30]. Zu Sichtmustern zählen der Blickpfad oder die Wechsel zwischen verschiedenen Bereichen [47, S. 5:31]. Schlussendlich beschreibt die visuelle Aufmerksamkeit die Zahl der Fixierungen in einer AOI oder den visuellen Aufwand, der betrieben werden musste, um die Aufgabe zu lösen [47, S. 5:32]. Eine Untersuchung dieser Fähigkeiten könnte helfen, wie PA Quellcode verstehen und wieso sie Missverständnisse entwickeln [76, S. 220].

2.5.2 Programmieraufgaben und Klausurergebnisse

In der Beschreibung der Problemstellung im Abschnitt 1.1 und im Unterabschnitt 2.3.2 wurde die Relevanz der Programmierfähigkeit erläutert. Zwar besitzt auch das Programmverständnis eine immer wichtiger werdende Rolle in der Ausbildung von PA, jedoch müssen diese in Klausuren vor allen Dingen Programme schreiben kön-

[3] Aufgrund des Rahmens der vorliegenden Arbeit wird nur diese beschrieben. Für weitere Methoden siehe [62]

nen. Aufgrund dessen kann anhand von Programmieraufgaben und den Klausurergebnissen, die zu einem Großteil aus Programmieraufgaben bestehen, evaluiert werden, ob sich die Programmierfähigkeit von Probanden verbessert hat. Im Unterabschnitt 2.2 wurden Schwierigkeiten durch eine erhöhte extrinsische kognitive Belastung beschrieben, welche durch eine fehlende Beherrschung der Syntax auftreten können. Falls PA von dem Treatment profitiert haben, sollten sie auch schneller und korrekter Programmieraufgaben lösen können sowie bessere Ergebnisse in der Klausur erzielen.

3 Related Work

Hohe Abbruchquoten, Schwierigkeiten und Fehler von PA stellen eine ständige Motivation für die Erforschung weiterer Lernstrategien und Werkzeuge zur Verbesserung von Programmiereinführungskursen dar. Es gibt eine Reihe von Werkzeugen, die zur Verbesserung der Programmierfähigkeiten und Problemlösungsstrategien von Studierenden eingesetzt werden. Zu diesen Werkzeugen gehören vereinfachte und visuelle/ blockbasierte Programmiersprachen sowie spezielle Aufgaben zum Erlernen der Grundlagen der Programmierung, wie etwa Tipp-, Syntax- und Programmieraufgaben[1]. Durch die Bearbeitung dieser Aufgaben kommt es zu einer Automatisierung der Fähigkeit syntaktisch korrekten Quellcode zu schreiben und damit zu einer Reduktion der extrinsischen kognitiven Belastung. Im folgenden Kapitel werden verwandte Arbeiten beschrieben, die den Einsatz der drei Aufgabentypen untersucht haben, welche in Programmiereinführungskursen eingesetzt werden können.

3.1 Syntaxaufgaben

Edwards et al. untersuchten in ihrer Studie die Aufteilung von Syntax und Problemlösung in Programmiereinführungskursen [18]. Dafür setzten sie Syntaxaufgaben ein, die im Laufe eines Kurses immer komplexer wurden, bspw. mussten Studierende zu Beginn eines Semesters nur den Schleifenkopf einer for-Schleife hinzufügen. Die simplen Aufgaben wurden eingesetzt, da die Forscher behaupteten, dass mehr Übung einen PA schneller zu einem Experten machen kann. An jedem Unterrichtstag bekamen die Probanden vor ihren Problemlösungsaufgaben Syntaxaufgaben [18]. Außerdem mussten Studierende Projekte per *pair programming* bearbeiten. Die Forscher untersuchten *competence* und *perceived value*. Competence besteht aus den Klausur- und Projektergebnissen und der Zeit für das Bearbeiten der Aufgaben [18]. Perceived value soll dabei helfen, die empfundene Wahrnehmung und das gesteigerte Interesse an der Materie zu verstehen. Diese besteht aus einer Befragung, bei der die Teilnehmenden am Ende eines Projekts ihre Programmiererfahrung angeben sollen. Außerdem wird sowohl vor als auch nach dem Kurs dieser von den Studierenden evaluiert. Die Evaluation des Kurses ist in diesem Artikel nicht angegeben. Der Einsatz von Syntaxaufgaben führte bei competence und perceived value zu besseren Ergebnissen, außer bei den Projekt- und Prüfungsergebnissen. Studierende, die in der Kontrollgruppe waren, konnten später optional an den Syntaxaufgaben teilnehmen. Als anekdotische Evidenz kann die Mail eines Studierenden gesehen werden, der den Einsatz dieser Aufgabe gerne von Beginn an gehabt hätte [18].

[1] Für eine genauere Beschreibung siehe Unterabschnitt 2.4.4

In einer weiteren Studie von Edwards et al. setzten diese Syntaxaufgaben als erste Vorgehensweise in einem CS1 Kurs ein [17]. Die Studie wurde über zwei Semester durchgeführt und die Probanden wurden in eine Test- und Kontrollgruppe aufgeteilt [17, S. 218]. Der Testgruppe wurden drei Wochen nach Beginn des Kurses Syntaxaufgaben vorgelegt, die in Zusammenhang mit den Themen des Kurses standen. Die Aufgaben sollten sie dreimal pro Woche vor ihrer Vorlesung bearbeiten, da in dieser die Themen vertieft werden [17, S. 218]. Für die Bearbeitung wurden die Aufgaben in Phanon eingebunden und die Probanden mussten 20 bis 60 Syntaxaufgaben durchführen [17, S. 218], die jeweils 20 bis 40 Sekunden dauerten [17, S. 219]. Wie in der vorherigen Studie von Edwards et al. [18] begannen die Studierenden mit simplen Aufgaben, wie dem Ändern der Endbedingung einer for-Schleife von 10 zu 12 [17, S. 219]. Im Laufe des Semesters stieg die Komplexität der Aufgaben an. Für jede Aufgabensammlung sollten die Studierenden maximal zehn Minuten benötigen [17, S. 219]. Edwards et al. behaupten, dass Studierende wahrscheinlich in den Schleifen-Aufgaben mehr Schleifen schreiben als sie sonst in einem ganzen Semester für ihre Programmierprojekte schreiben [17, S. 219]. Sie stellten drei Hypothesen auf: Die Ausfallrate sinkt, die Projekt- und Klausurergebnissen werden durchschnittlich besser sein und Studierende werden ihre Projekte schneller bspw. mit weniger Tastenanschlägen erledigen [17, S. 220]. Die Ausfallrate für das letzte Projekt sank, jedoch war sie bei einfacheren Projektaufgaben nicht so hoch [17, S. 220]. Die Studiengruppe schnitt überraschenderweise schlechter bei den Projektaufgaben ab, jedoch deutlich besser in der Klausur als erwartet [17, S. 220]. Die letzte Hypothese wurde auch widerlegt, da Probanden mehr Tastenanschläge benötigten als die Kontrollgruppe [17, S. 221]. Insgesamt arbeiteten sie eine Verbesserung der Abbruchquote, Verbesserung der Prüfungsergebnisse und eine Verringerung von Plagiaten heraus [17, S. 224]. Außerdem erfordern die Aufgaben sowohl vom Lehrenden, für die Implementierung, als auch von den Studierenden für die Bearbeitung der Aufgaben wenig Aufwand [17, S. 224].

Sullivan et al. erweiterten die Studie von Edwards et al. [17], indem sie die Einstellung von Studierenden gegenüber Syntaxaufgaben in CS1 untersuchten [67]. Dafür verwendeten sie einen qualitativen Ansatz und fanden heraus, dass den Studierenden bewusst war, wieso diese Übungen eingesetzt worden sind und welchen Effekt sie auf ihre Programmierfähigkeiten haben. In ihrer Studie wurden die Syntaxaufgaben nach und nach komplizierter und die Probanden mussten zunächst bswp. nur das Wort for bei einer for-Schleife einfügen und später ganze Schleifen selbst schreiben [67, S. 784]. Trotz Mehrarbeit für die Studierenden war ihre Einstellung positiv, was überraschend ist. 86 % der Studierenden fanden Phanon hilfreich, 84 % mochten die Aufgaben und 8% hatten Vergnügen bei der Ausführung der Aufgaben [67, S. 785]. Demgegenüber fanden nur 5 % der Studierenden die Aufgaben nervig und 5 % gefielen die Aufgaben nicht [67, S. 785]. Daraus lässt sich ein starkes Potenzial für einen praxisorientierten Lehrplan und eine Programmiereinführungskurs-Pädagogik ableiten, der zuerst die Syntax in den Vordergrund stellt. Außerdem konnten sie die Untersuchung von Edwards et al. bestätigen, indem der Einsatz von Syntaxaufgaben eine verstärkte Wirkung für die Kursteilnehmer von Programmiereinführungskursen,

durch eine besser zu bewältigende kognitive Belastung, hat [67, S. 787].

Gonzales setzte in ihrer Studie die Arbeit von Edwards et al. und Sullivan et al. fort [17, 67]. Sie führte eine Think-Aloud-Studie bei PA in einem Programmiereinführungskurs in Python durch [21, S. 10]. Die Studierenden mussten neben Programmieraufgaben auch Syntaxaufgaben in Phanon lösen [21, S. 10]. Gonzales führte ihre Think-Aloud-Studie in drei Phanon-Sitzungen durch, welche die Themen Bedingungen, for-Schleifen und verschachtelte for-Schleifen beinhalteten [21, S. 17f.]. Die Aufgaben waren in drei Kategorien unterteilt: *Modification Exercises*, *Debugging Exercises* und *Creation Exercises*. Bei Modification Exercises mussten die Probanden eine Variable im Programm verändern [21, S. 13]. Hingegen war bei Debugging Exercises eine Variable nicht deklariert, es fehlte ein Semikolon oder der Quellcode wurde falsch eingerückt [21, S. 13]. Im Gegensatz dazu bauen Creation Exercises auf Modification Exercises auf und Studierende mussten am Anfang eine Schleife schreiben, welche viermal die Zahl null ausgibt und danach eine Schleife, welche die Zahlen von null bis acht ausgibt. Bei diesen Aufgaben untersuchte Gonzales die Muster der Interaktionen der Studierenden bei Syntaxaufgaben und die Verhaltensweisen in Bezug auf die Leistung der Studierenden beim Erlernen der Syntax [21, S. 70]. Aus den Mustern wurde ersichtlich, dass die extrinsische kognitive Belastung während der Aufgaben gering war. Dadurch ist Phanon eine ideale Umgebung für das Erlernen der Syntax [21, S. 70]. Durch die Nutzung diese Tools werden die Studierenden schneller und präziser in der Bearbeitung von Programmieraufgaben, erkennen auch schneller Fehler im Code [21, S. 60] und können diesen besser debuggen [21, S. 71]. Außerdem wurde herausgefunden, dass die Probanden oft vorherige Übungen als Ressource für syntaktisch korrekten Quellcode genutzt haben [21, S. 65]. Insgesamt ist laut Gonzales Phanon eine ideale Umgebung für Studierende, sowohl zum Erlernen der Syntaxbeherrschung als auch für das Debuggen, welches modular möglich ist [21, S. 72]. Zudem wird durch die Syntaxaufgaben die extrinsische kognitive Belastung gesenkt, wodurch den PA mehr kognitive Ressourcen zur Verfügung stehen [21, S. 72].

Ly et al. [35] nutzen dieselben Syntaxaufgaben wie Edwards et al. [17], aber an einer anderen Universität und mit anderen Metriken. Sie untersuchten zwar die kurzzeitigen Effekte von Syntaxaufgaben, fragten jedoch auch Studierende nach ihrer Einschätzung zu den Syntaxaufgaben sowohl wöchentlich als auch am Ende des Semesters, wie Sullivan et al. [67]. Die Syntaxaufgaben wurden im ersten Semester eines Python-Kurses genutzt [35, S. 10]. Als Kontrollgruppe dienten Studierende im Wintersemester 2019 [35, S. 10]. Dem Lehrplan wurden nur die Syntaxaufgaben hinzugefügt, ansonsten wurde versucht alles gleich zu halten. Die Syntaxaufgaben beinhalteten Variablen, Bedingungen, Schleifen, Strings und die Nutzung gewöhnlicher String-Funktionen [35, S. 11]. In beiden Semestern wurde die Anzahl der Abgaben und die Zeit gemessen, welche Studierende für die Programmieraufgaben benötigten sowie, ob sie die Syntaxaufgaben bearbeitet haben oder nicht [35, S. 11]. Daten von Studierenden, die besonders oft die Aufgaben abgegeben haben oder längere Zeit nicht abgegeben haben, wurden entfernt [35, S. 11]. Zur Einschätzung der Programmiererfahrung wurde zu Beginn des Semesters ein Vortest durchge-

führt [35, S. 11]. Die Testgruppe konnte mit ähnlichen oder weniger Einreichungen und in ähnlicher oder kürzerer Zeit Programmieraufgaben lösen, die in Zusammenhang mit den Syntaxaufgaben stehen [35, S. 12]. Bei den Aufgaben, die keinen Zusammenhang zu den Syntaxaufgaben haben, benötigen die Studierenden ähnlich oder mehr Zeit und Einreichungen [35, S. 12]. Die Probanden der Kontrollgruppe benötigten sowohl während der Nutzung von Phanon in den ersten Wochen des Semesters mehr Zeit für alle Aufgaben, aber auch in den Wochen, in denen Phanon nicht eingesetzt worden war [35, S. 12]. Mehr als 65 % der Studierenden fanden die Syntaxaufgaben wenigstens etwas nützlich und nur 15 % fanden sie mindestens ziemlich schwierig [35, S. 12]. Am Ende des Semesters wurden die Studierenden befragt, welche Bestandteile und Materialien des Kurses sie am hilfreichsten fanden. Bei dieser Befragung schnitten die Syntaxaufgaben am schlechtesten ab [35, S. 12]. Jedoch waren diese für Studierende mit geringen Vorkenntnissen deutlich wichtiger und standen an zweiter Stelle [35, S. 12]. Als kurzfristiges Ergebnis konnte erkannt werden, dass Probanden in wöchentlichen Assignments besser wurden, wenn sie Probleme behandelten, die in den Syntaxaufgaben enthalten waren [35, S. 12]. Studierende benötigten 2020 grundsätzlich mehr Zeit für das Lösen von Programmen, die nicht auf den Phanon-Aufgaben aufbauten, mit einer Ausnahme [35, S. 13]. Die Autoren vermuten, dass dies an der Corona-Pandemie liegen könnte [35, S. 13]. Erwähnenswert ist, dass Studierende in Wochen, in denen Phanon-Aufgaben genutzt worden sind, durchschnittlich weniger Zeit für das Lösen aller Aufgaben benötigten, als in Wochen, in denen Phanon nicht eingesetzt worden war [35, S. 13]. Die Autoren glauben, dass dies am Zeigen ausgearbeiteter Lösungsbeispiele liegen könnte, die zu einer Reduzierung des Zeitaufwands geführt haben könnten [35, S. 13]. Wie in der Untersuchung von Sullivan et al. [67] fanden 65 % der Probanden den Einsatz der Syntaxaufgaben hilfreich. Besonders unerfahrene Programmierer sahen diese als nützlich an, erfahrene hingegen nicht [35, S. 13]. Jedoch wurden Syntaxaufgaben als am wenigsten nützliches Material bewertet, welches den Studierenden zur Verfügung stand [35, S. 13].

3.2 Tippaufgaben

Gaweda et al. nutzten Tippaufgaben in einem Programmiereinführungskurs [29], hierbei mussten Studierende Quellcode abtippen und durch ein Tool namens TYPOS wurden Zeilen mit einem Fehler markiert [20]. Die Forscher gehen davon aus, dass durch das Bearbeiten von Syntaxaufgaben kognitiv weniger anspruchsvolle Fähigkeiten gelernt werden. Durch das Beherrschen dieser Fähigkeiten haben sie mehr kognitive Ressourcen für das Bearbeiten von kognitiv anspruchsvolleren Fähigkeiten, wie dem Problemlösen [20, S. 111]. Sie setzten TYPOS drei Semester lang ein [20, S. 108]. In ihrer Studie verhinderten sie zudem ein Kopieren des Quellcodes des Programms, da dieses als Bild angezeigt worden ist [20, S. 108]. Während eines Semesters gab es 66 Aufgaben, die durchschnittlich 33 Zeilen Code beinhalteten. Die Aufgaben waren nicht hierarchisch geordnet und konnten be-

liebig oft wiederholt werden [20, S. 109]. Pro Aufgabe benötigten die Probanden im Durchschnitt 400 Sekunden [20, S. 109]. Auffällig bei der Ausführung der Aufgaben war, dass leistungsschwache Studierende den höchsten Lernzuwachs erzielten [20, S. 110]. Bei der Betrachtung von passiver und aktiver Nutzung der Software hat sich ergeben, dass die aktive Nutzung zu deutlich besseren Ergebnissen führte [20, S. 110]. Eventuell könnte der Mehrwert von Tippaufgaben nicht in dem Abtippen der Syntax liegen, sondern darin Tippfehler zu identifizieren und sie zu korrigieren [20, S. 111]. Alle Studierenden profitierten vom Abtippen der Beispiele. Für leistungsschwache Probanden war bereits das Zeigen der Quellcodestücke profitabel, wodurch ihnen mehr Beispiele zum Vergleichen und zur Nutzung in eigenen Programmen zur Verfügung standen [20, S. 111]. Studierende der Testgruppe hatten im Durchschnitt bessere Klausurergebnisse erzielt. Außerdem hatten sie je weniger Buildfehler in Programmierprojekten, desto mehr Aufgaben sie abgeschlossen hatten [20, S. 111]. Jedoch konnten die Studierenden selbst entscheiden, ob sie an der Studie teilnehmen. Die aktivsten Nutzer von TYPOS waren PA, die in ihrer ersten Klausur eine schlechte Note erhielten. Dadurch können laut den Forschern kausale Zusammenhänge infrage gestellt werden, denn diese Gruppe könnte auch andere Methoden genutzt haben, um ihre Ergebnisse zu verbessern [20, S. 111].

In der Studie von Leinonen et al. wurde ein ähnlicher Ansatz wie der von Gaweda et al. [20] genutzt. Studierende bearbeiteten Tippaufgaben in einem Java-Kurs, bevor diese mit den erlernten Programmierkonstrukten Probleme lösen mussten [29]. Der Einsatz dieser Aufgaben wurde durch die Bedeutung der Beherrschung der Syntax im Programmierenlernen motiviert [29, S. 89]. Die Komplexität der Syntax von Programmiersprachen kann eine Herausforderung für PA darstellen: so erkennen und verstehen sie bspw. häufig Kompilierungsfehler, die auf der Syntax beruhen, nicht [29, S. 85]. Zur Unterstützung der PA setzten sie ein Tool ein, das Syntaxfehler farblich markierte und in die Online-Lernmaterialien eingebunden war [29, S. 86]. Sie fanden heraus, dass in ihrem Programmierkurs die zusätzlichen Tippaufgaben keine sinnvolle Ergänzung waren. Denn in ihrem Kurs wurden bereits vorher viele kurze Programmieraufgaben eingesetzt [29, S. 89]. Sie stellten die Hypothese auf, dass Studierende, welche die Aufgaben ausgeführt hatten, weniger Tippfehler beim Schreiben von Programmen machen würden. Dies trat jedoch nicht ein [29, S. 87]. Außerdem benötigten Studierende, die an der Studie teilgenommen hatten, nicht weniger Zeit für Programmieraufgaben [29, S. 87]. Insgesamt lässt sich in ihrer Studie festhalten, dass Syntaxaufgaben in Form von Tippaufgaben keine Verbesserung der Studierenden bewirkten [29, S. 87]. Jedoch weisen sie in ihrer Untersuchung auf Mängel hin, denn die Probanden mussten nicht unbedingt das Tool nutzen und konnten ihre Programme auch ohne die Nutzung abgeben [29, S. 89].

3.3 Programmieraufgaben

Denny et al. untersuchten in ihrer Studie anhand von Programmieraufgaben syntaktische Probleme von PA mithilfe der Webseite CodeWrite [14, S. 209]. Die Studie

erfolgte in der Mitte des ersten Semesters eines Programmierkurses [14, S. 209] Zur Teilnahme mussten Studierende eine eigene Aufgabe erstellen sowie mindestens zehn andere Aufgaben, welche auch von Studierenden erstellt worden waren, korrekt beantworten [14, S. 209]. Der Fokus der Wissenschaftler richtete sich auf die Häufigkeit von Syntaxfehlern in Abgaben und ob diese mit der Leistung der Studierenden korrelierte sowie inwieweit Syntaxfehler die Studierenden daran hinderten, Aufgaben zu lösen [14, S. 209]. Die Programme zur Abgabe waren mit durchschnittlich acht Zeilen sehr kurz [14, S. 210]. Die häufigste Fehlerart waren syntaktische Fehler [14, S. 210] Sie fanden heraus, dass auch leistungsstarke Studierende Probleme mit der Syntax hatten, denn fast die Hälfte aller Abgaben der Studierenden im obersten Quartil enthielt Syntaxfehler, trotz der simplen Aufgaben, die durchschnittlich nur aus acht Zeilen Code bestanden [14, S. 210]. Jedoch wurde in dieser Studie nicht die Wirkung der Aufgaben auf die Leistungen der Studierenden im Kurs untersucht.

4 Methodik

Das vorliegende Kapitel setzt an den im Unterabschnitt 2.3.7 aufgezeigten Problemen, welche durch die Syntax einer Programmiersprache entstehen, an, mit dem Ziel, die Datenlage zu erweitern. Es wird das Experiment beschrieben, welches den Effekt des Einsatzes von Syntaxaufgaben bei PA aufzeigt und dabei den Fokus auf Programmverständnis und Programmierfähigkeit legt. Dieses Kapitel startet im Abschnitt 4.1 mit der Konkretisierung des Forschungsinteresses durch Definition der unabhängigen und abhängigen Variablen sowie der Forschungsfragen, welche durch Hypothesen gestützt werden. Im Abschnitt 4.2 werden die Teilnehmer des Experiments beschrieben. Als Nächstes folgt eine Beschreibung der Materialien und Aufgaben, die im Experiment genutzt wurden. Schlussendlich wird im Abschnitt 4.4 das experimentelle Design beschrieben.

4.1 Forschungsziel

Dieses Experiment zielt darauf ab, die Auswirkung von Syntaxaufgaben bei PA zu untersuchen. Dabei wird besonders das Verhalten und die visuelle Aufmerksamkeit der Probanden untersucht. Unterabschnitt 2.3.7 veranschaulicht, dass die Syntax einer Programmiersprache eine entscheidende Rolle beim Programmierenlernen spielt. Obwohl das Problem bereits seit mehreren Jahrzehnten bekannt ist, gibt es wenig Untersuchungen oder Ansätze für dessen Lösung. Außerdem ist unklar, ob spezielle Syntaxaufgaben einen Einfluss auf die Leistungen von PA haben. Aufgrund dessen besteht das Ziel der vorliegenden Arbeit darin, die Rolle der Syntax für PA herauszuarbeiten. Diesbezüglich ist zum einen die Untersuchung des Programmverständnisses von Relevanz, zum anderen ist die Untersuchung der Programmierfähigkeit von Interesse. Die Motivation dieser Arbeit besteht darin, herauszufinden, ob es Gemeinsamkeiten beim Lernen einer neuen Programmiersprache im Vergleich zu einer natürlichen Sprache gibt. Bei einer natürlichen Sprache muss zu Beginn der Grundwortschatz gelernt werden, deren einzelne Vokabeln eine geringe Elementinteraktivität aufweisen, da die verschiedenen Wörter unabhängig voneinander sind und es somit keine Interaktivität zwischen diesen gibt [19, S. 3]. Beim Einsatz von Syntaxaufgaben wird die extrinsische Belastung erheblich reduziert, weil sich PA keine Gedanken über eventuell auftretende Syntaxfehler machen müssen, wodurch den Studierenden mehr kognitive Kapazitäten für die Problemlösung zur Verfügung stehen.

4.1.1 Unabhängige Variablen

Das vorliegende Studiendesign enthält eine unabhängige Variable. Die unabhängige Variable dieses Experiments besteht aus den vier verschiedenen Syntaxaufgaben, welche die Studierenden ausführen mussten. Syntaxaufgaben wurden gewählt, weil die Syntax einer Programmiersprache bereits seit vielen Jahren ein Problem in der CEdR darstellt [14]. Im Vergleich zu anderen etwaigen Interventionsmöglichkeiten wurden diese gewählt, weil im theoretischen Rahmen bereits herausgearbeitet worden ist, dass visuelle Programmiersprachen für Informatikstudierende nur geringe Vorteile bringen, da sie keine grundlegenden Programmierkonzepte vermitteln. Hingegen stellen der Einsatz eines neuen Lehrplans [76], die Nutzung einer simpleren Programmiersprache oder auch der Einsatz von Syntaxaufgaben eine effektive Möglichkeit zur Verbesserung der Programmierfähigkeiten von PA dar. Denn alle drei Verfahren haben sich bereits in der Praxis als wirkungsvoll erwiesen. Aufgrund des Umfangs der vorliegenden Arbeit wäre ein neuer Lehrplan eine zu komplexe Intervention gewesen. Da im Kurs Datenstrukturen die Option besteht Prüfungsvorleistungen (PVL) als auch die Klausur in Python zu schreiben, fällt die Option der Einführung einer simpleren Programmiersprache weg. Aufgrund dessen liegt in dieser Arbeit der Fokus auf dem Einsatz von Syntaxaufgaben, welche als Treatment gewählt wurden.

Im folgenden Unterabschnitt werden syntaktische Fehler genannt, welche in den vergangenen Jahren in der CEdR herausgearbeitet worden sind. In den Anfangsjahren der CEdR wurden wenige syntaktische Schwierigkeiten von PA herausgearbeitet. Dabei sei jedoch zu beachten, dass es, laut Robins, in den Anfangsjahren wenige Studien gab, welche die Programmerstellung untersuchten und deutlich mehr, welche das Programmverständnis erforschten. Hierfür wurde syntaktisch korrekter Quellcode genutzt [53, S. 144]. Du Boulay nennt in seinem Artikel zu Schwierigkeiten beim Lernen des Programmierens mit dem Vergessen des Semikolons am Ende einer Zeile nur einen syntaktischen Fehler, welcher durch das Übergeneralisieren "from one part of the language to another" [15, S. 71] entsteht. Im Gegensatz zu den Anfangsjahren wurden in den letzten Jahren einige Studien durchgeführt, welche die syntaktischen Fehler bei PA untersucht haben [1, 13, 22, 24, 25, 73].

