HAW
HAMBURG

Bachelorarbeit

Jan-Niklas Jacobson

Anwendung von Progress-Networks in einer Online-Programmierübungs-Plattform am Beispiel von OPPSEE

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Jan-Niklas Jacobson

Anwendung von Progress-Networks in einer Online-Programmierübungs-Plattform am Beispiel von OPPSEE

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Axel Schmolitzky
Zweitgutachter: Prof. Dr. Philipp Jenke

Eingereicht am: 21. Juni 2022

Jan-Niklas Jacobson

Thema der Arbeit

Anwendung von Progress-Networks in einer Online-Programmierübungs-Plattform am Beispiel von OPPSEE

Stichworte

Progress-Networks, Programmieren, Feedback

Kurzzusammenfassung

Das Verhalten von Studierenden während der Bearbeitung einer Programmieraufgabe kann wichtige Erkenntnisse über Missverständnisse und Probleme im Bezug auf Lernstoff und Aufgabenstellung liefern. Mithilfe von Progress-Networks kann der Fortschritt einer Lerngruppe bei der Bearbeitung einer Aufgabe in einem Diagramm zusammengefasst und analysiert werden. In dieser Arbeit werden Progress-Networks als Analysewerkzeug in eine existierende Online-Programmierübungs-Plattform eingebunden. Die notwendigen Prozesse und dabei auftretenden Herausforderungen werden beschrieben. Der Fokus liegt dabei auf den Anpassungen der existierenden Aufgaben auf der Plattform. Zusätzlich wird eine Bibliothek zur dynamischen Generierung und Zeichnung von Progress-Networks entwickelt und eingebunden.

Jan-Niklas Jacobson

Title of Thesis

Application of Progress Networks in an Online Programming Platform using OPPSEE as an Example

Keywords

Progress Networks, Programming, Feedback

Abstract

The behavior of students during the completion of a programming task can provide important insights into misunderstandings and problems related to the learning material and the task. With the help of Progress Networks, the progression of a learning group through a task can be summarized in a diagram which then can be analyzed. In this thesis, Progress Networks are integrated as an analysis tool into an existing online programming platform. The necessary processes and encountered challenges are described. The focus is on the adaptations of existing tasks on the platform. Additionally, a library for dynamic generation and rendering of Progress Networks is developed and integrated into the platform.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **viii**

1 Einleitung **1**

1.1 Motivation 1

1.2 Fragestellung 2

1.3 Zielsetzung 2

1.4 Gliederung 2

2 Grundlagen **4**

2.1 Zentrale Begriffe 4

2.1.1 Online-Programmierübungs-Plattform 4

2.1.2 Testfall, Test 4

2.1.3 Microservices 5

2.1.4 OPPSEE-spezifische Begriffe 5

2.2 OPPSEE 5

2.2.1 Architektur 6

2.3 Progress-Networks 8

2.3.1 Progress-Network-Score 10

3 Ordnung der Testfälle **11**

3.1 Vorbereitungen 11

3.2 Aufgaben aus Level Eins 12

3.2.1 Aufgabe *Abgerundete Summe* 13

3.2.2 Aufgabe *Nachfolgebuchstabe Zyklisch* 14

3.2.3 Aufgabe *Zweidimensionale Punkte* 14

3.3 Aufgaben aus Level Fünf & Sechs 15

3.3.1 Aufgabe *Bildauswertung* 15

3.3.2 Aufgabe *Game Engine Abstract Monster* 16
3.3.3 Aufgabe *Generics Frachtschiff* 17
3.3.4 Aufgabe *Subtyping Generics Geometrische Körper* 18
3.3.5 Aufgabe *Geometrische Körper Subclassing* 19
3.4 Schwierigkeiten 21

4 Dynamische Generierung & Darstellung der Progress-Networks 22
4.1 Entwicklung einer NPM-Bibliothek 22
4.1.1 Generator 23
4.1.2 Renderer 24
4.2 Einbindung in OPPSEE 26
4.2.1 Änderungen an der Benutzeroberfläche 26
4.2.2 Umformung der Rohdaten 28
4.2.3 Darstellung in der Benutzeroberfläche 32

5 Fazit 34
5.1 Ausblick auf weitere Einsatzmöglichkeiten 35
5.1.1 Analyse des Quellcodes 35
5.1.2 Progress-Networks in weiteren Kontexten 35
5.1.3 Umgang mit mehreren Grader-Units 36
5.1.4 Vergleich von Progress-Networks 37
5.1.5 Darstellung des PN-Scores in der Kursübersichtsseite 37

Literaturverzeichnis 39

Selbstständigkeitserklärung 41

Abbildungsverzeichnis

2.1 Die Architektur von OPPSEE [8] 6

2.2 Ein Beispiel für die Konstruktion eines Progress-Network aus Traces nach McBroom et al. [14] 9

4.1 Ein Progress-Network aus Daten von McBroom et al. [14], welches mit der hier entwickelten Bibliothek gerendert wurde [12, 13] 26

4.2 Beispiel-Progress-Networks für die drei vorgestellten Möglichkeiten mit der Anforderung von Musterlösungen umzugehen. Der *MUSTER*-Knoten repräsentiert das Anfordern der Musterlösung 31

4.3 Ein Screenshot aus dem Admin-Panel in OPPSEE, welcher das dynamisch generierte und gerenderte Progress-Network eingebettet in das OPPSEE-Frontend zeigt 33

Tabellenverzeichnis

3.1 Alle Aufgaben in Level Eins [9]. Die ausgewählten Aufgaben sind markiert. 12
3.2 Alle Aufgaben in Level Fünf & Sechs [9]. Die ausgewählten Aufgaben sind markiert. 15

1 Einleitung

1.1 Motivation

Das Verhalten von Studierenden während der Bearbeitung einer Programmieraufgabe kann wichtige Erkenntnisse über Missverständnisse und Probleme im Bezug auf Lernstoff und Aufgabenstellung liefern [2, 3, 15]. Die Analyse der detaillierten Aufzeichnungen jedes individuellen Studierenden ist jedoch zeitaufwendig und bei großen Gruppen nicht durchführbar. Anfang 2021 stellten McBroom et al. [14] daher die sog. *Progress-Networks* vor. Sie funktionieren als ein Analysewerkzeug für die Auswertung von Studierendendaten in Programmieraufgaben. Diese leicht interpretierbaren Netzwerke fassen den Fortschritt einer Lerngruppe bei einer Programmieraufgabe in einem einzigen Diagramm zusammen und heben insbesondere die Stellen hervor, an denen die Studierenden Schwierigkeiten haben, Fortschritte zu erzielen. In dieser Arbeit werden Progress-Networks als Analysewerkzeug in eine existierende *Online-Programmierübungs-Plattform* (OPP) eingebunden und die dafür notwendigen Prozesse und möglichen auftretenden Probleme untersucht. Als Beispielplattform fungiert dabei das Projekt *OPPSEE* (Online Programming Practice for Software Engineering Education), welches seit 2020 an der Hochschule für Angewandte Wissenschaften Hamburg (HAW Hamburg) entwickelt wird [8, 9, 10]. Dort soll es Studierenden ermöglicht werden, in einer Webapplikation Programmieraufgaben zu lösen und eine automatisierte Rückmeldung zu dem eingereichten Programm zu bekommen. Lehrende sollen in OPPSEE Aufgaben und Kurse erstellen können und die Plattform so als Unterstützung in programmierbezogenen Veranstaltungen nutzen können. Die von McBroom et al. beschriebenen Progress-Networks erlauben einen Einblick in den Lösungsprozess einer Aufgabe seitens der Studierenden. Diese Art der Visualisierung könnte Lehrenden in OPPSEE einen anonymen Leistungsüberblick über einen Kurs geben. Zusätzlich könnten die Netzwerke für die Ersteller/-innen der Aufgaben wertvolles Feedback über Verständlichkeit und Schwierigkeitsgrad bieten.

1.2 Fragestellung

Diese Arbeit befasst sich primär mit der Fragestellung: *Wie können Progress-Networks in eine bestehende Online-Programmierübungs-Plattform eingebunden und dort genutzt werden?* Dies soll am Beispiel von OPPSEE erörtert werden. Eine entscheidende Voraussetzung für die Integration von Progress-Networks in eine OPP ist, dass die Testfälle der Aufgaben geordnet sind [14]. Die Testfälle müssen einen Fortschritt bei der Bearbeitung der Aufgabe repräsentieren, beginnend mit einem Ausgangsprogramm, welches keine Tests besteht, bis hin zu einem voll funktionsfähigen Programm, das alle Tests erfolgreich abschließt. Dies ist derzeit in OPPSEE nicht gegeben. Eine Teilfrage dieser Arbeit ist daher, wie die bestehenden Testfälle angepasst und geordnet werden können, um eine Einführung von Progress-Networks zu ermöglichen. Um Progress-Networks in einer OPP produktiv nutzen zu können, muss zudem eine Umformung der vorhandenen Rohdaten in ein geeignetes Format stattfinden und eine dynamische Generierung und Anzeige erarbeitet werden.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, Progress-Networks in OPPSEE einzubinden und ihren Nutzen zu evaluieren. Mithilfe von Metrikdaten sollen die Netzwerke dynamisch generiert und dann in der Weboberfläche von OPPSEE angezeigt werden. Die Arbeit soll Aufschluss darüber geben, wie Progress-Networks in einer produktiv genutzten Plattform in der realen Welt eingesetzt werden können. Dafür nötige Schritte und Anpassungen sowie etwaige Probleme und Nebeneffekte sollen beleuchtet und diskutiert werden. Schließlich soll ein Ausblick auf weitere Einsatzmöglichkeiten solcher Netzwerke gegeben werden.

1.4 Gliederung

Der verbleibende Teil dieser Arbeit ist wie folgt aufgebaut. In Kapitel 2 werden zunächst einige zentrale Begriffe erläutert, welche für das Verständnis dieser Arbeit von Bedeutung sind. Danach wird eine kurze Einleitung in das OPPSEE-System gegeben, welches in dieser Arbeit als Beispiel für eine reale Programmierplattform fungiert. Weiterhin wird in diesem Kapitel die Grundlagenarbeit von McBroom et al. [14] über Progress-Networks und damit das theoretische Fundament dieser Arbeit vorgestellt.

Kapitel 3 behandelt anschließend die Anpassungen, welche an OPPSEE durchgeführt wurden, um Progress-Networks implementieren zu können, insbesondere die namensgebende Ordnung der Testfälle. Anhand verschiedener Aufgaben werden die Denkprozesse und Überlegungen, die für eine sinnvolle Ordnung relevant sind, detailliert beschrieben. Es wird außerdem beispielhaft auf mögliche Probleme und Hindernisse eingegangen, welche bei diesem Schritt auftreten können.

Darauffolgend wird in Kapitel 4 eine Möglichkeit beschrieben, Progress-Networks dynamisch anhand von Rohdaten zu generieren. Dafür wird eine eigene TypeScript-Bibliothek entwickelt und in OPPSEE eingebunden. Dabei wird auch auf die Umwandlung der Rohdaten zuerst in Traces und dann in eine Netzwerkstruktur eingegangen und der Umgang mit verschiedenen Sonderfällen wie etwa der Kompilierfehler in Bezug auf Progress-Networks diskutiert.

Zum Schluss wird in Kapitel 5 ein Fazit der Arbeit gezogen. Dabei wird primär auf die Nutzbarkeit von Progress-Networks in OPP-Produktivsystemen eingegangen und damit die in Abschnitt 1.2 gestellte Forschungsfrage beantwortet. Zusätzlich wird ein Ausblick auf weitere mögliche Einsatzzwecke von Progress-Networks gegeben.

2 Grundlagen

2.1 Zentrale Begriffe

Im Folgenden werden die für das Verständnis dieser Arbeit wichtigen Begriffe *Online-Programmierübungs-Plattform*, *Testfall* und *Microservices* so definiert, wie sie in dieser Arbeit verwendet werden, sowie einige OPPSEE-spezifische Begriffe erläutert.

2.1.1 Online-Programmierübungs-Plattform

Online-Programmierübungs-Plattform (OPP) bezeichnet in dieser Arbeit eine Plattform, auf der es möglich ist, in einer Webapplikation Programmieraufgaben zu lösen und automatisierte Rückmeldungen zu den geschriebenen Programmen zu bekommen. Eine solche Plattform stellt beispielsweise OPPSEE dar, welche derzeit an der HAW Hamburg entwickelt wird. Diese wird im folgenden Abschnitt 2.2 näher erläutert.

2.1.2 Testfall, Test

Ein *Testfall* oder *(Unit-)Test* bezeichnet in dieser Arbeit einen dynamischen Softwaretest, welcher der Überprüfung einer oder mehrerer Eigenschaften eines Testobjektes dient. Testobjekte sind dabei individuelle, abgrenzbare Teile (Units) von Programmen. In der objektorientierten Programmierung sind dies etwa Klassen oder Methoden. Testfälle sind schnell zu schreiben und auszuführen und werden in OPPs häufig dafür verwendet, Studierenden eine Rückmeldung zu den eingereichten Programmen zu geben [11].

