Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jennifer Momsen

Evaluierung des Spoon-Frameworks zur Erstellung einer API für Whitebox-Tests in der Programmierlehre

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jennifer Momsen

Evaluierung des Spoon-Frameworks zur Erstellung einer API für Whitebox-Tests in der Programmierlehre

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Schmolitzky
Zweitgutachter: Prof. Dr. Schäfers

Eingereicht am: 6. Mai 2019

Jennifer Momsen

Thema der Arbeit
Evaluierung des Spoon-Frameworks zur Erstellung einer API für Whitebox-Tests in der Programmierlehre

Stichworte
Java, Whitebox-Test, statische Code-Analyse, Spoon-Framework, API-Design

Kurzzusammenfassung
Diese Arbeit beschäftigt sich mit der Entwicklung einer API zur Erstellung von Whitebox-Tests. Diese Whitebox-Tests sollen genutzt werden um den Quelltext von Studierenden, in praktischen Programmierprüfungen, zu untersuchen. Zunächst wird hierfür eine Anforderungsanalyse erstellt. Anschließend wird das Spoon-Framework vorgestellt, welches es ermöglicht Quelltexte zu analysieren. Mit den Möglichkeiten des Spoon-Frameworks und den Erkenntnissen aus der Anforderungsanalyse, wird dann eine eigene API für Whitebox-Tests entwickelt.
Jennifer Momsen

Title of the paper
Evaluating the Spoon-Framework to develop an API for Whitebox-tests to be used in programming education

Keywords
Java, Whitebox-Test, static Code-Analysis, Spoon-Framework, API-Design

Abstract
This thesis is about developing an API to implement Whitebox-Tests. These tests are used to analyse sourcecode, which is written by students in practical programming exams. At first there will be a requirement analysis on which functions are needed. Then the Spoon-Framework will be introduced, which has the ability to do static code-analysis. In the end both the Spoon-Framework and the results of the requirement analysis will be used to develop an API for Whitebox-tests.

Inhaltsverzeichnis

1. Einleitung **1**
- 1.1. Motivation und Ziel . . . 1
- 1.2. Themenabgrenzung . . . 2
- 1.3. Struktur . . . 2

2. Grundlagen API-Design **3**
- 2.1. Vor- und Nachteile einer API . . . 3
- 2.2. Gutes API-Design . . . 4
  - 2.2.1. Klassen-Design . . . 4
  - 2.2.2. Methoden-Design . . . 5
  - 2.2.3. Exception-Design . . . 5

3. Anforderungsanalyse **7**
- 3.1. Vorgehen . . . 7
- 3.2. Gemeinsamkeiten . . . 8
- 3.3. Unterschiede . . . 9
- 3.4. Weitere Anforderungen . . . 11
- 3.5. Zusammenfassung der Analyse . . . 11

4. Das Spoon-Framework **13**
- 4.1. Überblick . . . 13
- 4.2. Das Spoon Metamodell . . . 14
  - 4.2.1. Strukturelemente im Metamodell . . . 15
  - 4.2.2. Quelltextelemente im Metamodell . . . 15
  - 4.2.3. Referenzen im Metamodell . . . 16
  - 4.2.4. Graphische Darstellung des Metamodells . . . 17
- 4.3. Abfragen von Quelltext-Elementen . . . 19
  - 4.3.1. Getter . . . 19
  - 4.3.2. Filter . . . 20
  - 4.3.3. Weitere Möglichkeiten . . . 20

5. Die Cutlery Bibliothek **22**
- 5.1. Aufbau . . . 22
  - 5.1.1. Die Context-Klasse . . . 22
  - 5.1.2. Die Tester-Klassen . . . 23
  - 5.1.3. Die Methoden . . . 23

5.2. Proof of Concept 24
5.2.1. Hintergrund 25
5.2.2. Einbinden der Cutlery.jar in BlueJ 25
5.2.3. Tests schreiben mit Cutlery 25

6. Schluss **31**
6.1. Fazit 31
6.2. Ausblick 32

A. Inhalt der beiliegenden CD **33**

B. Benutzung von Cutlery in der Eclipse IDE **34**

Darstellungsverzeichnis

3.1. Übersicht Gemeinsamkeiten 9
3.2. Übersicht Anforderungen 12

4.1. Überblick statische Code-Analyse in Spoon 14
4.2. Ausschnitt der Strukturelemente 15
4.3. Ausschnitt der Quelltextelemente 16
4.4. Graphische Darstellung des Metamodells 17
4.5. Graphische Darstellung des Metamodells mit gefüllten Methodenrumpf 18

5.1. Übersicht Klassen mit Methoden der Cutlery-API 24
5.2. Geöffnetes Projekt in BlueJ 26
5.3. Die hinzugefügte Cutlery.jar 26

Listings

4.1. Befehl zur Erstellung der graphischen Darstellung des Metamodells 17
4.2. Hello World Beispiel 1 17
4.3. Hello World Beispiel 2 18
4.4. Anwendung von Getter-Methoden 19
4.5. NamedElementFilter in Spoon 20
4.6. TypeFilter in Spoon 20
4.7. Anwendung von Queries mit Java 8 Lambdas (Inria-Spoon, 2019) 20
4.8. Anwendung vom CtIterator (Inria-Spoon, 2019) 21

5.1. Erstellung eines Context-Objektes 23
5.2. Erzeugung eines FieldTesters 23
5.3. Import von Cutlery 26
5.4. Erstellung des Contexts und der benötigten Tester 27
5.5. Klasse implementiert gewünschtes Interface 28
5.6. Test Enum vorhanden 28
5.7. Enum wird als Parameter in Methode verwendet 29
5.8. Klasse hat Zustand und ein Feld benutzt *HashMap* 30
5.9. Alle Felder in Klasse *private* und ein Feld vom Typ *Map* existiert 30

1. Einleitung

In diesem Kapitel wird das Thema und das Ziel dieser Arbeit vorgestellt. Außerdem wird aufgezeigt, welche Themen in dieser Arbeit behandelt werden und welche nicht. Am Ende des Kapitels folgt dann eine Übersicht zur Struktur der Arbeit.

1.1. Motivation und Ziel

Um ihr Programmierwissen zu erweitern und zu festigen, können Informatikstudierende der HAW Hamburg Selbsttests absolvieren. Diese Tests werden online zur Verfügung gestellt und sind ein freiwilliges Angebot. Des Weiteren gilt es praktische Programmierklausuren (im Folgenden Laborprüfungen genannt) zu absolvieren. In beiden Fällen gibt es Aufgaben zur Lese- und zur Schreibkompetenz von Quelltexten. Um die Lesekompetenz der Studenten zu trainieren, werden Fragen zu vorgegebenen Quelltextfragmenten gestellt. Außerdem wird das allgemeine Verständnis der jeweiligen Programmiersprache überprüft. In den Aufgaben zur Schreibkompetenz müssen die Studierenden selber Quelltexte schreiben. Die Anforderungen an die Studierenden in den Laborprüfungen können von Professor zu Professor unterschiedlich sein, da jeder seine Vorlesung und Prüfung individuell gestaltet. Um zu überprüfen, ob die Anforderungen erfüllt sind, werden von jedem Professor automatisierte Tests verwendet. Dabei wird in Blackbox- und Whitebox-Tests unterschieden. Beim Blackbox-Testing wird das Verhalten der Anwendung getestet. Der Quelltext des Studierenden wird hierbei außer Acht gelassen, es wird lediglich ermittelt, ob die Anforderungen erfüllt wurden (vgl. Olsen u. a., 2018, S. 87). Es wird zum Beispiel geschaut, ob die Eingabe von gültigen Parametern zum erwarteten Ergebnis führt (Positivtest) oder ob bei Eingabe von ungültigen Parametern eine entsprechende *Exception* geworfen wird (Negativtest).
Wenn in der Aufgabenstellung gefordert wird, ein bestimmtes Programmierkonzept oder eine bestimmte Datenstruktur zu verwenden, ist es jedoch nötig die Implementation der Studierenden genauer zu untersuchen. Tests, bei denen die interne Struktur bekannt ist, werden Whitebox-Tests genannt (vgl. Olsen u. a., 2018, S. 87). Zurzeit werden diese Tests u.a. mit der Java *Reflection*-API oder mit *ANTLR* realisiert. Letzterer ist ein mächtiger Parsergenerator, der jedoch ein grundlegendes Verständnis von Syntaxbäumen und Grammatiken voraussetzt.