Bis zum Jahre 2015 wurden zur Erfassung syntaktischer Schwierigkeiten nur die Daten einiger Institutionen oder weniger Probanden genutzt. 2015 führten Altadmri und Brown die erste weltweite Studie durch, indem sie die Daten von etwa 265.000 Studierenden untersuchten [1, S. 522]. Die Teilnehmer nutzten die Entwicklungsumgebung BlueJ für die Programmiersprache Java und haben innerhalb eines Jahres etwa 37 Millionen Mal ihre Programme kompiliert [1, S. 522]. Fast die Hälfte dieser Kompilierungen konnte nicht erfolgreich ausgeführt werden und der häufigste Fehler bestand in einer falschen Klammersetzung [1, S. 523f.]. Auch in anderen Studien wurde die falsche Klammersetzung als häufiges Problem identifiziert [24, S. T4C-25; 22, S. 154; 25, S. 31; 13, S. 79]. Daneben vergessen PA oft ein Semikolon, das als Abschluss einer Anweisung benötigt wird [39, S. 5; 25, S. 31]. Außerdem werden häufig Variablen nicht deklariert, wodurch der Fehler "cannot resolve identifier" auftritt [13, S. 79; 24, S. T4C-25]. Hristova et al. fanden zudem auch falsche

Trennzeichen im Schleifenkopf einer for-Schleife [22, S. 154]. Für eine genauere Betrachtung verschiedener syntaktischer Fehler sei auf die oben verlinkten Studien hingewiesen. Außerdem sei auch noch darauf hinzuweisen, dass in einigen Studien semantische und Typfehler als häufigere Fehlerarten genannt werden [1, 9, 39].Anhand der Ergebnisse dieser Studien lässt sich festhalten, dass viele Kompilierungsfehler von PA auf die Syntax einer Programmiersprache zurückzuführen sind. Trotz der simplen Fehler, benötigen PA oft viel Zeit zum Verbessern dieser, wobei auch leistungsstarke Studierende Syntaxfehler machen. Als problematisch erweist sich dabei die zusätzliche extrinsische kognitive Belastung, auf welche im Unterabschnitt 2.3.7 genauer eingegangen wurde.

Probleme beim Meistern der Java-Syntax können ein wichtiges Anzeichen für den Erfolg von Studierenden in einem Programmiereinführungskurs sein [44, S. 703]. Dieses Treatment basiert auf der Idee, dass sich Studierende erst auf die Syntax fokussieren, um dadurch mehr kognitive Kapazitäten für die Problemlösung zur Verfügung zu haben. Dadurch kann die Beherrschung der Syntax als Grundlage für das Erlernen des Problemlösens gesehen werden [17, S.216]. Außerdem fokussieren sich Programmiereinführungskurse aktuell auf Algorithmen und Problemlösung und lassen in der Regel die Syntax außer Acht. Von den Studierenden wird erwartet, dass sie die Syntax selbst erlernen. Falls sie dabei Probleme haben, sind sie auf sich alleine gestellt und erhalten keine Aufgaben, die ihnen beim Erlernen einer Programmiersprache helfen könnten [14, S. 208]. Ein weiterer Grund für die Auswahl dieses Treatments liegt in der Einfachheit der Integrierung von Syntaxaufgaben in den Lehrplan, da dieser nicht unterbrochen wird und nicht viel Zeit für die Bearbeitung von Syntaxaufgaben benötigt wird [17, S. 217]. Schlussendlich könnte ein möglicher Grund für die Problematik der Syntax beim Programmieren in der Verschmelzung der zwei Probleme - Syntax und Problemlösen - liegen, die parallel von PA bewältigt werden müssen [17, S. 217]. Edwards et al. vergleichen dies mit der Verwendung eines Werkzeugs, ohne dieses bereits ausgiebig gekannt zu haben. Das Werkzeug stellt die Syntax dar und PA müssen viel mehr kognitive Ressourcen in die Beherrschung des Werkzeugs als in die Werkzeugverwendung investieren. Eine automatisierte Nutzung des Werkzeugs kann zur Freisetzung von kognitiven Kapazitäten führen, die für die Problemlösung zur Verfügung stehen [17, S. 217].

Zur Festigung der Syntax bei PA gibt es verschiedene Aufgabentypen. Diese werden nach Sullivan et al. in drei verschiedene Kategorien unterteilt: Programmieraufgaben, Tippaufgaben und Syntaxaufgaben [67, S. 783]. Bei Tippaufgaben wird Studierenden ein Programm gezeigt und sie müssen dieses eins zu eins abtippen, um dadurch die Syntax einer Programmiersprache zu lernen [67, S. 783]. PA benötigen für diesen Aufgabentyp kein syntaktisches Wissen der Programmiersprache, sie lernen diese nicht nur durch das Lesen von Programmen, sondern vor allen Dingen durch das Tippen [67, S. 783]. Im Gegensatz dazu stehen Programmieraufgaben. Bei diesen müssen Studierende einfache Programmieraufgaben lösen. Sullivan et al. nennen als Beispiel das Zählen einer Zahl in einem Array [67, S. 783]. Dieser Aufgabentyp beinhaltet jedoch neben der extrinsischen Belastung durch die Syntax auch noch die der Problemlösung und konzentriert sich mehr auf diese. Syntax-

aufgaben sind ein repetitiver Ansatz zum Lernen der Syntax einer Programmiersprache. Jedoch müssen sich Studierende bei diesem Aufgabentyp an die Syntax der Programmierkonstrukte erinnern und den Quellcode nicht nur wie bei den Tippaufgaben abtippen. Beispielsweise sollen Studierende ein Programm ergänzen, bei dem ein Teil einer Schleifen-Initialisierung fehlt oder auch Fehler verbessern, die eine Kompilierung des Programms unmöglich machen [67, S. 783].

In diesem Experiment wurden Syntaxaufgaben ausgewählt, da diese komplexer sind als Tippaufgaben, aber dennoch eine strikte Trennung der Syntax von anderen Aspekten bei der Programmierung, wie bspw. der Problemlösung, erlauben. Außerdem haben alle Teilnehmer bereits einen Einführungskurs in die Programmierung (Algorithmen und Programmierung (AuP)) belegt und sollten daher ein gewisses Maß an Programmiererfahrung mitbringen, auch wenn in diesem Kurs eine andere Programmiersprache (nämlich C) unterrichtet wurde. Nach der Aufteilung der Syntaxaufgaben von Gonzales, entsprechen die hier gewählten Aufgaben debugging exercises [21, S. 13]. PA müssen Fehler in einem kurzen Programm entdecken und diese beheben. Mithilfe der Syntaxübungen wird die hohe kognitive Belastung, die durch die Elementinteraktivität in Programmieraufgaben gegeben ist, verringert. Diese geringe kognitive Belastung ermöglicht das Einprägen der Syntax in das prozedurale Gedächtnis [67, S. 783]. Dadurch müssen sich PA nur auf das Beheben von syntaktischen Fehlern konzentrieren und nicht auf die Implementierung oder Ausführung eines Algorithmus. Im weiteren Verlauf des Programmierenlernens ist bei der Implementierung dieser Konstrukte eine geringere kognitive Belastung gegeben, womit mehr kognitive Ressourcen für die Problemlösung verfügbar sind. Hierdurch sollen die Leistungen der Studierenden verbessert werden. Für die Syntaxaufgaben wurden vier verschiedene Konstrukte - Bedingte Anweisungen, for-, while und do-while-Schleifen – genutzt, da diese seit vielen Jahren in der CEdR als Problem von PA identifiziert, aber immer noch nicht eliminiert worden sind [64]. Außerdem bauen neue Programmierkonstrukte und komplexere Aufgaben auf diesen Konstrukten auf [34, S. 26]. Wenn ein Studierender diese nicht versteht, wird er in seiner gesamten Programmierlaufbahn immer wieder Probleme bei der Einführung neuer Programmierkonstrukte bekommen.

Bedingungen wurden gewählt, da diese eine Grundlage vieler Programme sind, denn sie behandeln die Steuerung und den Ablauf eines Programms. Eine Bedingung beinhaltet eine Anweisung, die im Programmverlauf als wahr oder falsch ausgewertet wird. Studierende müssen hierbei boolesche Ausdrücke, den Programmfluss und die Kontrolle verstanden haben [21, S. 18].

Schleifen sind neben Bedingungen ein weiteres kritisches Thema bei der Programmierung, welches die Kontrolle und den Fluss eines Programms beinhaltet. Schleifen sind ein grundlegendes Konzept in der Informatik und sind in vielen Programmen enthalten. Durch den Einsatz von Schleifen werden Anweisungen wiederholt ausgeführt [2]. Schleifen wurden ausgewählt, da hierbei Probleme in Randsituationen und Abfolgen berücksichtigt werden müssen [21, S. 18].

In dem vorliegenden Experiment wurde nur eine Programmiersprache gewählt, da mehrere Sprachen den Rahmen der vorliegenden Arbeit gesprengt hätten. Die

Entscheidung für die Programmiersprache Java hängt von dem Curriculum des Kurses Datenstrukturen an der TUC ab. Außerdem haben viele Studierende in vorherigen Kursen Java noch nicht gelernt, wodurch ein größerer Effekt des Syntaxtrainings gegeben sein kann. Denn vorherige Studien analysierten die Syntax als wesentliche Hürde für den Einstieg in die Programmierung [14, 17, 67]. Die unabhängige Variable wird noch in vier verschiedene Aufgabentypen unterteilt - Bedingte Anweisungen, *for-*, *while-* und *do-while-Schleifen*. Zur Auswahl dieser vier Arten kam es durch den Aufbau grundlegender Programmierkonstrukte, die in drei aufeinander aufbauenden Ebenen kategorisiert werden können - *Sequences*, *Selection* und *Repetition*. Laut Böhm und Jacopini können durch den Einsatz von Programmierkonstrukten aus diesen drei Ebenen alle Programme geschrieben werden [11]. Sequences stellen den allgemeinen Ablauf eines Programms dar und sind trivial, weswegen sie in den Syntaxaufgaben nicht explizit gelernt werden können. Selections beschreiben die Auswahl einer Alternative, welche in Java entweder durch if-else- oder switch-Anweisungen implementiert werden können. Sie sind die Grundlage für die Syntaxaufgaben der bedingten Anweisungen. Die dritte Ebene besteht aus Repetitions, d.h. aus den vier verschiedenen Schleifenarten, welche es in Java gibt - for-, for-each-, while- und do-while-Schleifen. Wobei For-each-Schleifen nicht in die Syntaxaufgaben aufgenommen wurden, da diese den for-Schleifen zu sehr ähneln und der Aufwand für die Studierenden zu groß und damit nicht vertretbar gewesen wäre.

4.1.2 Abhängige Variablen

Als Messgrößen wurden sowohl das Programmverständnis, die Programmierfähigkeit und die Klausurergebnisse der Probanden gewählt, weil diese Aspekte wichtige Indikatoren für die Leistung der Studierenden darstellen. Das Programmverständnis kann dabei als Vorläufer der Programmierfähigkeit gesehen werden. Zur Messung des Programmverständnisses wurden Vergleichsaufgaben gewählt, da es sich bei diesen um eine zuverlässige Messgröße handelt, wie in Kapitel 2.5.1 dargelegt. Think-Aloud-Protokolle wurden ausgeschlossen, da der Kurs Datenstrukturen von mindestens 100 Teilnehmern besucht wurde. Falls ein Großteil dieser an dem Experiment teilgenommen hätte, wäre der zeitliche Aufwand zu hoch für diese Arbeit gewesen. Als abhängige Variablen wurden die Beantwortungszeit, die Korrektheit und die visuelle Aufmerksamkeit gewählt. Letztere wird nur bei den Programmverständnisaufgaben gemessen. In der vorliegenden Arbeit wird die Beantwortungszeiteit der Programmverständnisaufgaben folgendermaßen definiert – sie beginnt mit dem Aufrufen eines Quellcodestücks und endet mit der Abgabe einer Antwort. Die Korrektheit der Programmverständnisaufgaben gibt an, ob die Studierenden eine korrekte Antwort abgegeben haben (z. B. die Ausgabe der Anzahl der Vokale eines Strings). Die visuelle Aufmerksamkeit wird mit REyeker, einem Remote Eye Tracker, gemessen [43]. Dabei wird ein Bild des Quellcodestücks verwischt und aus der Ferne beobachtet, welche Zeilen die Probanden gerade betrachten, da durch einen Klick ein Teil des Quellcodes lesbar wird. Ein Klick macht einen rechteckigen Bereich

sichtbar, welcher ein Pixel hoch und 200 Pixel breit ist sowie einen Übergangsbereich von 30 Pixeln umfasst. Mithilfe von REyeker kann sowohl die Fixierungsanzahl der AOI als auch die Beantwortungszeit der Studierenden pro Aufgabe bestimmt werden. Wie im Unterabschnitt 2.5.1 erläutert, gehören diese zu den meist gewählten Eye-Tracking-Metriken in der CEdR. Als AOI werden die verschiedenen Programmierkonstrukte, die in den Syntaxaufgaben trainiert worden sind, definiert. Dabei wird das gesamte Konstrukt gezählt, mit einem gewissen Spielraum, der etwa eine halbe Zeile über bzw. unterhalb des Programmierkonstrukts zusätzlich einbezieht.

Hingegen stellen bei den Programmieraufgaben die abhängigen Variablen die Beantwortungszeit der Aufgaben sowie die Korrektheit der Aufgabe dar. Die Beantwortungszeit beginnt mit dem Starten der Aufgabe in Online-Plattform für Akademisches Lehren und Lernen (OPAL) und endet mit der Abgabe. Im Gegensatz zu den Programmverständnisaufgaben wird die Korrektheit der Programmieraufgaben anhand der semantischen und syntaktischen Fehler gemessen. Die maximale Anzahl liegt pro Fehlerart bei fünf. Denn bei Programmverständnisaufgaben ist es nicht möglich zu erkennen, an welcher Stelle die Probanden Fehler gemacht haben. Hingegen kann bei Programmieraufgaben geprüft werden, ob die Studierenden bspw. nur einen simplen Syntaxfehler oder einen grundlegend falschen Algorithmus programmiert haben. Dabei entspricht jedes Problem im Quellcode, das eine korrekte Kompilierung verhindert, einem Fehler. Bspw. wird bei einem grundlegend falschen Algorithmus, welcher nicht mit wenigen Anpassungen zum korrekten Ergebnis führt, die Höchstzahl an semantischen Fehlern vergeben.

Schlussendlich wird bei der Klausur nur die Korrektheit in Punkten gemessen.

4.1.3 Forschungsfragen

Anhand der verwandten Arbeiten, die in Kapitel 3 vorgestellt wurden, lauten die Forschungsfragen der vorliegenden Arbeit folgendermaßen:

> **Forschungsfrage 1**: Wie beeinflussen Syntaxaufgaben die Beantwortungszeit, Korrektheit und visuelle Aufmerksamkeit von PA in Programmverständnisaufgaben?
>
> **Forschungsfrage 2**: Wie beeinflussen Syntaxaufgaben die Beantwortungszeit und Korrektheit von PA in Programmieraufgaben?
>
> **Forschungsfrage 3**: Wie beeinflussen Syntaxaufgaben die Ergebnisse von PA in der Klausur?

4.1.4 Hypothesen

In diesem Unterabschnitt werden die Hypothesen diskutiert. Als Grundlage für die Hypothesen zur ersten Forschungsfrage dienen die im Unterabschnitt 2.3.7 dargelegten Befunde.

Hinsichtlich der Beeinflussung der Programmverständnisaufgaben werden folgende Hypothesen genannt:

H_1 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Beantwortungszeit bei den Programmverständnisaufgaben.

H_2 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Korrektheit bei den Programmverständnisaufgaben.

H_3 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Klickanzahl in den AOIs bei den Programmverständnisaufgaben.

Bezüglich der Beeinflussung der Ergebnisse von PA in Programmieraufgaben durch Syntaxaufgaben lassen sich folgende Hypothesen formulieren:

H_4 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Beantwortungszeit bei den Programmieraufgaben.

H_5 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Korrektheit bei den Programmieraufgaben.

Hinsichtlich der Klausurergebnisse wird folgende Hypothese definiert:

H_6 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Korrektheit in der Klausur.

4.2 Studienteilnehmer

Dieses Experiment richtet sich an PA, weswegen zunächst Teilnehmer gefunden werden mussten, die PA sind. Studienteilnehmer waren Informatikstudierende der TUC, die den Kurs Datenstrukturen im Sommersemester 2021 belegten. Dabei handelte es sich größtenteils um Bachelorstudierende, aber auch Masterstudierende, welche dem nicht-konsekutiven Master Informatik für Geistes- und Sozialwissenschaftler angehörten. Die Studie ist kein genereller Bestandteil des Kurses, aber die Probanden mussten bei einer Beteiligung weniger komplexe PVL bearbeiten, deren Bestehen eine Voraussetzung für die Teilnahme an der Klausur am Ende des Semesters ist.

4.3 Materialien und Aufgaben

Für diese Studie wurden 15 verschiedene Programmverständnis- und vier verschiedene Programmieraufgaben sowie 98 Syntaxaufgaben eingesetzt[1]. Außerdem wurde ein Video, ein Fragebogen, welcher demografische und Programmiererfahrungen erfasst sowie ein Fragebogen zur Messung der Schwierigkeit und etwaiger Probleme bei den Programmverständnisaufgaben, genutzt.

[1]Die eingesetzten Materialien und deren Analyse sowie Ergebnisse können auf GitHub abgerufen werden: https://github.com/gord6/master_arbeit

4.3.1 Einleitungsvideo

Bei Beginn der Studie wurde den Probanden ein Video angezeigt, welches eine Einführung in den Remote Eye Tracker REyeker [43] lieferte.

4.3.2 Fragebogen

Zur Bestimmung der demografischen Daten und Messung der Programmiererfahrung wurde ein validierter Fragebogen eingesetzt [63]. Dieser bestand aus vier Seiten. Auf der ersten Seite mussten Studierende in einer Multiple Choice Abfrage wählen, ob sie sich aktuell im Bachelor, Master oder einem anderen Abschluss, den sie in einem Textfeld hinzufügen konnten, befanden. Außerdem ist auf dieser Seite auch die Eintragung des aktuell belegten Studienganges vorgesehen, der als Text hinzugefügt werden musste. Als Letztes sollten die Teilnehmer ihr aktuelles (Fach-)semester eintragen. Alle drei Felder waren verpflichtend. Die zweite Seite des Fragebogens fragte den aktuellen beruflichen Status der Teilnehmer ab - Student (ohne Abschluss), Student (mit Abschluss), Universitätsangestellter, Programmierer/Entwickler oder anderes. Außerdem mussten die Teilnehmer angeben, wie viel Erfahrung sie in Jahren im Programmieren besitzen, sowohl beruflich als auch in der Bildung. Der letzte Punkt auf dieser Seite bezog sich darauf, wie die Studierenden ihre eigenen Programmiererfahrungen gegenüber den Mitstudierenden einschätzen. Auch auf dieser Seite waren wieder alle drei Fragen verpflichtend. Auf der vorletzten Seite wurden die Kenntnisse in vier bestimmten Programmiersprachen auf einer fünfstufigen Skala mit den Stufen - sehr unerfahren, unerfahren, mäßig erfahren, erfahren, sehr erfahren - abgefragt. Die Auswahl dieser vier Sprachen kam zustande, da Studierende der Fakultät Informatik an der TUC im Regelfall Kontakt zu diesen Programmiersprachen haben. In dem Vorkurs zu Datenstrukturen, AuP, lernen sie C und Python. C++ lernen sie bspw. in Grundlagen der Informatik, einem Kurs, den einige Studierende des Masters Informatik für Geistes- und Sozialwissenschaftler, ebenfalls belegen müssen. Währenddessen wird der Kurs Datenstrukturen sowohl in Java als auch in Python gelehrt. Um einen Überblick über weitere Programmierkenntnisse zu erhalten, wurde in der letzten Frage dieser Seite abgefragt, ob die Probanden noch andere Programmiersprachen beherrschen. Bis auf die letzte Frage waren alle Angaben verpflichtend. Auf der letzten Seite des allgemeinen Fragebogens wurde nach dem (biologischen) Geschlecht der Teilnehmer mit folgenden Auswahlmöglichkeiten gefragt: männlich, weiblich und keine Angabe. Die letzte Frage lautete "Wie alt sind sie?". Die Beantwortung der ersten Frage war obligatorisch, weil in früheren Studien herausgefunden worden ist, dass das Geschlecht eine Auswirkung auf die Programmierfähigkeiten haben kann. Im Gegensatz dazu war die Angabe des Alters optional.

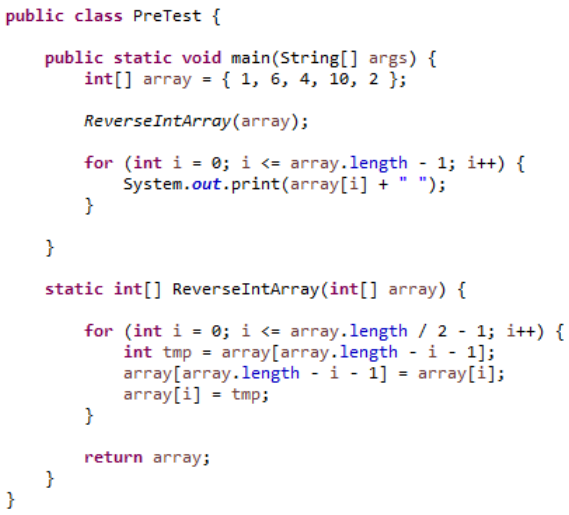
4.3.3 Programmverständnisaufgaben

Im Pre- und Posttest wurden insgesamt 15 verschiedene Programmverständnisaufgaben genutzt. Jeweils fünf Aufgaben waren im Pre- und Posttest gleich, enthielten jedoch eine andere Eingabe und damit auch eine andere Ausgabe. Zuerst werden die gleichen Aufgaben als V1 bis V5 beschrieben. Danach folgen V6 bis V10 und damit die letzten fünf Aufgaben des Pretests. Schlussendlich stellen V10 bis V15 die unterschiedlichen Aufgaben des Posttests dar.

Bei der ersten Programmverständnisaufgabe (V1) wird den Teilnehmern mit Common Chars eine Methode gezeigt, welche 2 Strings übergeben bekommt und die Anzahl gleicher Buchstaben an gleicher Stelle in beiden Wörtern ausgibt. Hingegen wird der Methode ReverseArray (V2) ein Integer-Array übergeben, dessen Reihenfolge der Elemente umgedreht wird. Die Ausgabe gibt das umgedrehte Array an. Die dritte Aufgabe (V3) Binary Search Strings sucht binär, ob ein Array aus Strings einen bestimmten String enthält. Falls in dem Array aus Strings der gesuchte String gefunden worden ist, wird dessen Index ausgegeben. Hingegen wird bei Multiples (V4) ein Integer übergeben und alle Zahlen von null an, die kleiner als die übergebene Zahl sind, sollen addiert werden, wenn sie das Vielfache von zwei oder drei sind. Die Summe dieser Zahlen wird ausgegeben. Bei der Methode InsertionSort (V5) wird der Methode ein unsortiertes Integer-Array übergeben, welches mithilfe des Sortieralgorithmus vom kleinsten zum größten Wert sortiert wird. Das sortierte Array wird ausgegeben.

Bei V6 werden der Methode zwei Strings übergeben und diese gibt zurück, ob der erste String den zweiten enthält. Falls dies der Fall ist, wird True ausgegeben, ansonsten False. Dem folgt ein Programm, welches die Anzahl der Vokale zählt (V7). Der Methode wird ein String übergeben und die Anzahl der Vokale wird ausgegeben. Als Nächstes soll in V8 die Kreuzsumme eines übergebenen Integers berechnet und zurückgegeben werden. Hingegen werden bei der neunten Programmverständnisaufgabe (V9) einem Programm zwei Variablen übergeben und getauscht. Die erste der beiden Variablen wird ausgegeben. Schlussendlich werden der letzten Aufgabe (V10), zwei Integer übergeben und der größte gemeinsame Teiler berechnet sowie ausgegeben.

Im Posttest folgen mit V11 bis V15 unterschiedliche Methoden als im Pretest. Die Komplexität entspricht V6 bis V10. Dadurch soll sichergestellt werden, dass PA nicht nur besser abschneiden, weil sie bereits die Methoden aus dem Pretest kennen. Bei V11 wird geprüft, ob ein übergebener String ein Palindrom ist. Falls dies der Fall ist, wird True ausgegeben, ansonsten False. Dem folgt mit V12 auch eine Methode, der ein String übergeben wird. Jedoch soll dieser umgedreht werden. Der umgedrehte String wird ausgegeben. Hingegen werden bei V13 zwei Integer übergeben. Dabei wird eine Zahl berechnet, die mit einer anderen potenziert wird. Der erste Integer stellt die Basis dar und der zweite den Exponenten. Das Ergebnis wird ausgegeben. Dem folgt mit V14 die Ausgabe des mittleren Elements eines Strings, wenn dieser ungerade viele Buchstaben enthält. Ansonsten werden die Elemente, welche um jeweils 1 von der Mitte abweichen, ausgegeben. Zum Schluss



```
public class PreTest {

    public static void main(String[] args) {
        int[] array = { 1, 6, 4, 10, 2 };

        ReverseIntArray(array);

        for (int i = 0; i <= array.length - 1; i++) {
            System.out.print(array[i] + " ");
        }

    }

    static int[] ReverseIntArray(int[] array) {

        for (int i = 0; i <= array.length / 2 - 1; i++) {
            int tmp = array[array.length - i - 1];
            array[array.length - i - 1] = array[i];
            array[i] = tmp;
        }

        return array;
    }
}
```

Abbildung 4.1: Programmverständnisaufgabe V2 - Als Bild eingebundener Quellcode in REyeker / SoSci Survey

werden bei V15 die Quadratwurzeln eines übergebenen Integer-Arrays berechnet. Das neu erstellte Array wird ausgegeben.

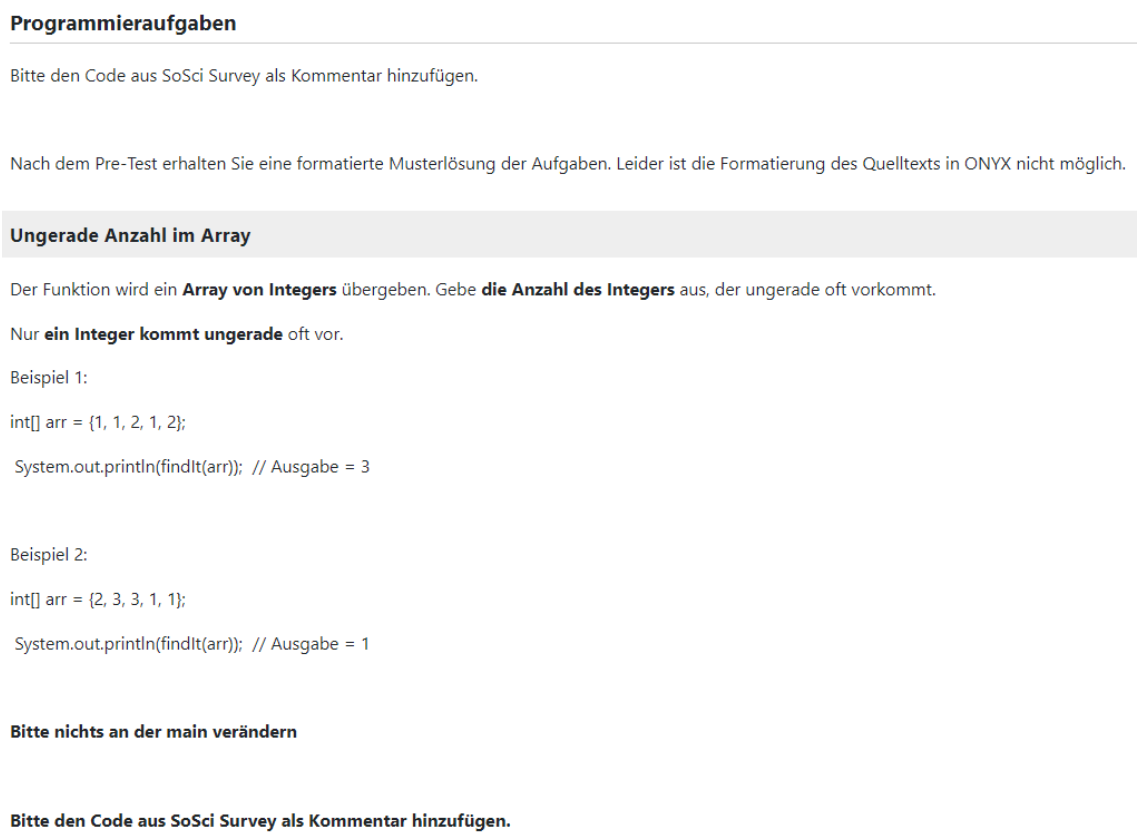
Bei allen Programmverständnisaufgaben wurden die Probanden aufgefordert, die endgültige Ausgabe einzugeben. In der Abbildung 4.1 ist die Aufgabe V2 dargestellt. Das Array wird dabei umgedreht und das korrekte Ergebnis lautet "2 10 4 6 1". Der Grund für die Festlegung der Aufgabe auf die Angabe der Ausgabe besteht darin, dass die Möglichkeit ausgeschlossen werden sollte, dass neben dem Algorithmus auch die Art der Aufgabe das Verständnis der Probanden beeinflusst.