2.1.3 Microservices

Microservices sind ein Architekturmuster in der Softwareentwicklung, bei denen eine komplexe Anwendung in mehrere Dienste (Services) aufgeteilt wird, welche jeweils einen kleinen Teil der Funktionalität anbieten. Diese Dienste kommunizieren untereinander mit sprachunabhängigen Interfaces wie etwa HTTP und sind ansonsten voneinander entkoppelt [16].

2.1.4 OPPSEE-spezifische Begriffe

Im Folgenden werden einige Begriffe erläutert, welche innerhalb von OPPSEE genutzt werden.

- *User* Der Begriff User umfasst alle Anwender/-innen von OPPSEE und damit sowohl Studierende als auch Lehrpersonal und Mitarbeiter/-innen des OPPSEE-Projektes.
- *Aufgabe* Eine Aufgabe repräsentiert eine Programmieraufgabe und alle dazugehörigen Informationen wie Aufgabentext, Quellcode und Testfälle.
- *Kurs* Ein Kurs ist eine Menge von Aufgaben. Eine Aufgabe kann mehreren Kursen zugeordnet werden. Der Fortschritt eines Users bei der Bearbeitung einer Aufgabe wird immer einer Aufgabe-Kurs-Verbindung zugeordnet. Es ist also möglich, dieselbe Aufgabe mehrfach in unterschiedlichen Kursen zu bearbeiten.
- *Kategorie* Eine Kategorie ist eine Menge von Kursen. Sie kann z.B. Kurse einer Programmiersprache oder eines Semesterkurses beinhalten.

2.2 OPPSEE

OPPSEE steht für *Online Programming Practice for Software Engineering Education* und ist eine OPP, welche derzeit an der HAW Hamburg entwickelt wird [8, 9, 10]. Bei OPPSEE ist es wie für eine OPP typisch möglich, online Programmieraufgaben zu absolvieren und dafür eine automatisierte Rückmeldung zu den abgegebenen Programmen zu erhalten. Diese Aufgaben können von Lehrenden erstellt, in Kursen zusammengefasst und als optionale Erweiterung des Lehrangebots bereitgestellt werden. OPPSEE ist also

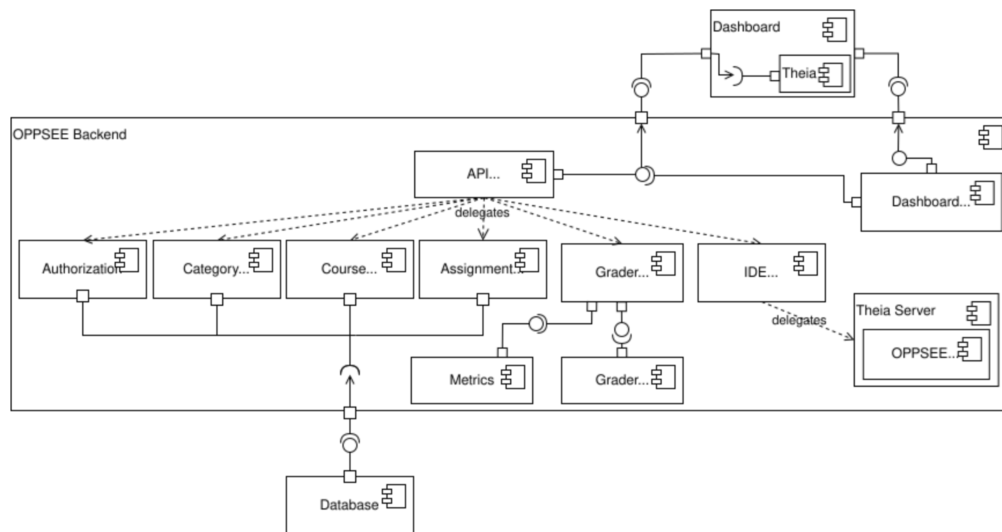

Abbildung 2.1: Die Architektur von OPPSEE [8]

ein freiwilliges Zusatzangebot für Informatik-Studierende an der HAW Hamburg, um die eigenen Programmierfähigkeiten zu verbessern. Der Anspruch, den Lehrbetrieb teilweise oder komplett über die Plattform abzuwickeln, wie bei anderen OPPs [4, 5], besteht bei OPPSEE nicht.

Perspektivisch sollen mithilfe von OPPSEE nicht nur Programmier-, sondern auch andere Software-Engineering-Fähigkeiten der Studierenden verbessert werden. Dazu sollen bei der Rückmeldung neben der funktionalen Qualität einer Lösung auch nicht-funktionale Aspekte wie etwa Code-Lesbarkeit, Architekturpattern und besonders Teamarbeit berücksichtigt werden [7].

2.2.1 Architektur

Die Microservice-Architektur von OPPSEE besteht aus einer Reihe von Services, welche untereinander mit HTTP-Interfaces kommunizieren. Nicht alle dieser Services sind für diese Arbeit von Relevanz. Im Folgenden werden die relevanten Services genauer erläutert.

Dashboard-Service

Das Dashboard ist das Frontend der Plattform und ist mit Vue 2 und TypeScript geschrieben. Auf der Startseite ist eine Liste von Kategorien dargestellt, welche wiederum eine Liste an Kursen enthalten. Bei Klick auf einen Kurs werden die entsprechenden Aufgaben des Kurses aufgelistet. Diese können hier von Studierenden ausgewählt und bearbeitet werden. Die Bearbeitung der Aufgaben und damit der IDE-Teil des Frontends baut auf der Eclipse-Theia-Plattform (kurz Theia) auf. Theia ist eine erweiterbare Plattform zur Entwicklung von Cloud- und Desktop-IDEs, basierend auf modernen Webtechnologien [6]. Damit bietet OPPSEE eine vollständige Entwicklungsumgebung im Browser, welche auf die Bedürfnisse der Plattform zugeschnitten ist.

Für Lehrende gibt es zusätzliche Optionen und Ansichten in der Applikation. So ist es Lehrenden beispielsweise möglich, Aufgaben in Kursen zusammenzufassen, welche dann etwa vorlesungsbegleitend eingesetzt werden können. Die Ansicht für Lehrende beinhaltet auch Statistikseiten für Kurse und einzelne Aufgaben. Hier kann für eine Aufgabe eingesehen werden, wie viele Studierende innerhalb des Kurses bereits an dieser gearbeitet haben. Zusätzlich sind einige simple Metrikdaten dargestellt, wie etwa die Erfolgsquote aller Abgaben oder die Anzahl der durchschnittlich benötigten Abgaben, bis alle Testfälle bestanden sind. Diese Statistikseite soll im Zuge dieser Arbeit durch Progress-Networks ergänzt werden, um Lehrenden die Analyse der Aufgabenbearbeitung zu erleichtern.

Grading-System

Das Grading-System in OPPSEE besteht aus einer erweiterbaren Menge an *Grader-Units* (i.F. nur noch Grader genannt) sowie einer *Grader-Registry*. Ein Grader ist für die Bewertung eines Abgabeversuches einer Aufgabe verantwortlich. Die Registry verwaltet die registrierten Grader und ist für die Wahl der korrekten Grader für einen jeweiligen Abgabeversuch verantwortlich. Dabei gibt es etwa für eine bestimmte Sprache einen eigenen Grader. In der Zukunft könnte es hier beispielsweise auch Unterscheidungen zwischen Unit-Test-Grader und etwaigen Code-Lesbarkeits-Gradern geben. Die klare Trennung erlaubt es, mit geringem Aufwand weitere Grader hinzuzufügen, um so die Funktionalität zu erweitern.

Da sich in dieser Arbeit auf Aufgaben in Java beschränkt wird, ist hier nur der Unit-Test-Grader für Java von Relevanz. In Kapitel 3 werden die notwendigen Änderungen des Graders näher beschrieben.

Metrics-Service

Der Metrics-Service stellt für das Speichern und Abfragen von verschiedenen Metrikdaten Schnittstellen bereit, welche von anderen Services genutzt werden können. Hierzu zählen auch sämtliche Abgabeversuche und deren Ergebnisse sowie Musterlösungsanforderungen. Diese Informationen werden für das Generieren von Progress-Networks benötigt, sodass dieser Service für diese Arbeit von besonderer Bedeutung ist.

2.3 Progress-Networks

Diese Arbeit baut zu großen Teilen auf einer Veröffentlichung der Entwickler von der australischen OPP *Grok Academy* auf. McBroom et al. [14] beschreiben dort eine Visualisierung in Form von sogenannten Progress-Networks, welche als Analysewerkzeug für die Auswertung von Studierendendaten in Programmieraufgaben funktionieren. Sie basieren auf *Interaction-Networks*, welche das Verhalten der Studierenden während der Lösung einer Aufgabe zusammenfassen. Um ein solches Interaction-Network aufzubauen, wird zuerst das Verhalten der Studierenden als sog. *Traces* aufgezeichnet. Ein Trace ist dabei eine Liste von aufeinanderfolgenden Zuständen des Quellcodes. Sind für alle Studierenden einer Gruppe Traces aufgezeichnet worden, können diese in ein Interaction-Network zusammengeführt werden. Dabei erhält man einen Pseudographen, bei welchem alle unterscheidbaren Zustände durch einen Knoten und Übergänge zwischen den Zuständen als gewichtete Kanten dargestellt werden. Mithilfe des Kantengewichts wird die Häufigkeit eines Übergangs abgebildet. Eine der größten Schwierigkeiten von Interaction-Networks ist das *State-Matching*: Funktionell äquivalente Programme können unterschiedlichster Struktur sein, repräsentieren aber möglicherweise trotzdem denselben Zustand und sind daher demselben Knoten zuzuordnen. Ein weiteres Problem ist die fehlende direkte Verbindung zwischen Zustand des Quellcodes und Fortschritt der Aufgabe. Bei der Betrachtung eines Traces ist nicht klar, ob dieser einen geradlinigen Weg zum Erfolg oder eher Schwierigkeiten beim Vorankommen darstellt. McBroom et al. stellen daher die oben genannten Progress-Networks als Alternative vor. Hier ersetzen geordnete Testfälle die

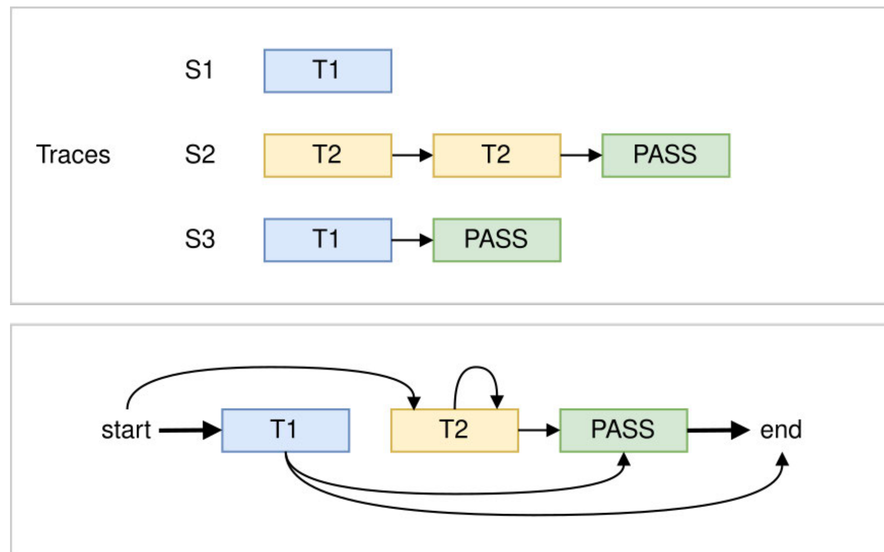

Abbildung 2.2: Ein Beispiel für die Konstruktion eines Progress-Network aus Traces nach McBroom et al. [14]

Zustände des Quellcodes, was die Anzahl der Knoten im Netzwerk stark reduziert. Die Notwendigkeit für State-Matching fällt weg, da die Testfälle klar definiert sind. Die Tests können von Experten geordnet werden, um einen Fortschritt in der Aufgabe bestmöglich abzubilden. Progress-Networks sind für die Visualisierung des Verhaltens von Studierenden in OPPs nützlich, da sie eine große Anzahl an Informationen komprimiert darstellen. Mithilfe von Progress-Networks können Lehrende Situationen identifizieren, bei denen Studierende Probleme haben, Fortschritt zu erzielen. Anschließend können die entsprechenden Gründe etwa durch Quellcodeanalyse ausfindig gemacht werden.

In Abbildung 2.2 ist die Konstruktion eines Progress-Networks aus Rohdaten in Form von Traces dargestellt. Diese bestehen aus einer Reihe von Abgabeversuchen. Jeder Abgabeversuch enthält die Information über den ersten fehlgeschlagenen Test. In den Traces ist zu sehen, dass S1 und S3 bei T1 starten und S2 bei T2. Dies wird im Progress-Network durch die vom Startknoten ausgehenden Kanten und deren Gewicht visualisiert. Ebenfalls zu erkennen ist das wiederholte Fehlschlagen von T2 im Trace von S2. Dies wird durch eine Kante visualisiert, welche T2 als Ursprung sowie als Ziel hat. Der Trace von S1 endet nach dem Fehlschlagen von T1. Die Bearbeitung wird damit als abgebrochen interpretiert. Dies wird durch die Kante repräsentiert, welche von T1 zum Endknoten führt.