Dieses kann zunächst zeitintensiv und aufwändig sein. Vor allem auch unter dem Aspekt, dass Programmieren zunehmend in informatikfremden Studiengängen gelehrt wird. Das bedeutet, eine einfachere Methode, die Tests zu realisieren bietet auch Lehrenden mit weniger Programmiererfahrung die Möglichkeit, den Aufgabenpool einfach zu erweitern und sinnvolle Überprüfungen der Anforderungen zu schreiben. Um das Schreiben der Whitebox-Tests zu vereinfachen, ist das Ziel dieser Arbeit eine Programmierschnittstelle zu entwickeln, welche leicht verwendet werden kann, ohne besondere Vorkenntnisse vorauszusetzen. Eine Programmierschnittstelle wird auch als Application Programming Interface (API) bezeichnet. Diese bietet eine Menge an Funktionen an, die unabhängig von ihrer Implementation vom Benutzer genutzt werden kann (vgl. Bloch, 2014). Zur Erstellung der Schnittstelle, soll das Spoon-Framework verwendet werden, welches Möglichkeiten anbietet Quelltexte zu analysieren und auch zu transformieren.

1.2. Themenabgrenzung

In dieser Arbeit geht es vor allem um die Erstellung einer Bibliothek, die gewissen Anforderungen entsprechen soll. Die Bibliothek wird mit dem Spoon-Framework erstellt, welches auch in dieser Arbeit vorgestellt und näher erläutert wird. Allerdings werden nur die Bereiche aus Spoon vorgestellt, die wichtig für die statische Code-Analyse ist. Außerdem werden Grundlagen im API-Design vermittelt. Grundwissen in Java wird vorausgesetzt und nicht weiter erklärt.

1.3. Struktur

Im zweiten Kapitel dieser Arbeit werden Grundlagen im API-Design vermittelt. Hier wird zunächst erläutert, welche Vor- und Nachteile eine API bietet. Danach werden Prinzipien vorgestellt, welche Anforderungen eine gute API erfüllen sollte. Im dritten Kapitel erfolgt die Anforderungsanalyse. Dort werden Hilfsklassen und -methoden, die von Professoren für ihre Laborprüfungen geschrieben wurden, analysiert, damit entschieden werden kann, wie die API für die Whitebox-Texts aussehen soll. Anschließend wird im vierten Kapitel dieser Arbeit das Spoon-Framework vorgestellt. Dabei liegt der Fokus auf der Quelltext-Analyse und wie es mit Spoon möglich ist einzelne Quelltext-Elemente zu analysieren.
Das fünfte Kapitel stellt die Cutlery-Bibliothek vor. Der Aufbau der Bibliothek, sowie deren Anwendung steht im Vordergrund. Beendet wird das Kapitel mit einem „Proof of Concept“. Am Ende gibt es ein Fazit und einen Ausblick.

2. Grundlagen API-Design

API ist nur ein Oberbegriff und kann in verschiedene API-Typen unterteilt werden. Dazu zählen Messaging-APIs, REST-APIs, Dateibasierte-APIs und Objektorientierte-APIs. Letztere ist sprachabhängig und kann zum Beispiel, wie im Fall dieser Arbeit, eine Java-API einer Bibliothek sein (vgl. Spichale, 2017, S. 7-8).

2.1. Vor- und Nachteile einer API

Vorteile einer API ergeben sich bereits aus der Definition von Bloch. Da API und Implementierung getrennt sind, ist der Austausch und die Änderbarkeit der Implementierung leichter. Wenn der Benutzer einer Softwarekomponente zu stark abhängig von den Implementierungsdetails ist, so ist der Quelltext instabil. Bei jeder Änderung muss die Softwarekomponente angepasst werden. Durch Nutzung einer API kann diese Abhängigkeit entkoppelt werden. Eine API folgt dem Geheimnisprinzip. Die Komplexität einer Implementation wird versteckt, so dass der Benutzer auch große Anwendungen beherrschen kann. Diese Aufteilung großer Anwendungen hat weiterhin Vorteile für die Arbeitsteilung und die Entwicklungskosten. Es geht allerdings nicht nur darum, Implementierungsdetails zu verbergen, sondern auch den Entwicklern Funktionen einer Softwarekomponente einfach zugänglich zu machen.
Eine Programmierschnittstelle hat auch Nachteile. Programmiersprachen APIs in unterschiedlichen Programmiersprachen sind nicht kompatibel zueinander. Allerdings können APIs, die auf Protokollen wie HTTP basieren, dieses Problem lösen. Diese sogenannten Remote-APIs können von unterschiedlichen Plattformen und Programmiersprachen genutzt werden. Ein weiterer Nachteil ist die eingeschränkte Änderbarkeit. Allerdings sollte man hier zwischen interner und veröffentlichter API unterscheiden. Letztere hat Benutzer, die unbekannt und nicht kontrollierbar sind. In diesem Fall dürfen keine Änderungen gemacht werden, die den Vertrag brechen. Bei einer internen API ist dies anders. Module können aus einem öffentlichem und einem verborgenem Teil bestehen. Die Kommunikation zwischen den Modulen läuft mit deren API ab. Hier dürfen Änderungen geschehen, da der von den APIs abhängige Quelltext intern kontrolliert wird und bekannt ist. Auch hier gilt es, so wenig wie möglich öffentlich zu machen, um Änderungen vornehmen zu können (vgl. Spichale, 2017, S. 7-11).

2.2. Gutes API-Design

Eine gute Programmierschnittstelle ist leicht zu erlernen und zu benutzen, auch ohne Dokumentation. Der Benutzer soll keine Möglichkeiten haben, die Schnittstelle falsch zu benutzen. Außerdem soll der Quelltext, den die API verwendet, leicht zu lesen und zu warten sein. Weitere Eigenschaften, die gutes Design ausmachen, sind Erweiterbarkeit, mächtig genug die Anforderungen zu erfüllen und die Schnittstelle soll an den Benutzer angepasst sein. Um diese Anforderungen zu erfüllen, müssen einige allgemeine Prinzipien eingehalten werden. Die Funktionalität muss einfach zu erklären sein, dabei sollen die Anforderungen erfüllt sein aber die API nicht Überladen. Besser ist es, die Anforderungen gut zu erfüllen, anstatt unnötige Funktionen hineinzubringen.
Implementationsdetails gehören nicht in die API Beschreibung. Um den Benutzer nicht mit zu viel Komplexität zu verwirren wird die Implementation strikt von der API getrennt. Darüber hinaus gilt das Geheimnisprinzip. Öffentliche Klassen haben private Felder. Klassen und Methoden sollen so privat wie möglich sein (Bloch, 2006, vgl.).
Die Benennungen der Methoden, Klassen, Felder und Variablen müssen selbsterklärend sein, damit der Benutzer auch ohne Dokumentation in der Lage ist die API zu benutzen. Konsistente Namen erleichtern das Verstehen und Erlernen der API (vgl. Spichale, 2017, S. 43 ff.). Dennoch ist eine gute Dokumentation unentbehrlich. Jede Klasse, Methode, jedes Interface, jeder Parameter und Konstruktor muss dokumentiert und ausreichend beschrieben sein. Dieses verhindert Missverständnisse zwischen Benutzer und API und verhindert, dass der Benutzer die API falsch benutzt (vgl. Bloch, 2006).

2.2.1. Klassen-Design

Die Klassen in einer API sind möglichst unveränderbar. Dieses hat den Vorteil, dass die Klassen einfach, Thread-safe und wiederverwendbar sind. Subklassen sollen nur verwendet werden, wenn es nötig ist (Bloch, 2006, vgl.). Dabei gilt das Liskovsche Substitutionsprinzip:

> „Sei *q(x)* eine beweisbare Eigenschaft von Objekten *x* des Typs *T*. Dann soll *q(y)* für Objekte *y* des Typs *S* wahr sein, wobei *S* ein Untertyp von *T* ist. “
> Liskov und Wing (1994)

Also gilt: Wenn Objekte einer Basisklasse verwendet werden, muss die Softwarekomponente auch mit Objekten einer abgeleiteten Klasse funktionieren, ohne dass an dieser Anpassungen vorgenommen werden müssen (vgl. Spichale, 2017, S. 78-79). Außerdem sollten öffentliche Klassen keine anderen öffentlichen Klassen als Unterklassen haben, um die Implementierung zu vereinfachen (Bloch, 2006, vgl.).