4.3.3.1 Fragebogen zu Programmverständnisaufgaben

Nach jedem Quellcodestück folgt eine Seite mit zwei Fragen, die erste zur Schwierigkeit des Programms, welche auf einer fünfstufigen Skala - Gar nicht so schwer, Irgendwie schwierig, Schwer, Sehr schwierig, Extrem schwierig - abgefragt wurde. Für die zweite Frage zu etwaigen Problemen gab es vier Antwortmöglichkeiten - Nein, Verständnis des Algorithmus, Benennung von Variablen, Technische Probleme (Internet Probleme, Bild hat nicht geladen, Wolken sind nicht verschwunden beim klicken) und Anderes.

4.3.4 Programmieraufgaben

Die Programmieraufgaben werden in der ONYX-Testsuite, einer Software, die das Erstellen elektronischer Prüfungen ermöglicht, erstellt. ONYX wurde genutzt, weil diese bereits in der Lernplattform OPAL, welche von den sächsischen Hochschulen genutzt wird, integriert ist. Dadurch hatten die Probanden über den OPAL-Kurs Datenstrukturen direkten Zugriff auf die Programmieraufgaben. Die Programmieraufgaben sollten dabei nicht zu simpel, aber auch nicht zu komplex sein, da die



Programmieraufgaben

Bitte den Code aus SoSci Survey als Kommentar hinzufügen.

Nach dem Pre-Test erhalten Sie eine formatierte Musterlösung der Aufgaben. Leider ist die Formatierung des Quelltexts in ONYX nicht möglich.

Ungerade Anzahl im Array

Der Funktion wird ein **Array von Integers** übergeben. Gebe **die Anzahl des Integers** aus, der ungerade oft vorkommt.

Nur **ein Integer kommt ungerade** oft vor.

Beispiel 1:

```
int[] arr = {1, 1, 2, 1, 2};
System.out.println(findIt(arr));  // Ausgabe = 3
```

Beispiel 2:

```
int[] arr = {2, 3, 3, 1, 1};
System.out.println(findIt(arr));  // Ausgabe = 1
```

Bitte nichts an der main verändern

Bitte den Code aus SoSci Survey als Kommentar hinzufügen.

Abbildung 4.2: Aufgabenbeschreibung der Programmieraufgabe P2

Lösung dieser die Programmierfähigkeit der Studierenden widerspiegeln sollten. Es wurde auf verschiedenen Programmierplattformen, wie bspw. CodeWars[2], nach Programmieraufgaben gesucht, welche dann an Kommilitonen weitergeleitet worden sind, um einschätzen zu lassen, ob die Aufgaben zu komplex oder zu einfach sind. Die Studierenden sollten in dem vorliegenden Experiment Aufgaben erhalten, die mit Basiskonstrukten der Programmiersprache auskommen und keine Einbindung weiterer Bibliotheken benötigen. Hierbei wurden im Pre- und Posttest jeweils eine Stringmanipulations- und eine Array-Aufgabe gewählt. Bei allen Aufgaben war die Klasse, die main-Methode sowie der Methodenkopf bereits vorgegeben. Zuerst folgt eine Darstellung der Aufgaben P1 und P2, welche für den Pretest genutzt worden sind. Bei der Aufgabe P1 müssen Studierende eine Methode schreiben, welche einen String übergeben bekommt. Dieser String besteht aus mehreren Worten, die mit einem Leerzeichen getrennt sind. Die Probanden sollten den String in die verschiedenen Wörter aufteilen und die Länge des kürzesten Wortes ausgeben. Hingegen wurde bei P2 der Methode ein Integer-Array übergeben. Eines dieser Elemente kommt ungerade oft vor. Die Probanden sollten die Anzahl dieses Elements angeben. Die genaue Aufgabenbeschreibung ist in der Abbildung 4.2 präsentiert.

[2]https://www.codewars.com/

Schlussendlich mussten von den Probanden auch im Posttest mit P3 und P4 zwei Aufgaben bearbeitet werden. Bei P3 wurde der Methode ein String und ein Integer übergeben. Der String sollte entsprechend dem Wert des übergebenen Integers oft wiederholt werden. Als Trennzeichen zwischen den Wörtern sollte ein Komma und ein Leerzeichen gesetzt werden. Jedoch nicht hinter dem letzten Wort. Bei der letzten Programmieraufgabe P4 bestand die Aufgabe darin, aus einem Integer-Array, welches einer Methode übergeben wird, das größtmögliche Produkt zweier benachbarter Elemente auszugeben.

4.3.5 Syntaxaufgaben

Als Treatment wurden Syntaxaufgaben gewählt. Der Schwerpunkt dieses Experiments liegt auf der Syntax einer Programmiersprache, weil, wie bereits erwähnt, in vielen Studien untersucht und festgestellt worden ist, dass die Syntax durch ihre extrinsische kognitive Belastung ein Problem für PA darstellt. Beim Lernen einer neuen natürlichen oder auch einer Programmiersprache ist es gewöhnlich, dass Lernende syntaktische Fehler machen [51, S. 1:4].

Insgesamt wurden pro Konstrukt 20-25 Aufgaben erstellt, für welche die Studierenden jeweils etwa 15 Minuten benötigen sollten. Außerdem wurden die Tests automatisiert korrigiert, wodurch die Probanden ein direktes Feedback erhielten. In den Programmen waren Syntaxfehler enthalten, welche die Studierenden verbessern mussten, damit das Programm kompilierbar war. Beispielhaft seien hier einige Fehlerarten zu nennen, die in den Syntaxaufgaben gestellt wurden: das Fehlen einer geschlossenen oder offenen Klammer, falsche Groß-/Kleinschreibung eines Schlüsselwortes, ein fehlendes Semikolon, fehlende Trennzeichen im Schleifenkopf einer for-Schleife oder das Fehlen einer Zählervariable in einer while- oder do-while-Schleife.

Zur besseren Veranschaulichung ist in der Abbildung 4.3 die 23. Aufgabe des if-else-Bedingungen-Tests zu sehen. In diesem Quellcodestück sind insgesamt drei Fehler enthalten. Die if-Bedingung in Zeile sechs muss von zwei runden Klammern, anstatt eckigen Klammern umgeben sein. Außerdem fehlt in Zeile acht hinter dem else eine öffnende geschweifte Klammer.

4.4 Experimentelles Design

Das vorliegende Experiment besteht aus einem Pretest, Syntaxaufgaben und einem Posttest. Zur Messung der abhängigen Variablen wurde ein Pretest-Posttest-Design gewählt, um den Einfluss des Treatments, der Syntaxaufgaben, auf die abhängigen Variablen durch Messwiederholungen zu quantifizieren. Dabei führten dieselben Probanden den Pretest, das Treatment und den Posttest durch. Somit entspricht das vorliegende Experiment einem within-subject-Design. Dadurch können die Ergebnisse innerhalb einer Gruppe und innerhalb der einzelnen Probanden sowohl vor als auch nach dem Einsatz des Treatments analysiert werden. In den folgenden Absätzen werden die Abläufe des Experiments, des Pretests, der Syntaxaufgaben und

Aufgabe 23

```
1  class Main {
2      public static void main(String [] args){
3          int a = 0;
4          int b = 5;
5
6          if [a >= b] {
7              System.out.println("a >= b");
8          } else
9              System.out.println("a < b");
10         }
11     }
12 }
```

Abbildung 4.3: Quellcode - Syntaxaufgabe if-else-Bedingungen 23

des Posttests erläutert.

Das Sommersemester 2021 bestand aus 14 Vorlesungswochen. Für alle drei Stufen des vorliegenden Experiments hatten die Studierenden zwei Wochen Zeit. Dadurch entwickelte sich folgender Ablaufplan: Innerhalb der ersten vier Wochen des Semesters fand der Pretest statt, innerhalb der nächsten vier Wochen bearbeiteten die Studierenden Syntaxaufgaben, innerhalb der darauffolgenden vier Wochen führten die Studierenden den Posttest durch. Der Posttest sollte nicht in das direkte Semesterende fallen, da die Teilnehmer sich zu diesem Zeitpunkt auf die Klausuren konzentrieren müssen und meist auch schon genügend Punkte für das Bestehen der PVL gesammelt haben. Dadurch könnten bei einer späteren Durchführung weniger Probanden am Posttest des Experiments teilnehmen.

Der Pretest bestand aus einem einleitenden Fragebogen und zwei praktischen Teilen – Programmverständnis- und Programmieraufgaben. Für den Fragebogen sowie die Programmverständnisaufgaben wurde SoSci Survey [28] genutzt. Aufgrund der aktuellen Pandemiesituation musste für Programmverständnisaufgaben auf einen Remote Eye Tracker zurückgegriffen werden, dabei wurde das Tool REyeker[3] genutzt. SoSci Survey wurde gewählt, da dies ein bekanntes Werkzeug ist und bereits in einer vorherigen Studie genutzt wurde, in der REyeker eingebunden worden ist [2]. Außerdem war dies die bestmögliche Option um auch während der Corona-Pandemie und der damit einhergehenden Maßnahmen der TUC diese Studie durchzuführen. In naher Zukunft wäre es möglich, die Studie zu replizieren, wobei ein stationärer Eye-Tracker genutzt werden kann. Außerdem könnte auch eine Think-Aloud-Studie durchgeführt werden, um genauere Auswirkungen der Syntaxaufgaben zu analysieren. Für die SoSci Survey Umfrage sollten die Studierenden

[3]Für weitere Informationen siehe: [43]

etwa 30 Minuten benötigen, für die beiden Programmieraufgaben auch etwa 30 Minuten, jeweils 15 Minuten pro Aufgabe. Nach jeder der zehn Programmverständnisaufgaben folgte ein Fragebogen, der eine verpflichtende fünfstufige Skala zur Schwierigkeit des Algorithmus und eine optionale Angabe zum Auftreten von Problemen beim Verstehen des Programms enthielt. Zu den Problemen zählen technische Probleme, der Algorithmus oder auch die ungenaue Benennung von Variablen. Am Ende der SoSci Survey-Umfrage folgte die Möglichkeit, weitere Anmerkungen zur Umfrage zu machen sowie die Erstellung eines einzigartigen Codes. Dieser Code war notwendig zur Anonymisierung, aber auch zur gleichzeitigen Verknüpfung verschiedener Studienergebnisse. Als Letztes folgte ein Dankesschreiben an die Probanden. Nach der Beantwortung der Programmverständnisaufgaben sollten die Probanden zwei Programmieraufgaben lösen.

Am Ende des Semesters folgte ein Post-Test, welcher entsprechend dem Pretest aus Programmverständnis- und Programmieraufgaben bestand. Die Programmverständnisaufgaben wurden in SoSci Survey erstellt und mithilfe des Tools REyeker umgesetzt. Erneut mussten die Probanden zehn Programme verstehen, von denen fünf den Pretest Programmen entsprachen. Jedoch haben diese andere Eingabe-Parameter und damit auch eine andere Ausgabe. Bei den restlichen fünf Programmen wurde darauf geachtet, dass die Schwierigkeit und Komplexität der Aufgaben denen des Pretests entsprach, um eine Vergleichbarkeit zu ermöglichen. Dem folgten zwei Programmieraufgaben, die in ONYX erstellt und in OPAL eingebunden wurden. Auch hierbei sollten diese eine ähnliche Schwierigkeit und auch einen ähnlichen Aufgabentyp aufweisen wie die Aufgaben des Pretests. Beide Aufgaben sollen jeweils wieder in etwa 15 Minuten von den Studierenden bewältigt werden können.

In der abschließenden Klausur konnten die Probanden ihren einzigartigen Code eintragen, um zu überprüfen, ob die Syntaxaufgaben nicht nur einen Effekt auf die Programmverständnis- und Programmieraufgaben des Pre- und Posttests hatten, sondern auch auf die Leistungen in der Klausur. Die Betrachtung der Klausurergebnisse war eine gute Ergänzung, da die Studierenden deutlich komplexere Aufgaben lösen mussten als in den Aufgaben des hier beschriebenen Experiments. Daneben ermöglichte die Betrachtung der Klausur ein umfassenderes Bild als die in der Studie gewählten Programmverständnis- und Programmieraufgaben. Außerdem konnten Studierende, die nicht an dieser Studie teilgenommen haben, als Kontrollgruppe betrachtet werden.

5 Ablauf

Im folgenden Kapitel wird die Durchführung der vorliegenden Studie beschrieben. Dabei werden die einzelnen Schritte, die zur Ausführung durchgeführt werden mussten, erläutert. Außerdem werden die Teilnehmer der Studie und die Erhebung der Daten dargestellt.

Zu Beginn des Sommersemesters 2021 wurden die Studierenden im Kurs Datenstrukturen auf die Studie hingewiesen. Im Forum der OPAL-Gruppe des Kurses wurde von Frau Prof. Dr. Siegmund ein Beitrag erstellt:

> "Wie einige bereits in AuP erfahren haben, wird es auch in dieser Vorlesung eine empirische Studie geben. Anders als in AuP wird diese Studie das ganze Semester laufen. Für alle Studierende, die teilnehmen, wird es spezielle Programmieraufgaben geben, die als Ersatz für einige der Programmieraufgaben dienen. Dazu wollen wir kognitive Fähigkeiten mit standardisierten Tests erheben, von denen wir wissen, dass sie mit Programmieren zusammenhängen. Mit dieser Studie wollen wir herausfinden, ob durch spezielle Programmieraufgaben der Erfolg des Kurses positiv beeinflusst werden kann. Insgesamt wird dadurch der Arbeitsaufwand nicht beeinflusst. Für die Teilnahme bis zum Ende des Kurses bekommt jede/r Teilnehmer/in zwei Punkte für die Prüfungsvorleistung angerechnet. Selbstverständlich werden die Daten anonymisiert verarbeitet. Tragen Sie sich bei Interesse bitte in der Gruppe Studie ein, damit wir entsprechend planen können."

Für die Teilnahme an der Studie erhielten die Probanden zwei Punkte für die PVL sowie weniger komplexe Aufgaben in diesen. Außerdem wurde eine gesonderte Gruppe in OPAL erstellt, in der sich Interessenten eintragen konnten. Dies taten 100 Studierende. Als Nächstes folgt eine Beschreibung des Pretests im Abschnitt 5.1. Anschließend wird das Treatment, die Syntaxaufgaben, im Abschnitt 5.2 dargestellt. Dieses Kapitel endet im Abschnitt 5.3 mit der Erläuterung des Posttests.

5.1 Pretest

Der Pretest der Studie startete zeitgleich mit der dritten PVL des Kurses Datenstrukturen am 03.05.2021. Die Probanden hatten bis zum 17.05.2021 00:00 Uhr und damit 14 Tage Zeit am Pretest teilzunehmen. Zur Teilnahme mussten die Studierenden im OPAL-Datenstrukturen-Kurs den linken Baustein Studie auswählen und hier

den Punkt "Prüfungsvorleistung 3 - Studie" anklicken. Bei der Auswahl von diesem öffnete sich eine neue OPAL-Seite, welche aus einer Überschrift sowie einer kurzen Beschreibung, in der die Studierenden gebeten wurden, zuerst die SoSci Survey-Umfrage und danach die Programmieraufgaben durchzuführen, bestand. Unterhalb der Beschreibungen waren zwei Verlinkungen sowohl zur SoSci Survey-Umfrage als auch zu den Programmieraufgaben des Pretests. Wenn die Probanden nun auf die Verlinkung der SoSci Survey-Umfrage klickten, öffnete sich ein neues Browserfenster mit der Umfrage. Auf der ersten Seite wurde ihnen im oberen Teil ein einleitender Text zur Studie sowie im unteren Teil ein Video zur Einführung der Bedienung der Umfrage gezeigt, siehe Abbildung 5.1. Mit einem Klick auf "Weiter" in der unteren rechten Ecke des Fensters gelangten die Probanden zum Fragebogen, der demografische Eigenschaften abfragte. Der Fragebogen bestand aus vier Seiten und auch hierbei mussten die Probanden, die, im Unterabschnitt 4.3.2 genannten, obligatorischen Fragen beantworten und nachdem Sie viermal auf "Weiter" (siehe Abbildung 5.2) klickten, gelangten Sie zur ersten Programmverständnisaufgabe.

Abbildung 5.1: SoSci Survey Fragebogen - Einleitungsvideo

Nach dem Klick auf "Weiter" befanden sich die Teilnehmer dann auf dem ersten von zehn Quellcodestücken, die mithilfe des Remote Eye Trackers REyeker eingebunden wurden.

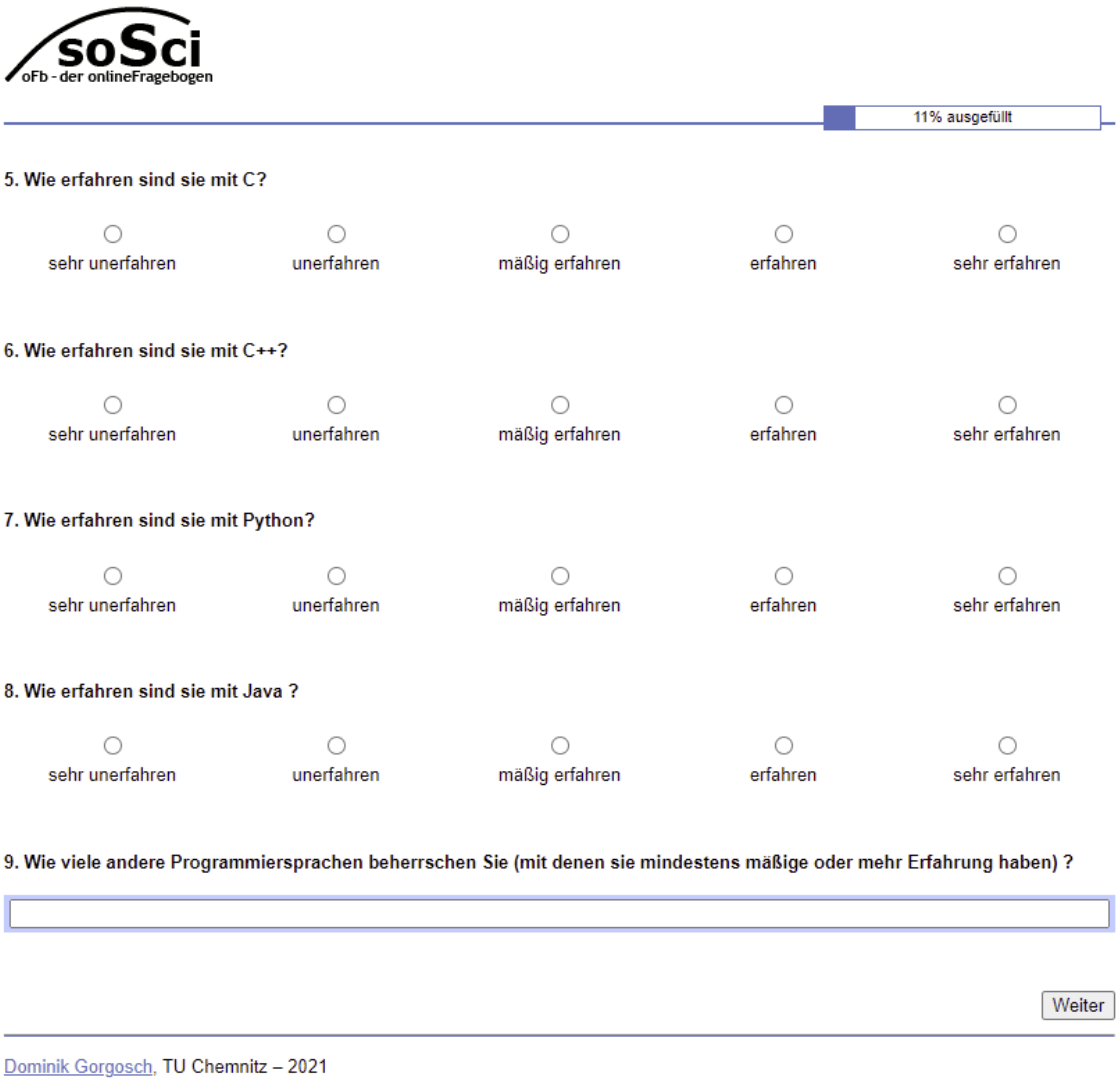


Abbildung 5.2: SoSci Survey Fragebogen - Angabe der Programmiererfahrung

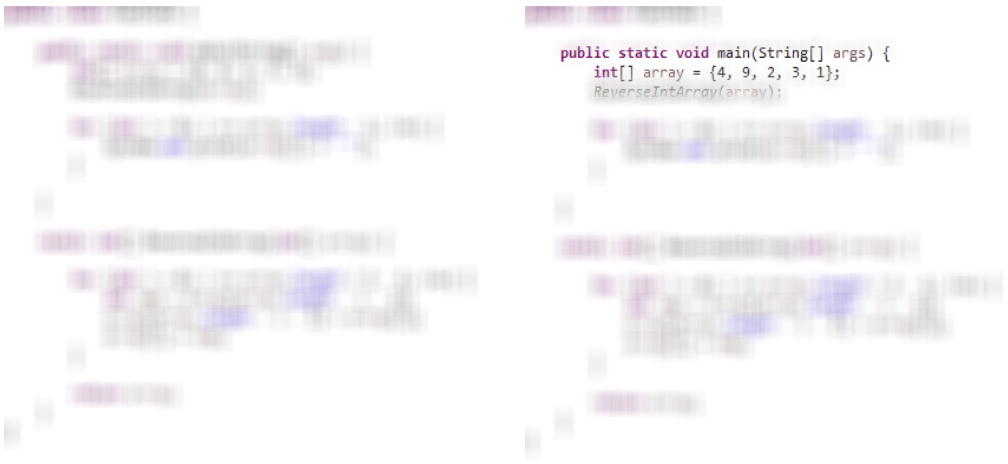
In der Abbildung 5.3 ist die erste Aufgabe der Programmverständnisaufgaben des Pretests zu sehen. Alle Seiten der Umfrage bestanden aus einem verschwommenen Quellcodestück, unterhalb dessen die Aufforderung "Geben Sie die Ausgabe des Code-Snippets ein:" und darunter ein Antwortfeld stand. In der unteren rechten Ecke des Browserfensters befand sich der "Weiter"-Button, mit dem man auf die nächste Seite gelang. In dieser Aufgabe mussten die Probanden die Ausgabe der von V1 eingeben. Dieser Methode werden zwei Strings übergeben und es wird die Anzahl der gleichen Buchstaben an gleichen Stellen ausgegeben. Der Quellcode ist durch die Einbindung REYekers verschwommen (siehe Abbildung 5.4(a)) und erst per Klick

Abbildung 5.3: SoSci Survey - Programmverständnisaufgabe V1

wird ein bestimmter Bereich, der vorher in REyeker eingestellt worden ist, sichtbar (siehe Abbildung 5.4 (b)). Durch das Aufdecken des verschwommenen Bereichs wird die Bewegung der Augen nachempfunden. Nachdem die Probanden sich so lange durch den Quellcode geklickt haben, bis sie die Antwort angeben können, mussten sie diese unterhalb des Quellcodes eingeben. Wenn die Teilnehmer eine Antwort eingegeben haben, können sie mithilfe des "Weiter"-Knopfs auf die nächste Seite gelangen.

Nach den Programmverständnisaufgaben gelangen die Probanden auf einen Fragebogen zur Schwierigkeit des Algorithmus sowie zu etwaigen Problemen (siehe Abbildung 5.5 und Unterabschnitt 4.3.2). Nach der Bearbeitung der zehn Programmverständnisaufgaben sowie der Beantwortung des Fragebogens mussten die Probanden noch ihren einzigartigen Code - Unique Code - erstellen. Danach hatten sie noch zwei Programmieraufgaben zu lösen.

Diese sind in ONYX bzw. OPAL erstellt worden. Die Studierenden mussten in



(a) (b)

Abbildung 5.4: REyeker - Einbindung V2: (a) Nutzersicht, wenn kein Bereich angeklickt wurde und (b) wenn ein Proband in das Bild geklickt hat

OPAL auf "Pre-Test" klicken. Bei einem Klick darauf öffnete sich eine neue Seite, die aus zwei Programmieraufgaben besteht. Die Probanden wurden aufgefordert den Unique Code aus SoSci Survey, als Kommentar hinzuzufügen. Nach der Bearbeitung von P1 wurde die Aufgabe mit einem Klick auf "Abgeben" in der linken unteren Ecke des Fensters gespeichert. Mit einem Klick auf "Weiter" in der rechten unteren Ecke gelangten die Studierenden zur zweiten und damit zur letzten Programmieraufgabe. Falls die Studierenden keine Veränderungen am vorgegebenen Quellcode vorgenommen haben, erhielten sie einen Hinweis - "Sie haben noch nicht alle Fragen auf dieser Seite beantwortet. Wenn Sie diesen Abschnitt nun auswerten lassen, können Sie Ihre Antworten auch nicht mehr ändern/ergänzen. Sind Sie sicher, dass Sie fortsetzen wollen?". Trotz des Hinweises konnten sie ihre Antwort abgeben, mit einem Klick auf den linken Knopf. Mit einem Klick auf den rechten Knopf konnten sie die Abgabe abbrechen und ihre Antwort eintragen. Falls sie ihre Aufgabe abgegeben haben, dann gelangen sie zu P2. Wie bei der ersten Aufgabe konnten die Studierenden in der linken unteren Ecke des Fensters die Aufgabe abgeben, jedoch war in der linken unteren Ecke zusätzlich ein Zurück-Button, mit dem sie wieder zur ersten Aufgabe springen konnten, in der rechten unteren Ecke konnte der Test abgeschlossen werden. Mit der Betätigung des "Test abschließen"-Knopfs erschien ein Hinweis, dass der Test damit beendet wird und ob der Teilnehmer den Test wirklich abgeben wolle. Nach der Abgabe konnte der Test nicht mehr bearbeitet werden und das Ergebnis war damit final. Die Auswertung erfolgte automatisiert durch Testfälle, die in ONYX implementiert werden konnten. Zusätzlich dazu gab es eine manuelle Prüfung vom Verfasser der Arbeit, da einige Aufgaben nicht kompilierbar waren, weil beispielsweise ein Semikolon am Ende einer Zeile fehlte oder es zu viele offene oder geschlossene Klammern gab.