2.3.1 Progress-Network-Score

McBroom et al. stellen zusätzlich eine Metrik vor, welche es erlaubt, die Aufgaben zu identifizieren, bei denen Studierende die meisten Probleme haben, Fortschritt zu erzielen. Mit diesem Wert, hier *Progress-Network-Score* (PN-Score) genannt, kann so bei einer Menge an Aufgaben eine Priorisierung nach potenziellem Nutzen einer Progress-Network-Analyse durchgeführt werden. Der PN-Score setzt sich aus der Anzahl von *Back-Steps* und *Repeat-Steps* in einer Aufgabe zusammen:

$$score(x) = back_steps(x) + repeat_steps(x).$$

Hierbei stehen Back-Steps für Abgabeversuche, bei denen mehr Tests fehlschlagen als zuvor und Repeat-Steps für Abgabeversuche, bei denen die selbe Anzahl an Tests fehlschlägt [14]. Der PN-Score gibt damit Aufschluss über die Schwierigkeit einer Aufgabe, welche an der Anzahl an Back- und Repeat-Steps festgemacht werden kann. Gleichzeitig berücksichtigt der PN-Score auch die Anzahl an Studierenden, die eine Aufgabe bearbeiten, denn die obige Gleichung kann auch wie folgt geschrieben werden:

$$score(x) = \frac{back_steps(x) + repeat_steps(x)}{num_students(x)} \times num_students(x).$$

Es haben also die Aufgaben einen hohen PN-Score, welche oft bearbeitet werden und bei denen Studierende Schwierigkeiten bei der Lösung haben. Verbesserungen an diesen Aufgaben haben daher die größte Wirkung. Der PN-Score kann somit zur Priorisierung von zu analysierenden Aufgaben verwendet werden.

Im späteren Verlauf dieser Arbeit werden Anpassungen an der Generierung des Progress-Networks diskutiert. Bei solchen Anpassungen kann es sinnvoll sein, auch die Berechnung des dazugehörigen PN-Score anzupassen. Dies gilt insbesondere dann, wenn weitere Kennzahlen hinzugefügt werden, welche Aufschluss über die Schwierigkeit der Aufgabe geben. Eine Anpassung bzw. Erweiterung des PN-Scores stellt dann sicher, dass dieser weiterhin zur Priorisierung von Aufgaben genutzt werden kann. Hierauf wird später in der Arbeit weiter eingegangen.

3 Ordnung der Testfälle

Um die Verwendung von Progress-Networks zu erlauben, müssen die Tests in OPPSEE geordnet werden, um einen Fortschritt repräsentieren zu können. In diesem Abschnitt werden beispielhaft die Tests einiger Aufgaben aus zwei Kursen in OPPSEE geordnet und wenn nötig umstrukturiert. Dabei werden die notwendigen Schritte beschrieben und auftretende Schwierigkeiten erläutert. Diese sind zu großem Teil allgemeingültig und können auch auf andere OPPs angewendet werden.

3.1 Vorbereitungen

Um eine Ordnung der Aufgaben in OPPSEE zu erlauben, ist keine Anpassungen am Java-Unit-Test-Grader (JUnit-Grader) selbst notwendig. Durch das Einfügen der JUnit4-Annotation

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

an den Kopf der Klasse wird die Ausführung der Testfälle in einer bestimmten Testklasse nach dem Namen aufsteigend sortiert. Eine Benennung der Testmethoden nach dem Muster `testXName`, wobei `X` der Index des Tests und `Name` der Name des Tests ist, bewirkt dann ein Ausführen der Tests in der beabsichtigten Reihenfolge.

Ein Nutzerinterface zur Erstellung von Aufgaben bzw. Testfällen existiert zur Zeit dieser Arbeit noch nicht. Demnach sind hier keine Anpassungen notwendig.

Die im Folgenden bearbeiteten Aufgaben stammen aus den Kursen Level Eins, Level Fünf und Level Sechs. Aus diesen Kursen wurde eine Untersumme an Aufgaben gewählt, welche eine möglichst heterogene Menge an Aufgaben bilden. Der Kurs Level Eins besteht

Aufgabe	# User	Abgabenerfolgsquote in %
Abgerundete Summe	30	16,1
Durchschnitt	26	16,1
Maximum	18	16,7
Maximum 2	17	29,4
Median	19	13,5
Minimum	20	23,4
Nachfolgebuchstabe Zyklisch	17	4,9
Nullstelle berechnen	12	22,0
Zweidimensionale Punkte	15	19,8

Tabelle 3.1: Alle Aufgaben in Level Eins [9]. Die ausgewählten Aufgaben sind markiert.

aus Aufgaben für Programmieranfänger/-innen. Für diese Aufgaben liegen bereits Realdaten vor, welche als Unterstützung für die Ordnung der Testfälle herangezogen werden können. Level Fünf & Sechs richten sich an Studierende im zweiten Semester und enthalten daher komplexere Aufgaben. Die getroffene Auswahl der Aufgaben in den Kursen wird am Anfang des jeweiligen Unterkapitels kurz erläutert.

3.2 Aufgaben aus Level Eins

Im Folgenden werden die Testfälle einiger Aufgaben aus Level Eins geordnet. Level Eins ist der erste Kurs und für Programmieranfänger/-innen, speziell Studierende im ersten Semester, gedacht. Komplexere Aufgaben folgen in Abschnitt 3.3.

Ein Vorteil der Aufgaben aus Level Eins ist, dass hierfür bereits Realdaten aus dem letzten Semester vorliegen, welche über eine logische Reihenfolge der Testfälle Aufschluss geben können. Diese Daten umfassen alle Abgabeversuche von Studierenden aus dem Erstsemesterkurs Programmieren 1 im Wintersemester 2021/2022 an der HAW Hamburg. Sie wurden aus der Datenbank des Metrics-Service exportiert und anschließend mithilfe eines Python-Scripts ausgewertet. Die Anzahl der Studierenden, welche die Aufgaben bearbeitet haben, unterscheidet sich pro Aufgabe. Sie liegt bei den hier bearbeiteten Aufgaben zwischen 15 und 30 Studierenden und kann in Tabelle 3.1 eingesehen werden.

In Tabelle 3.1 sind alle Aufgaben im Kurs Level Eins aufgelistet. Die Abgabenerfolgsquote berechnet sich aus der Gesamtzahl der Abgaben, dividiert durch die Anzahl der erfolgreichen Abgaben, d. h. derjenigen, bei denen alle Tests bestanden wurden. Die im

Folgenden bearbeiteten Aufgaben sind markiert. Die Aufgabe *Abgerundete Summe* wurde ausgewählt, da sie am häufigsten bearbeitet wurde und vom Aufbau her repräsentativ für die unkomplizierteren Aufgaben in Level Eins ist. Die Aufgabe *Nachfolgebuchstabe Zyklisch* ist mit einer Abgabenerfolgsquote von 4,9% die für die Studierenden komplizierteste Aufgabe und daher besonders relevant. Als Drittes wurde die Aufgabe *Zweidimensionale Punkte* ausgewählt, da sie repräsentativ ist für Aufgaben, welche in die objektorientierte Programmierung einführen.

3.2.1 Aufgabe *Abgerundete Summe*

Die Aufgabe *Abgerundete Summe* ist die erste Aufgabe in dem Kurs Level Eins. Hier soll eine Methode vervollständigt werden, welche zwei Parameter addiert und die Summe abgerundet zurückliefert. Ursprünglich existierten hier drei Unit-Tests. Je ein Test für positive und negative Parameter sowie ein Test mit einem positiven und einem negativen Parameter. Diese Tests bilden keinen Fortschritt ab. Um zu erreichen, dass nur einer oder zwei der Tests fehlschlagen, muss eine Differenzierung zwischen positiven und negativen Parametern seitens des Studierenden erfolgen. Dies ist nicht nötig, um die Aufgabe zu lösen und liegt daher nicht auf dem Lösungsweg. Um eine Reihenfolge mithilfe der Tests abzubilden, ist es hilfreich, sich die Tests wie Zwischenstationen auf dem Lösungsweg vorzustellen. Die Lösung eines erfahreneren Programmierers ist hier jedoch so kurz, dass es keinen nennenswerten Weg gibt. Dennoch kann diese Aufgabe in mindestens zwei Teilschritte unterteilt werden: Die Addition der Parameter und das Abrunden der dadurch entstandenen Summe. Für diese Teilschritte kann je ein Test erstellt werden. Der erste Test prüft, ob die Parameter korrekt addiert werden. Dabei werden nur Zahlen als Parameter übergeben, welche summiert eine ganze Zahl ergeben und so auf ein Testen des Abrundens verzichtet. Der zweite Test stellt dann sicher, dass ein Abrunden erfolgt. Eine etwaige Lösung, welche die Java-Methode `Math.round` nutzt, würde beispielsweise den ersten Test erfolgreich bestehen, aber bei dem zweiten Test durchfallen, da sie kaufmännisch rundet, anstatt abzurunden.

Bei der Auswertung der Realdaten wird ersichtlich, dass vor der Ordnung der Tests das Abrunden der Parameter vor der Summierung der häufigste Fehler innerhalb der kompilierbaren Abgaben war. Diese umgeformten Tests könnten dazu beitragen, diesen Fehler schneller zu finden und zu beheben, da zunächst eine korrekte Summierung der Parameter abgefragt wird. Bei einer Abrundung vor der Summierung würde dieser Test fehlschlagen (Bei bspw. $4.5 + 5.5$ wäre das Ergebnis 9 statt 10).

3.2.2 Aufgabe *Nachfolgebuchstabe Zyklisch*

Die Aufgabe *Nachfolgebuchstabe Zyklisch* behandelt sowohl den Typen `char` als auch Modulo-Arithmetik. Das Ziel ist es, eine Methode zu vervollständigen, welche den n-ten Nachfolger eines Buchstabens zurückgibt. Für `f('a', 1)` soll die Methode also `'b'` ausgeben. Nach dem Buchstaben z soll zyklisch wieder bei a begonnen werden. Für `f('z', 1)` ist die erwartete Ausgabe also `'a'`. Die Realdaten ergaben, dass die größte Schwierigkeit in dieser Aufgabe das Umsetzen der zyklischen Logik ist. Dies kann daraus geschlossen werden, dass 60% der Fehler bei dem ersten Testfall auftraten, welcher einen Buchstaben nach z erwartet, also einen zyklischen Ansatz voraussetzt. An dieser Stelle wurde häufig das Zeichen { statt a zurückgegeben, welches in der ASCII-Tabelle direkt nach dem Buchstaben z kommt. Der zweithäufigste Fehler (15%) tritt auf, wenn ein Nachfolger gesucht wird, welcher mehr als einen Zyklus weiter ist. Diese zwei häufigsten Fehler stellen damit die Hürden dar, die Studierende oft bei der Bearbeitung dieser Aufgabe haben. Diese Information ist nützlich für die Strukturierung der Tests.

Für diese Aufgabe existiert bisher nur ein einzelner Testfall, der sowohl Nachfolger innerhalb desselben Zyklus und des darauffolgenden Zyklus sowie mehrere Zyklen weiter testet. Um einen Fortschritt besser zu repräsentieren, wird dieser Test in drei Testfälle aufgeteilt. Der erste Testfall prüft Nachfolger im selben Zyklus. Hier ist also noch keine Logik notwendig, welche den Sprung von z zu a unterstützt. Dies wird im zweiten Test sichergestellt. Der dritte Test überprüft, ob auch Nachfolger einige Zyklen weiter noch korrekt zurückgegeben werden.

3.2.3 Aufgabe *Zweidimensionale Punkte*

In der Aufgabe *Zweidimensionale Punkte* soll eine Klasse, welche einen Punkt in einem zweidimensionalen Raum repräsentiert, vervollständigt werden. Zu diesem Zweck muss ein Konstruktor erstellt werden, der zwei Parameter x und y vom Typ `double` entgegennimmt und korrekt ablegt. Weiterhin müssen für x und y Getter geschrieben werden. Diese Aufgabe lässt sich gut in verschiedene Teilschritte aufteilen, welche je mit einem Test überprüft werden. Der erste Test stellt sicher, dass der Konstruktor mit den korrekten Parametern existiert. Der zweite Test überprüft, ob beide Getter existieren und der letzte Test testet die Funktionalität der Klasse, indem mehrere `Punkt`-Objekte erstellt werden und die zurückgegebenen Werte der Getter mit den an den Konstruktor übergebenen Werten verglichen wird.

Level	Aufgabe
5	Bildauswertung
	Game Engine Abstract Monster
	Game Engine Observer
	Generics Frachtschiff
	Subtyping Generics Geometrische Körper
6	Exception Training
	Exceptions Return To Exceptions
	Game Engine Goblin Hydra
	Geometrische Körper Subclassing

Tabelle 3.2: Alle Aufgaben in Level Fünf & Sechs [9]. Die ausgewählten Aufgaben sind markiert.