2.2.2. Methoden-Design

Beim Überladen von Methoden sollte man vorsichtig sein. Eine überladenen Methode muss dasselbe machen, wie eine andere Methode desselben Namens. Falls dies nicht der Fall ist, ist es besser eine neue Methode zu implementieren. Außerdem gilt es, Methoden mit gleichen Namen und gleicher Parameteranzahl zu vermeiden.
Bei der Parameterliste einer Methode ist zu beachten, dass die Reihenfolge der Parameter konsistent sind. Dieses gilt besonders, wenn die Parametertypen identisch sind. In der *java.util.Collections*-Klasse gilt für jede Methode, dass der erste Parameter immer eine *Collection* ist, die verändert oder abgefragt werden soll. Des Weiteren ist eine Parameterliste mit drei oder weniger Parametern zu bevorzugen. Für den Benutzer der API ist dies übersichtlicher und er ist weniger auf die Dokumentation angewiesen. Wenn eine lange Parameterliste nicht vermieden werden kann bieten sich zwei Techniken an. Entweder kann die Methode in mehrere aufgeteilt werden oder man erstellt eine Helferklasse, die die Parameter hält.
Das Verhalten einer Methode soll den Benutzer nicht überraschen. Besser ist, es mehr Aufwand in die Implementierung zu investieren, um das gewünschte Verhalten zu erreichen, auch wenn es nachteilig für die Performance der API ist (Bloch, 2006, vgl.).
Als Rückgabe sollte *Null* vermieden werden. Zum einen können unerwartete *NullPointerExceptions* beim Nutzer auftreten, wenn diese nicht ausreichend abgefangen werden, zum anderen können zu viele Nullüberprüfungen die Implementierung der API unübersichtlich machen. In diesen Fällen ist es besser eine leere *Collection*, ein Ergebnisobjekt oder ein *Optional* zu benutzen (vgl. Spichale, 2017, S. 63-64).

2.2.3. Exception-Design

Ausnahmesituationen können in jeder API auftreten. Um diese zu behandeln, gibt es *Exceptions*. Allerdings sollte man darauf achten, diese sparsam einzusetzen. Der Quelltext wird dadurch lesbarer und der Benutzer der API ist weniger frustriert. Außerdem können auch Performanceprobleme auftreten, wenn diese übermäßig benutzt werden. Die Ursachen dafür liegen beim Erzeugen des Stacktrace in Abhängigkeit seiner Tiefe und dem Aufräumen des Stacks.
Exceptions sollen nicht zum Steuern des Ablaufs benutzt werden, sondern zur Behandlung von Ausnahmen. Diese können in folgenden Situationen auftreten:

- Eine *Exception* wird aufgrund eines Programmierfehlers in der API geworfen. Der Benutzer kann in diesem Fall nichts tun, da der Fehler in der Implementierung der API liegt.

- Der Benutzer verwendet die API falsch und verletzt den API-Vertrag. Die API kann nützliche Rückmeldungen geben, so dass der Benutzer seinen Quelltext anpassen und den Fehler beheben kann.
- Bei einem Ausfall des Netzwerks, Speicherplatzmangel oder anderen Ressourcenausfällen werden auch *Exceptions* geworfen. Je nach Art des Ausfalls hat der Nutzer die Möglichkeit es später erneut zu versuchen oder die Verbindungsprobleme zu beheben (vgl. Spichale, 2017, S. 67).

In Java wird zwischen *Checked* und *Unchecked Exceptions* unterschieden. Checked Exceptions werden zur Compile-Time geprüft. Werden diese nicht behandelt, kann der Quelltext nicht kompiliert werden. Das bedeutet, dass diese Exceptions mittels *try/catch*-Blöcken gefangen werden müssen oder zumindest mit *throws* weitergeworfen werden sollten. Die Verpflichtung die *Exceptions* zu behandeln, kann für den Benutzer der API eine große Last darstellen. Es werden dann häufig leere *Catch-Blöcke* verwendet, da der Benutzer oftmals nicht weiß, wie die *Exceptions* behandelt werden können oder nicht den Zugriff zu den benötigten Daten hat (vgl. Spichale, 2017, S. 67). Unchecked Exceptions werden nicht zur Compile-Time überprüft, sondern zur Laufzeit.
Wenn eine *Exception* geworfen wird sollten Informationen dazu beigefügt werden. Das erleichtert dem Benutzer das Reparieren und Erholen von der Ausnahmesituation. Im Falle von Unchecked Exceptions genügt eine ausreichend genaue Nachricht. Bei den Checked Exceptions sollten Informationen über Zugriffsmöglichkeiten mitgegeben werden (vgl. Bloch, 2006). Es gilt die Empfehlung Unchecked Exceptions zu bevorzugen und Checked Exceptions nur zu verwenden, wenn der Benutzer diese behandeln kann (vgl. Bloch, 2018).

3. Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die zu erstellende API festgelegt. Dabei wird der vorhandene Quelltext von Professoren, die automatische Tests für ihre Laborprüfungen anbieten, analysiert. Der Quelltext der zur Verfügung gestellt wurde ist von Herrn Prof. Dr. Schmolitzky, Herrn Prof. Dr. Schäfers und Herrn Prof. Dr. Jenke, außerdem wurden Quelltext-Auszüge von automatischen Tests für Online-Selbsttests von dem studentischen Mitarbeiter Niels Gandraß und eine E-Mail mit Anforderungen von Herrn Prof. Dr. Schäfers zur Verfügung gestellt.
Eine wichtige Anforderung, die bereits im Kapitel 2 erwähnt wurde, ist, eine gute API vollkommen von der Implementation zu abstrahieren. Das bedeutet, dass in diesem Fall keine Kenntnisse vom Spoon-Framework bekannt sein müssen. Dieses fördert die leichte Erlernbarkeit und einfache Benutzung der Bibliothek. Noch zu erwähnen ist, dass die Dokumentation der API in Englisch verfasst ist.

3.1. Vorgehen

Zunächst musste der vorhandene Quelltext gelesen werden und die Methoden auf ihre Funktion analysiert werden. Um einen besseren Überblick zu bekommen wurden alle Methoden, die wichtig für die Spezifikation sind, aufgelistet. Dabei wurden allerdings einige Methoden außer Acht gelassen:

- Private Helfermethoden
- Methoden, die die Lesbarkeit der Testergebnisse erhöhen
- Methoden, die die Benutzbarkeit der eigenen Tests zu verbessern

Die relevanten Methoden wurden nach ihrem Autor in eine Tabelle einsortiert. Anschließend wurden gleiche oder ähnliche Methoden in eine Zeile sortiert. Dabei haben sich bereits einige Gemeinsamkeiten gezeigt, so dass es Sinnvoll war Kategorien zu erstellen, um die Lesbarkeit der Tabelle zu fördern. Anhand dessen, was die Methoden prüfen, haben sich folgende Kategorien ergeben: *Klassen*, *Felder*, *Interfaces*, *Enums*, *Methoden*, *Konstruktor*, *Exceptions*, *Sonstiges* und

Einzelfälle. Letztere beinhaltete Methoden, die nur von einem Professor benutzt wurden und nicht, was diese prüfen. Um die Kategorien konsistent zu halten, wurden die Methoden der Kategorie *Einzelfälle* in andere Kategorien einsortiert, so dass die Kategorien repräsentieren, was die Methoden testen. Die Kategorie *Exception* wurde komplett entfernt, da sie nur eine Methode enthielt. Diese prüfte, ob in einer Methode eine *Exception* geworfen wird. Es wird also eine Methode, auf eine Eigenschaft geprüft. Deswegen wurde diese Methode der Kategorie *Methoden* zugeordnet. In der Kategorie *Sonstiges* wurden alle Methoden einsortiert, die nicht in die anderen Kategorien passten. Dazu gehören zum Beispiel Methoden, die schauen, ob alle Klassen-, Methoden-, und Feldernamen den Quelltextkonventionen entsprechen.
Die nächsten Abschnitte stellen weitere Details der Anforderungen vor. Dazu gehören die Gemeinsamkeiten und Unterschiede, die in den analysierten Quelltexten gefunden wurden.

3.2. Gemeinsamkeiten

In diesem Abschnitt werden die Gemeinsamkeiten in den Quelltexten aufgezeigt. Also Methoden, die die gleiche Funktion erfüllen. Da die Methoden zuvor in Kategorien unterteilt wurden, werden die gemeinsamen Methoden pro Kategorie hier vorgestellt werden. Dieses soll die Übersichtlichkeit verbessern. Außerdem werden auch die Methoden, die sehr häufig (in drei von vier Fällen) vorkommen, berücksichtigt. Alle anderen Methoden werden als Einzelfall angesehen.
In der Kategorie *Interface* gibt es eine Methode, die von nahezu jedem benutzt wird. Hierbei handelt es sich um die Frage, ob eine Klasse von einem Interface implementiert wird. Interessant ist hierbei, dass sogar die Parameterlisten nahezu identisch sind. Es wird immer eine Klasse und ein Interface übergeben, dabei unterscheiden sich nur die Typen. Je nach persönlicher Präferenz wird der Name als *String* übergeben oder Objekte der zu testenden Klassen.
Bei den Konstruktoren wird abgefragt, ob ein spezifischer Konstruktor mit der übergebenen Parameterliste vorhanden ist. Auch in diesen Fällen wird eine nahezu identische Parameterliste benutzt: die Klasse, in der sich der Konstruktor befindet und die Parameterliste, die der Konstruktor haben soll. Außerdem existiert in zwei Quelltexten eine Methode, die testet, welche Sichtbarkeit ein spezifischer Konstruktor hat. Die Frage nach der Sichtbarkeit von Elementen taucht immer wieder auf. Meistens in Form von eigenen Methoden, die explizit Fragen, ob zum Beispiel ein Konstruktor *public* ist.
Beim Testen von Feldern wird auch häufig nach der Sichtbarkeit gefragt. Dabei wird eine Methode benutzt, bei der man einen (Sichtbarkeits)-Modifizierer als Parameter übergibt. Damit kann man nicht nur die Sichbarkeit überprüfen, sondern auch, ob andere Modifizierer wie *final*