Abbildung 5.5: SoSci Survey Fragebogen - Abfrage Schwierigkeit des Quellcodestücks

5.2 Syntaxaufgaben

Die fünfte PVL startete am 17.05.2021 und zeitgleich starteten auch die vier Syntaxaufgaben - if-else-Bedingungen sowie for-, while- und do-while-Schleifen. Die Studierenden hatten bis zum 31.05.2021 00:00 Uhr und damit 14 Tage Zeit daran teilzunehmen. Die Probanden mussten im OPAL-Kurs in der linken Spalte auf "Prüfungsvorleistung 5 - Studie" klicken und dort die vier verschiedenen Syntaxaufgaben aufrufen. Jeder einzelne Test wurde in ONYX bzw. OPAL erstellt. Zum Starten der Aufgaben mussten sie in der Mitte des Browserfensters auf "Test starten" klicken, danach öffnet sich in diesem Browserfenster der Test. Dieser wurde in zwei Sektionen "Organisatorisches" und "Aufgaben" unterteilt, denn die Aufgaben sollten jedes Mal randomisiert angezeigt werden, jedoch sollten die Probanden immer zuerst in der Sektion "Organisatorisches" ihren Unique Code eingeben. Deswegen wurde als Erstes diese Sektion aufgerufen, in der sie aufgefordert wurden, ihren Unique Code zur Verknüpfung der verschiedenen Studienbestandteile, einzugeben. Die zweite Sektion beinhaltet die verschiedenen Aufgaben des jeweiligen Tests. Bei der Auswahl dieser Sektion steht im oberen Teil des Bildschirmfensters eine Anleitung zur Bearbeitung der Aufgabe, dass die Studierenden syntaktische Fehler im

Quellcode verbessern sollen und dass in den Programmen immer geschweifte Klammern genutzt werden und nicht nur Einrückungen sowie ein vorgegebenes Beispiel für die Nutzung dieser Klammern. Zu den syntaktischen Fehlertypen zählten falsche Klammertypen, fehlende Semikolons, unterschiedlich viele geschlossene und offene Klammern oder falsche Initialisierungen der Schleifen sowie nicht deklarierte Variablen. Bei den if-else-Bedingungen gab es eine komplexere Aufgabe, deswegen bestand dieser Test aus insgesamt 23 Aufgaben. Hingegeben bestanden die Tests zu den verschiedenen Schleifenvariationen aus jeweils 25 Aufgaben. Nachdem die Studierenden ihren Unique Code eingegeben haben, gelangten sie mit einem Klick auf "Weiter" in der rechten unteren Ecke oder der Auswahl einer Aufgabe im linken Menü-Baustein zu den Syntaxaufgaben des jeweiligen Tests. In diesen wurden sehr kurze Java-Programme gezeigt, die einen oder zwei Syntaxfehler enthielten. Wenn die Probanden das Programm verbessert hatten, konnten sie unten links ihre "Antworten abgeben". Falls sie nichts am Quellcode geändert haben, erschien der Hinweis "Sie haben noch nicht alle Fragen auf dieser Seite beantwortet. Wenn Sie diesen Abschnitt nun auswerten lassen, können Sie Ihre Antworten auch nicht mehr ändern/ergänzen. Sind Sie sicher, dass Sie fortsetzen wollen?". Nachdem sie ihre Antwort abgegeben haben, können sie entweder über die Navigationsleiste im linken Bereich oder unten rechts auf "Weiter" klicken, um zur nächsten Aufgabe zu gelangen. Jeder dieser vier Tests beinhaltete 23-25 Aufgaben. Die Auswertung der Aufgaben erfolgte automatisch und die Studierenden bekamen nach der Abgabe jeder Aufgabe den korrekten Quellcode gezeigt. Leider war dieser nicht formatierbar, weil diese Option aktuell in ONYX/ OPAL nicht verfügbar ist. Dadurch fehlten Einrückungen im korrekten Quellcode. Beim Abschluss aller Aufgaben eines Tests wurde den Studierenden die erreichte Punktzahl sowie die benötigte Zeit für das Bearbeiten der Aufgaben angezeigt.

5.3 Posttest

Die neunte PVL startete am 14.06.2021, gleichzeitig mit dem Posttest der Studie. Die Studierenden hatten 14 Tage, bis zum 28.06.2021 00:00 Uhr, Zeit, am Posttest teilzunehmen. Um an den verschiedenen Aufgaben des Posttests teilzunehmen, mussten die Studierenden auf der Seite des OPAL-Kurses Datenstrukturen auf der linken Seite den Reiter "PVL 9 - Studie" aufrufen. Hierbei öffnete sich eine neue OPAL-Seite, die aus einer Überschrift sowie einer kurzen Beschreibung bestand. In dieser wurden die Studierenden gebeten, zuerst die SoSci Survey-Umfrage und danach die Programmieraufgaben durchzuführen. Unterhalb der Beschreibungen befanden sich zwei Verlinkungen sowohl zur SoSci Survey-Umfrage als auch zu den Programmieraufgaben des Posttests. Wie beim Pretest wurden die Probanden aufgefordert, als Erstes auf den Link zur SoSci Survey-Umfrage zu klicken. Bei der Auswahl öffnete sich dann ein neues Browserfenster mit dieser Umfrage. Auf der ersten Seite stand eine kurze Einleitung zum Posttest. Mit einem Klick auf "Weiter" in der unteren rechten Ecke des Fensters gelangen die Probanden zu den zehn Pro-

grammverständnisaufgaben. Die Codestücke wurden auch im Posttest per REyeker eingebunden. Mit einem Klick wird ein bestimmter Bereich des Quellcodes sichtbar, der ansonsten vollständig verschwommen ist. Die Teilnehmer müssen eine Antwort abgeben, falls sie dies nicht tun, erschien bei der Betätigung des "Weiter"-Knopfs in der rechten unteren Ecke ein Hinweis. Nach jedem der zehn Programme erschien eine Seite, die zwei Fragen enthält. Die erste Frage befasst sich mit der Schwierigkeit des Codestücks, welche auf derselben fünfstufigen Skala wie im Pretest bewertet werden sollte. Diese Frage war verpflichtend und beim Anklicken des "Weiter"-Knopfs ohne vorherige Auswahl eines Schwierigkeitsgrades, würde ein Hinweis erscheinen. Im Gegensatz dazu war die zweite Frage optional und befragte die Probanden nach möglichen Schwierigkeiten, die denen des Pretests entsprechen. Nach Beantwortung aller zehn Quellcodestücke sowie der dazugehörigen Frageseiten, mussten die Studierenden auf der vorletzten Seite ihren Unique Code eintragen. Auf der letzten Seite der SoSci Survey-Umfrage bedankte der Verfasser der Arbeit sich für die Mithilfe der Probanden und für die Mitteilung weiterer Anmerkungen teilte dieser seine Mailadresse mit. Damit waren die Programmverständnisaufgaben beendet und die Studierenden konnten das Browserfenster schließen.

Nach den Programmverständnisaufgaben folgten nun zwei Programmieraufgaben, welche in ONYX bzw. OPAL erstellt worden sind. Die Studierenden mussten im OPAL-Kurs Datenstrukturen zuerst den linken Baustein Studie und danach "Prüfungsvorleistung 9 - Studie" auswählen. Nun konnten sie entweder direkt den "Post-Test" auswählen oder in der Mitte des Browserfensters auf die Verlinkung zum Post-Test klicken. Dadurch öffnete sich ein neues OPAL-Overlay. In der Mitte von diesem mussten sie den Knopf "Test starten" auswählen und es öffnete sich wiederum ein neues Overlay. In diesem war der Test in zwei Sektionen unterteilt - Organisatorisches und Programmier-Aufgaben, da die Probanden zuerst ihren Unique Code eingeben sollten. Denn im Pretest hatten einige Studierenden ihren Unique Code nicht in den Quellcode als Kommentar hinzugefügt. Beim Öffnen der Sektion Organisatorisches wurde eine Aufforderung zur Eingabe des Unique Codes dargestellt, den die Studierenden in ein freies Textfeld eintippen sollten. Mit einem Klick auf "Antworten abgeben" in der unteren linken Ecke des Browserfensters wurde ihre Eingabe gespeichert. Zur Auswahl der Programmieraufgaben konnten die Studierenden entweder in der rechten unteren Ecke auf "Weiter" klicken oder im linken Reiter eine der beiden Programmieraufgaben auswählen. Die Aufgaben bestanden aus den Aufgaben P3 und P4.

Beim Öffnen der Aufgabe P3 wurde eine Beschreibung der Aufgabe angezeigt. Die Probanden mussten die Methode vervollständigen. Die Auswertung erfolgte automatisiert anhand von fünf Testfällen. Zur Abgabe dieser Aufgabe mussten die Studierenden in der unteren linken Ecke "Antworten abgeben" anklicken und entweder über den linken Reiter oder per Klick auf "Weiter" in der rechten unteren Ecke die nächste Programmieraufgabe String-Wiederholung auswählen. Alle Abgaben wurden von dem Verfasse der Arbeit noch einmal manuell geprüft, um eventuelle Problematiken der automatisierten Auswertung zu verhindern. Entsprechend der Aufgabe P3 mussten die Probanden auch bei P4 zur Lösung der Aufgabe

die Methode mit Quellcode füllen. Diese Aufgabe musste wieder per "Antworten abgeben" in der linken unteren Ecke abgegeben werden und zur Beendigung des Posttests, und damit des letzten Schrittes der Studie, mussten die Studierenden in der unteren rechten Ecke "Test abschließen" auswählen.

Im nächsten Kapitel werden die Ergebnisse der Studie dargestellt.

6 Ergebnisse

In diesem Kapitel werden die Ergebnisse des Experiments beschrieben. Jedoch werden sie nicht interpretiert, um die Ergebnisse so objektiv wie möglich darzustellen. Die Datenvorbearbeitung wird im Abschnitt 6.1 besprochen. Im Abschnitt 6.2 werden die deskriptiven Statistiken beschrieben. Dem folgt das Testen der Hypothesen im Abschnitt 6.3.

6.1 Vorbereitung und Datenvorbearbeitung

Im ersten Schritt der Vorbereitung wurde kontrolliert, wie viele Probanden an allen acht Aufgaben – Pretest (Programmier- und Programmverständnisaufgaben), vier Syntax-Aufgaben (if-else-Bedingungen, for-, while-, do-while-Schleifen) und Posttest (Programmier- und Programmverständnisaufgaben) – teilgenommen haben. Von ursprünglich 52 PA, die am Pretest teilnahmen, haben 21 das gesamte Experiment beendet. Aufgrund des festgelegten Ziels den Effekt von Syntaxaufgaben auf PA zu messen, wurden daraufhin nur deren Ergebnisse ausgewertet.

Die Programmmverständnisaufgaben mussten zur weiteren Analyse vorbearbeitet werden. Der erste Schritt war das Herunterladen der Daten des Pre- und Posttests aus SoSci Survey im xlsx-Format. Beide Tabellen beinhalteten aber auch für die Studie irrelevante Spalten, wie etwa die Bildschirmgröße, die Browserversion oder das Betriebssystem. Daneben bestanden die relevanten Variablen aus schlecht verständlichen Buchstaben- und Zahlenkombinationen. Mithilfe der Python Bibliothek Pandas wurden die Daten bearbeitet. Die Tabelle musste eingelesen werden, wobei zuerst die irrelevanten Spalten entfernt und die schlecht lesbaren Variablennamen durch besser lesbare ersetzt wurden, bspw. PD14_01 zu Unique Code. Nach diesem ersten Schritt wurden in den beiden Excel-Dateien Probanden entfernt, die nicht an allen Teilen der Studie teilgenommen hatten. Da SoSci Survey die Beantwortungszeit automatisch gespeichert hat, bedurfte dies keiner weiteren Vorbearbeitung. Bei der Bewertung der Korrektheit der Programmverständnisaufgaben wurden auch falsch formatierte Antworten als richtig akzeptiert. Bspw. wurde bei V5 sowohl 1 3 4 7 9 als auch 1, 3, 4, 7, 9 als korrekt gewertet. Zudem mussten Ausreißer identifiziert und entfernt werden. Antworten wurden als Ausreißer klassifiziert, wenn die Beantwortungszeit höher war als die der dritten Quartile addiert mit dem 1,5-fachen Interquartilsabstand oder wenn sie niedriger war als die der ersten Quartile subtrahiert durch den 1,5-fachen Interquartilsabstand und wenn die Korrektheit der Antwort falsch war. Wenn ein Proband häufiger als zweimal als Ausreißer in einem Test deklariert worden war, wurde er aus dem Datensatz entfernt. Es gab zwar Aus-

reißer, wobei keiner der Probanden mehr als zwei Ausreißer hatte. Dementsprechend wurde niemand entfernt. Jedoch hatte ein Proband ein technisches Problem bei V3, weil der Quellcode nicht geladen worden ist. Dadurch wurde der Proband, weil es sich um ein within-subject-Design handelt, sowohl im Pre- als auch im Posttest bei V3 entfernt.

Demgegenüber benötigten die REyeker-Daten für die Heatmaps keine gesonderte Vorbearbeitung, jedoch mussten die AOI definiert werden, indem mithilfe einer Funktion der REyeker-Analyse-Bibliothek die Bilder der Quellcodestücke in ein 2D-Array umgewandelt worden sind. Dadurch konnte anhand von x-y-Koordinaten die genaue Position der AOI bestimmt werden.

Die Datenvorbearbeitung der Beantwortungszeit der Programmieraufgaben benötigte keine Vorbearbeitung, da in OPAL die Zeit automatisch gemessen worden ist. Deswegen begann die Vorbearbeitung mit dem Aussortieren der Probanden, welche nicht an allen acht Aufgaben teilgenommen haben. Darauf folgte eine manuelle Prüfung der eingereichten Antworten und damit der Korrektheit. Zur Messung der Ergebnisse wurde die Anzahl und Art der Fehler gezählt, welche die Probanden machten. Obwohl pro Aufgabe fünf Testfälle implementiert waren, könnten Studierende bereits beim Vergessen eines Semikolons oder einem ähnlichen syntaktischen Fehler keinen dieser Testfälle bestehen. Dadurch würden sie die volle Fehleranzahl erhalten. Damit dies nicht geschieht, wurden die abgegebenen Aufgaben manuell ausgewertet. Dafür wurden die Testfälle in der Eclipse IDE implementiert und der abgegebene Quellcode manuell gedebuggt. Dabei wurde die Anzahl der Fehler gezählt und diese in logische und syntaktische Fehler unterteilt.

Nach der Datenvorbearbeitung folgen nun die deskriptiven Statistiken der Teilnehmer und der verschiedenen Aufgabentypen.

6.2 Deskriptive Statistik

Im folgenden Unterabschnitt werden zunächst die Probanden beschrieben. Darauf folgen die deskriptiven Statistiken bei den Programmverständnisaufgaben. Schlussendlich werden die Programmieraufgaben dargestellt.

6.2.1 Beschreibung der Probanden

In der Tabelle 6.1 wird die gesamte Testgruppe dargestellt. Der obere Teil der Tabelle zeigt das Geschlecht, Alter und Fachsemester der Studierenden an. Im unteren Teil wird die Programmiererfahrung in Jahren in den Bereichen Programmieren lernen, Beruflich programmieren und in der Programmiersprache Java präsentiert. Außerdem wird die Einschätzung ihrer Programmierfähigkeiten gegenüber ihren Kommilitonen dargestellt.

An dem Experiment haben insgesamt 21 Studierende teilgenommen, von denen 16 männlich und fünf weiblich waren. Die Probanden teilen sich in 16 Bachelorstudierende und fünf Masterstudierende auf. Der Kurs Datenstrukturen sollte

Demografische Daten	
Männlich	16
Weiblich	5
Alter	21.38 ± 3.76
Fachsemester	2.48 ± 1.17
Programmieren lernen	2.81 ± 1.99
Beruflich programmieren	0.43 ± 0.93
Java	2.76 ± 1.22
Einschätzung ihrer Fähigkeiten gegenüber Kommilitonen	3.0 ± 0.89

Tabelle 6.1: Demografische Daten der Probanden

in der Regel im zweiten Fachsemester besucht werden. Der Mittelwert der Probanden betrug 2,48 Semester, mit einer Standardabweichung von 1,17 Semestern.

Die Probanden haben durchschnittlich bereits 2,81 Jahre Programmieren gelernt, bei einer Standardabweichung von 1,99 Jahren. Hingegen beträgt die berufliche Praxis nur 0,43 Jahre, mit einer Standardabweichung von 0,93 Jahren. Außerdem programmieren die Probanden seit 2,76 Jahren in Java, bei einer Standardabweichung von 1,22 Jahren. Die Einschätzung ihrer Fähigkeit gegenüber Kommilitonen ist mit 3,0 auf der mittleren Ebene der Skala, damit schätzen sich die Probanden ähnlich kompetent ein wie ihre Kommilitonen. Beachtenswert ist, dass keiner der Probanden seine Fähigkeiten geringer als 3 eingeschätzt hat. Außerdem sei darauf hinzuweisen, dass keiner der Probanden weniger als zwei Jahre Erfahrung in mindestens einer der Programmiersprachen besitzt.

6.2.2 Programmverständnisaufgaben

Als Nächstes folgen die Antworten der Probanden sowohl im Pretest als auch im Posttest in den Programmverständnisaufgaben. Die Antworten und Zeitstempel der Programmverständnisaufgaben wurden in SoSci Survey gespeichert. Zudem wurde zur Auswertung der Augenbewegungen REyeker eingesetzt und in SoSci Survey eingebunden. Dafür musste eine Variable in SoSci Survey erstellt werden, die zur Speicherung der Klickvektoren eingesetzt wurde. Bei der Betrachtung der Programmverständnisaufgaben werden nur die gleichen Aufgaben V1 bis V5 angezeigt sowie eine Darstellung der gesamten Ergebnisse von V1 bis V5 zusammengerechnet. Ein Vergleich verschiedener Algorithmen gestaltet sich als schwierig, weswegen diese nicht betrachtet werden und als Distraktoren bei den Programmverständnisaufgaben gesehen werden können. Durch einen zeitlichen Abstand von mindestens vier Wochen, wie in Kapitel 5 dargestellt, kann davon ausgegangen werden, dass die Probanden sich nicht mehr an die Aufgaben des Pretests erinnern konnten.

Die Beantwortungszeit bei den Programmverständnisaufgaben ist der Tabelle 6.2 und der Abbildung 6.1 zu entnehmen. Bei V1 betrug die durchschnittliche Beant-

Aufgabe	Test	Anzahl der Antworten	Korrektheit in %	Mittelwert Beantwortungszeit [s]	Standardabweichung Beantwortungszeit [s]	Mittelwert Klickanzahl AOI	Standard-Abweichung Klickanzahl AOI
V1	Pretest	21	85,71%	121	39	18	12
	Posttest	21	85,71%	74	29	11	6
V2	Pretest	21	80,95%	121	65	13	13
	Posttest	21	90,48%	72	33	9	7
V3	Pretest	20	75,00%	187	136	45	37
	Posttest	20	70,00%	86	57	29	20
V4	Pretest	21	52,38%	126	44	10	7
	Posttest	21	42,86%	112	47	9	5
V5	Pretest	21	90,48%	119	115	21	17
	Posttest	21	100,00%	88	70	17	17
V1-V5	Pretest	21	84,17%	135	91	21	23
	Posttest	21	80,33%	86	51	15	14

Tabelle 6.2: Programmverständnisaufgaben im Pre- und Posttest, ihre entsprechende Korrektheit in %, der Mittelwert und die Standardabweichung ihrer Beantwortungszeit sowie der Mittelwert und die Standardabweichung bei der Klickanzahl in AOI pro Proband

wortungszeit im Pretest 121 Sekunden, im Posttest 74 ± 29 Sekunden. Im Pretest wurde V2 in 121 ± 39 Sekunden beantwortet, im Posttest benötigten die Probanden nur 72 ± 33 Sekunden. V3 wurde innerhalb von 187 ± 136 Sekunden im Pre- und 86 ± 57 Sekunden im Posttest bearbeitet. Die Beantwortungszeit bei V4 betrug im Pretest 126 ± 44 Sekunden und im Posttest 112 ± 47 Sekunden. Schlussendlich wurde V5 im Posttest in 119±115 Sekunden beantwortet und im Posttest innerhalb 88 ± 70 Sekunden. Insgesamt benötigten die Probanden für die Beantwortung derselben Algorithmen im Pretest 135 ± 91 Sekunden und im Posttest 86 ± 51 Sekunden.

In der Abbildung 6.2 wird die Korrektheit der Algorithmen, die sowohl im Pre- als auch im Posttest genutzt wurden, verglichen. Bei V1 – gab es mit 85,71% das gleiche Ergebnis. Demgegenüber hat sich das Ergebnis bei V2 verbessert, da im Pretest die Korrektheit 80,95% und im Posttest 90,48% betrug. Bei V3 gab es einen marginalen Unterschied in der Korrektheit, im Pretest waren die Probanden mit 75% marginal besser als im Posttest mit 70%. V4 wurde im Pretest von 52,38% positiv beantwortet, im Posttest von 42,86%. Schlussendlich fiel im Pretest V5 mit 90,48% schlechter aus als im Posttest mit 100%. Wenn die Ergebnisse aller fünf Algorithmen zusammengerechnet werden, wird deutlich, dass die Unterschiede nur marginal sind, denn im Pretest haben 76,9% der Probanden die Programmverständnisaufgaben derselben Algorithmen korrekt beantwortet, im Posttest 77,81%.

Neben der Korrektheit und der Beantwortungszeit soll auch noch die Fixierungsanzahl pro Quellcodestück in den AOI untersucht werden. Die AOI sind die in den Syntaxaufgaben behandelten Programmierkonstrukte. Zur Messung der Klickan-

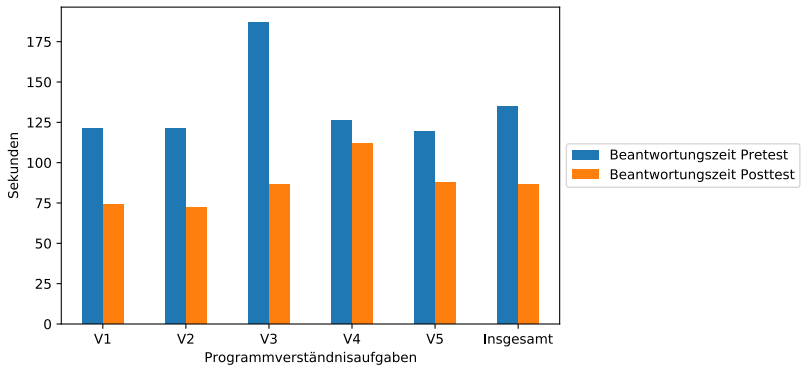


Abbildung 6.1: Programmverständnisaufgaben - Durchschnittliche Beantwortungszeit

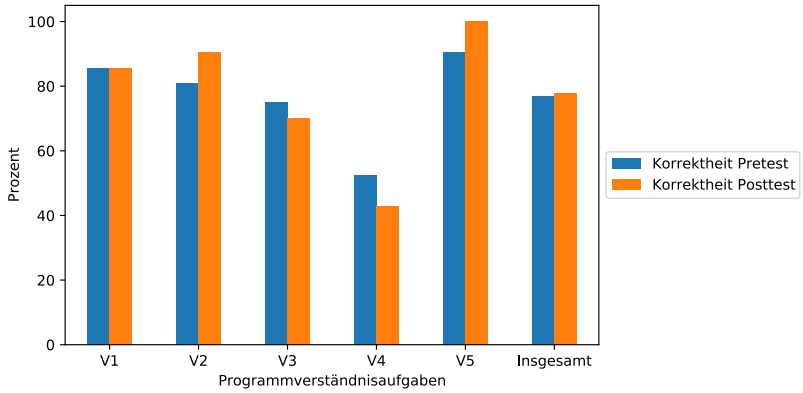


Abbildung 6.2: Programmverständnisaufgaben - Durchschnittliche Korrektheit

zahl innerhalb dieser Konstrukte wurde mithilfe der REyeker-Daten die Anzahl der Klicks bestimmt. In REyeker wird der Klick als x-y-Koordinate auf dem Quellcode dargestellt. Jeder Klick, der sich im Bereich eines der getesteten Programmierkonstrukte befand, wurde als Klick in eine AOI gewertet. Zudem musste durch die REyeker-Einstellungen ein Toleranzbereich definiert werden, der dem des REyeker-Radius entsprach. In der Abbildung 6.3 werden die durchschnittlichen Fixierungsanzahlen pro Algorithmus des Pre- und Posttests dargestellt. Bei denselben Algorithmen wurde bei jeder Aufgabe die durchschnittliche Fixierungsanzahl gesenkt, jedoch gibt es Unterschiede zwischen den Algorithmen. Bei V1 gab es eine Senkung von 18 $\pm$ zwölf auf elf $\pm$ sechs Klicks in den AOI. Im Vergleich dazu fiel die Senkung von V2 mit 13 $\pm$ 13 auf neun $\pm$ sieben Klicks niedriger aus. Hingegen gab es bei V3 einen größeren Unterschied, mit einer Senkung von 45 $\pm$ 37 auf 29 $\pm$ 20 Klicks. V4 war sowohl im Pre- als auch im Posttest die Aufgabe mit den wenigsten Klicks. Der Unterschied fällt nur marginal aus, mit einer Senkung von zehn $\pm$ sieben auf neun $\pm$ fünf. Schlussendlich ist die Reduzierung der Klickanzahl bei V5 von 21 $\pm$ 17 auf 17 $\pm$ 17 Klicks etwas deutlicher als bei V4. Insgesamt hat sich der Mittelwert der Klickanzahl ist bei den Aufgaben V1-V5 von 21 $\pm$ 23 auf 15 $\pm$ 14 gefallen.

Bei der Betrachtung der Heatmaps wurden Unterschiede in den verschiedenen

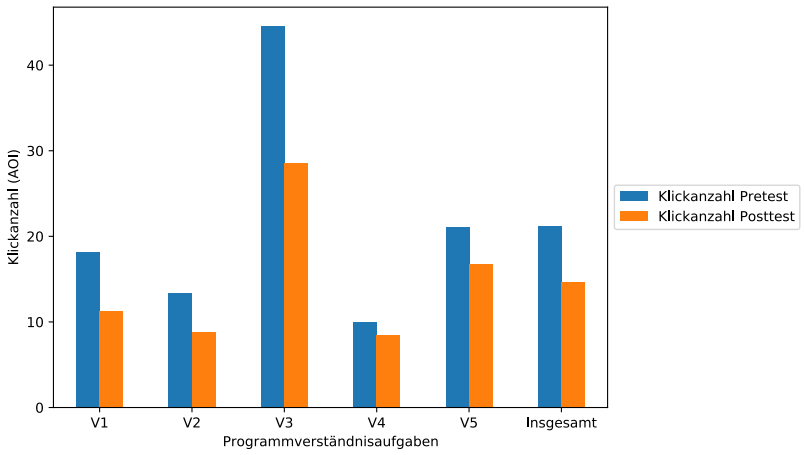


Abbildung 6.3: Programmverständnisaufgaben - Durchschnittliche Klickanzahl in AOI

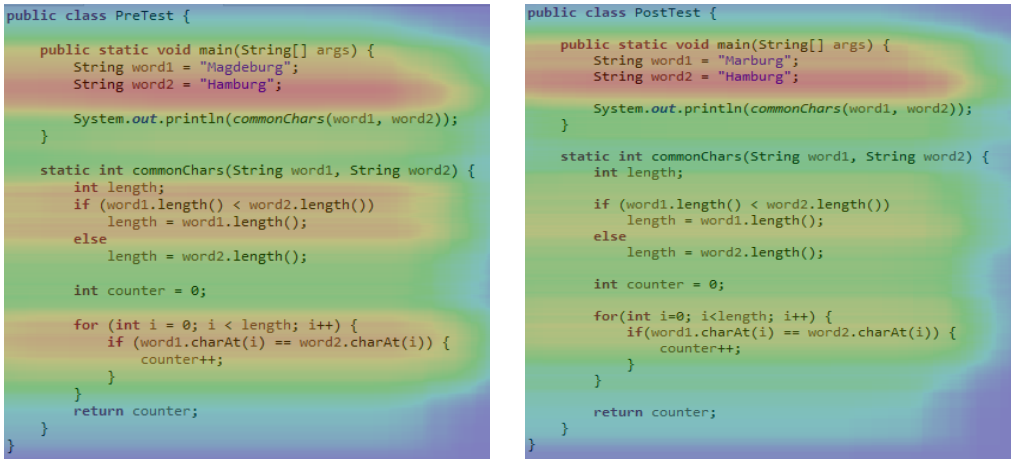
Algorithmen identifiziert. Es werden die Programmverständnisaufgaben V1 und V3 sowohl im Pre- als auch im Posttest präsentiert. Die Aufgaben V2, V4 und V5 befinden sich zwischen diesen beiden Heatmaps, welche die Extremwerte abbilden. Der Unterschied bei der Aufgabe V1 ist deutlich ersichtlich. In der Abbildung 6.4(a) haben die Probanden wesentlich häufiger in die wichtigen Schritte des Algorithmus geklickt, welche sich in der if-else-bedingung und der for-Schleife befanden. Im Posttest (siehe Abbildung 6.4(b)) haben die Studierenden deutlich seltener in die if-else-Bedingung und in die for-Schleife geklickt. Hingegen wurde im Posttest deutlich häufiger in die main-Methode geklickt. Diese enthält zwei Strings, die der Methode als Parameter übergeben werden sowie den Methodenaufruf.