3.3 Aufgaben aus Level Fünf & Sechs

Im Folgenden werden nun die Testfälle einiger Aufgaben aus den Kursen Level Fünf & Sechs aus OPPSEE umgeformt. Da diese Aufgaben für Studierende im zweiten Semester gedacht sind, sind sie komplexer als die Aufgaben im vorangegangenen Kapitel. Für diese Aufgaben liegen keine Realdaten vor.

In Tabelle 3.2 sind alle Aufgaben aus den Kursen Level Fünf & Sechs aufgelistet. Die im Folgenden bearbeiteten Aufgaben sind markiert. Bis auf die Aufgabe *Game Engine Observer*, welche einige Überschneidungen zu *Game Engine Abstract Monster* aufweist, wurden alle Aufgaben aus Level Fünf bearbeitet. An Stelle von *Game Engine Observer* wurde die Aufgabe *Geometrische Körper Subclassing* aus Level Sechs gewählt.

3.3.1 Aufgabe *Bildauswertung*

In der Aufgabe *Bildauswertung* soll eine Klasse `BildauswertungImpl` geschrieben werden, welche ein Interface `Bildauswertung` implementiert. Die Klasse soll ein Satellitenbild im Infrarotspektrum auswerten, welches durch die Klasse `Infrarotbild` repräsentiert wird. Ein solches Bild besteht aus Infrarotwerten zwischen 0 und 1000, welche in einem zweidimensionalen Koordinatensystem angeordnet sind. Über den Wert der Infrarotstrahlung kann die Art der Wolken im jeweiligen Bereich ermittelt werden. Dabei soll in der Aufgabe zwischen Schnee- und Regenwolken sowie klarem Himmel unterschieden werden. Bei klarem Himmel kann außerdem mithilfe des Wertes die Temperatur der Erdoberfläche berechnet werden. Die entsprechende Formel ist in der Aufgabenstellung

beschrieben. Die zu implementierende Klasse `BildauswertungImpl` soll für ein Infrarotbild den Anteil an Schnee- und Regenwolken sowie klarem Himmel herausfinden und, sofern möglich, die Durchschnittstemperatur berechnen. Diese Werte sollen über jeweilige Funktionen abgefragt werden können. Zusätzlich soll die Übergabe von ungültigen Parametern durch Java-Assertions abgefangen werden.

Für diese Aufgabe bestehen bereits sechs Testfälle, welche verschiedene Funktionalitäten sicherstellen. Vier Fälle testen die Funktionalität der Wolkentypanteile. Dabei je ein Fall für jeden Wolkentyp alleine und ein zusätzlicher Testfall welche ein Infrarotbild mit mehreren Wolkentypen als Parameter an die Testklasse übergibt. Die zweite Funktionalität, die Berechnung der Durchschnittstemperatur, wird in einem weiteren Testfall überprüft. Der letzte Testfall testet dann noch ungültige Parameter wie etwa `null` statt einem Infrarotbild-Objekt oder Infrarotwerte welche außerhalb des Grenzbereiches liegen.

Die Reihenfolge der Tests kann von den Abhängigkeiten zwischen den Funktionalitäten abgeleitet werden. So ist die Erkennung der Wolkentypen eine Voraussetzung für die Berechnung der Durchschnittstemperatur. Daraus kann geschlossen werden, dass die Funktionalität der Wolkentyperkennung und damit auch das Zählen des Anteils vor der Durchschnittstemperaturberechnung getestet werden sollte. Das Testen der Behandlung von ungültigen Parametern ist unabhängig von der restlichen Logik und kann daher sowohl am Anfang als auch am Schluss einsortiert werden.

3.3.2 Aufgabe *Game Engine Abstract Monster*

Die Aufgabe *Game Engine Abstract Monster* erfordert die Implementierung der abstrakten Klasse `AbstractMonster` einer fiktiven Videospiel-Engine. Die Klasse repräsentiert dabei die Basisfunktionalität eines Monsters in der Engine. Ein Monster kann so etwa den Spieler oder andere Kreaturen angreifen sowie angegriffen und geheilt werden. Es sind zwei Interfaces vorgegeben, welche von der Klasse `AbstractMonster` implementiert werden sollen. Zusätzlich sollen ein vorgegebener Konstruktor und eine abstrakte Methode hinzugefügt werden. Außerdem sollen alle möglichen ungültigen Parameter mit Assertions abgefangen werden, um zu verhindern, dass das `AbstractMonster` in einem ungültigen Zustand geführt werden kann. Da die unterschiedlichen Assertions nicht vorgegeben werden, muss selbst erarbeitet werden, was ein ungültiger Zustand bedeuten kann und wie dieser zu verhindern ist.

Die 15 Testfälle dieser Aufgabe sind bereits ausreichend unterteilt und müssen lediglich sortiert werden. Zuerst wird die Initialisierung und die Setter getestet. Diese sind der Grundstein für die weitere Funktionalität. Dann folgen die verschiedenen implementierenden Methoden. Da diese keine Abhängigkeiten untereinander haben ist ihre Reihenfolge irrelevant. Zum Schluss folgt, ähnlich der vorherigen Aufgabe, das Testen der Assertions. Da die Sicherstellung des Vertragsmodells in dieser Aufgabe komplexer ist, hat jede erforderliche Assertion ihren eigenen Testfall. Die Reihenfolge der Assertion-Testfälle gleicht dabei der Reihenfolge der Funktionalitätstests.

3.3.3 Aufgabe *Generics Frachtschiff*

Die Aufgabe *Generics Frachtschiff* behandelt generische Klassen. Das Ziel ist es, eine generische Klasse `Frachtschiff` zu implementieren. Deren generischer Typ muss dabei das Interface `Transporteinheit` implementieren. Dieses ist vorgegeben und wird durch zwei ebenfalls vorgegebene Klassen `Container` und `Stueckgut` implementiert. Mit diesen Klassen wird das zu implementierende `Frachtschiff` getestet. Dafür müssen verschiedene Methoden implementiert werden. Die Funktionalität der Klasse kann in verschiedene Teile aufgeteilt werden:

- *Name* Ein `Frachtschiff` hat einen Namen. Dieser soll über den Konstruktor übergeben werden und mit einem Getter `gibName` wieder abgerufen werden können.
- *Ladeliste* Ein `Frachtschiff` hat eine Liste des generischen Typen, welcher die Fracht repräsentiert. Bei der Erstellung ist diese leer. Auf die Liste soll über `gibLadeliste` von außen zugegriffen werden können. Hierbei ist zu beachten, dass eine Kopie gefordert wird. Eine Veränderung der zurückgegebenen Liste darf die Fracht selbst nicht beeinflussen.
- *Be- und Entladen* Um die Ladeliste von außen zu modifizieren, sollen zwei Methoden `lade` und `loesche` implementiert werden. Beide sollen sowohl einzelne Transporteinheiten als auch eine Liste als Parameter bekommen können.
- *Zuladungsgewicht* Eine `Transporteinheit` hat ein Gewicht. Eine Methode `gibZuladungsgewicht` soll das Gewicht der gesamten Fracht, also aller Transporteinheiten, zurückgeben.

Versucht man diese Funktionalitäten in eine Reihenfolge zu bringen, stellt man fest, dass es keine allgemeingültige Reihenfolge gibt. Die verschiedenen Funktionalitäten sind teilweise voneinander unabhängig. Dies erschwert es für Aufgaben dieser Art eine sinnvolle Ordnung zu finden. Ein Kompromiss für voneinander unabhängige Funktionalitäten ist es, die dazugehörigen Testfälle nach Komplexität der Funktionalitäten zu ordnen. Nach dieser Logik wird zuerst die Namensfunktionalität getestet. Diese ist unabhängig von den anderen Funktionalitäten und einfach implementierbar.

Als zweites wird die Ladeliste getestet. Es ist zu beachten, dass die Funktionen zum Be- und Entladen eng mit dieser verknüpft sind und ein eigenes Testen daher nicht möglich ist. Was aber getestet werden kann, ist die Länge der Liste nach Erstellen des `Frachtschiff`-Objektes. Diese muss dabei leer sein. Die nächsten vier Tests dienen der Überprüfung der Be- und Entladelogik. Beide Methoden werden je mit einzelnen sowie mit einer Liste von Transporteinheiten getestet. Danach wird die Methode `gibZuladungsgewicht` getestet und zum Schluss sichergestellt, dass die in `gibLadeliste` zurückgegebene Liste eine Kopie ist und Änderungen daran die Liste im Objekt selber nicht modifizieren.

Testen der generischen Programmierung

All diese Funktionalitäten lassen sich einfach testen, sind jedoch nicht der Schwerpunkt der Aufgabe. Dieser liegt bei der Umsetzung der Generics selbst und ist somit syntaktischer Natur. Ein Testen ist daher komplex und nicht durch Unit-Tests allein durchführbar. Gleichzeitig ist der Nutzen begrenzt, da die Syntax der generischen Programmierung in Java leicht nachgeschlagen werden kann und moderne IDEs wie OPPSEE oft Hilfestellungen liefern können. Zu bedenken ist dennoch, dass ein Großteil der Fehler in dieser Aufgabe voraussichtlich syntaktischer Natur sein werden und daher meist keinen spezifischen Testfall betreffen. Dieses Thema wird in Unterunterabschnitt 4.2.2 ausführlicher diskutiert.

3.3.4 Aufgabe *Subtyping Generics Geometrische Körper*

In der Aufgabe *Subtyping Generics Geometrische Körper* soll eine statische Hilfsklasse zur Verarbeitung von geometrischen Körpern implementiert werden. Diese soll zwei Methoden erhalten. Die erste Methode heißt `entferneUngueltige` und soll aus einer

übergebenen Liste ungültige Körper, d.h. solche mit einem Volumen kleiner oder gleich null, entfernen. Die zweite Methode ist `sammleDieNKleinsten` und soll als Parameter zwei Listen und eine Integer-Zahl n übergeben bekommen. Aus der ersten Liste soll die Methode die n Körper mit dem kleinsten Volumen in die zweite Liste kopieren, ohne die Listen anderweitig zu modifizieren.

Für diese Aufgaben existierten bisher nur zwei Testfälle, also ein Test pro Methode. Da dies nur ungenügend einen Fortschritt repräsentiert, müssen die Tests aufgeteilt bzw. erweitert werden. Der Testfall, welcher die Methode `entferneUngueltige` testet, kann in zwei Tests aufgeteilt werden. Hierzu ist es hilfreich, die erwartete Funktionalität der Methode in Teilfunktionen herunterzubrechen. Die Methode soll ungültige Körper entfernen. Daraus lässt sich im Umkehrschluss schließen, dass gültige Körper nicht entfernt werden dürfen. Daraus lassen sich zwei Testfälle ableiten. In einem Testfall wird sichergestellt, dass keine gültigen Körper aus der Liste entfernt werden und im anderen, ob ungültige Körper korrekt entfernt werden. Der Test der zweiten Methode kann ebenfalls aufgeteilt werden. Auch hier ist es von Nutzen Teilfunktionen zu identifizieren. Die Hauptfunktion der Methode ist das Kopieren einer Anzahl n an Körpern von der ersten Liste zur zweiten Liste. Zwei weitere Teilfunktionen sind, dass keine der Listen mehr modifiziert werden als nötig. Die erste Liste darf also gar nicht modifiziert werden, und zu Liste zwei dürfen nur die n Körper hinzugefügt werden. Diese Funktionen können dann in drei Testfällen überprüft werden.

Um diese Testfälle nun in eine sinnvolle Reihenfolge zu bringen, bietet es sich an, zuerst die Hauptfunktionen zu testen, und danach die weniger elementaren Nebenfunktionen und Eigenschaften. Die Hauptfunktion der ersten Methode ist das Entfernen ungültiger Körper. Der Testfall, welcher dies sicherstellt, ist also der erste in der Reihenfolge. Der zweite Testfall überprüft dann die daraus abgeleitete zweite Funktion, dass gültige Körper nicht entfernt werden. In der zweiten Methode stellt demnach der erste Testfall sicher, dass die korrekten Körper in der übergebenen Anzahl kopiert werden. Die zwei letzten Testfälle testen dann die oben genannten Teilfunktionen der zweiten Methode.

3.3.5 Aufgabe *Geometrische Körper Subclassing*

In der Aufgabe *Geometrische Körper Subclassing* sollen einige Klassen implementiert werden, welche geometrische Körper repräsentieren. Dabei ist eine abstrakte Klasse Koerper vorgegeben, welche zwei Methoden definiert: `gibVolumen` und `gibOberflaeche`.

Diese wenden Standardformeln an und greifen dabei auf zwei abstrakte Methoden zu: `gibGrundflaeche` und `gibMantelflaeche`. Es sollen fünf Klassen implementiert werden. Die ersten drei Klassen sind `Quader`, `Zylinder` und `Pyramide`. Alle erben von der Klasse `Koerper`. Die Klasse `Pyramide` ist dabei abstrakt, da es keine Formel gibt, um für eine allgemeine `Pyramide` Mantel- und Grundfläche zu berechnen. Daher sollen zwei weitere Klassen für die quadratische Pyramide und die rechteckige Pyramide implementiert werden. Beide erben von der Klasse `Pyramide`.