oder *static* benutzt werden. Außerdem haben fast alle Professoren eine Methode verfasst, die alle Felder testet, ob diese *private* sind. Ein weiteres Anliegen ist der Wert eines Feldes. Dieses wird in zwei Fällen überprüft. Dieses mal mit einer identischen Parameterliste. Die Instanz eines Objektes wird als *Object* und der Name des Feldes als *String* übergeben.
In der Kategorie zum Testen von Methoden sind zwei Überprüfungen häufig vertreten. Zum einen die Frage, ob eine bestimmte Methode mit der angegebenen Signatur in einer Klasse vorhanden ist und zum anderen, ob die erwartete Sichtbarkeit einer Methode mit der tatsächlichen übereinstimmt. Die Überprüfung, dass eine *Exception* in einer Methode geworfen wird, kommt auch in mehreren Quelltexten vor. Dabei wird jedoch nur geschaut, ob eine *Exception*-Klasse in der Methode vorkommt. Die gesuchte *Exception* wird in der Parameterliste definiert. Außerdem gibt es in zwei Quelltexten eine Methode die prüft, ob in einer bestimmten Methode eine Rekursion vorliegt.
Enums werden nur von zwei Professoren untersucht. In beiden Fällen gibt es die Methode *getEnum*. Im ersten Fall wird in einer Klasse nach einem *Enum* mit dem übergebenen Namen gesucht. Im anderen Fall wird in einer *CompilationUnit* nach einem *Enum* anhand seines Names gesucht. Wird ein Enum gefunden, wird dieses dann zurückgegeben.

Category	**Similarity**
Enums	getEnum
Fields	allFieldsPrivate hasFieldModifier getFieldValue
Interfaces	classImplementsInterface
Constructor	constructorExists hasConstructorRequestedAccessModifier
Methods	methodExists hasMethodRequestedAccessModifier methodThrowsException isMethodRecursive

Darst. 3.1.: Übersicht Gemeinsamkeiten

3.3. Unterschiede

Die Quelltexte der Professoren sind direkt zugeschnitten auf deren Anwendungsfall. In diesem Fall die praktische Programmierprüfung. Diese werden von jedem Professor unterschiedlich gestaltet. Dementsprechend unterschiedlich sind auch die Methoden, die zum Testen dieser

Prüfungen geschrieben wurden. Sichtbarkeiten werden zwar von fast allen Professoren geprüft, aber die Art und Weise, wie dieses geschieht kann variieren. In einem Fall wurden Methoden erstellt die konkret fragen, ob ein Feld *private* ist oder ein Konstruktor *public*. In einem anderen Fall wurden Prädikate erstellt: *isPublic*, *isPrivate*, *isStatic* und *isFinal*. Diese Prädikate werden dann in anderen Methoden verwendet, wie *hasMethodModifier(class, methodName, paramList, accessModifier)*.
Des Weiteren fällt auf, dass der Fokus in den Tests und somit auch in den Prüfungen, auf unterschiedlichen Programmierkonzepten liegt. *Enums* werden von Prof. Jenke sehr ausführlich getestet. Es wird nicht nur nach einem *Enum* gefragt, sondern auch Methoden benutzt, die die *Enumkonstanten* und *Enumordinale* abfragen. Wohingegen in anderen Quelltexten keine Methoden auftreten, die sich mit *Enums* beschäftigen. In der Testklasse von Prof. Schäfers werden Klassen ausführlich auf ihre Merkmale getestet. Es gibt dort Methoden wie *isClass(classUnderTest)*, *isExtending(classUnderTest, RequestedSuperTypeName)* und *isInterface(classUnderTest)*, die in keiner anderen Testklasse vorkommen. Außerdem wird bei den Methodentests in Prof. Schäfers Klasse unterschieden in Prozeduren und Funktionen. So gibt es Methoden mit den Namen *isFunction* und *isProcedure*. Zusätzlich zu diesen, gibt es noch die Methoden *isFunctionWithoutParameter* und *isProcedureWithoutParameter*. Diese überprüfen, ob eine Funktion(oder Prozedur) ohne Parameterliste, mit der angegebenen Signatur existiert. Ein anderer Einzelfall ist die Überprüfung, ob die Quelltextkonventionen eingehalten wurden. Hierfür existieren Methoden zum Testen der Schreibweise von Methodennamen, Feldnamen und Klassennamen. Noch eine Besonderheit, die nur in einer Testklasse vorkommt, ist die Überprüfung, ob verbotene Typen in einer Klasse vorkommen. Also Typen, die der Student laut Aufgabenstellung in der Klausur nicht verwenden darf.
Des Weiteren gibt es einen interessanten Unterschied beim Testen von Feldern. Die Sichtbarkeit der Felder ist für jeden Professor interessant. Allerdings wird nur von einem Professor abgefragt, ob eine bestimmte Klasse überhaupt Felder hat. Auch interessant ist, dass in einem Fall getestet wird, ob ein bestimmter Typ in der Parameterliste einer Methode vorkommt. Wohingegen die anderen Testklassen allgemeiner prüfen, ob es eine Methode mit einer bestimmten Signatur gibt.
Natürlich gibt es noch viele weitere kleinere Unterschiede, wie die Benennung von Methoden, die Struktur der Testklassen oder andere Implementationsdetails. Damit dieser Abschnitt übersichtlich bleibt, wurden hier nur die interessantesten und markantesten Unterschiede vorgestellt.

3.4. Weitere Anforderungen

Zusätzlich zu den bereitgestellten Quelltexten der Professoren und des studentischen Mitarbeiters, gab es auch noch eine E-Mail von Prof. Schäfers mit Anforderungen. In dieser E-Mail wurden die Anforderungen in zwei Kategorien, nach ihrer Wichtigkeit, sortiert. In der ersten Kategorie ist die Frage nach lokalen Elementen, da diese mit Java *Reflections* nicht gefunden werden können. Hierzu zählt zum Beispiel der Wunsch nach einer Methode, die eine *Collection* zurückgibt, mit allen Typen(inklusive innere Klassen), die in einem *Package* deklariert worden sind. Außerdem werden noch folgende Anforderungen als besonders interessant angesehen:

- Enthält der Quelltext Lambdas?
- Ist eine Methode direkt rekursiv? Oder Teil einer indirekten Rekursion?
- Enthält eine Methode Iterationen?
- Wird ein Knoten(oder ähnliches) wirklich verwendet?
- Sind zwei Lösungen gleich?

Am Ende der E-Mail werden noch zwei Anforderungen mit geringerer Priorität gestellt. Hierbei handelt es sich um die Fragen, ob ein Typ rekursiv ist und ob *Exceptions* im Quelltext auftreten. Letzteres soll unabhängig davon sein, ob die *Exceptions* tatsächlich geworfen werden.

3.5. Zusammenfassung der Analyse

In dieser Anforderungsanalyse wurden die Gemeinsamkeiten und Unterschiede der verschiedenen Quelltexte vorgestellt. Dieses gilt als Grundlage, welche Funktionalitäten die API haben soll. Der Schwerpunkt liegt hierbei auf allen Methoden, die von jedem Professor für ihre Tests geschrieben wurden. Damit die API gewohnt benutzt werden kann, wie die Klassen zuvor, sollen so viele Funktionalitäten wie möglich implementiert werden. Dabei soll die API übersichtlich bleiben und leicht erlernbar sein.
Des Weiteren sollen die zusätzlichen Anforderungen soweit wie möglich in die neue API integriert werden, um die Funktionalität zu ergänzen. In Darstellung 3.2 sieht man eine Übersicht der Anforderungen, die aus der Analyse hervorgegangen sind.