Im Gegensatz dazu ist der Unterschied bei V3 zwischen den Heatmaps in der Abbildung 6.5 nur minimal. Im Posttest wurde prozentual minimal häufiger in den Methodenaufruf, das Eingabe-Array und das gesuchte Wort geklickt. In der Methode und damit dem eigentlichen Algorithmus gibt es kaum einen Unterschied. Dieselben Zeilen, die das String-Array durchlaufen und dessen Elemente mit dem gesuchten Wort vergleichen, wurden ähnlich häufig angeklickt.

In der Abbildung 6.6 werden Boxplots für die Beantwortungszeit (a) und Klickanzahl in den AOI (b) dargestellt. Für die Korrektheit ist keine Darstellung möglich, da es hierbei nur die Werte wahr oder falsch gibt. Bei der Beantwortungszeit betrug der Median im Pretest 121, im Posttest sank dieser auf 77. Hingegen fiel bei der Klickanzahl der Median von 15 im Pretest auf 10 im Posttest.

6.2.3 Programmieraufgaben

In OPAL ist es nur möglich, die Beantwortungszeit eines Tests und damit beider Programmieraufgaben anzeigen und ausgeben zu lassen. Deswegen können die Aufgaben nicht einzeln analysiert werden. In der Tabelle 6.3 und der Abbildung 6.7 werden die Beantwortungszeiten der Programmieraufgaben des Pre- und Posttests



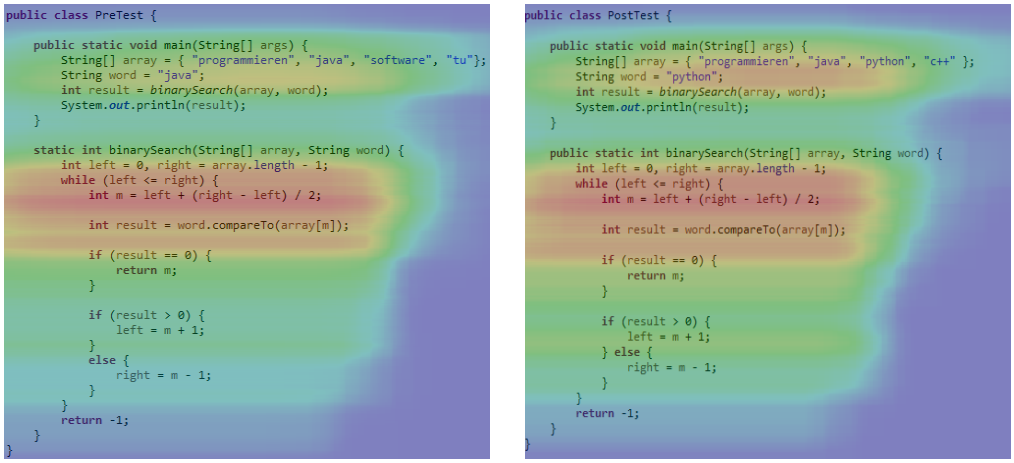
```java
public class PreTest {

    public static void main(String[] args) {
        String word1 = "Magdeburg";
        String word2 = "Hamburg";

        System.out.println(commonChars(word1, word2));
    }

    static int commonChars(String word1, String word2) {
        int length;
        if (word1.length() < word2.length())
            length = word1.length();
        else
            length = word2.length();

        int counter = 0;

        for (int i = 0; i < length; i++) {
            if (word1.charAt(i) == word2.charAt(i)) {
                counter++;
            }
        }
        return counter;
    }
}
```

(a)

```java
public class PostTest {

    public static void main(String[] args) {
        String word1 = "Marburg";
        String word2 = "Hamburg";

        System.out.println(commonChars(word1, word2));
    }

    static int commonChars(String word1, String word2) {
        int length;

        if (word1.length() < word2.length())
            length = word1.length();
        else
            length = word2.length();

        int counter = 0;

        for(int i=0; i<length; i++) {
            if(word1.charAt(i) == word2.charAt(i)) {
                counter++;
            }
        }

        return counter;
    }
}
```

(b)

Abbildung 6.4: Programmverständnisaufgaben - Durchschnittliche Heatmaps von V1 im Pretest (a) und Posttest (b)



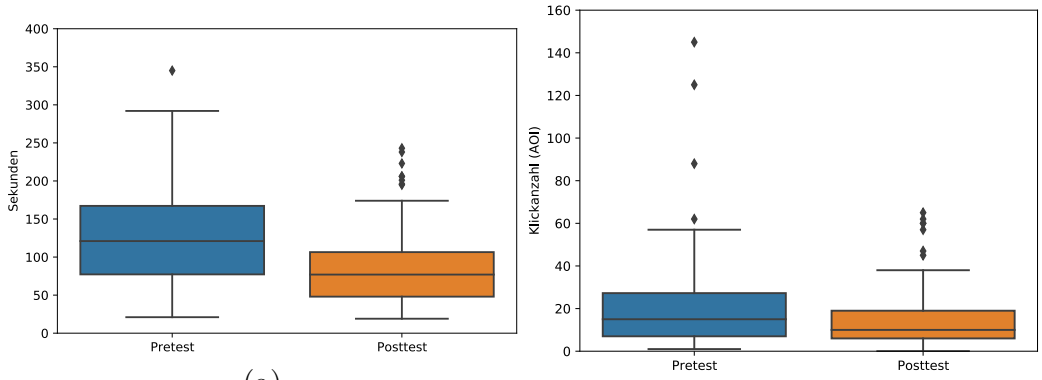
```java
public class PreTest {

    public static void main(String[] args) {
        String[] array = { "programmieren", "java", "software", "tu"};
        String word = "java";
        int result = binarySearch(array, word);
        System.out.println(result);
    }

    static int binarySearch(String[] array, String word) {
        int left = 0, right = array.length - 1;
        while (left <= right) {
            int m = left + (right - left) / 2;

            int result = word.compareTo(array[m]);

            if (result == 0) {
                return m;
            }

            if (result > 0) {
                left = m + 1;
            }
            else {
                right = m - 1;
            }
        }
        return -1;
    }
}
```

(a)

```java
public class PostTest {

    public static void main(String[] args) {
        String[] array = { "programmieren", "java", "python", "c++" };
        String word = "python";
        int result = binarySearch(array, word);
        System.out.println(result);
    }

    public static int binarySearch(String[] array, String word) {
        int left = 0, right = array.length - 1;
        while (left <= right) {
            int m = left + (right - left) / 2;

            int result = word.compareTo(array[m]);

            if (result == 0) {
                return m;
            }

            if (result > 0) {
                left = m + 1;
            } else {
                right = m - 1;
            }
        }
        return -1;
    }
}
```

(b)

Abbildung 6.5: Programmverständnisaufgaben - Durchschnittliche Heatmaps von V3 im Pretest (a) und Posttest (b)



(a) (b)

Abbildung 6.6: Programmverständnisaufgaben - Boxplots: (a) Beantwortungszeit und (b) Klickanzahl in den AOI

Aufgabe	Anzahl der Antworten	Mittelwert Anzahl der Fehler	Standard-Abweichung Anzahl der Fehler	Mittelwert Beantwort-ungszeit [hh:mm:ss]	Standard-abweichung Beantwortungs-zeit [hh:mm:ss]
P1	21	1,52	1,63		
P2	21	1,76	1,84		
P3	21	0,29	0,64		
P4	21	0,86	0,91		
P1+P2	21	3,29	2,97	00:48:17	00:41:28
P3+P4	21	1,14	1,15	00:18:36	00:07:41

Tabelle 6.3: Programmieraufgaben im Pre- und Posttest, ihre entsprechende Korrektheit gemessen am Mittelwert und der Standardabweichung in der Anzahl an Fehlern, der Mittelwert und die Standardabweichung ihrer Beantwortungszeit pro Proband

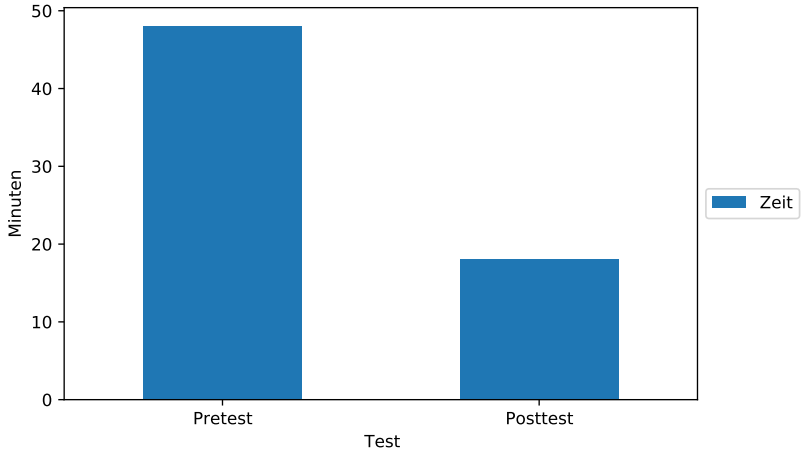


Abbildung 6.7: Programmieraufgaben - Durchschnittliche Beantwortungszeit

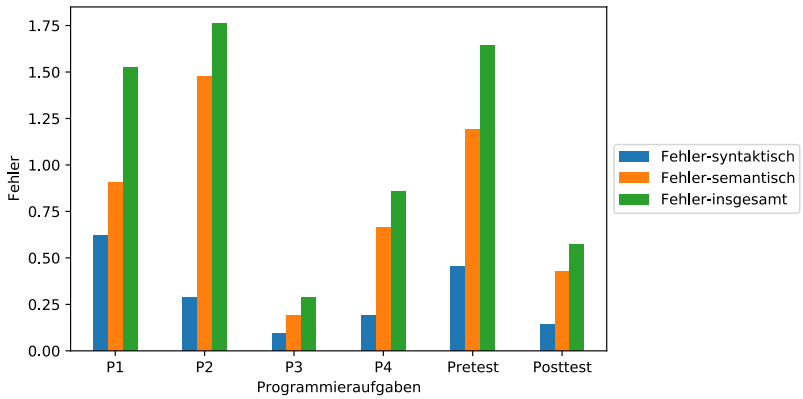
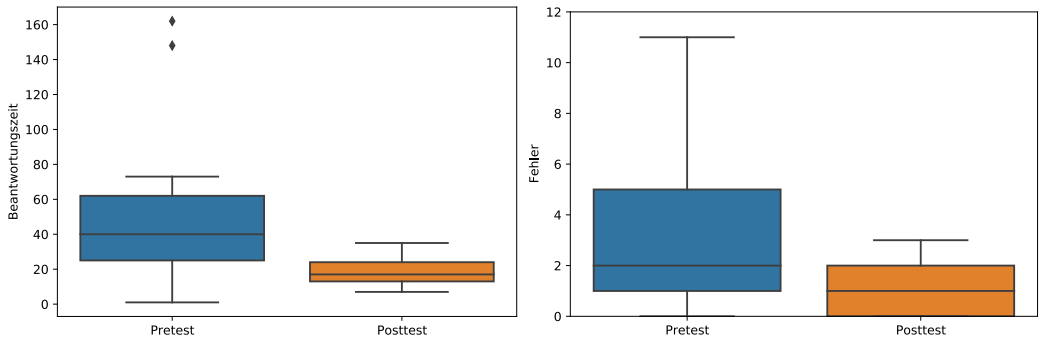


Abbildung 6.8: Programmieraufgaben - Durchschnittliche Fehleranzahl

in Minuten dargestellt. Für die Beantwortung der Pretest-Programmieraufgaben benötigten PA durchschnittlich 48 Minuten und 17 Sekunden bei einer Standardabweichung von 41 Minuten und 28 Sekunden. Jedoch gab es einige Probanden, die weit über eine Stunde gebraucht habe und wieder andere haben die beiden Programme in unter einer Minute abgegeben. Der schnellste PA benötigte im Pretest 57 Sekunden, hingegen der langsamste 161 Minuten und 33 Sekunden. Im Gegensatz dazu benötigten die Studierenden für die Beantwortung der Programmieraufgaben des Posttests durchschnittlich 18 Minuten und 36 Sekunden (Standardabweichung: Sieben Minuten und 41 Sekunden). Im Posttest benötigte der langsamste Proband 34 Minuten und 54 Sekunden, der schnellste sechs Minuten und 31 Sekunden.

Neben der Beantwortungszeit wurde bei den Programmieraufgaben auch die Korrektheit, anhand der Anzahl an Fehlern pro Programmieraufgabe gemessen (siehe Tabelle 6.3 und Abbildung 6.8). Die Höchstzahl an Fehlerpunkten pro Fehlerart - syntaktisch und semantisch - beträgt fünf. Die beiden Programmieraufgaben des Pretests waren P1 und P2, die des Posttests P3 und P4. In der Aufgabe P1 machten die Probanden durchschnittlich 1,52 ± 1,63 Fehler, davon entfielen 0,62 auf syntak-



(a) (b)

Abbildung 6.9: Programmieraufgaben - Boxplots: (a) Beantwortungszeit und (b) Anzahl der Fehler

tische Fehler und 0,9 auf semantische Fehler. Hingegen machten die Probanden bei P2 durchschnittlich 0,29 syntaktische und 1,48 semantische Fehler, somit insgesamt 1,76 ± 1,83 Fehler. Im Vergleich dazu machten die Probanden im Posttest deutlich weniger Fehler. Eine mögliche Erklärung dafür wird im Unterabschnitt 7.1.5.3 erläutert. Bei P3 machten PA durchschnittlich nur 0,29 ± 0,64 Fehler, von denen 0,1 auf syntaktische und 0,19 auf semantische Fehler entfielen. Demgegenüber machten Probanden bei P4 0,86 ± 0,91 Fehler - 0,19 syntaktische und 0,67 semantische Fehler.

Sowohl bei der Array- als auch bei der Stringmanipulationsaufgabe sind die Fehlerquoten stark gesunken. Bei der Stringmanipulationsaufgabe wurden im Posttest 80,92% weniger Fehler als im Pretest gemacht. Bei der Arrayaufgabe ist die Fehleranzahl auf 76,11% gesunken. Bei einer genaueren Betrachtung syntaktischer und semantischer Fehler fällt auf, dass zwischen P1 und P3 die Anzahl syntaktischer Fehler um 83,87% gesunken ist, die semantischen Fehler um 78,89%. Zwischen den Aufgaben P2 und P4 war der Unterschied geringer, weil syntaktische Fehler um 34,48% verringert werden konnten. Jedoch war der Unterschied bei semantischen Fehlern deutlicher, da diese um mehr als die Hälfte - 54,73% - reduziert werden konnten. Bei der Betrachtung der Aufgaben des Pretests im Vergleich zu den Aufgaben des Posttests wurden im Pretest insgesamt 3,29 Fehler und im Posttest 1,14 Fehler gemacht. Besonders auf der syntaktischen Ebene konnten die Kompetenzen verbessert werden, da diese durchschnittlich von 0,9 Fehlern auf 0,29 Fehler gefallen sind. Auch die Anzahl semantischer Fehler konnte von 2,38 auf 0,86 mehr als halbiert werden.

In der Abbildung 6.9 sind die Boxplots für die Beantwortungszeit und Anzahl der Fehler der Programmieraufgaben visualisiert. Der Median im Pretest betrug für die Beantwortungszeit 40 Minuten, im Posttest sank dieser auf 17 Minuten. Bei den Fehlern lag der Median im Pretest bei zwei und konnte im Posttest auf einen verringert werden.

6.2.4 Klausurergebnisse

Neben den Programmverständnis- und Programmieraufgaben wurde auch der Einfluss auf das Klausurergebnis untersucht. An der Klausur haben insgesamt 102 Studierende teilgenommen. Von 21 Probanden haben zwei nicht an der Klausur teilgenommen. Damit haben 19 der 102 Studierendem an dem Experiment partizipiert und 83 taten dies nicht. Alle 19 Probanden haben die Klausur bestanden. Hingegen fielen 15 der 83 Studierenden durch die Klausur. Dies entspricht 18,07%. Außerdem hatten die Probanden ein besseres Ergebnis, da sie von möglichen 100 Punkten durchschnittlich 76,42 Punkte erreichten, bei einer Standardabweichung von 12,93 Punkten. Die Kontrollgruppe hat mit durchschnittlich 64,51 Punkten und einer Standardabweichung von 18,59 Punkten schlechter abgeschnitten. In der Abbildung 6.10 wird der Boxplot des Klausurergebnisses dargestellt. Die Testgruppe stellt dabei die Probanden der Studie dar, alle anderen stellen die Kontrollgruppe dar. Der Median der Probanden im Klausurergebnis beträgt 75,5 Punkte. Im Gegensatz dazu steht der Median der restlichen Studierenden mit 66,5 Punkten. Auffallend sind die Ausreißer nach unten in der Kontrollgruppe. Dem Gegenüber hat selbst der schlechteste Proband mit 54,5 Punkten mehr als die Hälfte der Punkte in der Klausur erreicht.

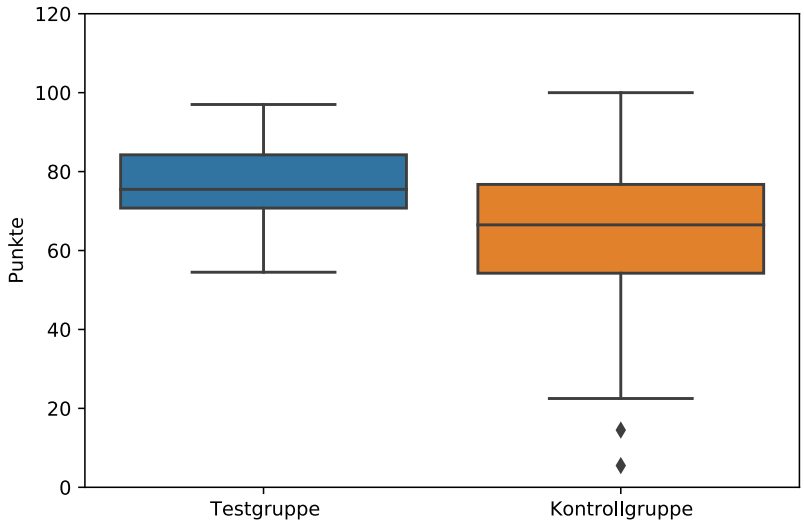


Abbildung 6.10: Klausur - Boxplot erreichter Punkte

6.3 Inferenzstatistik

Zum Testen der Hypothesen müssen diese zunächst in statistische Hypothesen umgewandelt werden. Im folgenden Abschnitt werden nur die Ergebnisse genannt. Die Interpretation der Ergebnisse erfolgt im Kapitel 7.

Programm-verständnis-aufgaben	Mittelwert Beantwortungszeit [s]		Statistische Tests	
	Pretest	Posttest	Wilcoxon	Cliff's-Delta
V1-V5	135	86	W: 746,5 | p: <0,001	0,41

Tabelle 6.4: Programmverständnisaufgaben - Beantwortungszeit

6.3.1 Programmverständnisaufgaben

In den folgenden Unterabschnitten werden die Hypothesen der Programmverständnisaufgaben getestet. Zuerst wird die Hypothese bezüglich der Beantwortungszeit betrachtet. Danach folgt die Korrektheit. Schlussendlich wird getestet, ob es einen signifikanten Unterschied in der Klickanzahl in den AOI gibt.

6.3.1.1 Beantwortungszeit

Die Hypothese lautet:

> H_1 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Beantwortungszeit bei den Programmverständnisaufgaben.

Die statistische Nullhypothese ist:

> H_{01} Bei den Programmverständnisaufgaben V1 bis V5 gibt es durch die Teilnahme an den Syntaxaufgaben keinen signifikanten Unterschied in der Beantwortungszeit.

Die Tabelle 6.4 stellt die Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests dar. Dieser Test wurde gewählt, weil die Stichprobe weniger als 30 Teilnehmer umfasst, abhängig und nicht normal verteilt ist. Die Normalverteilung wurde mithilfe des Shapiro-Wilk-Tests bestimmt. Neben der Messung der Signifikanz wurde für die Effektstärke Cliffs-Delta genutzt.

Das Signifikanzniveau beträgt 5%. Bei der Beantwortungszeit der Programmverständnisaufgaben liegt ein folgendes Ergebnis vor (W: 746,5, p: $<$ 0,001). Dies bedeutet, dass der beobachtete Unterschied in der Beantwortungszeit signifikant ist. Somit kann die in der Operationalisierung aufgestellte Hypothese H_1 angenommen werden. Neben der Signifikanz soll auch noch auf deren Effektstärke eingegangen werden. Die Verbesserung der Beantwortungszeit hatte laut Cliff's Delta eine mittlere Effektstärke.

Programm-verständnis-aufgaben	Korrektheit in % Pretest	Posttest	Statistische Tests Wilcoxon	Cliff's-Delta
V1-V5	76,9 %	77,81%	W: 132 | p: 0,85	0,01

Tabelle 6.5: Programmverständnisaufgaben - Korrektheit

Programm-verständnis-Aufgaben	Mittelwert Klickanzahl in AOIs Pretest	Posttest	Statistische Tests Wilcoxon	Cliff's-Delta
V1 - V5	21	15	W: 1426,5 | p: <0,001	0,19

Tabelle 6.6: Programmverständnisaufgaben - Durchschnittliche Klickanzahl

6.3.1.2 Korrektheit

Die Hypothese wurde in der Operationalisierung folgendermaßen definiert:

> H_2 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Korrektheit bei den Programmverständnisaufgaben.

Die aufgestellte Nullhypothese lautet:

> H_{02} Bei V1 bis V5 gibt es durch die Teilnahme an den Syntaxaufgaben keinen signifikanten Unterschied in der Korrektheit.

Für das Testen dieser Hypothese wurde der Wilcoxon-Vorzeichen-Rang-Test genutzt, da die Stichprobe aus weniger als 30 Teilnehmern besteht sowie abhängig ist. Zudem hat der Shapiro-Wilk-Test ergeben, dass die Korrektheit nicht normal verteilt war. In der Tabelle 6.5 ist die Korrektheit der Programmverständnisaufgaben dargestellt. Der Unterschied bei den Aufgaben V1 bis V5 war nicht signifikant, denn das Signifikanzniveau von 5% wurde deutlich überschritten (W: 132, p: 0,85). Deswegen wird H_{02} bestätigt. Damit kann die aufgestellte Hypothese, dass PA durch den Einsatz der Syntaxaufgaben die Programmverständnisaufgaben korrekter beantworten, verworfen werden. Außerdem kann auch anhand der Effektstärke dargestellt werden, dass diese mit 0,01 unerheblich ist.

6.3.1.3 Visuelle Aufmerksamkeit

Die in der Operationalisierung aufgestellte Hypothese wurde folgendermaßen formuliert:

Programmier-aufgaben	Mittelwert Beantwortungszeit [hh:mm:ss]		Statistische Tests	
	Pretest	Posttest	Wilcoxon	Cliffs Delta
P1-P2/P3-P4	00:48:17	00:18:36	W: 22 | p: 0.002	0,55

Tabelle 6.7: Programmieraufgaben - Beantwortungszeit

Programmier-aufgaben	Mittelwert Korrektheit in Anzahl an Fehlern		Statistische Tests	
	Pretest	Posttest	Wilcoxon	Cliff's Delta
P1-P2/P3-P4	3,29	1,14	W: 12,5 | p: <0,001	0,39

Tabelle 6.8: Programmieraufgaben - Korrektheit in Anzahl an Fehlern

> H_3 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verringert sich die Klickanzahl in den AOIs bei den Programmverständnisaufgaben.

Für die Überprüfung wird die Nullhypothese definiert:

> H_{03} Bei den Programmverständnisaufgaben V1 bis V5 gibt es durch die Teilnahme an den Syntaxaufgaben keinen Unterschied in der Klickanzahl.

Der Wilcoxon-Vorzeichen-Rang-Test wurde angewendet, um diese Hypothesen zu testen. Denn die Stichprobe enthält weniger als 30 Teilnehmer, ist abhängig und durch den Shapiro-Wilk-Test wurde ermittelt, dass die Klickanzahlen nicht normal verteilt sind. In der Tabelle 6.6 werden die durchschnittlichen Klickanzahlen im Pre- und Posttest angezeigt. Diese betrugen im Pretest 21 und im Posttest 15. Der Unterschied zwischen Pre- und Posttest ist signifikant, da der p-Wert des Wilcoxon-Vorzeichen-Rang-Tests (W: 1426,5, p: <0,001) kleiner als das Signifikanzniveau von 5% ist. Die Effektstärke entspricht nach Cliff's Delta einem schwachen Effekt.

6.3.2 Programmieraufgaben

In den folgenden Unterabschnitten werden die beiden Hypothesen über die Beantwortungszeit und Korrektheit der Programmieraufgaben getestet.

6.3.2.1 Beantwortungszeit

In der Operationalisierung wurde die Hypothese wie folgt formuliert:

> H_4 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Beantwortungszeit bei den Programmieraufgaben.

Die Nullhypothese lautet:

> H_{04} Bei P1-P2/P3-P4 gibt es durch die Teilnahme an den Syntaxaufgaben keinen signifikanten Unterschied in der Beantwortungszeit.

Auch für das Testen dieser Hypothese wurde der Wilcoxon-Vorzeichen-Rang-Test eingesetzt, da die Stichprobe weniger als 30 Teilnehmer umfasste sowie abhängig ist und durch den Shapiro-Wilk-Test herausgefunden worden ist, dass die Beantwortungszeit nicht normal verteilt war. In Tabelle 6.7 wird die durchschnittliche Dauer der Beantwortungszeit der Programmieraufgaben dargestellt. Der Unterschied zwischen den Aufgaben des Pretests und des Posttests ist signifikant, weil der p-Wert kleiner als das Signifikanzniveau von 5% ist (W: 12,5, p: <0,001). Dadurch lässt sich die Hypothese bestätigen, dass durch die Teilnahme an den Syntaxaufgaben die Beantwortungszeit in den Programmieraufgaben gesenkt werden kann. Nach Cliffs-Delta ist dies eine mittlere Effektstärke.

6.3.2.2 Korrektheit

Die in der Operationalisierung festgelegte Hypothese bezüglich der Korrektheit der Programmieraufgaben ist:

> H_5Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Korrektheit bei den Programmieraufgaben.

Die Nullhypothese wird folgendermaßen formuliert:

> H_{05} Bei P1/P3 gibt es durch die Teilnahme an den Syntaxaufgaben keinen signifikanten Unterschied in der Korrektheit.