Für die Aufteilung der Tests in dieser Aufgabe bieten sich zwei Möglichkeiten an. Entweder wird in jedem Testfall eine Methode an allen Klassen getestet, oder jeder Testfall testet alle Methoden einer Klasse. Die bestehende Struktur nutzt die zweite Aufteilung. Diese hat den Vorteil, dass sie die vermutlich natürlichere Umsetzung der Aufgabe unterstützt, wenn davon ausgegangen wird, dass zuerst eine Klasse vervollständigt wird, anstatt eine Methode in jeder Klasse. Daher wird diese Strukturierung beibehalten.

Es existieren also vier Testfälle, die fünfte, abstrakte Klasse `Pyramide` wird in den zwei konkreten Pyramide-Klassen mitgetestet. Diese müssen nun noch geordnet werden. Dafür bietet sich, wie in der vorherigen Aufgabe, eine Ordnung nach Komplexität der Klassen an, da die Reihenfolge der Umsetzung nicht vorgegeben ist. Zuerst wird der Quader getestet, da die Berechnung von Fläche und Volumen die am wenigsten komplexe ist. Als zweites wird die Klasse `Zylinder` getestet. Es folgt die quadratische Pyramide und zum Schluss die rechteckige Pyramide als komplexeste Klasse.

Da für alle Klassen je dieselben vier Methoden getestet werden, kann auch hier eine Ordnung festgelegt werden. Dies ist nicht relevant für die Progress-Networks, da diese auf dem Level der Testfälle erstellt werden, kann aber dennoch für die Rückmeldung an die Studierenden relevant sein, da stets nur die erste falsche Annahme kommuniziert wird. Um auch hier eine Ordnung nach Komplexität zu implementieren, kann das Testen der Methoden geordnet werden. Die nach Komplexität der Berechnung geordneten Attribute sind dabei in aufsteigender Reihenfolge geordnet: Grundfläche, Volumen, Mantelfläche und Oberfläche. So wird zunächst die Funktionalität der einfacheren Methoden getestet.

3.4 Schwierigkeiten

Nachdem nun die Tests einiger Aufgaben in OPPSEE umgeformt und geordnet worden sind, können zwei primäre Schwierigkeiten bei der Anpassung von Tests für die Nutzung mit Progress-Networks identifiziert werden.

- *Kurze Aufgaben* Aufgaben, welche so kurz sind, dass sie keinen nennenswerten Lösungsweg haben, sind nur schwierig auf ein Progress-Network abzubilden. Solche Aufgaben haben oft nur einen Test oder mehrere Tests, welche aber nie bis selten unterschiedliche Ergebnisse haben. Das Resultat ist ein Progress-Network, welches kaum Informationen enthält.

- *Schwierig zu testende Konzepte* Besonders Aufgaben, welche sich an Programmieranfänger/-innen richten, haben als Schwerpunkt häufig Konzepte, welche sich nur schwierig mit Unit-Tests testen lassen, wie z. B. objektorientierte oder generische Programmierung. Diese Aufgaben enthalten meist kaum testbare Logik. Außerdem führt eine fehlerhafte Umsetzung oftmals zu einem Kompilierfehler im Testobjekt oder in der Testklasse.

Am geeignetsten für die Anwendung von Progress-Networks sind demnach Aufgaben, die einen gewissen Grad an Komplexität aufweisen und ihren Schwerpunkt auf Logik haben. Es ist zu beachten, dass sich diese Einschränkungen auf die in dieser Arbeit untersuchte Nutzung von Progress-Networks mit Unit-Tests beziehen. Progress-Networks können auch bei anderen Arten von Tests Anwendung finden, etwa bei solchen, die statische Code-Analyse durchführen. Die einzige Voraussetzung ist, dass eine Menge an geordneten Tests vorliegt, welche einen Fortschritt repräsentieren.

4 Dynamische Generierung & Darstellung der Progress-Networks

Ein essenzieller Teil der Implementierung von Progress-Networks in eine OPP ist die dynamische Generierung der Netzwerke aus den vorliegenden Daten. Im Folgenden wird eine TypeScript-Bibliothek entwickelt, welche aus Rohdaten Progress-Networks generieren kann [12, 13]. Diese wird dann in das Frontend von OPPSEE eingebunden.

4.1 Entwicklung einer NPM-Bibliothek

Da in OPPSEE ein TypeScript-Frontend genutzt wird, bietet sich die Entwicklung einer JavaScript-kompatiblen Bibliothek zur Generierung von Progress-Networks an. So existiert eine klare Schnittstelle zwischen der Umwandlung der OPPSEE-spezifischen Daten und der Generierung von Progress-Networks, welche unabhängig von der Applikation ist. Die Bibliothek ist in TypeScript geschrieben, um Typen für verschiedene Strukturen wie etwa Traces definieren zu können.

Um die Bibliothek in OPPSEE einfach einbinden zu können, ist diese in *NPM* (ehemals Node Package Manager) veröffentlicht [12]. NPM ist ein Paketmanager für die JavaScript-Laufzeitumgebung Node.js. Hier veröffentlichte Pakete bzw. Bibliotheken können in einem beliebigen Node.js-Projekt eingebunden werden. Eine Nutzung der Bibliothek in OPPSEE und ggf. auch anderen OPPs ist somit möglich. Der Quellcode der Bibliothek ist Open Source und kann auf *GitHub* (github.com) eingesehen werden [13].

Die Funktionalität der Bibliothek kann in zwei Hauptfunktionen unterteilt werden: Erstens die Umformung von Traces in eine Struktur, welche ein Progress-Network repräsentiert und zweitens das Darstellen des Netzwerkes. Daher wird die Bibliothek in zwei Komponenten geteilt. Die erste Komponente ist der *Generator* und behandelt die Umformung

der Traces in eine Progress-Network-Struktur. Die zweite Komponente, der *Renderer*, ist für die Darstellung des Netzwerkes zuständig.

4.1.1 Generator

Der Generator erstellt aus einer Liste von Traces ein Netzwerk. Hierfür ist die Definition einiger Typen notwendig. Zuerst sind die Inputtypen zu definieren, also die Typen der Daten, welche in den Generator hereingegeben werden.

- `Attempt` Ein Attempt repräsentiert einen Abgabeversuch. Er enthält ein Flag, ob der Test bestanden wurde. Wurde der Test nicht bestanden, ist zusätzlich noch der Index des fehlgeschlagenen Tests im Attempt enthalten.
- `Trace` Ein Trace enthält eine zeitlich geordnete Liste von Attempts. Der erste Attempt ist in der Liste also an erster Stelle.

Zusätzlich zu der Liste von Traces wird eine geordnete Liste der Testfälle in den Generator gegeben. Dies ist notwendig, da nicht alle Testfälle zwingend in den Attempts enthalten sind. Es ist möglich, dass ein Testfall in keinem Attempt als erstes fehlschlägt und dementsprechend nicht auftaucht. Da hier bereits die Namen aller Testfälle enthalten sind, reicht es im Attempt statt dem Namen nur den Index des Tests mitzugeben.

Aus diesen Inputdaten wird das Netzwerk generiert und vom Generator zurückgegeben. Das Netzwerk wird ebenfalls als Typ definiert und gehört damit zu den Outputtypen.

- `Edge` Eine Edge repräsentiert eine Kante im Netzwerk. Sie enthält den Namen des Ursprung- und Zielknotens sowie ein Gewicht.
- `ProgressNetwork` Das Progress-Network enthält eine geordnete Liste der Namen aller Knoten im Netzwerk und eine Liste der Edges.

Um aus den Inputdaten ein Netzwerk zu generieren, wird als erstes die Liste der Knoten erstellt. Die Knoten sind wie die Testfälle nach Fortschritt geordnet. Die besteht aus dem Startknoten, den geordneten Testfällen, einem Pass-Knoten, wenn eine Abgabe alle Tests besteht und dem Endknoten. Dann werden die Kanten aus den Traces generiert. Dabei wird zunächst für jeden Attempt eine Kante mit dem Gewicht eins erstellt. Der Ursprungsknoten ist dabei entweder der Startknoten oder der Zielknoten des letzten Attempts desselben Traces. Der Zielknoten ist der des fehlgeschlagenen Tests oder der

Endknoten. Für jeden Trace wird außerdem am Ende eine weitere Kante erstellt, welcher vom letzten Zielknoten zum Endknoten führt. Im zweiten Schritt werden alle Kanten mit demselben Ursprungs- und Zielknoten in eine Kante zusammengeführt. Das Gewicht dieser Kante wird dabei durch die Anzahl der zusammengeführten Kanten bestimmt. Anschließend wird das für die Anzeige fertige Netzwerk zurückgegeben.

Berechnung des Progress-Network-Scores

Da die Berechnung des PN-Scores ein wichtiger Teil des Analysewerkzeuges ist, ist es sinnvoll, diese in die Bibliothek zu integrieren. Dafür reicht eine einfache Methode, welche für ein übergebenes Netzwerk der oben eingeführten Struktur den PN-Score berechnet und zurückgibt. Die Berechnung besteht dabei aus zwei Schritten. Im ersten Schritt findet sowohl das Filtern der relevanten Kanten sowie das Abbilden auf das Kantengewicht statt. Dafür werden die Kanten, die Back- oder Repeat-Steps darstellen (bei denen also der Zielindex größer als der Ursprungsindex ist) auf ihr Kantengewicht gesetzt. Die restlichen Kanten werden stattdessen auf null gesetzt und sind somit effektiv herausgefiltert. So kann ein weiteres Iterieren der Kanten vermieden werden. Die resultierende Liste von Zahlen wird dann summiert, um den PN-Score zu erhalten. Anhand dieses Wertes kann so später eine Priorisierung der Analyse stattfinden.

4.1.2 Renderer

Das fertige Netzwerk kann nun von der zweiten Komponente als SVG gerendert werden. Für das Darstellen von Daten in Webseiten gibt es viele Javascript-Bibliotheken. Ein bekannter Vertreter dieser Art ist die Open Source-Bibliothek *D3.js* [1], im folgenden D3 genannt. D3 erleichert die Manipulation des *Document Object Models* (DOM) zur dynamischen Visualisierung von Daten. Es ist am ehesten vergleichbar mit anderen Dokumenttransformatoren wie JQuery oder CSS. Im Gegensatz zu diesen ist D3 jedoch datenbasiert. Das heißt, es ist möglich, Elemente aufgrund von Daten hinzuzufügen, zu entfernen und anzupassen. D3 kann sowohl mit herkömmlichen HTML-Elementen arbeiten, als auch mit SVG-Elementen, welche sich für komplexere Visualisierungen besser eignen. SVG steht für *Scalable Vector Graphics* und ist eine Spezifikation zur Beschreibung zweidimensionaler Vektorgrafiken. Es basiert auf XML und kann von allen relevanten Webbrowsern dargestellt werden. D3 ist daher optimal für das Rendern von Progress-Networks und wird in dieser Arbeit dafür verwendet.

Die Renderer-Komponente bekommt als Input-Parameter ein `ProgressNetwork`, wie es in Unterabschnitt 4.1.1 definiert ist, die gewünschte Höhe und Breite des SVGs sowie die ID eines Elementes im DOM. Als erster Schritt der Generierung wird das `<svg>`-Element mit den übergebenen Maßen erstellt und als Kind des Elements mit der übergebenen ID in den DOM eingefügt. Dann wird eine Gruppe, also ein `<g>`-Element, eingefügt. Dieses fungiert als Wurzelelement und erlaubt es, das Netzwerk am Ende innerhalb des SVGs zu zentrieren.

Der nächste Schritt ist das Rendern der Nodes. Hierfür wird zuerst für jede Node eine Gruppe erstellt, die später den Kreis (das `<circle>`-Element) sowie den Namen des Knotens beinhalten wird. Jede Gruppe wird dabei bereits an die korrekte Position transformiert. Dies geschieht mithilfe der scaleBand-Funktion von D3 und erlaubt ein dynamisches Platzieren der Nodes auf der Grundlage der gegebenen Breite. In jede Node-Gruppe, mit Ausnahme des Start- und Endknotens, wird dann der Kreis gerendert, welcher durch ein `<circle>`-Element dargestellt wird. Schließlich wird ein zentriertes `<text>`-Element mit dem Namen des Knotens hinzugefügt. Die Knoten sind nun fertig gerendert.

Die Darstellung der Kanten ist komplexer. Zunächst wird auch hier wieder für jede Kante eine Gruppe erstellt. Die Reihenfolge ist dabei abhängig von dem Kantengewicht, da eine schwerere Kante stets nach einer leichteren Kante gezeichnet werden sollte. Dadurch wird sichergestellt, dass die wichtigsten Informationen, die durch die Kanten mit dem höchsten Gewicht dargestellt werden, im Vordergrund sind. Als zweites werden die Kanten als `<path>`-Elemente gezeichnet und an die jeweilige Gruppe angehängt. Die Strichstärke und Deckkraft werden dabei von dem Kantengewicht beeinflusst. Kanten mit höherem Gewicht sind breiter und kräftiger. Anschließend wird das Kantengewicht als Text an den Scheitelpunkt der Kante gerendert.