Category	Requirement
Enums	getEnum getEnumConstant getEnumOrdinal
Fields	allFieldsPrivate hasFieldModifier getFieldValue usesImplAsField
Interfaces	classImplementsInterface
Class	isExtending containsForbiddenType hasClassModifier hasClassCollectionWithElemType hasLambda
Constructor	constructorExists hasConstructorRequestedAccessModifier
Methods	methodExists hasMethodRequestedAccessModifier methodThrowsException isMethodRecursive isMethodStatic isRecursive containsLoop
Other	hasInvalidFieldName hasInvalidMethodName hasInvalidClassName getTypes

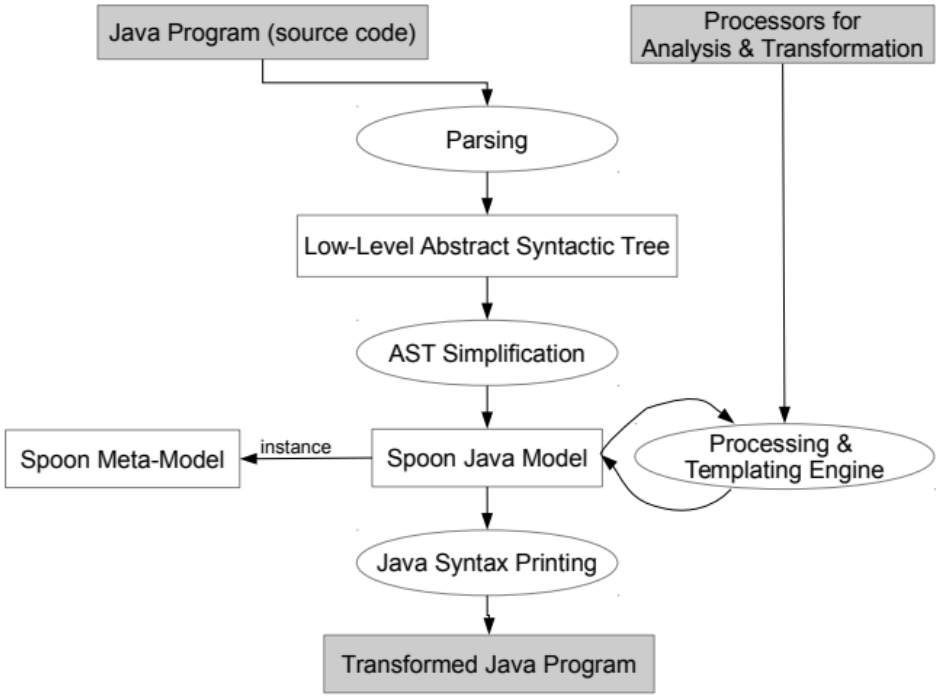
Darst. 3.2.: Übersicht Anforderungen

4. Das Spoon-Framework

Spoon ist ein Framework, welches Schnittstellen zur Analyse und Transformation von Java-Quelltext anbietet. Entwickelt wurde es im Inria (Französisches Institut für Digitalforschung) in Lille. Das Projekt wurde ursprünglich von Renaud Pawlak, Nicolas Petitprez und Carlos Noguera 2005/2006 ins Leben gerufen. Inzwischen arbeiten verschiedene Entwickler an dem Projekt, da es als Open-Source-Projekt online frei verfügbar ist. Es wird regelmäßig gewartet und weiterentwickelt, so dass neue Features alle 1-2 Monaten veröffentlicht werden. In den Veröffentlichungen werden vor allem Fehler beseitigt und die API erweitert, so dass auch der Quelltext von neuen Java-Versionen analysiert und transformiert werden kann (vgl. Inria-Spoon, 2019).
In diesem Kapitel wird das Spoon-Framework vorgestellt. Dabei wird ausführlich auf die Quelltext-Analyse eingegangen. Die Transformation wird nicht behandelt, da diese für diese Arbeit nicht relevant ist.

4.1. Überblick

Eines der Hauptziele von Spoon ist es, für Anwendungsentwickler ein einfaches Tool bereitzustellen, welches es ihnen ermöglicht, ihre eigenen domainspezifischen Analysen für Quelltexte zu schreiben (vgl. Pawlak u. a., 2015, S. 2). Im Falle von Spoon handelt es sich um statischen Code-Analyse. Bei einer statischen Analyse wird der Quelltext, im Gegensatz zu einer dynamischen Analyse, nicht ausgeführt (vgl. Ayewah u. a., 2008).
Damit Spoon den Quelltext untersuchen kann, liest das Framework diesen ein. Aus dem eingelesen Quelltext wird ein abstrakter Syntaxbaum (AST) erstellt, der dann weiter von Spoon vereinfacht wird. Dieses ist nötig, damit der AST vom Entwickler möglichst einfach analysiert und gegebenenfalls transformiert werden kann. Das Modell ist eine Instanz des Spoon-Metamodells (vgl. Pawlak u. a., 2015, S. 2-3).



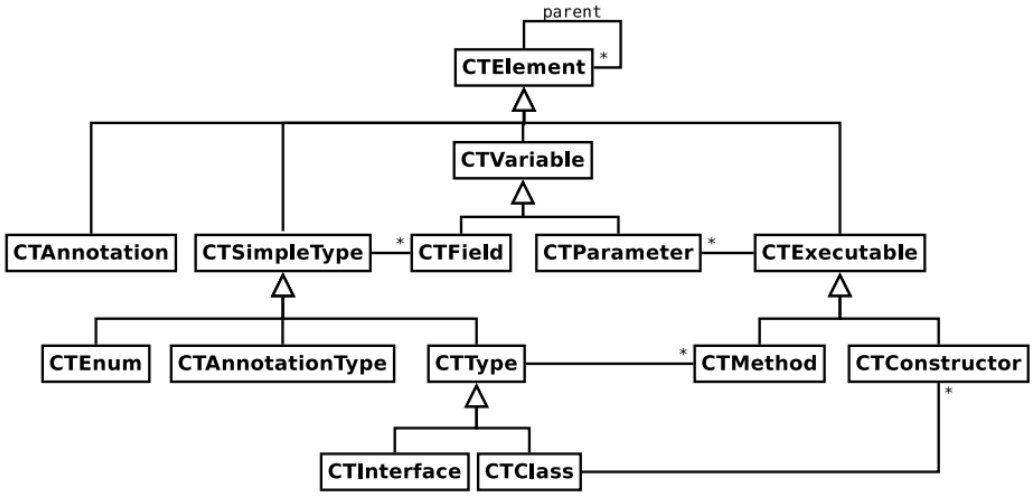
Darst. 4.1.: Überblick statische Code-Analyse in Spoon
(Quelle: Pawlak u. a., 2015)

4.2. Das Spoon Metamodell

Eine der fünf Haupteigenschaften von Spoon ist ein einfach zu lesendes Metamodell, welches einfach zu verstehen und zu manipulieren ist. In diesem Abschnitt wird die Struktur des Modells näher erläutert, sowie einige Elemente vorgestellt.
Das Spoon-Metamodell besteht aus drei Teilen. Der erste Teil ist der strukturelle Teil, welcher die Deklarationen von Programmelementen wie Interfaces, Klassen, Variablen, Methoden, Annotationen und Enums enthält. Der zweite Teil enthält den ausführbaren Quelltext, wie er zum Beispiel in Methodenrümpfen vorkommt. Im dritten Teil werden die Referenzen auf andere Programmelemente modelliert. Die Namen der Elemente im Spoon-Metamodell sind alle mit dem Präfix „CT“ (für Compile-Time) versehen. (vgl. Pawlak u. a., 2015).

4.2.1. Strukturelemente im Metamodell

Ein Ausschnitt der Strukturelemente im Metamodell werden in Darstellung 4.2 veranschaulicht. Allerdings ist dieser Graph nicht trivial. Zunächst einmal ist zu erwähnen, dass alle Elemente von *CTElement* erben. Die Eltern-Kind-Beziehungen, wie sie im Spoon-Metamodell gelten, werden allerdings in umgekehrter Reihenfolge gelesen, welches zunächst etwas ungewohnt ist. Der *CTClass*-Knoten ist Elternknoten von dem *CTMethod*-Knoten und dieser wiederum ist Elternknoten von *CtExecutable*. Der Aufbau des Metamodells ist dennoch für Java-Entwickler leicht zu erlernen, da der Aufbau von Java-Programmen bekannt ist. Eine Klasse kann mehrere Methoden enthalten, somit gilt auch im Diagramm eine *CTClass* enthält n-*CTMethod* (vgl. Inria-Spoon, 2019). Deutlicher wird dies später im Unterkapitel 4.2.4, wo die graphische Darstellung des Metamodells, an einem konkreten Beispiel, näher erläutert wird.