Für das Testen dieser Hypothesen wurde der Wilcoxon-Vorzeichen-Rang-Test durchgeführt. Denn die Stichprobe besteht aus weniger als 30 Teilnehmern und ist abhängig. Die Anwendung des Shapiro-Wilk-Tests ergab, dass die Genauigkeit nicht normal verteilt war. In der Tabelle 6.8 ist ersichtlich, dass der p-Wert des Wilcoxon-Vorzeichen-Rang-Tests (W: 12,5, p: $< 0{,}001$) geringer ist als das Signifikanzniveau. Außerdem ist die Effektstärke nach Cliff's Delta mittel.

6.3.3 Klausur

Die Hypothese bezüglich der Ergebnisse in der Klausur wurde in der Operationalisierung wie folgt beschrieben:

> H_6 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihr Ergebnis in der Klausur.

Klausur	Mittelwert Punkte in der Klausur		Statistische Tests	
	Kontrollgruppe	Testgruppe	Mann-Whitney-U	Cliff's-Delta
Klausurergebnis	64,51	76,42	U: 1100,5 | p: 0,01	0,4

Tabelle 6.9: Klausur - Durchschnittliche Punktzahl

Die Nullhypothese wird folgendermaßen definiert:

> H_{06} Bei der Klausur gibt es durch die Teilnahme an den Syntaxaufgaben keinen Unterschied in den Ergebnissen.

Für das Testen dieser Hypothese wurde der Mann-Whitney-U-Test eingesetzt, da die Stichprobe weniger als 30 Teilnehmer enthält sowie unabhängig ist. Zudem wurden durch den Shapiro-Wilk-Test ermittelt, dass die Anzahl der Punkte in der Klausur nicht normal verteilt war. In der Tabelle 6.9 werden die Ergebnisse des Mann-Whitney-U-Tests sowie die Effektstärke nach Cliff's Delta dargestellt. Der Unterschied zwischen den Probanden des Experiments und den anderen Teilnehmern der Klausur ist signifikant, da der p-Wert des Tests kleiner als das Signifikanzniveau von 5% ist (U: 1100,5, p: 0,01). Außerdem gilt die Effektstärke nach Cliff's-Delta mit 0,4 als mittel.

7 Diskussion

Basierend auf den Ergebnissen des vorliegenden Experiments sowie den bestätigten und abgelehnten statistischen Hypothesen werden im folgenden Kapitel die Ergebnisse diskutiert. Im Abschnitt 7.1 wird auf die Auswirkungen auf die drei abhängigen Variablen - Korrektheit, Beantwortungszeit und visuelle Aufmerksamkeit - eingegangen. Darauf folgen im Abschnitt 7.2 Einschränkungen der Validität, zu denen die Konstruktvalidität, interne, externe und statistische Validität zählt.

7.1 Bewertung der Ergebnisse und Implikationen

Im folgenden Abschnitt werden die im Kapitel 6 herausgearbeiteten Ergebnisse bewertet und Implikationen aus diesen herausgearbeitet. Dieser Abschnitt beginnt mit den Programmverständnisaufgaben. Danach folgt die Interpretation der Programmieraufgaben. Schlussendlich werden die Ergebnisse der Klausur besprochen.

7.1.1 Programmverständnisaufgaben

In den folgenden Unterabschnitten werden die Beantwortungszeit, Korrektheit und visuelle Aufmerksamkeit in den Programmverständnisaufgaben diskutiert.

7.1.1.1 Beantwortungszeit

Die Hypothese wurde in der Methodik wie folgt formuliert:

> H_1 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Beantwortungszeit bei den Programmverständnisaufgaben.

Der Wilcoxon-Vorzeichen-Rang-Test hat ergeben, dass der Unterschied in der Beantwortung der Programmverständnisaufgaben signifikant war. Dadurch kann die Hypothese bestätigt werden. Dies kann durch den Einsatz der Syntaxaufgaben erklärt werden. Jedoch sei auch darauf hinzuweisen, dass PA am Anfang signifikante Fortschritte in ihrer Programmierkompetenz machen. Demnach kann der Unterschied von vier bis acht Wochen auch einen Teil zu dem Unterschied in der Beantwortungszeit beigetragen haben. Die Konsequenz, die aus den Ergebnissen gezogen werden kann, ist, dass Syntaxaufgaben helfen können, kurze Programme schneller zu verstehen. Die Beantwortungszeit von Programmverständnisaufgaben könnte durch den Einsatz von Syntaxaufgaben unterstützt werden. Anhand der Diskussion meiner

Ergebnisse kann die in der Operationalisierung aufgestellte Hypothese belegt werden. Damit diese Aussage jedoch pauschalisiert werden kann, sollte die externe Validität erhöht werden. Dafür könnten die Aufgaben an mehreren Universitäten durchgeführt oder verschiedene Quellcodestücke hierfür genutzt werden. Außerdem wäre der Einsatz von Programmierexperten oder anderen Programmiersprachen interessant, da sich bspw. die Komplexität der Syntax von Java und Python unterscheidet. Es könnten zudem auch längere und komplexere Programme eingesetzt werden.

7.1.1.2 Korrektheit

In diesem Unterabschnitt soll folgende Hypothese diskutiert werden:

> H_2 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Korrektheit bei den Programmverständnisaufgaben.

Durch den Einsatz des Wilcoxon-Vorzeichen-Rang-Tests musste H_2 abgelehnt werden. Aufgrund dessen muss nach einer Erklärung für den nicht signifikanten Unterschied gesucht werden. Eine mögliche Erläuterung könnte sein, dass PA zwar die Syntax der Programmierkonstrukte besser verstanden und dementsprechend auch schneller lesen konnten. Aber dies hat zu keinem Unterschied beim Verständnis der Semantik der Programmierkonstrukte und somit auch dem Programmverständnis geführt. Für weitere Erklärungen wären Think-Aloud-Protokolle eine mögliche Lösung. Außerdem kann davon ausgegangen werden, dass die extrinsische Belastung durch die Syntax einer Programmiersprache bei Programmverständnisaufgaben gering ist, da PA syntaktisch korrekten Quellcode gezeigt bekommen haben. Zur Generalisierung dieser Aussage sollte die Korrektheit und die einzelnen Schritte, die zur korrekten Beantwortung eines Programms, führen, genauer untersucht werden. Dafür könnten größere Programme eingesetzt werden und Zwischenergebnisse abgefragt werden.

7.1.1.3 Visuelle Aufmerksamkeit

Die Annahme über die visuelle Aufmerksamkeit von AOI lautet:

> H_3 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verringert sich die Klickanzahl in den AOI bei den Programmverständnisaufgaben.

Mithilfe des Wilcoxon-Vorzeichen-Rang-Tests konnte die aufgestellte Hypothese bezüglich der Verbesserung der Klickanzahl in den AOI für die Aufgaben V1 bis V5 belegt werden. Bei der Betrachtung der Heatmaps und der Klickanzahl in den AOI können die Unterschiede auf die Komplexität verschiedener Algorithmen und der dort eingesetzten Programmierkonstrukten zurückgeführt werden. In der Abbildung 6.4 sind die Unterschiede in den Heatmaps bei V1 sichtbar. Insgesamt haben PA im Posttest weniger in die Methode und in die relevanten Schritte dieser geklickt.

Den beiden Heatmaps kann entnommen werden, dass Studierende durch den Einsatz der Syntaxaufgaben schneller verstanden haben, was in den Programmierkonstrukten abläuft und für welchen Zweck diese in der Methode verwendet werden. Die erste if-Bedingung fragt ab, welcher der beiden übergebenen Parameter länger ist. In der folgenden for-Schleife wird geprüft, ob in beiden übergebenen Strings die gleichen Buchstaben an der gleichen Stelle stehen. Wenn bei V1 Heatmap und Klickanzahl in den AOI zusammen betrachtet werden, ist ersichtlich, dass es sowohl absolut als auch prozentual einen Rückgang in der visuellen Aufmerksamkeit der AOI gab. Im Gegensatz dazu ist bei V3 der Rückgang der Klickanzahl nur absolut signifikant reduziert worden. Die prozentuale Verteilung der Klicks entspricht in etwa denen des Pretests. Dadurch kann bei dieser Aufgabe darauf geschlossen werden, dass Studierende das Programm zwar insgesamt schneller gelesen haben, jedoch die Funktion der AOI nicht schneller erkannt wird. Bei V2 und V5 liegen die Ergebnisse zwischen denen von V1 und V3. Hingegen gab es bei V4 keinen sichtbaren Unterschied in den Heatmaps und auch nur einen marginalen Unterschied bei der Klickanzahl in den AOI.

Dennoch lässt sich diese Hypothese grundsätzlich bestätigen. Bei einer Verstärkung der externen Validität, indem deutlich längere und komplexere Programme eingesetzt werden würden, könnte diese Annahme generalisiert oder widerlegt werden.

7.1.2 Programmieraufgaben

Nach der Diskussion der Programmverständnisaufgaben werden nun die Ergebnisse der Programmieraufgaben interpretiert.

7.1.2.1 Beantwortungszeit

In der Methodik wurde folgende Hypothese definiert:

> H_4 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Beantwortungszeit bei den Programmieraufgaben.

Durch die Nutzung des Wilcoxon-Vorzeichen-Rang-Tests wurde eine signifikante Verbesserung in der Beantwortungszeit der Programmieraufgaben festgestellt. Daher kann darauf geschlossen werden, dass die Syntaxaufgaben eine Auswirkung auf die Beantwortungszeit bei Programmieraufgaben haben. Jedoch gibt es verschiedene Einschränkungen bei dieser Hypothese. Zum einen benötigten einige Probanden nur wenige Minuten für die Abgabe ihres Programms. Zum anderen gab es unterschiedliche Komplexitätsgrade bei den Stringmanipulationsaufgaben P1 und P3. Außerdem wurde von einigen Probanden berichtet, dass die Aufgabenbeschreibung von P2 unklar war, wodurch es zu einer höheren extrinsischen kognitiven Belastung gekommen ist. Zur Interpretation wird nun auf die drei Problematiken eingegangen.

Anhand der kurzen Beantwortungszeit einiger Probanden muss davon ausgegangen werden, dass diese ihre Aufgaben bereits vorher in einer anderen Entwicklungsumgebung programmiert oder zumindest geplant haben. Aufgrund dessen, muss die Verbesserung der Beantwortungszeit kritisch betrachtet werden, denn die genaue Beantwortungszeit, welche PA für das Lösen der Programmieraufgaben benötigt haben, konnte nicht gemessen werden. Jedoch sei darauf hinzuweisen, dass diese Möglichkeit sowohl im Pre- als auch im Posttest gegeben war.

Aufgrund der unterschiedlichen Komplexität kann davon ausgegangen werden, dass Probanden für die Bearbeitung von P3 allein durch diesen Unterschied weniger Zeit benötigt haben. In OPAL ist es nicht möglich, die Beantwortungszeit der einzelnen Aufgaben auszuwerten, jedoch werden durchschnittliche Beantwortungszeiten gemessen. Aber diese enthalten auch Ergebnisse von Studierenden, die letztendlich nicht an allen Teilen der Studie teilgenommen haben. Bei dieser Betrachtung ist jedoch auffällig, dass die Bearbeitung von P1 durchschnittlich 22 Minuten und sechs Sekunden gedauert hat. Hingegen benötigten die Probanden im Posttest mit sieben Minuten und vier Sekunden weniger als ein Drittel der Zeit. Dem gegenüber hat sich die Beantwortungszeit zwischen P2 zu P4 von 26 Minuten und zehn Sekunden auf elf Minuten und sechs Sekunden nur etwas mehr als halbiert. Dennoch ist der Effekt auch bei einem Vergleich der Aufgaben P2 und P4 signifikant. Deswegen kann die signifikante Verbesserung der Beantwortungszeit durch den Einsatz der Syntaxaufgaben erklärt werden. Jedoch sei auch hier auf die etwa sechs Wochen zwischen Pre- und Posttest hinzuweisen, wodurch PA ihre Programmierfähigkeiten generell verbessert haben sollten.

Die erhöhte extrinsische Belastung der Aufgabenbeschreibung von P2 hat wahrscheinlich zu einer leichten Verzerrung der Ergebnisse geführt, jedoch geht der Autor der vorliegenden Arbeit davon aus, dass dies eher die Korrektheit beeinflusst hat und die Beantwortungszeit nur marginal.

Zum Pauschalisieren dieser Aussagen sollte die Konstruktvalidität und die externe Validität erhöht werden. Für die Erhöhung der Konstruktvalidität sollte ein geeigneter Bewertungsmaßstab für die Programmieraufgaben erstellt werden. Somit wären unterschiedliche Komplexitätsgrade ausgeschlossen. Hingegen könnten für eine gesteigerte externe Validität mehr Probanden mit einbezogen werden sowie auch eine randomisierte Zuteilung erfolgen. Außerdem könnten verschiedene Programmiersprachen oder auch komplexere Programme untersucht werden. Eine weitere Option wäre die Teilnahme von Programmierexperten.

7.1.2.2 Korrektheit

Folgende Hypothese für die Korrektheit in Programmieraufgaben wurde in der Operationalisierung formuliert:

> H_5 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihre Korrektheit bei den Programmieraufgaben.

Durch den Einsatz des Wilcoxon-Vorzeichen-Rang-Tests konnte die Hypothese belegt werden, denn es ist eine signifikante Verbesserung der Korrektheit ermittelt worden. Als mögliche Erklärung kann der Einsatz der Syntaxaufgaben gelten. Die Anwendung des Treatments hat zu einer Senkung der extrinsischen kognitiven Belastung geführt. Auch wenn beachtet wird, dass P3 im Vergleich zu P1 zu simpel war und nur P2 und P4 verglichen werden, ist eine Reduzierung der Anzahl der Fehler von PA signifikant. Daneben sei noch auf die Aufgabenstellung von P2 hinzuweisen, denn einige Probanden haben die Aufgabe falsch verstanden und dies auch in Übungen zur Vorlesung erläutert. Insgesamt haben fünf der 21 Probanden die Aufgabe falsch verstanden. Jedoch führte bei drei von fünf der Abgaben eine Änderung der return-Anweisung zum richtigen Ergebnis. Eine signifikante Reduzierung der Anzahl an Fehlern bleibt jedoch bestehen, selbst wenn die Fehler dieser Probanden herausgerechnet werden würden.

Mit besonderer Beachtung der Syntax ist auffallend, dass im Pretest deutlich mehr syntaktische Fehler gemacht worden sind. Im Pretest wurden von den Probanden bei P1 zur Ermittlung der Länge des Strings length anstatt length() genutzt. Außerdem haben drei Studierende mindestens ein Semikolon am Ende einer Anweisung vergessen. Ein weiteres Problem war das Vergessen eines Trennzeichens in der String split-Methode. Vier Probanden hatten mindestens einen Tippfehler, wie etwa lenght anstatt length, in ihrem Programm. In einer for-Schleife wurde in der Bedingung ein Komma anstatt eines Semikolons als Trennzeichen genutzt. Des Weiteren hat ein Proband die Zählvariable einer while-Schleife nicht hochgezählt. Außerdem haben zwei Studierende unterschiedlich viele geschweifte Klammern genutzt. Ein Teilnehmer hat eine Variable mit einem Schlüsselwort (short) deklariert. Außerdem hat ein Studierender die Methodensignatur doppelt hinzugefügt.

Im Gegensatz dazu haben im Posttest drei Studierende mindestens einmal ein Semikolon als Abschluss einer Anweisung vergessen. Außerdem wurde zur Bestimmung der Länge eines Strings zweimal length() anstatt length genutzt. Zudem hat ein Proband size anstatt length genutzt. Außerdem hat ein Proband eine geschweifte Klammer zu viel genutzt. Insgesamt wurden im Posttest in absoluten Zahlen deutlich weniger syntaktische Fehler gemacht als im Pretest.

Jedoch hat sich der Anteil syntaktischer Fehler an der Anzahl aller Fehler nur marginal verändert. Der Anteil betrug im Pretest 27,39% und im Posttest 25,44%. Aber es wurden in deutlich weniger Aufgaben syntaktische Fehler gemacht, im Pretest hatten 14 Programme von neun unterschiedlichen Probanden syntaktische Fehler. Im Vergleich dazu hatten im Posttest nur fünf Probanden in fünf Aufgaben syntaktische Fehler. Demnach liegt der Hauptaspekt der Verbesserung der Korrektheit in der Verringerung semantischer Fehler. Hierbei wird die These unterstützt, dass der primäre Aspekt der Syntaxaufgaben nicht das Erlernen der Syntax einer Programmiersprache ist, sondern die Reduzierung der extrinsischen kognitiven Belastung. Dadurch können mehr lernförderliche Ressourcen für die Bildung von Schemata für die Problemlösung entstehen.

In weiteren Studien sollte die Konstruktvalidität erhöht werden, indem ein Maßstab für die Beurteilung der Komplexität von Programmieraufgaben eingesetzt werden

sollte. Außerdem könnte eine genauere Betrachtung der Effektivität bezüglich der gelernten Programmierkonstrukte erfolgen, indem diese beim Zählen der Fehler gesondert berücksichtigt werden.

7.1.3 Klausur

Die Hypothese lautet:

> H_6 Wenn Studierende an den Syntaxaufgaben teilgenommen haben, dann verbessert sich ihr Ergebnis in der Klausur.

Der Mann-Whitney-U-Test hat eine Bestätigung dieser Hypothese ergeben, da der Vergleich zwischen den beiden Gruppen einen signifikanten Effekt auf das Ergebnis in der Klausur ergab. Anhand der Ergebnisse in der Tabelle 6.9 wird ersichtlich, dass die Teilnahme an den Syntaxaufgaben einen signifikanten Effekt auf die Ergebnisse der Klausur hatte. Studierende, die an der Studie teilgenommen haben, schnitten durchschnittlich besser ab. Es sei jedoch darauf hinzuweisen, dass die Teilnehmenden nicht randomisiert gewählt worden sind. Dadurch könnten vor allen Dingen Studierende, die generell motivierter sind und versuchen ihre Ergebnisse zu verbessern, an der Studie teilgenommen haben. Eine Verbesserung der Ergebnisse kann durch eine verringerte extrinsische kognitive Belastung durch die Java-Syntax entstanden sein. Im Sommersemester wurde die Klausur aufgrund von Corona-Maßnahmen online durchgeführt. Dadurch konnten die Teilnehmenden mögliche Lösungen im Internet suchen und ihre Aufgaben in einer Entwicklungsumgebung testen. Bei einer klassischen Klausur, die mit Stift und Papier stattfindet, könnten die Effekte größer sein. Außerdem könnten dabei auch mögliche syntaktische Fehler gezählt und verglichen werden. In weiteren Studien könnten die Klausurergebnisse genauer untersucht werden, wobei geprüft werden könnte, an welchen Stellen PA Probleme haben und Fehler machen und ob diese andere Fehler als die Kontrollgruppe machen. Nach der Diskussion der sechs Hypothesen folgt nun die Beantwortung der drei Forschungsfragen.

7.1.4 Beantwortung der Forschungsfragen

Im folgenden Abschnitt sollen die drei Forschungsfragen der vorliegenden Arbeit beantwortet werden.

> **Forschungsfrage 1**: Wie beeinflussen Syntaxaufgaben die Beantwortungszeit, Korrektheit und visuelle Aufmerksamkeit von PA in Programmverständnisaufgaben?

In den Unterabschnitten 2.3.4 und 2.3.5 wurde herausgearbeitet, dass Programmverständnis eine zentrale Fähigkeit ist, die PA beherrschen müssen. Der Verfasser der Arbeit kommt zu dem Schluss, dass der Einsatz der Syntaxaufgaben zwar einen Effekt auf die Verbesserung der Beantwortungszeit und die visuelle Aufmerksamkeit

hat, aber dieser nicht mit der Korrektheit korreliert. Insbesondere bei Aufgabe V4, die keinen häufig genutzten Algorithmus enthielt, gab es eine Verschlechterung in der Korrektheit und auch die Verbesserung der Beantwortungszeit war nur marginal. Anhand der anderen Aufgaben ist jedoch folgendes ersichtlich: Erstens fördert der Einsatz von Syntaxaufgaben das schnelle Lesen von Programmen. Denn bei genauerer Analyse der Beantwortungszeit wird deutlich, dass diese sich bei allen Aufgaben verringert hat, die Unterschiede jedoch zwischen den unterschiedlichen Programmverständnisaufgaben variieren. Aber es sollte beachtet werden, dass zwischen dem Pre- und Posttest etwa sechs Wochen liegen, wodurch PA generell ihr Programmverständnis verbessert haben könnten. Zweitens führt die Fähigkeit Programme schneller lesen zu können nicht zu einer gesteigerten Korrektheit, die mit verbesserten Tracingfähigkeiten und einem verbesserten Programmverständnis assoziiert werden kann.

Jedoch wird in der Vorlesung und auch in den Übungen das Lesen und Verstehen von Quellcode nicht explizit gelehrt und geübt, weswegen die Lesegeschwindigkeit ein guter Marker zur Analyse der Lesefähigkeit von Studierenden sein kann. Somit beeinflussen Syntaxaufgaben die Beantwortungszeit von Programmverständnisaufgaben insofern, dass PA diese Aufgaben signifikant schneller beantworten. Hingegen wird die Korrektheit der Programmverständnisaufgaben nur marginal durch den Einsatz von Syntaxaufgaben verbessert. Obwohl die Probanden die Aufgaben zu etwa einem Drittel schneller beantwortet haben, haben sie diese nur geringfügig korrekter beantwortet. Anhand dieser Ergebnisse wird ersichtlich, dass Syntaxaufgaben die Korrektheit von Programmverständnisaufgaben nicht beeinflussen. Bei Betrachtung der kognitiven Belastung kann festgehalten werden, dass durch das Beherrschen der Syntax nur wenige lernförderliche Ressourcen für das Programmverständnis freigesetzt werden. Beim Lesen von fehlerfreien Programmen besteht für Studierende keine extrinsische kognitive Belastung, weil die Syntax der zu lesenden Programme fehlerfrei ist. Dementsprechend hat ein separates Lernen von Syntax und Problemlösung keinen Effekt auf die Korrektheit von Programmverständnisaufgaben.

Schlussendlich soll zur Beantwortung der ersten Forschungsfrage auch noch die Beeinflussung der visuellen Aufmerksamkeit erwähnt werden. Bei der Analyse der Klickanzahl in den AOIs ist die Verbesserung signifikant, auch wenn die Effektstärke nach Cliff's Delta nur schwach ist. Neben der Betrachtung der Klickanzahl werden auch die durchschnittlichen Heatmaps der verschiedenen Programmverständnisaufgaben betrachtet. Insgesamt haben sich die Heatmaps, bis auf bei V1, nur marginal verändert. Dementsprechend beeinflussen Syntaxaufgaben die visuelle Aufmerksamkeit von PA bei Programmverständnisaufgaben, jedoch nicht bei jeder Aufgabe gleich stark. Insgesamt lässt sich festhalten, dass Syntaxaufgaben die Beantwortungszeit und visuelle Aufmerksamkeit positiv beeinflussen, wobei die Beantwortungszeiten signifikanter beeinflusst werden. Hingegen wird die Korrektheit nicht beeinflusst. Somit kann zusammenfassend angemerkt werden, dass der Einsatz von Syntaxaufgaben zur Verbesserung der Lesefähigkeit von PA führen kann. Aber dies sollte in weiteren Studien untersucht werden, da vor allen Dingen zwei Aspekte nicht abschließend beantwortet werden können. Zum einen kann anhand des vorliegen-

den Experiments nicht herausgearbeitet werden, an welchen Stellen Studierende zu Fehlern beim Programmverständnis neigen. Zum anderen, ob der Einsatz der Syntaxaufgaben zu einer Veränderung möglicher Fehlerquellen beigetragen hat.

Aufgrund der Verknüpfung der Programmierfähigkeit als auch der Klausurergebnisse werden Forschungsfrage 2 und 3 gemeinsam beantwortet. Jedoch wird genauer auf die einzelnen Attribute eingegangen.

> **Forschungsfrage 2**: Wie beeinflussen Syntaxaufgaben die Beantwortungszeit und Korrektheit von PA in Programmieraufgaben?
>
> **Forschungsfrage 3**: Wie beeinflussen Syntaxaufgaben die Ergebnisse von PA in der Klausur?

In den Programmieraufgaben und der Klausur wurde die wichtigste Fähigkeit, die PA in einem Programmierkurs lernen - Programmieren - untersucht. In der Analyse und Diskussion der Ergebnisse konnte sowohl bei der Klausur auch bei den Programmieraufgaben ein signifikanter Effekt beobachtet werden. Zuerst wird nun auf die Beeinflussung der Beantwortungszeit und Korrektheit von PA in Programmieraufgaben eingegangen. Danach folgt eine Erläuterung der Beeinflussung bezüglich der Ergebnisse von PA in der Klausur.

In den Unterabschnitten 7.1.2.1 und 7.1.2.2 wurden die Ergebnisse der Programmieraufgaben-Hypothesen dargelegt. Anhand dieser wurde ersichtlich, dass PA deutlich schneller Programme geplant und geschrieben haben, die dazu auch noch weniger Fehler enthielten und somit korrekter waren. Probanden beantworteten nach dem Treatment die Programmieraufgaben in weniger als der Hälfte der Zeit. Außerdem verringerte sich die Anzahl an Fehlern auf fast ein Drittel.

Ähnlich den Programmieraufgaben wurde auch in der Klausur festgestellt, dass Probanden sich deutlich verbessert haben. Die Klausurergebnisse können als wichtigster Erfolgsfaktor für die Kompetenz nach einem Programmierkurs gesehen werden. Denn in diesen werden komplexere Aufgaben abgefragt als in den Programmieraufgaben des vorliegenden Experiments. Außerdem wurden die Programmieraufgaben nicht benotet, da die Bewertung keine Auswirkungen auf die Abschlussnote hatte. Durch die Teilnahme an den Syntaxaufgaben haben PA profitiert, denn jeder Proband hat die Klausur bestanden und durchschnittlich haben sie eine signifikant höhere Punktzahl in den Klausuren erhalten. Dadurch haben sie Datenstrukturen mit einer besseren Note bestanden und haben ihre Programmierkompetenz verbessert. Im Abschnitt 1.1 wurden grundlegende Probleme in Programmiereinführungskursen dargestellt. Besonders auffällig waren die hohen Durchfall- und Abbruchquoten von Studierenden. Der Einsatz der Syntaxaufgaben hat dazu beigetragen, dass nur zwei der 21 Probanden die Klausur nicht mitgeschrieben haben. Dies entspricht einem Anteil von 9,52%. Die restlichen 19 Probanden haben die Klausur bestanden. Dadurch hat sich die Durchfallquote erheblich verbessert und auch die Abbruchquote ist gering. Zudem ist nicht bekannt, wieso die beiden Probanden die Klausur nicht mitgeschrieben haben. Somit hat der Einsatz der Syntaxaufgaben den gesamten Lernprozess der PA unterstützt. Dies ist anhand

der Klausurergebnisse ersichtlich. Denn die höhere Punktzahl kann als Beweis dafür gesehen werden, dass die Probanden bessere Programmierkenntnisse nach dem Kurs Datenstrukturen aufwiesen als die restlichen Teilnehmer.