Ganz zum Schluss wird die Wurzelgruppe, welche jetzt das komplette Netzwerk enthält, innerhalb des SVGs zentriert. Das Progress-Network ist damit fertig gerendert und kann auf einer beliebigen Webseite angezeigt werden. Ein fertig gerendertes Beispielnetzwerk ist in Abbildung 4.1 zu sehen.

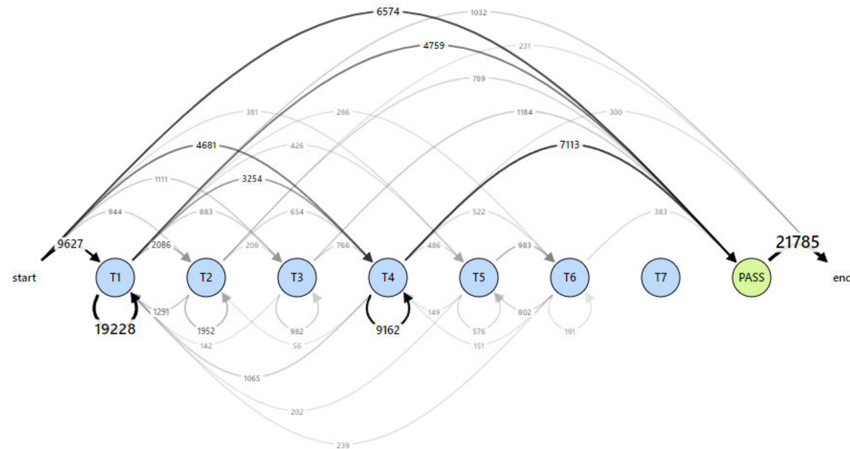

Abbildung 4.1: Ein Progress-Network aus Daten von McBroom et al. [14], welches mit der hier entwickelten Bibliothek gerendert wurde [12, 13]

4.2 Einbindung in OPPSEE

Nachdem die Datenverarbeitung und Visualisierung der neuen Bibliothek fertig implementiert ist, kann diese in OPPSEE als Beispiel-OPP implementiert werden. Dazu sind zunächst einige Änderungen an der Benutzeroberfläche nötig. Darauffolgend müssen die individuell strukturierten Daten dieser spezifischen OPP in allgemeingültige Traces verwandelt werden, aus dem die Bibliothek ein Netzwerk generieren kann.

4.2.1 Änderungen an der Benutzeroberfläche

Es sind einige Änderungen an der Benutzeroberfläche nötig, um generierte Progress-Networks so informativ wie möglich zu machen. Im Folgenden werden diese aufgezählt und begründet.

Anzeige des Ergebnisses des letzten Abgabeversuches

Wird in der IDE in OPPSEE ein Abgabeversuch durchgeführt, so wird die Rückmeldung dieses Versuches auf der rechten Seite angezeigt. Wird die Seite nun aber neu geladen, etwa weil eine längere Pause innerhalb der Bearbeitung stattfindet, ist die Rückmeldung nicht mehr zu sehen, da diese nicht im aktuellen Stand der Bearbeitung gespeichert wird.

Um zu sehen, welche Tests noch fehlschlagen, ist es notwendig einen erneuten Abgabeversuch durchzuführen. Dies wird vom Progress-Network als Repeat-Step interpretiert und verwässert so die Daten, aus welchen das Progress-Network generiert werden soll. Es ist daher von Vorteil für die Aussagekraft des Netzwerkes, das Ergebnis des letzten Abgabeversuchs zu speichern und bei einem erneuten Laden der IDE darzustellen.

Anzeige der durchlaufenen Testfälle

Ursprünglich werden in OPPSEE nach einem Abgabeversuch nur fehlgeschlagene Tests angezeigt. Bestandene Tests sind nicht sichtbar. Gerade bei der nun vorliegenden Ordnung der Testfälle ist dies nicht optimal, da es Studierenden das Einschätzen des eigenen Fortschritts erschwert. Die notwendige Änderung ist damit das Visualisieren von bestandenen Tests.

Im Gegensatz zu McBroom et al., wo die Testausführung unterbrochen wird, sobald ein Test fehlschlägt, werden bei dem JUnit-Grader in OPPSEE stets alle Testfälle ausgeführt. Da Unit-Tests jedoch nach Definition isoliert und unabhängig voneinander sind, ist dies zu vernachlässigen. Wichtig ist allerdings, dass in der Nutzeroberfläche von OPPSEE die Ergebnisse jedes Testfalles angezeigt werden, während bei McBroom et al. naturgemäß nur die Testfälle bis zum ersten fehlgeschlagenen Testfall angezeigt werden. Dies kann Einfluss auf die Reihenfolge der Problemlösung der Studierenden haben. Schlagen bspw. bei einer Aufgabe mit drei Testfällen die Tests zwei und drei fehl, so kann bei einer Anzeige nach McBroom et al. davon ausgegangen werden, dass der Fehler in Test zwei zuerst behoben wird, da die Fehlermeldung aus Test drei nicht einsehbar ist. Werden beide Tests und damit beide Fehler angezeigt, so kann es sein, dass zuerst der Fehler aus Test drei behoben wird. Nimmt man an, dass beide Wege später zur selben Lösung führen, mag dies zuerst nach einem vernachlässigbaren Unterschied klingen. Für die Visualisierung mit Progress-Networks entsteht jedoch ein entscheidender Nachteil. Ein vorgezogenes Beheben von fehlgeschlagenen Testfällen, welche in der Reihenfolge nach einem anderen fehlgeschlagenen Test kommen, ist in einem Progress-Network nicht von wiederholten Lösungsversuchen dieses primären Testfalls zu unterscheiden. Die Folge ist nicht nur ein Informationsverlust, sondern überdies eine Verzerrung. Repeat-Steps können nicht mehr eindeutig einem Testfall zugeordnet werden. Daher ist es von Vorteil, bei einem fehlgeschlagenen Testfall gegebenenfalls existierende nachfolgende Testfälle zu maskieren.

4.2.2 Umformung der Rohdaten

Die Datenstruktur, wie sie in OPPSEE gespeichert wird, muss in Traces umgewandelt werden, damit daraus ein Netzwerk generiert werden kann. Jeder Abgabeversuch wird in OPPSEE in der Datenbank abgelegt. Gespeichert werden dabei Anlegedatum, Studierendenkennung, Kurs, Aufgabe, das Programm zum Zeitpunkt der Abgabe sowie eine Liste der Testberichte. Ein Testbericht besteht aus einem Flag, ob der Test bestanden wurde, dem Bericht, den der Grader zurückgegeben hat, und Informationen zu dem genutzten Grader. Um die Abgaben für eine bestimmte Aufgabe zu erhalten, kann nun einfach nach einer Kurs-Aufgaben-Kombination gefiltert werden. Für die Generierung eines Progress-Networks wird nur ein Bruchteil der gespeicherten Informationen benötigt, ebenso muss die Struktur der Daten angepasst werden. Diese Umwandlung geschieht dabei im OPPSEE-Frontend, bevor die Daten von dem Progress-Network-Generator der Bibliothek in die finale Progress-Network-Struktur transformiert werden.

Im ersten Schritt der Umwandlung muss eine Gruppierung der Abgabeversuche nach Studierenden stattfinden, um die Daten in Traces nach McBroom et al. zu gruppieren. Informationen über die Studierenden selbst werden dabei nicht benötigt. Ein Trace besteht lediglich aus einer Liste von Abgabeversuchen einer einzelnen Person. Ein Abgabeversuch muss dabei nur die Information enthalten, ob der Versuch erfolgreich war, also alle Tests bestanden wurden, und gegebenenfalls zusätzlich den ersten Test, welcher fehlgeschlagen ist.

Die Liste der Tests wird vorher herausgeschrieben, da sie individuell als eigener Parameter in den Progress-Network-Generator gegeben wird. In den Traces muss ein Abgabeversuch dann nur noch den Index des fehlgeschlagenen Tests in der Testliste enthalten. Die für die Liste benötigten Titel der Tests sind nicht explizit in einem Testbericht-DTO enthalten und müssen aus dem HTML-Code des Berichts extrahiert werden. Da die Testtitel zu lang sind, um sie wortwörtlich in das Netzwerk zu übernehmen, werden sie vorher auf Abkürzungen (T1, T2, etc.) abgebildet, wie sie bereits in den Beispielnetzwerken in dieser Arbeit genutzt wurden (siehe Abbildung 2.2, Abbildung 4.2). Eine Legende mit einer Liste der Testtitel und der dazugehörigen Abkürzung wird neben dem Netzwerk angezeigt, um die Auswertbarkeit nicht zu beeinträchtigen.

Um aus der Liste aller Abgabeversuche Traces zu extrahieren, werden diese zunächst nach dem Erstelldatum aufsteigend sortiert. Die Liste wird dann durchiteriert, wobei die Abgabeversuche in eine Map herausgeschrieben werden, welche die Studierenden-ID

als Key und den dazugehörigen Trace als Value hat. Zu beachten ist, dass nach einem erfolgreichen Abgabeversuch alle weiteren Abgabeversuche des jeweiligen Studierenden ignoriert werden müssen, da die Aufgabe hier bereits als erfolgreich gelöst gilt. Ist die Liste der Abgabeversuche vollständig verarbeitet, wird aus den Values der entstandenen Map eine Liste an Traces generiert, welche später als Parameter in den Progress-Network-Generator gegeben wird.

Sonderfall Kompilierfehler

Fehler, die bereits während des Kompilierens ausgelöst werden, unterscheiden sich von Laufzeitfehlern dadurch, dass sie auftreten, bevor das Programm und damit auch die Unit-Tests ausgeführt werden können. Es schlägt also kein individueller Test fehl, stattdessen wird eine Fehlermeldung des Compilers ausgegeben. McBroom et al. diskutieren diesen Sonderfall, und wie dieser innerhalb von Progress-Networks behandelt wird, nicht. Möglicherweise wurden Kompilierfehler bereits vorher aussortiert oder dem ersten Testfall zugeordnet. Es wird klar, dass es verschiedene Möglichkeiten gibt, mit Kompilierfehlern umzugehen:

- a) *Aussortieren* Alle Kompilierfehler werden aussortiert und sind daher im Progress-Network nicht sichtbar.
- b) *Testfall 1 zuordnen* Alle Kompilierfehler werden dem ersten Testfall zugeordnet. Dies entspricht dabei der Logik, dass tatsächlich alle Tests fehlschlagen, wenn das Programm nicht kompiliert werden kann.
- c) *Zusätzlicher Testfall* Alle Kompilierfehler werden einem zusätzlichen Testfall zugeordnet, welcher vor den Unit-Testfällen steht.

Ein Aussortieren der Kompilierfehler hat einen Informationsverlust zur Folge: Fast die Hälfte aller Fehler in der ersten Aufgabe *Abgerundete Summe* sind Kompilierfehler. Es kann jedoch dagegen argumentiert werden, dass ein bloßes Darstellen dieser Fehler keinen Mehrwert bietet, da es unterschiedlichste Gründe für Fehler zur Kompilierzeit geben kann, die aus einem Progress-Network nicht ersichtlich werden. Es können keine Rückschlüsse über beispielsweise missverständliche Aufgabenstellungen oder Gebiete, bei denen Studierende Schwierigkeiten haben, gezogen werden. Höchstens der Grad der syntaktischen Bewegungssicherheit der Studierenden wird dadurch sichtbarer. Dies ist jedoch

auch durch ein Anzeigen der Anzahl an Kompilierfehlern zu erreichen. Kompilierfehler sind speziell für Programmieranfänger eine nicht unbedeutende Hürde, wie etwa die Analysen des Datenerfassungsprojektes Blackbox zeigen [2, 3]. Andere Arbeiten, wie etwa von Qian und Lehman [15], argumentieren dagegen, dass syntaktische Fehler zwar die häufigsten unter Studierenden seien, aber trotzdem unbedeutend, oberflächlich und leicht zu beheben wären. Im Kontext der Progress-Networks könnte dies bedeuten, dass ein Einbinden der Kompilierfehler wichtigere Informationen verwässert. Auch diskutabel ist, ob b) und c) die Repräsentation des Fortschritts durch die Testfälle unterstützen. Angenommen ein/-e Student/-in hat ein Programm, welches zwei von drei Tests besteht. Im Laufe der Behebung des Fehlers wird ein Semikolon vergessen. Bei dem nächsten Abgabeversuch kompiliert dieses Programm nun nicht mehr. Der Fehler ist leicht ersichtlich und schnell behoben. Trotzdem wird dies als Back-Step gewertet. Einerseits kann argumentiert werden, dass das sonst korrekte Programm auch mit Syntaxfehler näher an einer fertigen Lösung ist als ein Programm, welches nur zwei von drei Unit-Tests besteht, andererseits besteht ein nicht kompilierendes Programm streng genommen keinerlei Tests, da es überhaupt nicht lauffähig ist.