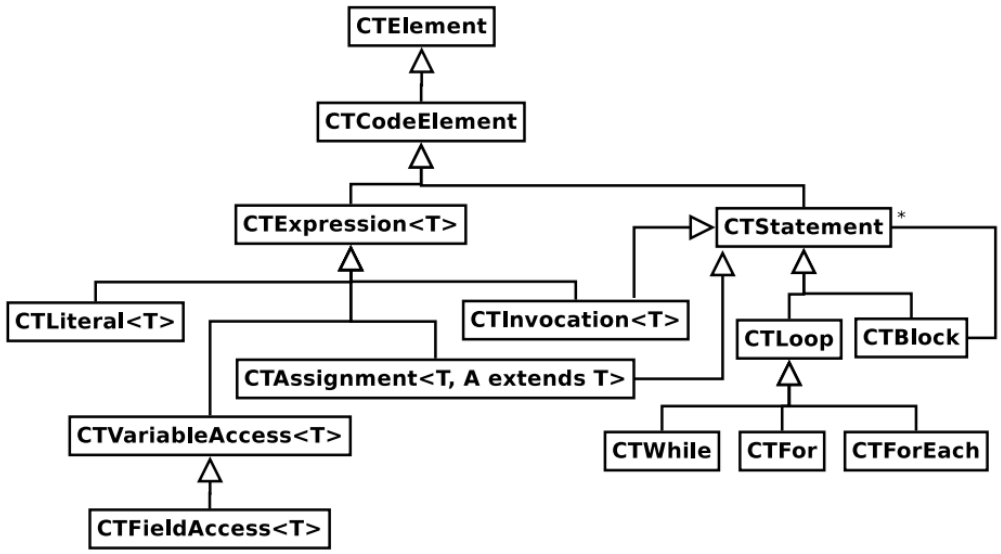


Darst. 4.2.: Ausschnitt der Strukturelemente
(Quelle: Pawlak u. a., 2015)

4.2.2. Quelltextelemente im Metamodell

Das Diagramm 4.3 zeigt einen Auschnitt der Quelltextelemente im Metamodell. Es gibt zwei Hauptzweige in diesem Modell: *CTExpression<T>* und *CTStatement*.
*CTStatement*s sind untypisierte top-level Anweisungen, die direkt in einem Quelltextabschnitt verwendet werden können. *CTExpression*s sind Elemente, die in einem *CTStatement* verwendet werden können. Damit das Diagramm lesbarer ist, ist dies nicht darin dargestellt. Ein *CTLoop* ist

ein *CTStatement* und zeigt auf eine *CTExpression<T>*, welches in dem Fall den *Boolean*-Ausdruck darstellt. Es gibt auch Elemente, die sowohl eine *CTExpression<T>* und ein *CTStatement* sind, z. B. *CTInvocation<T>* (vgl. Inria-Spoon, 2019).



Darst. 4.3.: Ausschnitt der Quelltextelemente
(Quelle: Pawlak u. a., 2015)

4.2.3. Referenzen im Metamodell

Das Referenzmodell in Spoon orientiert sich daran, dass womöglich nicht alle vom Programm referenzierten Elemente in das Metamodell übernommen worden sind, da sie aus einer externen Bibliothek stammen, die nicht Teil des eingelesenen Quelltextes ist. Ein *CTExpression<T>*-Knoten, welches ein *String* zurückgibt, ist an eine Referenz vom Typ *String* gebunden und nicht etwa an das Übersetzungszeit-Modell aus der *java.util.String*-Klasse, da dieser Quelltext normalerweise nicht Teil des zu analysierenden Quelltextes ist. Diese schwachen Referenzen, machen es einfacher ein Metamodell zu erzeugen und modifizieren, ohne dabei zu stark an alle referenzierten Elemente gebunden zu sein (vgl. Pawlak u. a., 2015, S.4).

4.2.4. Graphische Darstellung des Metamodells

Spoon bietet die Möglichkeit das erstellte Metamodell graphisch darzustellen. Dieses kann sehr einfach in der Konsole erzeugt werden. Dazu benötigt man eine *.jar*-Datei mit einer Spoon Version und Quelltext, aus dem das Metamodell generiert werden soll. In Listing 4.1 ist der Befehl für die Kommandozeile dargestellt. Wichtig ist hier, den richtigen Pfad zur *.jar*-Datei und zum Quelltext anzugeben (vgl. Inria-Spoon, 2019).

```
java -cp path/to/spoon/jar spoon.Launcher -i path/to/sourcecode/
   HelloWorld.java --gui --noclasspath
```

Listing 4.1: Befehl zur Erstellung der graphischen Darstellung des Metamodells

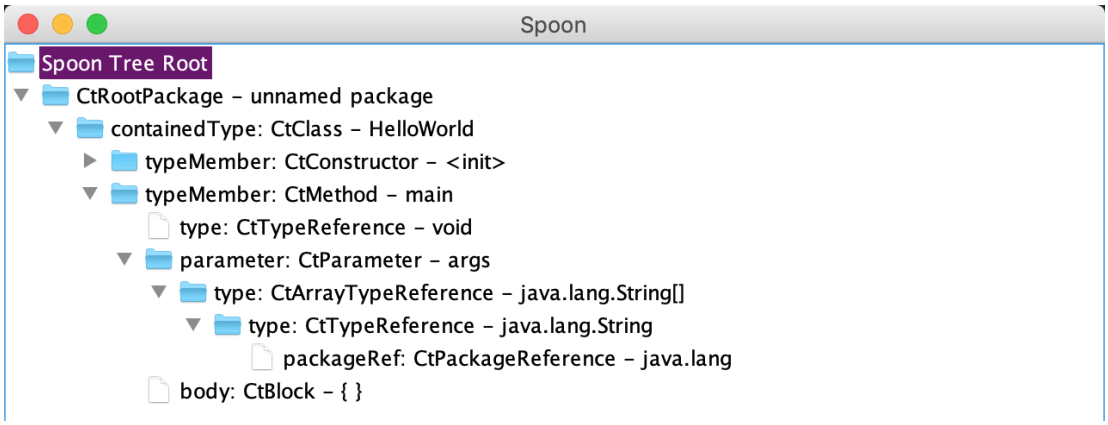
Als Beispiel wurde das Metamodell aus einer einfachen *HelloWorld.java*-Klasse erstellt. Diese Klasse trägt den Namen *HelloWorld* und hat zunächst eine leere *main*-Methode (Listing 4.2).

```
public class HelloWorld
{
    public static void main(String[] args){

        }
}
```

Listing 4.2: Hello World Beispiel 1

Wenn man den in Listing 4.1 gezeigten Befehl ausführt, öffnet sich ein Fenster, welches das Metamodell der *HelloWorld.java*-Klasse darstellt (s. Darstellung 4.4).



Darst. 4.4.: Graphische Darstellung des Metamodells

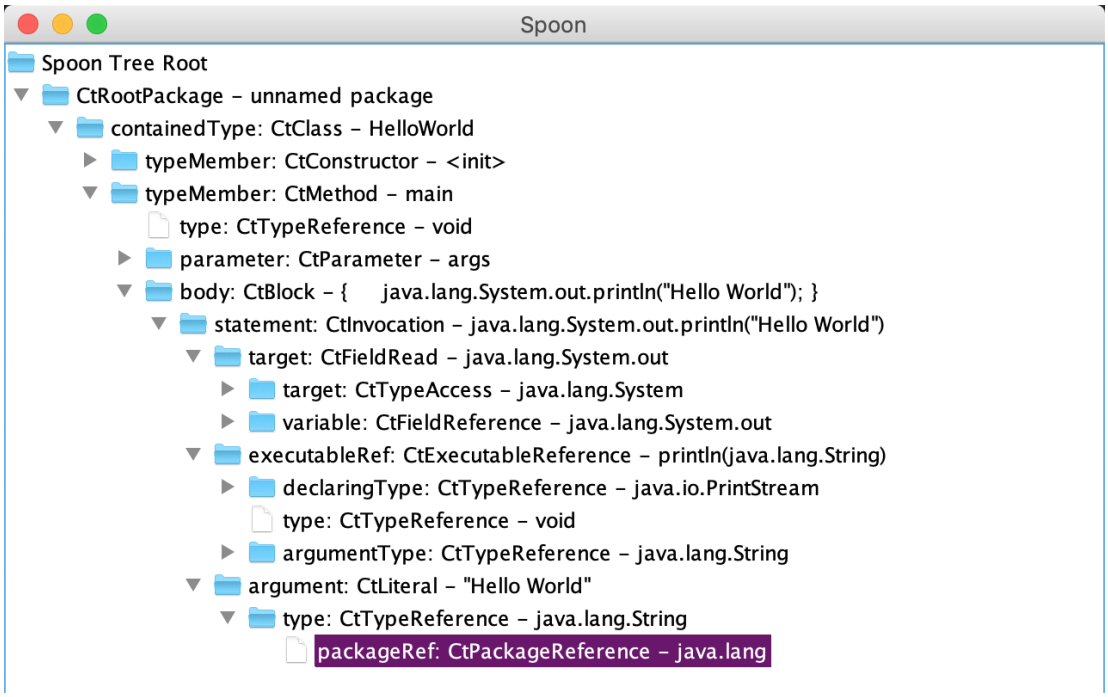
In dem Fenster sieht man den Aufbau der Klasse. Es gibt eine *CTClass* mit dem Namen *HelloWorld* und eine *CTMethod* mit dem Namen *main*, welche den Bezeichnungen aus dem