Den Syntaxaufgaben einen derart maßgeblichen Effekt zuzuweisen mag zum Zweifeln anregen. Jedoch wurde im theoretischen Hintergrund die Rolle der kognitiven Belastung für das Lernen der Programmierung im Unterabschnitt 2.3.7 beschrieben. Demnach haben die Syntaxaufgaben zu einer Verringerung der extrinsischen kognitiven Belastung von PA beim Bearbeiten einer Klausur geführt. Dabei war der primäre Effekt der Syntaxaufgaben nicht das Verbessern syntaktischer Kenntnisse, sondern eine Reduzierung der extrinsischen Belastung durch stärker ausgeprägte syntaktische Kenntnisse, wodurch mehr lernförderliche Ressourcen für komplexere Fähigkeiten, wie etwa die Problemlösung, akkumuliert werden konnten. Denn Programmierenlernen umfasst mehr Fähigkeiten als das Lernen einer formalen Sprache, weil in dieser Sprache auch Probleme gelöst werden müssen. Durch den Einsatz von Syntaxaufgaben wird das Prinzip der begrenzten Veränderung nicht verletzt, wodurch PA schrittweise Schemata entwickeln und automatisieren können. Hierfür dient die Beherrschung syntaktischer Kenntnisse als Fundament.

Diese lernförderlichen Ressourcen führten zu einem besseren Lernergebnis, was in eine höhere Programmierkompetenz resultierte. Der Einsatz von Syntaxaufgaben als Grundlage für das Programmierenlernen hat zur Bildung komplexer Schemata geführt, die von PA in der Klausur automatisiert genutzt werden konnten. Zusammenfassend kann rekkurierend auf die zweite und dritte Forschungsfrage festgehalten werden, dass Syntaxaufgaben die Ergebnisse von PA in der Datenstrukturenklausur positiv beeinflusst haben.

Nach der Diskussion der Ergebnisse folgt nun die Reflektion des Experiments der vorliegenden Arbeit.

7.1.5 Reflektion

Während der Evaluierung der Daten der vorliegenden Arbeit sind einige positive, aber auch negative Aspekte des Experiments sichtbar geworden.

7.1.5.1 Studiendesign

Der zeitliche Ablauf der Studie wurde gut geplant. Zwischen dem Start des Pretests und dem Start des Posttests lagen nur sechs Wochen. Dadurch konnte die interne Validität des Experiments hochgehalten werden, da der zeitliche Faktor keine tragende Rolle spielt. Dennoch war der zeitliche Abstand groß genug, damit die Studierenden sich nicht mehr an die Aufgaben des Pretests erinnern konnten. Außerdem war die geschätzte Beantwortungszeit für die verschiedenen Bestandteile des Experiments gut gewählt. Die Bearbeitungsdauer der einzelnen Bestandteile der Studie sollten nicht zu lange dauern, um keinen zu großen Eingriff in den Kurs Datenstrukturen darzustellen. Für den Fragebogen sowie die Programmverständnisaufgaben benötigten die Studierenden etwa 30 Minuten, für die Programmieraufgaben

durchschnittlich 48 Minuten. Für die Syntaxaufgaben benötigten Sie durchschnittlich etwa 50 Minuten. Auffällig ist, dass PA für die ersten beiden Tests - if-else-Bedingungen und for-Schleifen - mit 15 Minuten 56 Sekunden und 15 Minuten 26 Sekunden deutlich länger benötigt haben als für die letzten beiden Tests - do-while (zehn Minuten 47 Sekunden) und while (acht Minuten 25 Sekunden). Ein Ziel dieser Aufgaben war ein schnelles Bearbeiten durch die Studierenden, wodurch sie zügig syntaktische Fehlerquellen erkennen und beheben können. Dies wurde erreicht, denn pro Test lagen die durchschnittlichen Beantwortungszeiten pro Aufgabe bei 20 bis 41 Sekunden. Im Vergleich zum Pretest benötigten die Probanden für die Programmverständnisaufgaben etwa zehn Minuten weniger und für die Programmieraufgaben nur 18 Minuten und waren somit 30 Minuten schneller. Bei einer größeren Stichprobe hätten die Probanden in verschiedene Gruppen aufgeteilt werden können. Somit hätte untersucht werden können, ob die generelle Programmiererfahrung oder die Programmiererfahrung in Java den Effekt der Syntaxaufgaben beeinflusst. Außerdem hätten sowohl das biologische Geschlecht als auch der Einfluss verschiedener Studiengänge untersucht werden können. Dabei wäre vor allen Dingen der nichtkonsekutive Masterstudiengang Informatik für Geistes- und Sozialwissenschaftler interessant, weil dessen Studierende bereits ein geistes- oder sozialwissenschaftliches Bachelorstudium abgeschlossen haben.

Nach der Beschreibung des generellen Studienablaufs soll an dieser Stelle auf die einzelnen Bestandteile - Programmverständnisaufgaben, Programmieraufgaben und Syntaxaufgaben - eingegangen werden.

7.1.5.2 Programmverständnisaufgaben

Die Programmverständnisaufgaben wurden gut ausgewählt. Die Schwierigkeit und Beantwortungszeit ähnelten sich im Pre- und Posttest. Dies kann anhand der Beantwortungszeiten der Aufgaben V1-V5 im Vergleich zu den Aufgaben V6-10 bzw. V11-V15 gesehen werden. Bei der Korrektheit gab es sowohl bei gleichen Algorithmen als auch bei verschiedenen Algorithmen keinen signifikanten Effekt. Demnach haben Syntaxaufgaben scheinbar keinen Effekt auf die Verbesserung der Lesefähigkeiten der Studierenden bzgl. der Korrektheit. In der Abbildung 6.1 wird ersichtlich, dass die Probanden im Posttest alle gleichen Aufgaben, durch den Einsatz der Syntaxaufgaben, schneller beantwortet haben. Bei der Betrachtung aller gleichen Algorithmen wurde nachgewiesen, dass die Verringerung statistisch signifikant ist. Jedoch unterscheiden sich die verschiedenen Aufgaben. Besonders bei V3 gab es eine durchschnittliche Verringerung um 53,84 %. Die durchschnittliche Bearbeitungszeit von 187 Sekunden im Pretest kann durch die Komplexität der Aufgabe, der Zeilenanzahl sowie an der String-Methode compareTo() liegen. Denn im Pretest gaben fünf der 51 Probanden an, dass sie die compareTo()-Methode googlen mussten. Drei dieser fünf haben an allen acht Aufgaben des Tests teilgenommen und werden somit in meinen Ergebnissen berücksichtigt. Wenn davon ausgegangen wird, dass nicht jeder Proband dies angemerkt hat, dann mussten noch mehr die Nutzungsweise dieser Methode googlen, was zu der langen Beantwortungszeit geführt haben kön-

nte. Außerdem führte dies zu einer höheren extrinsischen kognitiven Belastung, weil das Problem durch den Einsatz einer anderen Methode hätte vermieden werden können. In einer weiterführenden Studie sollte dementsprechend darauf geachtet werden, dass keine Methoden von Bibliotheken genutzt werden oder diese vorher explizit erklärt werden müssen. Bei den anderen gleichen Aufgaben lag die Beantwortungszeit zwischen 119 und 126 Sekunden. Die Beantwortungszeit von V1 und V2 verringerte sich im Posttest ähnlich stark. Hingegen sank die Beantwortungszeit bei V4 nur marginal, V5 lag zwischen den drei Aufgaben.

Neben der Betrachtung der gleichen Algorithmen sollen auch die unterschiedlichen interpretiert werden. Allgemein konnte die Beantwortungszeit aller unterschiedlichen Algorithmen um durchschnittlich 31,59% verringert werden und ähnelte dabei der Verringerung bei gleichen Programmen (35,94%). PA haben im Pretest besonders lange für die Beantwortung von V6, V8 und V10 benötigt. Im Posttest haben sie nur für V14 ähnlich viel Zeit gebraucht. Vier Probanden haben angemerkt, dass sie die Funktionsweise der String-Methode substring() nicht kennen. Diese Unkenntnis führt zu einer höheren extrinsischen kognitiven Belastung, die ein Problem beim Bearbeiten der Aufgabe darstellt, weil Studierende entweder nach der Funktionsweise der Methode suchen mussten oder bei ihrer Antwort mutmaßen. Anhand der Verringerung der durchschnittlichen Beantwortungszeit bei den unterschiedlichen Algorithmen kann ein ähnliches Fazit gezogen werden, wie bei den gleichen Algorithmen, weil sich auch hier die Beantwortungszeit signifikant verringert hat.

Besonders auffällig ist die geringe Korrektheit bei V4. V4 ist das einzige Programm, welches keinen Algorithmus enthält, der häufig in der Lehre eingesetzt wird. Außerdem gibt der Name der Methode - Multiples - keinen direkten Hinweis auf die Funktionsweise des Programms, wie dies etwa bei einer Methode MultiplesOfTwoAndThree der Fall wäre. Trotz des schlechten Ergebnisses haben die Probanden die Aufgabe als einfach eingeschätzt, im Pretest durchschnittlich 1,19 und im Posttest 1,24. Dabei ist auch auffällig, dass V4 die einzige Programmverständnisaufabe der gleichen Aufgaben ist, die von den Studierenden im Posttest als schwieriger eingestuft worden ist. Das boolesche oder könnte eine andere Schwierigkeit darstellen, da V4 die einzige Aufgabe ist, die dieses enthält. Ein anderes Problem könnte der Einsatz des Modulo-Operators sein. Jedoch lässt sich in den falschen Antworten kein Muster entdecken, die Antworten sind auf einem Spektrum zwischen 1 und 38 verteilt. Auch im Posttest gibt es kein Muster bei den falschen Antworten. So wurden sowohl 6, 7, 8 als auch Antworten rund um die richtige Lösung von 32 abgegeben. Im Posttest haben auch nur zwei Personen 42 abgegeben. Dies wäre die richtige Lösung, wenn die Schleife nicht bis 9, sondern bis einschließlich 10 gelaufen wäre. Somit ist in der Tabelle 6.5 die durchschnittliche Korrektheit von V4 sowohl im Pre- als auch im Posttest im Vergleich zu allen anderen Aufgaben besonders niedrig. In einem Think-Aloud-Protokoll könnte festgestellt werden, welche Probleme PA beim Lösen dieser Aufgabe hatten. Zum Vergleich könnten verschiedene Methoden mit einem unpräzisen Namen verglichen werden oder auch unterschiedliche Methoden, die ein boolesches oder enthalten. Trotz dessen beinhaltete V3 einen deutlich komplexeren Algorithmus. Daraus kann geschlossen werden, dass PA besonders gut Lehralgorith-

men kennen und diese auch besser als bisher unbekannte Programme beantworten. Dies könnte an einer häufigeren Verwendung klassischer Lehralgorithmen liegen.

Unterschiedliche Algorithmen sollen den gleichen gegenübergestellt werden. Die Korrektheit bei unterschiedlichen Algorithmen ist von 91,43 % auf 82,86% gesunken. Dabei sticht vor allen Dingen V14 heraus. Bereits im vorherigen Absatz wurde auf die erhöhte extrinsische kognitive Belastung eingegangen, die durch eine unbekannte String-Methode entsteht. Diese Aufgabe wurde von nur 47,62% korrekt beantwortet. Neun der 21 Probanden haben die Funktionsweise der Methode nicht verstanden, weil sechs ein dreistelliges Ergebnis eingetragen haben, zwei ein einstellig und jemand sogar kommentiert hat, dass er die Funktionsweise nicht kannte. Der Methode werden als Parameter der Start- und Endindex des Strings übergeben. Jedoch werden nur die Buchstaben zwischen Startindex und Endindex, ohne den Endindex einzuschließen, ausgegeben. Daher kann das schlechte Ergebnis auf diese Methode zurückgeführt werden. Demnach sollte bei einer weiterführenden Studie auf den Einsatz unbekannter Methoden verzichtet werden oder diese vorher eingeführt werden. Bei den restlichen Aufgaben veränderte sich die Korrektheit zwischen Pre- und Posttest kaum. Es ist zwar kompliziert unterschiedliche Algorithmen gegenüberzustellen, aber anhand der Beurteilung der Schwierigkeit durch die Probanden als auch meiner Einschätzung, soll dies nun versucht werden. V6 entsprach im Pretest einer Schwierigkeit von 1,67 und 85,71% der Studierenden haben die korrekte Antwort abgegeben, dem steht V14 gegenüber mit einer durchschnittlichen Schwierigkeit von 1,38 und einer Korrektheit von 47,62%. V7 wird V11 gegenübergestellt: 100% zu 90,48% bei einer Schwierigkeit von 1,05 im Pretest und 1 im Posttest. Außerdem werden V8 90,48% (Schwierigkeit: 1,05) und V13 90,48% (Schwierigkeit: 1) verglichen. Bei der Betrachtung von V9 90,48% (Schwierigkeit: 1,05) gegenüber V11 100% (Schwierigkeit: 1,05) wird eine gesteigerte Korrektheit, bei gleichbleibender Schwierigkeit sichtbar. Hingegen ist beim Vergleich zwischen V10 90,48% (Schwierigkeit: 1,24) und V15 mit 85,71% (Schwierigkeit: 1,29) eine leicht gestiegen Schwierigkeit und gesunkene Korrektheit ersichtlich. Die Schwierigkeit aller Aufgaben ist von 1,23 auf 1,14 gesunken. Jedoch ist die Korrektheit nicht gestiegen, sondern von 91,43% auf 82,86% gefallen.

Neben der generellen Auswahl der Programmverständnisaufgaben sei auch noch REyeker zu nennen. Die Einstellungen hätten in einer Pilotstudie getestet werden und daraufhin angepasst werden müssen. Durch die Einstellungen konnten Probanden nicht nur eine Zeile, sondern fast drei Zeilen sehen (siehe Abbildung 5.4 (b)). Dadurch war es schwierig herauszufiltern, ob sie in die AOI geklickt haben. Problematisch hierbei war, dass sich die Bibliothek zur Analyse der REyeker-Daten noch in der Entwicklung befindet. Demnach sollte bei einer weiterführenden Studie eine Pilotstudie durchgeführt werden, um zu evaluieren, ob die Einstellungen passend für das Forschungsziel sind. Dadurch würde das Arbeiten mit einem Spielraum, rund um die relevanten Zeilen und Konstrukte des Quellcodes, wegfallen. Ein weiterer verbesserungswürdiger Aspekt in REyeker war die Implementierung der Quellcodestücke. Diese werden aktuell als Bilder hochgeladen, wodurch sich deren Größe unterscheiden kann. In der vorliegenden Arbeit waren die Bilder der Quell-

codestücke etwas zu klein gewählt, weswegen in einer weiteren Studie größere Bilder genutzt werden sollten. Außerdem waren die Bilder der Programme, wenn auch nur minimal, unterschiedlich groß. Dementsprechend wird an einer aktualisierten Form von REyeker gearbeitet, in der Quellcode und nicht nur Screenshots von Quellcodestücken direkt eingebunden werden können. Dadurch könnte sich die Qualität der Ergebnisse verbessern, weil eventuelle Schwierigkeiten, die durch unterschiedlich große Screenshots auftreten können, wie etwa unterschiedliche Sichtbereiche durch einen Klick, verhindert werden können.

Schlussendlich könnte neben einer Optimierung der Auswahl der Programmverständnisaugaben sowie der REyeker-Einstellungen in einem weiteren Schritt eine Korrelation zwischen schnellerer Beantwortungszeit und der Korrektheit untersucht werden. Zudem könnte eine Korrelation zwischen der Selbsteinschätzung der Schwierigkeit und der Korrektheit untersucht werden.

7.1.5.3 Programmieraufgaben

Für die Programmieraufgaben wurden vorher unterschiedliche Websites durchsucht, auf denen verschiedene Programmieraufgaben angeboten werden (bspw. CodeWars). Jedoch wurden zu simple Aufgaben für den Posttest ausgewählt. Die Aufgaben wurden anhand meiner Einschätzung sowie der von Kommilitonen ausgewählt. Für eine bessere Einstufung müsste ein Maßstab gefunden werden, anhand dessen die Programmieraufgaben eingestuft werden. Zum einen wäre ein Abwandeln vorheriger PVL oder Klausuraufgaben möglich, sodass diese innerhalb von 30 bis 60 Minuten von PA bearbeitet werden können.

Außerdem war die Aufgabenbeschreibung bei P1 mit einer hohen extrinsischen Belastung verbunden. Aufgrund dessen wurden die Aufgabenbeschreibungen des Posttests von zwei Mitarbeiterinnen der Professur Softwaretechnik auf etwaige Problematiken überprüft. Dies hätte auch im Pretest stattfinden müssen. Eine weitere Optimierungsmöglichkeit besteht in einem verpflichtenden Beantwortungsfeld für die Eintragung des Unique Codes. Im Pretest sollten die Probanden ihren Unique Code im Quellcode kommentieren, aber dies machten einige nicht.

Die Evaluierung der Beantwortungszeit pro Programmieraufgabe stellte sich als schwierig dar, denn in OPAL werden nur die Zeiten von Tests gespeichert und nicht einzelner Aufgaben. Bei einer Replikation oder Weiterführung des Experiments sollte entweder eine andere Plattform genutzt oder die einzelnen Aufgaben als Tests in OPAL eingebunden werden. Außerdem kann nicht ausgeschlossen werden, dass PA die Aufgaben bereits vorher geplant oder in einer Entwicklungsumgebung programmiert haben. Falls Think-Aloud-Protokolle genutzt werden würden, könnte dies überprüft werden.

Problematisch bei der Bewertung der Korrektheit war eine Unterteilung in semantische und syntaktische Fehler. In einer weiterführenden Studie sollte ein genauerer Bewertungsmaßstab zur Evaluierung der Programmieraufgaben genutzt werden. Die Messbarkeit von Programmieraufgaben könnte bspw. durch die Indexbildung, welche Ebert in seiner Dissertation vorgestellt hat [16, S. 49-56], überprüft werden. Außer-

dem könnten die Programmieraufgaben als PVL eingesetzt werden, wodurch PA diese als relevanter wahrnehmen könnten und dies eine Beeinflussung der Ergebnisse verursachen könnte. Außerdem könnte ein mehrmaliges Abgeben der Aufgaben eine Möglichkeit darstellen, damit PA selbstständig syntaktische und simple Fehler verbessern können. Hierbei könnten nur wenige Testfälle genutzt werden, die bspw. keine Randfälle abfangen, aber die grundlegende semantische und syntaktische Struktur des Programms prüfen. Falls mehrere Programmieraufgaben eingesetzt werden würden, könnten so die Programmierfähigkeit und das Verhalten bei der Evaluierung von Programmen überprüft werden.

7.1.5.4 Syntaxaufgaben

Neben den Programmverständnis- und Programmieraufgaben soll auch auf die Syntaxaufgaben genauer eingegangen werden. Die Probanden haben bei allen Aufgaben sehr gut abgeschnitten und fast alle Fehler in den Programmen entdeckt und behoben. Der Mittelwert beträgt beim Syntaxtraining der if-else-Bedingungen 21,33 von 23 Punkten. Bei for-Schleifen 24,14 von 25 Punkten. Bei do-while-Schleifen 24,38 von 25 Punkten und bei while-Schleifen 24,52 von 25 Punkten. Im Folgenden werden nun Aufgaben pro Test beschrieben, die von mehr als drei PA falsch beantwortet worden sind.

Bei den Syntaxaufgaben zu if-else-Bedingungen wurden am häufigsten die Aufgabe 23 falsch beantwortet, gefolgt von Aufgabe 10, 14 und 18. Bei Aufgabe 23 waren sowohl eckige anstatt runde Klammern um die Bedingung gesetzt als auch eine schließende geschweifte Klammer fehlte. Bei Aufgabe 10 wurde hinter die Bedingung ein Semikolon vor der öffnenden geschweiften Klammer gesetzt. Wiederum bei Aufgabe 14 enthielt die Abfrage der Bedingung ein Gleichheitszeichen zu viel. Bei Aufgabe 18 wurden anstatt runder Klammern eckige um die Bedingung gesetzt. Die Aufgabe 20 wurde am häufigsten beim for-Vokabeltraining nicht richtig korrigiert. Die zwanzigste Aufgabe enthielt zwei Fehler, es fehlte eine schließende geschweifte Klammer der ersten for-Schleife sowie ein Plus bei der Erhöhung der Variable j in der zweiten Schleife. Beim while-Vokabeltraining wurde die erste Aufgabe von sechs PA falsch beantwortet. Die erste Aufgabe wurde zu einer Endlosschleife, da dort die Zählvariable innerhalb der while-Schleife nicht hochgezählt worden ist. Die Aufgabe neun des do-while-Vokabeltrainings wurde von sieben Probanden falsch beantwortet. Die neunte Aufgabe enthielt mit einem Doppelpunkt anstatt eines Semikolons einen falschen Abschluss der Schleife.

Die Aufgaben sind nach der Reihenfolge der Einbindung in OPAL dargestellt. Auffallend ist, dass die Probanden vor allen Dingen beim ersten Test, in den if-else-Bedingungen die meisten Fehler gemacht haben. Außerdem enthielt ein Großteil der Aufgaben jeweils nur einen Fehler. Die Probanden haben vor allem bei den Aufgaben Fehler gemacht, die mehr als einen Fehler enthielten, wie bei der 23. Aufgabe der if-else-Bedingungen und der 20. Aufgabe der for-Schleifen. In einem weiteren Schritt könnte die Anzahl der Fehler genannt werden und auch die Anzahl der Fehler könnte zwischen den Aufgaben stärker variieren. Außerdem waren alle

Aufgaben Debugging-Aufgaben, bei denen PA Fehler finden und beheben mussten. Die Aufgaben könnten in verschiedene Aufgaben unterteilt werden und auch in der Schwierigkeit variieren. Dazu zählen Modifizierungs-, Debugging- und Programmieraufgaben, wie im Unterabschnitt 2.4.4 erläutert. Außerdem könnte die Schwierigkeit innerhalb eines Tests ansteigen, denn in den Syntaxaufgaben der vorliegenden Arbeit entsprachen diese dem gleichen Schwierigkeitsgrad.

7.2 Einschränkung der Validität

Es ist wichtig, verschiedene Einschränkungen der Validität zu nennen, denn dadurch können die Ergebnisse beeinflusst und eventuell unbrauchbar werden. Infolgedessen kann durch die identifizierten Fehler ein deutlicher Forschungspfad für künftige wissenschaftliche Auseinandersetzungen aufgezeigt werden. Zuerst werden Einschränkungen für die interne Validität genannt. Danach folgt die externe Validität. Schlussendlich werden Bedrohungen für die statistische Validität präsentiert.

7.2.1 Konstruktvalidität

Für die Konstruktvalidität ist relevant zu hinterfragen, ob tatsächlich die Programmverständnis- und Programmierfähigkeit von Studierenden gemessen worden ist. Im Rahmen der vorliegenden Masterarbeit ist diese durch die Auswahl an Programmverständnis- und Programmieraufgaben gegeben, jedoch sollte für weitere Untersuchungen ein genauerer Maßstab zur Klassifizierung der abhängigen Variablen genutzt werden. Bspw. könnte ein Error-Quotient für die Bestimmung der Programmieraufgaben eingesetzt werden [26]. Trotz dieser Einschränkungen der Konstruktvalidität kann der Programmiererfolg im Rahmen dieses Experiments durch den Einsatz der Syntaxaufgaben gemessen werden.

7.2.2 Interne Validität

Ein mögliches Problem stellen Störfaktoren dar, die trotz einer gründlichen Kontrolle auftreten können. Dazu zählen etwa falsche Angaben der Probanden oder eine unbewusste Beeinflussung durch den Verfasser der vorliegenden Arbeit. Als weiteres Problem kann die Auswahl der Quellcodestücke angesehen werden. In zwei verschiedenen Programmverständnisaufgaben, V3 und V14, wurden String-Methoden genutzt, die nicht alle Probanden kannten. Deswegen sollte bei einer Replikation der Studie darauf geachtet werden, dass keine unbekannten Methoden eingebunden wurden. Besonders problematisch kann eine Stichprobenverzerrung genannt werden, denn die Probanden wurden dem Experiment nicht randomisiert zugeordnet. Dadurch könnte es dazu gekommen sein, dass besonders motivierte Studierende an der Studie teilgenommen haben, wodurch die Ergebnisse beeinflusst worden sind. In einer weiterführenden Studie sollten die Teilnehmer randomisiert zugeteilt werden. Ein weiterer Grund der Bedrohung der internen Validität besteht darin, dass die

Studierenden die Aufgabe online und unbeaufsichtigt bearbeitet haben. Dadurch können Ablenkungen nicht ausgeschlossen werden, bspw. haben einige Probanden bei den Programmieraufgaben besonders viel Zeit benötigt. Dies könnte daran liegen, dass sie nebenbei etwas anderes gemacht haben. Deswegen sollte weiterführende Studien diese Problematik beachten und womöglich ein Überwachungssystem für auszuführende Aufgaben in Betracht ziehen. Demnach sind die gesammelten Daten zwar nicht nutzlos, aber es sollten gewisse Parameter bei weiteren Studien besser überwacht werden.

7.2.3 Externe Validität

Im vorliegenden Experiment wurde angestrebt, die interne Validität zu maximieren, um die Auswirkungen auf das Programmverständnis und die Programmierfähigkeit zuverlässig zu messen. Dementsprechend wurde das Experiment nur in einem Programmierkurs ausgeführt, wodurch nur Studierende daran teilgenommen haben. Außerdem wurden die Aufgaben in einer einzigen Programmiersprache gestellt. Die Anzahl der Programmieraufgaben war auf vier und die Anzahl der Quellcodestücke auf 15 verschiedene beschränkt. Die herausgearbeiteten Ergebnisse sind nur auf die Probanden und auf den Rahmen des Experiments anwendbar. Zur Verallgemeinerung der Ergebnisse sollten verschiedene Parameter der vorliegenden Arbeit, wie etwa die Probanden, die genutzte Programmiersprache und die Aufgaben, verändert werden. Das Experiment könnte in anderen Programmierkursen oder an mehreren Universitäten durchgeführt werden. Dadurch könnten unterschiedlich erfahrene Programmierer untersucht werden. Außerdem könnten auch professionelle Programmierer an zukünftigen Experimenten teilnehmen. Andere Möglichkeiten wären die Verwendung weiterer Programmiersprachen. Schlussendlich könnten andere Quellcodestücke oder Programmieraufgaben ausgewählt werden. Eine Reduzierung der Einschränkungen durch die externe Validität ist ein wichtiger Schritt für an diese Studie anknüpfende Experimente. Dennoch war der Aufbau des Experiments repräsentativ, denn besonders PA arbeiten mit Programmen dieser Schwierigkeitsstufe und müssen auch Probleme lösen, die den ausgewählten Programmieraufgaben entsprechen.

7.2.4 Statistische Validität

Die statistische Validität kann durch die kleine Stichprobe bedroht sein. Diese Validitätsart gibt die Gültigkeit von Schlussfolgerungen über die Korrelation zwischen Treatment und Ergebnis an. Die kleine Stichprobe führte zu einer geringen statistischen Kraft. Für eine Stärkung der statischen Validität sollte in einer Replikation oder in einer weiterführenden Studie eine größere Stichprobe genutzt werden.

7.3 Vergleich zu Related Work

Beim Vergleich zu anderen Studien ist grundlegend festzustellen, dass diese keine Programmverständnisaufgaben genutzt haben. Die Fähigkeit Programme zu lesen und verstehen zu können wurde in keiner Studie getestet, in der auch Syntaxaufgaben eingesetzt worden sind. Außerdem wurden auch in keiner dieser Experimente ein Remote Eye Tracker oder Eye Tracker genutzt. Aufgrund dessen werden in diesem Abschnitt insbesondere die Klausurergebnisse verglichen, denn in anderen Studien wurden in der Regel die Beantwortungszeit und die Korrektheit von Programmierprojekten sowie der Klausur verglichen.