In dieser Arbeit wurde sich für Möglichkeit a) entschieden, auch wenn b) und c) ebenfalls denkbar gewesen wären. Dennoch ist es mit wenig Aufwand möglich, die Implementierung auf eine andere Variante umzubauen oder sogar mehrere Varianten zu ermöglichen, zwischen denen dann in einem User-Interface gewählt werden könnte. So kann die Relevanz der Kompilierfehler, die bei Programmieranfängern höher ist als bei fortgeschritteneren Entwickler/-innen, von den Lehrenden selbst eingeschätzt werden.

Sonderfall Musterlösungen

In OPPSEE ist es möglich, innerhalb der Web-IDE eine von OPPSEE bereitgestellte Musterlösung anzufordern. Ähnlich wie bei den Kompilierfehlern in Unterunterabschnitt 4.2.2 gibt es auch hier verschiedene Möglichkeiten, mit der Nutzung dieser Funktion umzugehen:

- a) *Ignorieren* Das Anfordern der Musterlösung wird ignoriert. Die Aufgabe kann danach normal gelöst werden, ohne dass das Anfordern der Musterlösung im jeweiligen Trace und damit im Progress-Network ersichtlich ist.

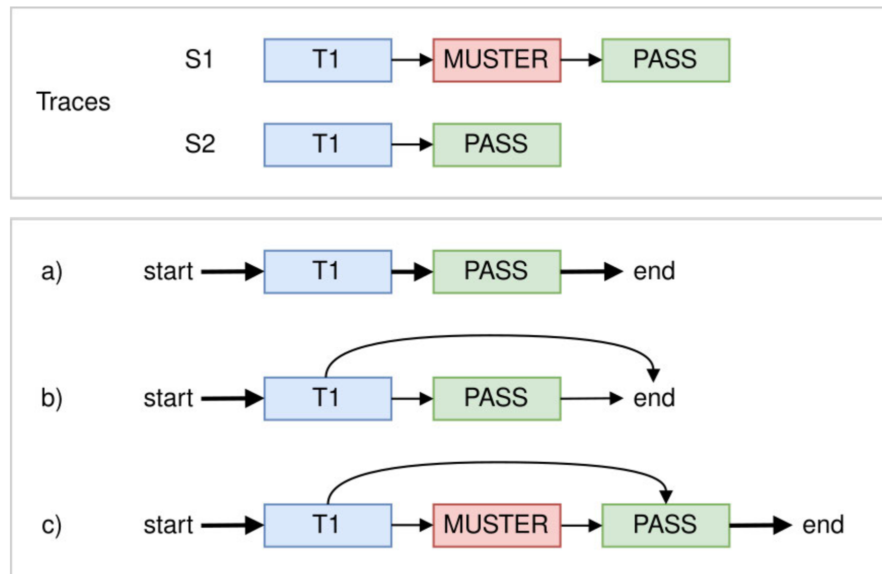

Abbildung 4.2: Beispiel-Progress-Networks für die drei vorgestellten Möglichkeiten mit der Anforderung von Musterlösungen umzugehen. Der *MUSTER*-Knoten repräsentiert das Anfordern der Musterlösung

- b) *Als Trace-Ende interpretieren* Das Progress-Network interpretiert das Anfordern der Musterlösung als Abbruch der Bearbeitung (Kante zum Endknoten). Alle darauffolgenden Abgabeversuche werden dementsprechend ignoriert.
- c) *Zusätzlicher Knoten* Das Anfordern der Musterlösung wird innerhalb des Progress-Networks mit einer Kante zu einem neuen Knoten *Musterloesung* visualisiert. Für die Kante des nächsten Abgabeversuches ist dieser demnach der Ursprungsknoten.

Die drei Möglichkeiten sind in Abbildung 4.2 anhand je eines Progress-Networks dargestellt. Jedes der Netzwerke ist dabei aus denselben Trace-Daten S1 und S2 erstellt worden. Sie unterscheiden sich nur im Umgang mit Anforderungen der Musterlösung.

Möglichkeit a) ist die einfachste, da sie keinerlei weitere Anpassungen benötigt. Sie hat allerdings den entscheidenden Nachteil, dass Lösungen, für welche die Musterlösung genutzt wurde, nicht von regulären Lösungen zu unterscheiden sind. Eine Möglichkeit, um dem entgegenzuwirken, wäre es, den Anteil der Studierenden, welche die Musterlösung vor der Beendigung der Aufgabe angefordert haben, als eine zusätzliche Kennzahl anzuzeigen. Möglichkeit b) filtert die Bearbeitungen nach der Musterlösungsanforderung aus dem Progress-Network heraus. Dadurch ist die Informationsquantität geringer, aber die

Qualität besser, da sicherer ist, dass der im Diagramm visualisierte Fortschritt selbst erarbeitet ist. Die dritte Möglichkeit c) ist in der Umsetzung am aufwendigsten, allerdings gehen hier am wenigsten Informationen verloren.

In dieser Arbeit wurde sich für die Möglichkeit c) entschieden um so viele Informationen wie möglich zu erhalten. Um die Musterlösungen in die Progress-Networks miteinzubinden, müssen zunächst das Anfordern der Musterlösung mit in die Traces eingearbeitet werden. Dafür werden die Musterlösungsanforderungen zusätzlich zu den Abgabeversuchen vom Grader-Service geladen. Diese werden dann zu einer Liste zusammengeführt und nach dem Erstelldatum aufsteigend sortiert. Diese Liste wird iteriert, um die Traces für jeden Nutzer herauszuschreiben. Die Testliste muss dafür um einen neuen Knoten erweitert werden, wie in Abbildung 4.2 c) gezeigt. Die Musterlösungsanforderungen werden dann ähnlich wie ein fehlgeschlagener Abgabeversuch behandelt, bei dem der Index des ersten fehlgeschlagenen Tests der des *MUSTER*-Knotens ist. Dabei muss beachtet werden, dass nur die jeweils erste Anforderung der Musterlösung einer Studentin oder eines Studenten in den Trace geschrieben wird. Jede folgende Anforderung hat keinen Einfluss auf die Bearbeitung der Aufgabe und sollte daher ignoriert werden, um ein Verwässern des Progress-Networks zu vermeiden.

Da Progress-Networks und der Progress-Network-Score eine Einheit bilden, sollte bei Anpassungen am Netzwerk auch über die Anpassung des PN-Scores nachgedacht werden. Eine hohe Anzahl an Anforderungen der Musterlösung lässt auf Schwierigkeiten mit der Aufgabe schließen. Um das Netzwerk besser abzubilden, wird daher zusätzlich zu den Back- und Repeat-Steps die Anzahl der Musterlösungsanforderungen (nur max. eine Anforderung pro Person wird gezählt) zum Progress-Network-Score addiert.

4.2.3 Darstellung in der Benutzeroberfläche

Entsprechen die fertigen Traces und die Liste der Testnamen dann den benötigten Input-Typen des Generators, kann die Bibliothek das Netzwerkobjekt generieren. Aus diesem kann dann der PN-Score berechnet werden, welcher in OPPSEE ergänzend neben dem Netzwerk angezeigt wird. Schließlich kann die Renderer-Komponente mithilfe des Netzwerkobjekts das fertige Netzwerk rendern. Dafür wird neben Breite und Höhe sowie dem Netzwerkobjekt noch die ID eines HTML-Elements mitgegeben, unter welchem das Netzwerk schließlich in das *Document Object Model* eingefügt wird. Dies geschieht auf

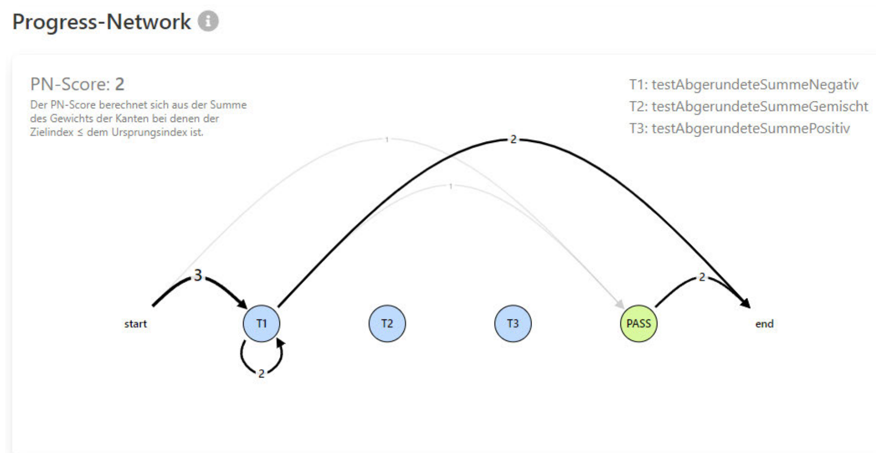

Abbildung 4.3: Ein Screenshot aus dem Admin-Panel in OPPSEE, welcher das dynamisch generierte und gerenderte Progress-Network eingebettet in das OPPSEE-Frontend zeigt

der Statistikseite einer jeweiligen Aufgabe im Kontext eines Kurses innerhalb des Admin-Panels von OPPSEE und reiht sich damit in andere Metriken ein, welche zur Bewertung der Aufgabe und den Studierenden nützlich sind. In Abbildung 4.3 ist ein Screenshot eines dynamisch generierten und gerenderten Progress-Networks im Admin-Panel zu sehen. Durch Schweben mit dem Mauszeiger über dem Info-Icon kann ein Informationstext aufgerufen werden, welcher die Funktionsweise der Progress-Networks in einigen Sätzen erklärt.

5 Fazit

In dieser Arbeit wurde demonstriert und erläutert, wie die Progress-Networks aus der Arbeit von McBroom et al. [14] als Werkzeug zur Analyse von Bearbeitungen von Programmieraufgaben durch Studierende in OPPs genutzt werden können. Die dafür notwendigen Änderungen an der Plattform wurden am Beispiel von OPPSEE beschrieben. Der wichtigste Teil ist dabei die Anpassungen der Aufgaben und insbesondere der Ordnung der Testfälle. Diese müssen notwendigerweise einen Fortschritt bei der Bearbeitung der Aufgabe repräsentieren können. Die Ordnung der Testfälle und Anpassung von Aufgaben wurde an verschiedenen Beispielaufgaben in OPPSEE demonstriert. Weiterhin wurde eine Bibliothek in TypeScript entwickelt, mit deren Hilfe es möglich ist, zur Laufzeit Rohdaten im Trace-Format nach McBroom et al. in eine Netzwerkstruktur zu bringen und diese als Progress-Network zu visualisieren. Die Bibliothek ist allgemein gehalten und kann so in eine beliebige OPP (mit einem JavaScript- oder TypeScript-Frontend) eingebunden werden. Die Einbindung wurde am Beispiel von OPPSEE demonstriert und kann dort im Produktivbetrieb genutzt werden. Die Implementierung von Progress-Networks in OPPSEE hat gezeigt, dass eine Integration von zur Laufzeit generierten Progress-Networks in Produktivsystemen möglich ist. Der Aufwand ist dabei stark abhängig von der Anzahl an existierenden Aufgaben, da die dazugehörigen Testfälle von Experten geordnet werden müssen.

Eine Limitierung dieser Arbeit ist das Fehlen von Realdaten, welche nach der Integration von Progress-Networks in OPPSEE aufgezeichnet wurden. Dies ist einerseits dem Zeitplan dieser Arbeit und andererseits der noch wenigen Nutzer von OPPSEE geschuldet. Dies führt dazu, dass die Änderungen an den Aufgaben und Testfällen nicht auf ihren Einfluss in die Bearbeitung ebendieser durch Studierende überprüft werden können. Auch kann deshalb keine Auswertung von Realdaten mithilfe der Progress-Networks in OPPSEE stattfinden.

5.1 Ausblick auf weitere Einsatzmöglichkeiten

Im Folgenden wird ein Ausblick auf weitere Einsatzmöglichkeiten von Progress-Networks in OPPs gegeben, auf die in dieser Arbeit bisher nicht eingegangen wurde.

5.1.1 Analyse des Quellcodes

Mithilfe von Progress-Networks ist es zwar möglich, Situationen zu identifizieren, in denen Studierende Schwierigkeiten haben, Fortschritt zu erzielen, jedoch kann eine solche Identifikation nur auf der Granularität des jeweiligen Testfalls erfolgen. Für eine detaillierte Analyse des Ursprungs dieser Schwierigkeiten ist ein Einblick in den Quellcode hilfreich. Dies ist momentan in OPPSEE nur manuell durch Zugriff auf die Datenbank möglich. Für eine nahtlose Integration von Progress-Networks in OPPSEE wäre es besser, wenn die Analyse der entsprechenden Stellen im Quellcode direkt innerhalb der Nutzeroberfläche von OPPSEE stattfinden könnte. So könnte sich bei einem Klick auf einen Testfall-Knoten im Netzwerk beispielsweise die Entwicklungsumgebung mit einem zufällig ausgewählten Quellcode eines Studierenden öffnen, bei welchem der jeweilige Test fehlgeschlagen ist. Alternativ könnte aus einer Liste der Programme, bei welchen dieser Test fehlgeschlagen ist, gewählt werden. So könnte die komplette Analyse nahtlos in OPPSEE stattfinden.