Quelltext entsprechen. Weiter unten kann man sehen, dass der Methodenrumpf *CTBlock* leer ist. Die Darstellung des Metamodells ist interaktiv und man kann die Elternknoten aufklappen. In dem Beispiel sieht man, dass die *CTClass* mit dem Namen „HelloWorld“ aufgeklappt wurde. In dieser befindet sich ein *CTConstructor* und eine *CTMethod* mit dem Namen *main*. Die *CTMethod* hat wiederum drei Kinder: eine *CTTypeReference*, *CTParameter* und ein *CTBlock*. Dieser zeigt uns in diesem Beispiel einen leeren Methodenrumpf. In einem zweiten Beispiel wird der Methodenrumpf nun gefüllt (s. Listing 4.3).

```
1 public class HelloWorld
2 {
3     public static void main(String[] args){
4                 System.out.println("Hello World")
5         }
6 }
```

Listing 4.3: Hello World Beispiel 2

Das Metamodell hat sich entsprechend verändert und ist komplexer geworden. Wie man in Darstellung 4.5 sieht, ist der *CTBlock* nicht mehr leer, sondern hat mehrere Kindknoten. Unter anderem findet man dort ein *CTLiteral* mit *"Hello World"*. Genauso, wie in dem Quelltext.



Darst. 4.5.: Graphische Darstellung des Metamodells mit gefüllten Methodenrumpf

4.3. Abfragen von Quelltext-Elementen

Nachdem im vorherigen Abschnitt erläutert wurde, wie das Spoon-Metamodell aufgebaut ist, stellt sich nun die Frage, welche Möglichkeiten es gibt, um an die benötigten Quelltext-Elemente zu gelangen. Dieses ist von großem Interesse zum implementieren der gewünschtem API. In diesem Abschnitt werden einige dieser Möglichkeiten vorgestellt und mit Beispielen erläutert. Alle hier erläuterten Techniken sind der Spoon-Webseite entnommen (vgl. Inria-Spoon, 2019). Im vorherigen Kapitel wurden die Anforderungen analysiert und erwähnt, welche Funktionen die API bieten soll. Spoon soll bei der Analyse der Quelltexte helfen. Dabei ist es nötig die Quelltext-Elemente näher zu untersuchen. Die Implementation der Methode `allFieldsPrivate` erfordert es an alle Felder einer Klasse zu gelangen, um dann zu prüfen, ob alle Felder *private* sind.

4.3.1. Getter

Eine Möglichkeit, um an bestimmte Quelltext-Elemente zu gelangen, ist den generierten AST zu durchlaufen. Alle Elemente bieten *getter*-Methoden an, die es ermöglichen, unkompliziert an das gewünschte Element zu gelangen. Von einer *CtClass* kann man sich zum Beispiel, alle Methoden oder alle Felder, geben lassen.

```
1  public boolean allFieldsPrivate() {
2          CtClass<?> clazz = findClass();
3          Collection<CtField<?>> fieldList = clazz.getFields();
4          for(CtField<?> f : fieldList ) {
5                  if(!(f.isPrivate())) {
6                          return false;
7                  }
8          }
9          return true;
10 }
```

Listing 4.4: Anwendung von Getter-Methoden

Wie in Listing 4.4 zu sehen, gibt es die Methode `getFields()`, mit der man sehr einfach an alle Felder einer Klasse gelangen kann. Die Methode gibt eine `Collection<CtField<?>>` zurück. Diese wird anschließend in einer Schleife durchlaufen. In dieser wird geprüft, ob wirklich alle Felder in einer Klasse *private* sind.

4.3.2. Filter

CtElement bietet mit *getElement(Filter))* eine Methode an, mit denen man nach bestimmten Kriterien filtern kann. Ein Filter definiert ein Prädikat in Form einer *matches*-Methode, d.h. es wird *true* geliefert wenn ein Element, welches den Filterangaben entspricht, gefunden wird. Der Suchalgorithmus des Filters ist durch eine Tiefensuche implementiert. Beim Traversieren des ASTs werden die Elemente, die das Prädikat des Filters erfüllen aufgesammelt.
In der API soll es eine Methode geben, die überprüft, ob es einen Aufzählungstypen mit einem bestimmten Namen gibt. In Listing 4.5 wird gezeigt, wie mit Hilfe eines `NamedElementFilters` nach allen *CTEnum*, mit dem gegebenen Namen, gefiltert wird.

```
1 METAMODEL.getElements(new NamedElementFilter<>(CtEnum.class, "Foo"))
    ;
```

Listing 4.5: NamedElementFilter in Spoon

Es werden verschiedene Filter von Spoon zur Verfügung gestellt. Ein weiteres Beispiel ist die Verwendung eines `TypeFilters`, wie in Listing 4.6 gezeigt. Hier wird nach allen *CTAssignments* innerhalb eines Methodenrumpfes gefiltert.

```
1 list1 = methodBody.getElements(new TypeFilter(CTAssignment.class)}
```

Listing 4.6: TypeFilter in Spoon

4.3.3. Weitere Möglichkeiten

Spoon bietet eine Vielzahl an weiteren Möglichkeiten an, um an Quelltext-Elemente zu gelangen. Drei weitere Alternativen, zu den bereits genannten, sollen in diesem Abschnitt kurz vorgestellt werden.

Queries

Queries wurden mit der Spoon Version 5.5 eingeführt und bieten verbesserte Funktionen zum Filtern an. Queries können Java 8 Lambdas benutzen, verkettet werden und an verschiedenen Elementen wiederverwendet werden.

```
1 // returns a list of String
2 list = package.map((CtClass c) -> c.getSimpleName()).list();
```

Listing 4.7: Anwendung von Queries mit Java 8 Lambdas (Inria-Spoon, 2019)

Scanner

Mit dem *CtScanner* ist es möglich einen Elternknoten und seine Kinder zu besuchen. Damit kann man zum Beispiel die Anzahl der Kindknoten ermitteln, die einem bestimmten Typ entsprechen. Der erstellte Scanner kann dabei auf jeden Elternknoten angewendet werden.

Iteratoren

Der *CtIterator* wird ähnlich wie der Iterator in Java verwendet. Dabei wird über die Knoten des ASTs iteriert. Dieses geschieht mittels Tiefensuche, allerdings gibt es zusätzlich einen *CTBFSIterator*, der die Breitensuche verwendet.

```
1 CtIterator iterator = new CtIterator(root);
2 while (iterator.hasNext()) {
3     CtElement el = iterator.next();
4     //do something on each child of root
5 }
```

Listing 4.8: Anwendung vom CtIterator (Inria-Spoon, 2019)

Dieses Kapitel sollte einen kurzen Einblick in das Framework geben und die wichtigsten Funktionen erläutern. Im nächsten Kapitel wird, die mit Hilfe von Spoon entwickelte Programmierschnittstelle, „Cutlery“ vorgestellt.

5. Die Cutlery Bibliothek

Die Bibliothek, die aus den Anforderungen hervorgegangen ist trägt den Namen „Cutlery“. Übersetzt in das Deutsche bedeutet es „Besteck “. Eine Anspielung auf das verwendete Spoon-Framework. Dies soll verdeutlichen, dass nicht nur Löffel gefunden werden können, sondern auch andere Werkzeuge, um eine Quelltextanalyse durchzuführen. Hinzuzufügen ist, dass die API komplett in Englisch verfasst wurde.

5.1. Aufbau

Cutlery ist in zwei Teile aufgeteilt. Zum einen gibt es die Tester-Klassen, die es ermöglichen verschiedene Quelltext-Elemente zu analysieren und zu testen. Zum anderen die Context-Klasse, die den Einstiegspunkt in die Benutzung der API markiert. Beide Teile werden in diesem Kapitel vorgestellt und deren Funktionen erläutert.