Es gibt unterschiedliche Ergebnisse beim Einsatz von Syntax- bzw. Tippaufgaben. Bei Edwards et al. [18] gab es keinen signifikanten Unterschied in der Verbesserung der Noten in der Klausur, die in der Semesterhälfte stattfand. Hingegen schnitt bei einer weiterführenden Studie von Edwards et al. die Testgruppe in einer ersten Klausur, die drei Wochen nach Beginn des Semesters stattfand, durchschnittlich sieben Prozentpunkte besser ab als die Kontrollgruppe [17, S. 220]. Dieser Effekt verringerte sich bei der zweiten Klausur, welche von den Studierenden zwei Wochen nach dem Ende der Syntaxaufgaben bearbeitet wurde. Hierbei erhielt die Testgruppe durchschnittlich nur noch zwei Prozentpunkte mehr als die Kontrollgruppe [17, S. 221]. Bei Gaweda et al. hatten aktive Teilnehmer der Studie auch bessere Klausur- und Kursnoten [20, S. 110]. Jedoch weisen die Forscher darauf hin, dass besonders viele Studierende, die bereits die Klausur ein zweites Mal schrieben, ihr Tool genutzt haben, wodurch kausale Zusammenhänge angezweifelt werden können [20, S. 111]. Ly et al. [35] und Leinonen et al. [29] haben nicht die Klausurergebnisse, sondern Programmieraufgaben untersucht. Bei Ly et al. [35, S. 12] schnitten Studierende bei Programmieraufgaben, die in direktem Zusammenhang zu den Syntaxaufgaben standen, besser ab. Hingegen gab es bei Leinonen et al. [29] keinen messbaren Effekt bei Programmieraufgaben. Jedoch weisen die Forscher darauf hin, dass in ihrem Programmiereinführungskurs bereits viele kurze Programmieraufgaben eingesetzt wurden.

8 Fazit und Ausblick

Die Syntax einer Programmiersprache ist ein relevanter Faktor für das Programmierenlernen. Im Laufe der vorliegenden Ausarbeitung wurde ersichtlich, dass das Beherrschen der Syntax einen signifikanten Effekt auf das Programmierenlernen hat. Dabei liegt der primäre Einfluss von Syntaxaufgaben nicht auf der Steigerung syntaktischer Kenntnisse, sondern auf einer Reduzierung der extrinsischen kognitiven Belastung. Durch diese können mehr lernförderliche Ressourcen freigesetzt werden, die zur Entwicklung von Schemata und deren Automatisierung führen. Somit können syntaktische Kenntnisse einer Programmiersprache als Grundlage für ein effektives Lernen der Problemlösung in einer Programmiersprache gesehen werden. Aufgrund der geringen Elementinteraktivität bei Syntaxaufgaben besitzen diese eine geringere kognitive Belastung als Programmieraufgaben, wodurch das Prinzip der begrenzten Veränderung nicht verletzt wird. Im Laufe der vorliegenden Arbeit wurde aufgezeigt, dass insbesondere die Programmierfähigkeit verbessert worden ist, wohingegen jedoch das Programmverständnis nicht signifikant besser wurde. Die Probanden haben zwar den Quellcode deutlich schneller gelesen und auch die AOI seltener angeklickt, jedoch hat sich die Korrektheit bei diesen Aufgaben nicht erhöht. Es mag verwundern, dass der Einsatz von Syntaxaufgaben, für die Studierende durchschnittlich etwa 50 Minuten benötigen, einen derart erheblichen Effekt hat. Durch diese kurze Bearbeitungsdauer ist eine Implementierung in einen Lehrplan unproblematisch, weil der Lehrplan nur minimal angepasst werden muss und auch der Aufwand für die Erstellung der Syntaxaufgaben hält sich in Grenzen. Außerdem sei noch darauf hinzuweisen, dass der Einsatz der Syntaxaufgaben keinen zusätzlichen Lernaufwand für die Probanden darstellte, denn sie mussten weniger komplexe PVL bearbeiten. Somit hat ein zielgerichteter Einsatz der Zeit zu einer Verbesserung der Programmierfähigkeit und als auch der Klausurergebnisse von PA geführt. Zusammenfassend lässt sich folgendes Fazit ziehen: Programmierenlernen stellt einen komplexen und schwierigen Prozess dar, dessen Lehre einer kontinuierlichen Optimierung bedarf. Es sollte in weiteren Experimenten untersucht werden, ob der Einsatz von Syntaxaufgaben zur Optimierung der Lehre beitragen kann. Die vorgestellten Ergebnisse führen zu einer Reihe an Fragen, die in weiteren Studien untersucht werden könnten.

Ein Startpunkt für weitere Studien ist das Anknüpfen an den Ergebnissen des vorliegenden Experiments. Es wurde herausgearbeitet, dass der Einsatz von Syntaxaufgaben die Beantwortungszeit von PA bei Programmverständnisaufgaben verbessert hat, jedoch nicht die Korrektheit. Beim Einsatz von Think-Aloud-Protokollen könnten Problematiken beim Programmverständnis genauer herausgearbeitet werden. Außerdem könnte untersucht werden, ob eine schnellere Beantwortungszeit mit der

Korrektheit korreliert. Think-Aloud-Protokolle könnten auch genauere Einblicke in die kognitive Belastung der Probanden liefern. Des Weiteren könnten die Programmverständnisstrategien detaillierter untersucht werden. Dazu zählen die Analyse der Dauer der Betrachtung einer Stelle im Quellcode sowie die Betrachtung der Lesemuster. Dafür könnten die REyeker-Einstellungen angepasst werden oder ein stationärer Eye Tracker genutzt werden. Dadurch könnte genauer analysiert werden, ob Syntaxaufgaben zu einer Verbesserung der Expertise in der Programmverständnisdomäne führen. Denn eine Veränderung der Nutzung von einem Bottom-up-Modell zu einem Top-down-Modell beim Lesen von Programmen könnte eine Steigerung der Kompetenz bedeuten, da erfahrene Programmierer in bekannten Domänen einen Top-down-Ansatz verwenden. Daneben könnten auch die Klausurergebnisse mehrerer Jahre gespeichert und später betrachtet werden und somit könnte über mehrere Semester und Jahre hinweg evaluiert werden, zu welchem Effekt Syntaxaufgaben führen.

In einer weiterführenden Studie könnte die Teilnahme am Pre- und Posttest in Datenstrukturen verpflichtend gemacht werden. Dadurch würde sich die Teilnehmerzahl auf etwa 100 Studierende erhöhen. Somit könnte mit einer großen Kontroll- und Testgruppe gearbeitet werden und die Teilnehmer der Syntaxaufgaben könnten randomisiert zugeteilt werden. Daneben könnten für mehrere Programmierkonstrukte Syntaxaufgaben, wie in der Reflektion im Unterabschnitt 7.1.5.4 erläutert, kreiert werden. Somit könnten auch Studierende untersucht werden, die nicht alle Syntaxaufgaben gelöst haben. Außerdem könnte eine größere Stichprobe in ihre Studiengänge, ihre biologischen Geschlechter und ihre Programmiererfahrung unterteilt werden. Anhand dieser Aufteilungen könnte untersucht werden, ob es unterschiedliche Auswirkungen anhand dieser Eigenschaften gibt. Im Abschnitt 2.3 wurde als eine Herausforderung von PA individuelle Schwächen genannt. Falls das explizite Lernen der Syntax weniger leistungsstarke Studierende auf eine Stufe bei ihren Vorkenntnissen setzt, kann dies untersucht werden. Daneben könnten die Programmverständnis- und Programmieraufgaben des Pretests die Programmiererfahrung, anhand der neo-piagetischen Stufen, der Probanden bestimmen. Demnach könnte untersucht werden, welche Auswirkungen Syntaxaufgaben auf die neopiagetische Klassifizierung von Studierenden haben.

Außerdem könnte eine andere Software zur Erstellung der Syntaxaufgaben genutzt werden, die mehr Ereignisse der Studierenden speichern kann, wie etwa die genaue Beantwortungszeit oder die Anzahl an Tastaturanschlägen. Anhand dessen könnte untersucht werden, ob sich bei diesen Aspekten etwas verändert hat. Beispielsweise könnte Phanon [17] eingesetzt werden, jedoch ist diese Software nur in Python und nicht in Java nutzbar.

Als Möglichkeit der Verbesserung des Programmverständnisses könnten auch Syntaxaufgaben für die Entwicklung des Programmverständnisses eingesetzt werden. Hierbei könnte sich an den Ansätzen von Xie et al. und Nelson zum expliziten Lernen der Lesefähigkeiten von Studierenden orientiert werden [45, 46, 76]. Mit einem Fokus auf die Syntax könnte Studierenden vor der Bearbeitung der Syntaxaufgaben bereits syntaktisch korrekter Quellcode gezeigt werden, den sie tracen müssen. Die

Schwierigkeit sollte denen der Syntaxaufgaben entsprechen. Demnach wären Aufgaben wie das Lesen einer Schleife ein möglicher Startpunkt.

Eine andere Option für weitere Studien besteht darin, die Syntaxaufgaben in AuP, dem Vorgänger von Datenstrukturen, einzusetzen. Hierbei handelt es sich um den ersten Programmierkurs, den Studierende an der TUC belegen. Außerdem wird in AuP mit C eine andere Programmiersprache unterrichtet. Bei einem Einsatz in AuP könnten Gemeinsamkeiten und Unterschiede im Vergleich zum Einsatz in Datenstrukturen gezogen werden sowie die Unterschiede zu den Phanon-Studien, welche auch in einem ersten Programmierkurs stattfanden, gezogen werden [17, 18, 21, 67]. Falls dies geschieht, könnten Unterschiede des Effektes von Syntaxaufgaben in einem ersten Programmierkurs und dem darauffolgenden analysiert werden. Dementsprechend könnte evaluiert werden, zu welchem Zeitpunkt der Einsatz von Syntaxaufgaben effektiver ist. Hierbei wäre auch eine Unterteilung in die Programmiererfahrung und die Vorkenntnisse der Probanden interessant.

Neben dem Einsatz in AuP könnten die Aufgaben auch an verschiedenen Universitäten sowohl in Deutschland als auch weltweit eingesetzt werden. Dabei könnte bspw. analysiert werden, ob es Unterschiede bei Programmiereinführungskursen mit einem anderen Curriculum gibt. Außerdem könnte hierbei der Einsatz weiterer Programmiersprachen betrachtet werden. Anhand dieser Beispiele ist ersichtlich, dass es eine Vielzahl möglicher weiterer Studien gibt. Deswegen sollte das Ziel sein, die Rolle der Syntaxaufgaben in der Lehre detaillierter zu untersuchen und deren Einsatz in verschiedenen Szenarien zu evaluieren.

Literaturverzeichnis

[1] Altadmri, A., Brown, N.C.: 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education. S. 522–527. ACM, Kansas City Missouri USA (2015), https://doi.org/10.1145/2676723.2677258

[2] Aqeel, A., Peitek, N., Apel, S., Mucke, J., Siegmund, J.: Understanding Comprehension of Iterative and Recursive Programs with Remote Eye Tracking. Proceedings of the Annual Workshop of the Psychology of Programming Interest Group (PPIG) (2021)

[3] Ayres, P.: Can the isolated-elements strategy be improved by targeting points of high cognitive load for additional practice? Learning and Instruction 23, S. 115–124 (2013), https://doi.org/10.1016/j.learninstruc.2012.08.002

[4] Bay, W., Thiede, B., Wirtz, M.: Die Theorie der kognitiven Belastung (Cognitive Load Theory). In: Gretsch, P., Holzäpfel (eds.) Lernen mit Visualisierungen. Erkenntnisse aus der Forschung und deren Implikationen für die Fachdidaktik, S. 123–134. Waxmann Verlag GmbH, Münster, New York (2016)

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

[6] Bennedsen, J., Caspersen, M.E.: Failure rates in introductory programming. ACM SIGCSE Bulletin 39(2), S. 32–36 (2007), https://doi.org/10.1145/1272848.1272879

[7] Bennedsen, J., Caspersen, M.E.: Failure rates in introductory programming: 12 years later. ACM Inroads 10(2), S. 30–36 (2019), https://doi.org/10.1145/3324888

[8] Brooks, R.: Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies 18(6), S. 543–554 (1983), https://doi.org/10.1016/S0020-7373(83)80031-5

[9] Brown, N.C., Altadmri, A.: Investigating novice programming mistakes: educator beliefs vs. student data. In: Proceedings of the tenth annual conference on International computing education research. S. 43–50. ICER '14, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2632320.2632343

[10] Buck, D., Stucki, D.J.: Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. ACM SIGCSE Bulletin 32(1), S. 75–79 (2000), https://doi.org/10.1145/331795.331817

[11] Böhm, C., Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM 9(5), S. 366–371 (May 1966), https://doi.org/10.1145/355592.365646

[12] The Joint Task Force on Computing Curricula, C.: Computing curricula 2001. Journal on Educational Resources in Computing 1(3es), S. 1–es (2001), https://doi.org/10.1145/384274.384275

[13] Denny, P., Luxton-Reilly, A., Tempero, E.: All syntax errors are not equal. In: Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. S. 75–80. ITiCSE '12, Association for Computing Machinery, New York, NY, USA (2012), https://doi.org/10.1145/2325296.2325318

[14] Denny, P., Luxton-Reilly, A., Tempero, E., Hendrickx, J.: Understanding the syntax barrier for novices. In: Proceedings of the 16th annual joint conference on Innovation and technology in computer science education. S. 208–212. ITiCSE '11, Association for Computing Machinery, New York, NY, USA (2011), https://doi.org/10.1145/1999747.1999807

[15] Du Boulay, B.: Some Difficulties of Learning to Program. Journal of Educational Computing Research 2(1), S. 57–73 (1986), https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[16] Ebert, M.W.: Webbasierte Ad-hoc-Programmieraufgaben zur Vermittlung von grundlegenden Konzepten der Programmierung in Vorlesungen. Diss. (2018)

[17] Edwards, J., Ditton, J., Trninic, D., Swanson, H., Sullivan, S., Mano, C.: Syntax Exercises in CS1. In: Proceedings of the 2020 ACM Conference on International Computing Education Research. S. 216–226. ICER '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3372782.3406259

[18] Edwards, J., Fulton, E., Holmes, J., Valentin, J., Beard, D., Parker, K.: Separation of syntax and problem solving in Introductory Computer Programming (2018), https.//doi.org/10.1109/FIE.2018.8658852

[19] Garner, S.: Reducing the Cognitive Load on Novice Programmers. Association for the Advancement of Computing in Education (AACE) (2002)

[20] Gaweda, A.M., Lynch, C.F., Seamon, N., Silva de Oliveira, G., Deliwa, A.: Typing Exercises as Interactive Worked Examples for Deliberate Practice in CS Courses. In: Proceedings of the Twenty-Second Australasian Computing Education Conference. S. 105–113. ACE'20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3373165.3373177

[21] Gonzales, S.: An In-Depth Look at Learning Computer Language Syntax in a High-Repetition Practice Environment. All Graduate Theses and Dissertations (2021), https://doi.org/10.26076/b01c-6c80

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

[23] Informatik, F.: Studienordnung für den konsekutiven Studiengang Informatik für Geistes- und Sozialwissenschaftler mit dem Abschluss Master of Science (M.Sc.) an der Technischen Universität Chemnitz vom 8. Juni 2011 (2011)

[24] Jackson, J., Cobb, M., Carver, C.: Identifying Top Java Errors for Novice Programmers. S. T4C–T4C (2005), https://doi.org/10.1109/FIE.2005.1611967

[25] Jadud, M.C.: A First Look at Novice Compilation Behaviour Using BlueJ. Computer Science Education 15(1), S. 25–40 (2005), https://doi.org/10.1080/08993400500056530

[26] Jadud, M.C.: An Exploration of Novice Compilation Behaviour in BlueJ. Diss., University of Kent (2006)

[27] Kummerfeld, S., Kay, J.: The Neglected Battle Fields of Syntax Errors. In: ACE (2003)

[28] Leiner, D.: SoSci Survey (Version 3.1.06) (2019)

[29] Leinonen, A., Nygren, H., Pirttinen, N., Hellas, A., Leinonen, J.: Exploring the Applicability of Simple Syntax Writing Practice for Learning Programming. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. S. 84–90. ACM, Minneapolis MN USA (2019), https://doi.org/10.1145/3287324.3287378

[30] Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B., Thomas, L.: A multi-national study of reading and tracing skills in novice programmers.

ACM SIGCSE Bulletin 36(4), S. 119–150 (2004), https://doi.org/10.1145/1041624.1041673

[31] Lister, R., Fidge, C., Teague, D.: Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. ACM SIGCSE Bulletin 41(3), S. 161–165 (Jul 2009), https://doi.org/10.1145/1595496.1562930

[32] Luxton-Reilly, A.: Learning to Program is Easy. In: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education. S. 284–289. ACM, Arequipa Peru (2016), https://doi.org/10.1145/2899415.2899432

[33] Luxton-Reilly, A., Becker, B.A., Cao, Y., McDermott, R., Mirolo, C., Mühling, A., Petersen, A., Sanders, K., Simon, Whalley, J.: Developing Assessments to Determine Mastery of Programming Fundamentals. In: Proceedings of the 2017 ITiCSE Conference on Working Group Reports. S. 47–69. ITiCSE-WGR '17, Association for Computing Machinery, New York, NY, USA (Jan 2018), https://doi.org/10.1145/3174781.3174784

[34] Luxton-Reilly, A., Petersen, A.: The Compound Nature of Novice Programming Assessments. In: Proceedings of the Nineteenth Australasian Computing Education Conference. S. 26–35. ACE '17, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3013499.3013500

[35] Ly, A., Edwards, J., Liut, M., Petersen, A.: Revisiting Syntax Exercises in CS1. In: Proceedings of the 22st Annual Conference on Information Technology Education. S. 9–14. SIGITE '21, Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3450329.3476855

[36] Mannila, L., Peltomäki, M., Salakoski, T.: What About a Simple Language? Analyzing the Difficulties in Learning to Program. Computer Science Education 16 (2006), https://doi.org/10.1080/08993400600912384

[37] Mason, R.: Designing introductory programming courses: the the role of cognitive load. Diss. (2012)

[38] Mayrhauser, A.v., Vans, A.M.: From Program Comprehension to Tool Requirements for an Industrial Environment. In: In Proceedings of IEEE Workshop on Program Comprehension. S. 78–86. IEEE Comp. Soc. Press (1993)

[39] McCall, D., Kölling, M.: Meaningful categorisation of novice programmer errors. In: 2014 IEEE Frontiers in Education Conference (FIE) Proceedings. S. 1–8 (2014), https://doi.org/10.1109/FIE.2014.7044420

[40] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.D., Laxer, C., Thomas, L., Utting, I., Wilusz, T.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In: Working group reports from ITiCSE on Innovation and technology in computer science education. S. 125–180. ITiCSE-WGR '01, Association for Computing Machinery, New York, NY, USA (2001), https://doi.org/10.1145/572133.572137

[41] Mciver, L.: The Effect of Programming Language on Error Rates of Novice Programmers. In: Paper presented at 12th Annual Workshop of Psychology of Programmers Interest Group (PPIG. Cozenza, Italy (2000)

[42] Medeiros, R.P., Ramalho, G.L., Falcão, T.P.: A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. IEEE Transactions on Education 62(2), S. 77–90 (2019), https://doi.org/10.1109/TE.2018.2864133

[43] Mucke, J., Schwarzkopf, M., Siegmund, J.: REyeker: Remote Eye Tracker. In: ACM Symposium on Eye Tracking Research and Applications. S. 1–5. ACM, Virtual Event Germany (2021), https://doi.org/10.1145/3448018.3457423

[44] Munson, J.P., Zitovsky, J.P.: Models for Early Identification of Struggling Novice Programmers. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. S. 699–704. SIGCSE '18, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3159450.3159476

[45] Nelson, G.L.: Teaching and Assessing Programming Language Tracing. Diss. (2021)

[46] Nelson, G.L., Xie, B., Ko, A.J.: Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In: Proceedings of the 2017 ACM Conference on International Computing Education Research. S. 2–11. ACM, Tacoma Washington USA (2017), https://doi.org/10.1145/3105726.3106178

[47] Obaidellah, U., Al Haek, M.: Evaluating gender difference on algorithmic problems using eye-tracker. S. 1–8 (2018), https://doi.org/10.1145/3204493.3204537

[48] Oliver, D., Dobele, T.: First Year Courses in IT: A Bloom Rating. Journal of Information Technology Education: Research 6, S. 347–360 (2007), https://doi.org/10.28945/220

[49] Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., Paterson, J.: A survey of literature on the teaching of introductory programming. In: Working group reports on ITiCSE on Inno-

vation and technology in computer science education. S. 204–223. ITiCSE-WGR '07, Association for Computing Machinery, New York, NY, USA (2007), https://doi.org/10.1145/1345443.1345441

[50] Powers, K., Ecott, S., Hirshfield, L.M.: Through the looking glass: teaching CS0 with Alice. In: Proceedings of the 38th SIGCSE technical symposium on Computer science education. S. 213–217. SIGCSE '07, Association for Computing Machinery, New York, NY, USA (2007), https://doi.org/10.1145/1227310.1227386

[51] Qian, Y., Lehman, J.: Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. ACM Transactions on Computing Education 18(1), S. 1–24 (2017), https://doi.org/10.1145/3077618

[52] Raigoza, J.: A study of students' progress through introductory computer science programming courses. In: 2017 IEEE Frontiers in Education Conference (FIE). S. 1–7 (2017), https://10.1109/FIE.2017.8190559

[53] Robins, A., Rountree, J., Rountree, N.: Learning and Teaching Programming: A Review and Discussion. Computer Science Education 13(2), S. 137–172 (2003), https://doi.org/10.1076/csed.13.2.137.14200

[54] Robins, A.V.: Novice Programmers and Introductory Programming. In: Robins, A.V., Fincher, S.A. (eds.) The Cambridge Handbook of Computing Education Research, S. 327–376. Cambridge Handbooks in Psychology, Cambridge University Press, Cambridge (2019)

[55] Robins, A.V., Margulieux, L.E., Morrison, B.B.: Cognitive Sciences for Computing Education. In: Robins, A.V., Fincher, S.A. (eds.) The Cambridge Handbook of Computing Education Research, S. 231–275. Cambridge Handbooks in Psychology, Cambridge University Press, Cambridge (2019)

[56] Salleh, S.M., Shukur, Z., Judi, H.M.: Scaffolding Model for Efficient Programming Learning Based on Cognitive Load Theory S. 77–83 (2018)

[57] Schulte, C.: Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In: Proceedings of the Fourth international Workshop on Computing Education Research. S. 149–160. ICER '08, Association for Computing Machinery, New York, NY, USA (Sep 2008), https://doi.org/10.1145/1404520.1404535

[58] Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., Paterson, J.H.: An introduction to program comprehension for computer science educators. In: Proceedings of the 2010 ITiCSE working group reports. S. 65–86. ITiCSE-WGR '10, Association for Computing Machinery, New York, NY, USA (Jun 2010), https://doi.org/10.1145/1971681.1971687

[59] Sharafi, Z., Sharif, B., Guéhéneuc, Y.G., Begel, A., Bednarik, R., Crosby, M.: A practical guide on conducting eye tracking studies in software engineering. Empirical Software Engineering 25(5), S. 3128–3174 (2020), https://doi.org/10.1007/s10664-020-09829-4

[60] Shneiderman, B.: Teaching programming: A spiral approach to syntax and semantics. Computers & Education 1(4), S. 193–197 (1977), https://doi.org/10.1016/0360-1315(77)90008-2

[61] Siegmund, J.: Framework for Measuring Program Comprehension. Diss. (2012)

[62] Siegmund, J.: Program Comprehension: Past, Present, and Future. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). S. 13–20. IEEE, Suita, Osaka, Japan (2016), http://ieeexplore.ieee.org/document/7476769/

[63] Siegmund, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring and modeling programming experience. Empirical Software Engineering 19(5), S. 1299–1334 (2014), http://link.springer.com/10.1007/s10664-013-9286-4

[64] Sleeman, D., Putnam, R.T., Baxter, J., Kuspa, L.: Pascal and High School Students: A Study of Errors. Journal of Educational Computing Research 2(1), S. 5–23 (1986), https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77

[65] Sorva, J.: Visual Program Simulation in Introductory Programming Education. Diss. (2012)

[66] Stefik, A., Siebert, S.: An Empirical Investigation into Programming Language Syntax. ACM Transactions on Computing Education 13(4), S. 19:1–19:40 (2013), https://doi.org/10.1145/2534973

[67] Sullivan, S., Swanson, H., Edwards, J.: Student Attitudes Toward Syntax Exercises in CS1. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, S. 782–788. Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3408877.3432399

[68] Sweller, J.: Cognitive Load Theory: Recent Theoretical Advances. In: Plass, J.L., Brünken, R., Moreno, R. (eds.) Cognitive Load Theory, S. 29–47. Cambridge University Press, Cambridge (2010), https://www.cambridge.org/core/books/cognitive-load-theory/cognitive-load-theory-recent-theoretical-advances/E61D8A35BFEF87651CD5A6155BE26569

[69] Sweller, J., Merrienboer, J.J.G.v., Paas, F.G.W.C.: Cognitive Architecture and Instructional Design. Educational psychology review 10(3), S. 251–296 (1998), https://doi.org/10.1023/A:1022193728205

[70] Teague, D., Lister, R.: Programming: reading, writing and reversing. In: Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14. S. 285–290. ACM Press, Uppsala, Sweden (2014), http://https://doi.org/10.1145/2591708.2591712

[71] Utting, I., Tew, A.E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersen, M., Kolikant, Y.B.D., Sorva, J., Wilusz, T.: A fresh look at novice programmers' performance and their teachers' expectations. In: Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports. S. 15–32. ITiCSE -WGR '13, Association for Computing Machinery, New York, NY, USA (2013), https://doi.org/10.1145/2543882.2543884

[72] Vihavainen, A., Airaksinen, J., Watson, C.: A systematic review of approaches for teaching introductory programming and their influence on success. In: Proceedings of the tenth annual conference on International computing education research - ICER '14. S. 19–26. ACM Press, Glasgow, Scotland, United Kingdom (2014), https://doi.org/10.1145/2632320.2632349

[73] Vihavainen, A., Paksula, M., Luukkainen, M.: Extreme apprenticeship method in teaching programming for beginners. In: Proceedings of the 42nd ACM technical symposium on Computer science education. S. 93–98. SIGCSE '11, Association for Computing Machinery, New York, NY, USA (2011), https://doi.org/10.1145/1953163.1953196

[74] Weintrop, D., Killen, H., Munzar, T., Franke, B.: Block-based Comprehension: Exploring and Explaining Student Outcomes from a Read-only Block-based Exam. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. S. 1218–1224. SIGCSE '19, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3287324.3287348

[75] Winslow, L.E.: Programming pedagogy; a psychological overview. ACM SIGCSE Bulletin 28(3), S. 17–22 (1996), https://doi.org/10.1145/234867.234872

[76] Xie, B., Loksa, D., Nelson, G.L., Davidson, M.J., Dong, D., Kwik, H., Tan, A.H., Hwa, L., Li, M., Ko, A.J.: A theory of instruction for introductory programming skills. Computer Science Education 29(2-3), S. 205–253 (2019), https://doi.org/10.1080/08993408.2019.1565235

TECHNISCHE UNIVERSITÄT CHEMNITZ

Studentenservice – Zentrales Prüfungsamt

Selbstständigkeitserklärung

Name: Gorgosch Vorname: Dominik geb. am: 06.06.1992 Matr.-Nr.: 478540	**Bitte beachten:** 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: 29.11.2021 Unterschrift: .....

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.