5.1.2 Progress-Networks in weiteren Kontexten

Progress-Networks wurden in dieser Arbeit, wie von McBroom et al. [14] definiert, als Werkzeug zur Visualisierung von Fortschritt einer Menge von Studierenden bei der Bearbeitung von Programmieraufgaben beschrieben. Auf den ersten Blick erscheinen aber noch weitere Einsatzmöglichkeiten im Kontext von OPPs und insbesondere OPPSEE denkbar. Als naheliegendes Beispiel kann dafür etwa die Anwendung von Progress-Networks im Kontext von OPPSEE-Kursen genannt werden. Hier wäre etwa ein zusätzliches Progress-Network auf Kursebene vorstellbar, welches den Fortschritt innerhalb eines Kurses visualisiert. Bei näherer Betrachtung zeigen sich hier jedoch einige Probleme. Zuallererst muss sich bewusst gemacht werden, welche Informationen mit Progress-Networks gewonnen werden sollen. Primäres Ziel ist es, Schwierigkeiten von Studierenden bei der Bearbeitung einer Aufgabe zu erkennen und zu spezifizieren. Dieses Ziel lässt sich auf Kurse übertragen. Auch hier kann es helfen, zu erfahren, bei welchen Assignments (im

folgenden Absatz wird für OPPSEE-Aufgaben der englische Begriff Assignment genutzt, um eine Verwechslung mit einer allgemeinen Lernaufgabe, auf dessen Fortschritt sich ein übliches Progress-Network bezieht, zu vermeiden) innerhalb eines Kurses die größten Schwierigkeiten auftreten. Allerdings kann argumentiert werden, dass dies bereits durch den PN-Score abgebildet wird, welcher die Schwierigkeiten mit einer Aufgabe auf eine Metrik reduziert. Abgesehen von dem Nutzen steht auch die Umsetzung vor einigen Hindernissen. So können Assignments innerhalb eines Kurses semantisch nicht eins zu eins auf Testfälle innerhalb einer Aufgabe abgebildet werden. Während Testfälle im Kontext von Progress-Networks als Indikator für den Fortschritt bei der Entwicklung des jeweiligen Programmes fungieren, sind Aufgaben komplett voneinander unabhängige Teilaufgaben, welche einzeln bearbeitet werden müssen. Die Bearbeitung eines Kurses setzt damit die Bearbeitung eines jeden enthaltenen Assigments voraus, während die Bearbeitung einer Aufgabe ununterbrochen und vollständig geschehen kann und durch die Testfälle lediglich auf ihre Korrekt- und Vollständigkeit überprüft wird. Dies zieht einige Probleme mit sich. So muss verhindert werden, dass Studierende Aufgaben in einer anderen Reihenfolge als der gewollten abschließen, damit die Assignments im Kurs einen Fortschritt darstellen können. Das wiederum bedeutet, dass es keine Back-Steps im Netzwerk geben kann, da jedes Assignment vor dem nächsten komplett gelöst werden muss. Die resultierenden Progress-Networks im Kurskontext würden sich also zu einem großen Teil von üblichen Progress-Networks unterscheiden.

5.1.3 Umgang mit mehreren Grader-Units

Zum Zeitpunkt dieser Arbeit wird in OPPSEE nur eine Grader-Unit produktiv eingesetzt: Der Java-Unit-Test-Grader (JUnit-Grader). Es befindet sich jedoch auch bereits ein Unit-Test-Grader für die Sprache C in Entwicklung. Weitere Grader, wie etwa zur Auswertung der Code-Lesbarkeit oder Architektur, sind geplant. Die in dieser Arbeit beschriebene Umformung der Rohdaten in Traces behandelt die Datenstruktur, wie sie vom JUnit-Grader gespeichert wird. Es ist jedoch möglich, dass die Ergebnisse von Unit-Test-Gradern anderer Sprachen eine davon abweichende Struktur aufweisen. Langfristig ist eine Modularisierung der Rohdatenumformung in Traces daher ein notwendiger Schritt für die Nutzung von Progress-Networks in OPPSEE. Ein weiterführendes Forschungsthema ist außerdem die Integration der Ergebnisse von anderen Grader-Typen in Progress-Networks oder verwandte Visualisierungen.

5.1.4 Vergleich von Progress-Networks

McBroom et al. stellen in ihrer Arbeit eine Möglichkeit vor, zwei Progress-Networks miteinander zu vergleichen. Sie nutzen dafür ein dreistufiges Verfahren. Zunächst werden die Kantengewichte der zwei jeweiligen Netzwerke normalisiert, indem jedes Gewicht durch die Gesamtzahl der Abgabeversuche geteilt wird. Zweitens werden die Knoten der beiden Netzwerke abgeglichen, bspw. auf der Grundlage semantischer Äquivalenz. Dieser Schritt fällt weg, wenn zwei Progress-Networks derselben Aufgabe verglichen werden, etwa mit unterschiedlichen Gruppen von Studierenden. Als Drittes wird dann die Differenz der normalisierten Kantengewichte berechnet. Daraus ergibt sich ein neues Progress-Network mit Kantengewichten zwischen -1 und 1, i.F. Differenznetzwerk genannt. Positive Werte repräsentieren hier solche Kanten, welche im ersten Netzwerk häufiger vorkommen, und negative Werte repräsentieren Kanten, die im zweiten Netzwerk häufiger vorkommen. Dies lässt sich veranschaulichen, indem die Kanten umso breiter gezeichnet werden, je weiter ihr Gewicht von 0 entfernt ist und für positive und negative Werte unterschiedliche Farben gewählt werden.

Diese Differenznetzwerke können etwa genutzt werden um zwei Gruppen von Studierenden (bspw. zwei Semestergruppen) miteinander zu vergleichen, um Auswirkungen von Änderungen an Aufgaben oder sogar der Plattform selbst zu überprüfen (z.B. im Zuge der Validierung von A/B-Tests) oder um die Performance einer Gruppe von Studenten in ähnlichen Aufgaben miteinander zu vergleichen.

Die Progress-Network-Generator-Bibliothek könnte zukünftig erweitert werden, um das Vergleichen zweier Netzwerke zu unterstützen. Eine beispielhafte Erweiterung könnte aus Folgendem bestehen: Erstens einem weiteren Netzwerktypen, der als Kantengewicht nur Werte von -1 bis 1 unterstützt, zweitens eine Comparer-Komponente, welche aus zwei normalen Progress-Networks ein solches Differenznetzwerk-Objekt generiert und schließlich einer Anpassung der Renderer-Komponente, um auch diese Differenznetzwerke korrekt darstellen zu können.

5.1.5 Darstellung des PN-Scores in der Kursübersichtsseite

Der PN-Score ist nach McBroom et al. [14] eine Metrik, die es Lehrenden erlaubt, die Aufgaben im Hinblick auf Progress-Networks zu identifizieren, bei denen Studierende die

meisten Probleme haben, Fortschritt zu erzielen. Dafür ist es jedoch notwendig, die verschiedenen Scores auf einen Blick miteinander zu vergleichen, oder sogar die Aufgaben nach dem PN-Score sortieren zu können. Momentan wird der Score jedoch lediglich neben dem Netzwerk auf der Detail-Statistikseite einer Aufgabe angezeigt, nicht auf der Kursübersichtsseite. Das Darstellen des PN-Score auf dieser Seite hätte den Vorteil, dass auf den ersten Blick die Aufgaben mit den höchsten PN-Scores, und damit die Aufgaben, auf denen der Fokus der Auswertung liegen sollte, identifiziert werden könnten. Hier zeigt sich jedoch eine Schwachstelle der Generierung der Progress-Networks im Frontend, also im Browser der Seitenbesucher/-innen. Die Generierung des Netzwerkes ist nötig, um den PN-Score zu berechnen. Er kann zwar auch aus den Rohdaten kalkuliert werden, dies erfordert jedoch einen deutlich komplexeren und im Bezug zur Generierung des Netzwerkes redundanten Algorithmus. Beide dieser Möglichkeiten erfordern die Umformung von Rohdaten während dem Aufruf der Kursübersichtsseite. In Anbetracht der hier möglichen Datenmengen könnte das an dieser Stelle einen unverhältnismäßig großen Rechenaufwand bedeuten. Diese Schwierigkeit ist ein Argument für eine Berechnung der Netzwerkstruktur im Backend, beispielsweise in einem zu OPPSEEs Microservice-Architektur passenden eigenen Service. Das Ergebnis kann so zudem auf Serverseite für mehrere Nutzer zwischengespeichert werden, um den Rechenaufwand zu verringern. Anstelle der Rohdaten könnten folglich die individuellen Scores eines gesamten Kurses von einem eigenen Endpunkt abgefragt werden. Für jede Aufgabe könnten dann die jeweiligen Progress-Network-Daten in der bereits fertigen Netzwerkstruktur vom Service geladen werden. Die Frontend-Bibliothek wäre demnach nur noch für das Rendern des Netzwerkes zuständig.

Literaturverzeichnis

[1] Bostock, Michael ; Ogievetsky, Vadim ; Heer, Jeffrey: D3: Data-Driven Documents. In: *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). – URL http://vis.stanford.edu/papers/d3

[2] Brown, Neil C. ; Altadmri, Amjad: Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. New York, NY, USA : Association for Computing Machinery, 2014 (ICER '14), S. 43–50. – URL https://doi.org/10.1145/2632320.2632343. – ISBN 9781450327558

[3] Brown, Neil Christopher C. ; Kölling, Michael ; McCall, Davin ; Utting, Ian: Blackbox: A Large Scale Repository of Novice Programmers' Activity. (2014), S. 223–228. – URL https://doi.org/10.1145/2538862.2538924. ISBN 9781450326056

[4] Code FREAK: *Coding Education Platform that supports Teachers and Students*. – URL https://codefreak.org. – Zugriffsdatum: 17.12.2021

[5] Codeboard.io: *A web-based IDE to teach programming in the classroom*. – URL https://codeboard.io. – Zugriffsdatum: 17.12.2021

[6] Eclipse Theia: *Eclipse Theia*. – URL https://projects.eclipse.org/projects/ecd.theia/. – Zugriffsdatum: 09.05.2022

[7] Gandrass, Niels ; Hinrichs, Torge ; Schmolitzky, Axel: Towards an Online Programming Platform Complementing Software Engineering Education. In: *Software Engineering im Unterricht der Hochschulen 2020* Bd. 2531, RTWH Aachen, 2020, S. 27–35. – URL http://hdl.handle.net/20.500.12738/12559. – ISSN 1613-0073

[8] HAW Hamburg: *HAW Hamburg GitLab - OPPSEE*. – URL https://git.haw-hamburg.de/oppsee. – Zugriffsdatum: 02.06.2022

[9] HAW Hamburg: *OPPSEE.* – URL https://oppsee.haw-hamburg.de/. – Zugriffsdatum: 07.06.2022

[10] HAW Hamburg: *OPPSEE - Online Programming Practice for Software Engineering Education.* – URL https://oppsee.informatik.haw-hamburg.de/. – Zugriffsdatum: 02.01.2022

[11] Ihantola, Petri ; Ahoniemi, Tuukka ; Karavirta, Ville ; Seppälä, Otto: Review of Recent Systems for Automatic Assessment of Programming Assignments. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research.* New York, NY, USA : Association for Computing Machinery, 2010 (Koli Calling '10), S. 86–93. – URL https://doi.org/10.1145/1930464.1930480. – ISBN 9781450305204

[12] Jan-Niklas Jacobson: *@jnjacobson/progress-network-generator - npm.* – URL https://www.npmjs.com/package/@jnjacobson/progress-network-generator. – Zugriffsdatum: 02.06.2022

[13] Jan-Niklas Jacobson: *jnjacobson/progress-network-generator: Generate progress networks from raw data..* – URL https://github.com/jnjacobson/progress-network-generator. – Zugriffsdatum: 07.06.2022

[14] McBroom, Jessica ; Paassen, Benjamin ; Jeffries, Bryn ; Koprinska, Irena ; Yacef, Kalina: Progress Networks as a Tool for Analysing Student Programming Difficulties. In: *Australasian Computing Education Conference.* New York, NY, USA : Association for Computing Machinery, 2021 (ACE '21), S. 158–167. – URL https://doi.org/10.1145/3441636.3442366. – ISBN 9781450389761

[15] Qian, Yizhou ; Lehman, James: Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. In: *ACM Trans. Comput. Educ.* 18 (2017), oct, Nr. 1. – URL https://doi.org/10.1145/3077618

[16] Ren, Zhongshan ; Wang, Wei ; Wu, Guoquan ; Gao, Chushu ; Chen, Wei ; Wei, Jun ; Huang, Tao: Migrating Web Applications from Monolithic Structure to Microservices Architecture. In: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware.* New York, NY, USA : Association for Computing Machinery, 2018 (Internetware '18). – URL https://doi.org/10.1145/3275219.3275230. – ISBN 9781450365901

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original