5.1.1. Die Context-Klasse

Als Erstes musste die Frage geklärt werden, wie der Benutzer die API verwenden soll und wie der zu analysierende Quelltext übergeben werden soll. Die erste Idee war, dieses in den entsprechenden Tester-Klassen zu machen. Bei Erstellung eines Testers wird diesem der Pfad zum Quelltext übergebenden. Allerdings gab es dadurch duplizierten Quelltext, da jeder Tester zunächst den übergebenden Quelltext parsen musste. Des Weiteren ergab sich das Problem, dass, wenn man den gleichen Quelltext von verschiedenen Testern analysieren lassen möchte, Spoon im Hintergrund den selben AST mehrmals erstellte. Daraus entstand die Idee, eine eigene Klasse zu erstellen, die den Einstiegspunkt für den Benutzer darstellt und zentral für das Erstellen des Spoon-Metamodells zuständig ist, welches dann in den einzelnen Testern benutzt werden kann. Außerdem geschieht so eine weitere Abstraktion von der Spoon-Logik, von der der Nutzer nichts wissen soll.
Wie in Listing 5.1 zu sehen wird bei der Erstellung der `Context`-Klasse als Parameter der absolute oder relative Pfad zu einem Ordner oder einer Datei übergeben. Dieser Pfad führt

zum Quelltext, der untersucht werden soll. Anschließend kann ein *Tester* erstellt werden, der als Parameter ein *Context*-Objekt erhalten muss.

```
1 Context context = new Context("src/test/java/resources");
```

Listing 5.1: Erstellung eines Context-Objektes

5.1.2. Die Tester-Klassen

Die Methoden, die sich aus der Anforderungsanalyse ergaben, wurden zuerst sortiert. Methoden, die Gemeinsamkeiten aufwiesen wurden zusammengeführt. Danach wurden sie Kategorien zugeordnet. In diesem Fall haben sich folgende Kategorien abgezeichnet: Methoden, Klassen, Interfaces, Enums, Konstruktoren, Felder und Sonstiges. Diese Kategorien haben sich ergeben aus der Frage, wie die vorhandenen Methoden in Verbindung stehen und was diese testen. Folglich wurden die Kategorien zu Klassen. Da diese Klassen beschreiben, was getestet wird, hat es sich angeboten das Wort Tester anzuhängen, so dass sich die API aus den folgenden Klassen zusammensetzt: *ClassTester*, *ConstructorTester*, *EnumTester*, *FieldTester*, *InterfaceTester*, *MethodTester* und *OtherTester*. Wie oben bereits erwähnt, muss beim Erstellen der jeweiligen Tester der *Context* mitgegeben werden. In dem *ConstructorTester*, *FieldTester* und *MethodTester* muss zusätzlich noch der Klassenname, der zu untersuchenden Klassen angegeben werden.

```
1 FieldTester fTester   = new FieldTester(context, "MyClass");
```

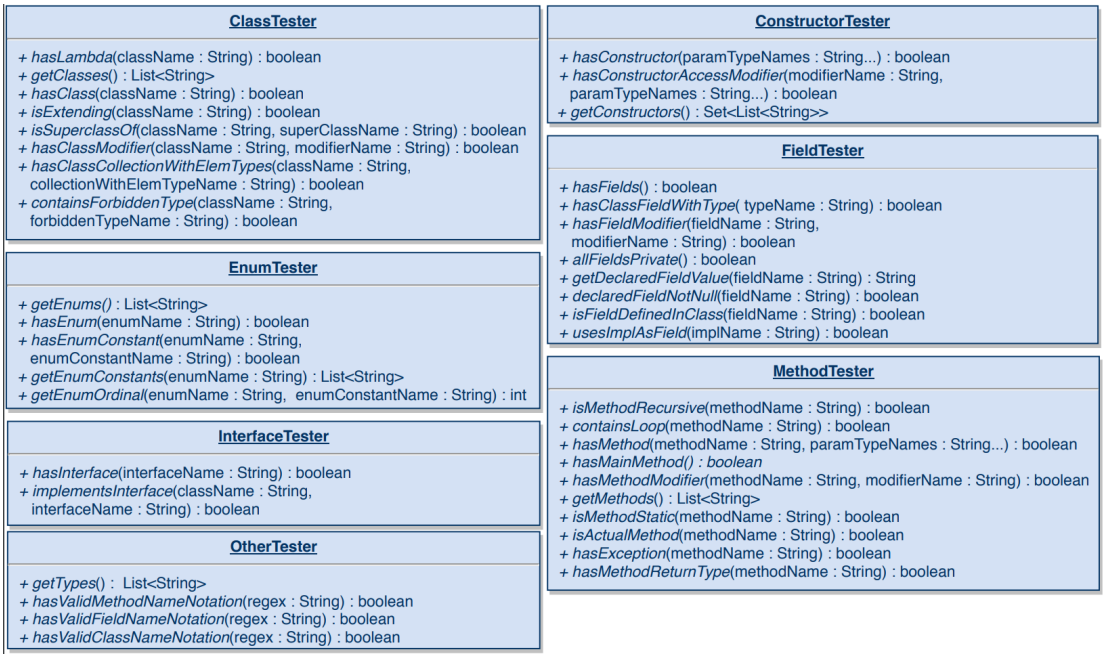
Listing 5.2: Erzeugung eines FieldTesters

Dieses hat den Vorteil, dass in den einzelnen Methoden des Testers die Übergabe des Klassennamens als Parameter nicht erfolgen muss. Somit bleibt die Parameterliste kurz und leicht verständlich. Der *Context*, der dem Tester mitgegeben wird, kann entweder zu einem Package führen oder direkt zur Klasse. Das Package muss in dem Fall die Klasse mit dem übergebenen Namen enthalten (siehe Listing 5.2).

5.1.3. Die Methoden

Die Methoden wurden anhand dessen einsortiert, was sie testen. Hierbei gab es allerdings ein paar Entscheidungen, die getroffen werden mussten, da dies nicht immer eindeutig war. In dem *InterfaceTester* gibt es nur zwei Methoden: *hasInterface(String interfaceName)* und *implementsInterface(String className, String interfaceName)*. Letztere prüft, ob eine Klasse ein Interface implementiert. Diese Methode, hätte auch in den *ClassTester* gepasst, da sie eine Klasse analysiert. Allerdings wird in keiner anderen Klasse der Name eines Interfaces oder andere

Informationen über Interfaces benötigt, so dass entschieden wurde, aus Konsistenzgründen diese Methoden zusammen in einer Klasse unterzubringen. Dieser Grundsatz zieht sich durch alle Tester-Klassen: Felder sind nur im *FieldTester* präsent, Methoden nur im *MethodTester*, Enums nur im *EnumTester*. Eine Ausnahme bilden die Methoden im *OtherTester*. Diese analysieren den gesamten *Context*, der übergeben wurde. So kann man sich mit *getTypes()* alle Typen aus einem Packet, Projekt oder einer Klasse geben lassen. Außerdem kann geprüft werden, ob die Quelltextkonventionen von Methoden-, Felder- und Klassennamen eingehalten wurden.
Zur besseren Übersicht ist ein Diagramm der Tester-Klassen mit deren Methoden in Darstellung 5.1 zu sehen.



Darst. 5.1.: Übersicht Klassen mit Methoden der Cutlery-API

5.2. Proof of Concept

Damit gezeigt werden kann, dass Cutlery zum Schreiben von Whitebox-Tests geeignet ist, wurde eine bereits vorhandene Testklasse aus einer Programmierprüfung von Prof. Schmolitzky mit Cutlery neu implementiert. Zugleich wird daran veranschaulicht, wie die API benutzt wird.

5.2.1. Hintergrund

Prof. Schmolitzky verwendet in seiner Vorlesung, zur Einführung in das Programmieren mit Java, die Entwicklungsumgebung BlueJ. Diese wird dann auch in der praktischen Programmierprüfung am Ende des Semesters verwendet. Das bedeutet, die Studierenden benutzen BlueJ, um die gestellten Aufgaben zu lösen. Dabei wird überprüft, ob sie die gelehrten Programmierkonzepte aus der Vorlesung verstanden haben. Die Tests schreibt Prof. Schmolitzky und diese können in der Prüfung von den Studierenden benutzt werden, um einen Anhaltspunkt zu bekommen, ob der geschriebene Quelltext die Anforderungen der Aufgabe erfüllt. Dabei werden zum einen Blackbox-Tests verwendet, um die reine Funktionalität zu testen. Zum anderen existieren Whitebox-Tests, die unter anderem überprüfen, ob alle Felder *private* sind. Im Folgenden wird gezeigt, wie Cutlery in BlueJ verwendet werden kann, um Tests für eine Programmierprüfung zu erstellen.

5.2.2. Einbinden der Cutlery.jar in BlueJ

Als erstes muss die Cutlery-Bibliothek einem BlueJ-Projekt hinzugefügt werden, damit es ohne Probleme verwendet werden kann.[1] Hierzu wird zunächst das vorhandene Projekt wie gewohnt geöffnet. In diesem Fall ist es eine Programmierprüfung aus dem Wintersemester 2017 (s. Darst. 5.2).

Anschließend kann man in den Einstellungen einen Reiter *Libraries* finden. In diesem kann man externe Bibliotheken hinzufügen. Mit dem Button *Add File* öffnet sich das Verzeichnissystem des Computers und man kann nach der *Cutlery.jar*-Datei suchen. Nachdem man diese hinzugefügt hat, erscheint der Pfad zu der Datei in dem *User Libraries*-Fenster (s. Darst. 5.3). Sollte in den Klammern hinter dem Pfad „not loaded “ stehen, muss BlueJ einmal neu gestartet werden, um die Änderungen wirksam zu machen.

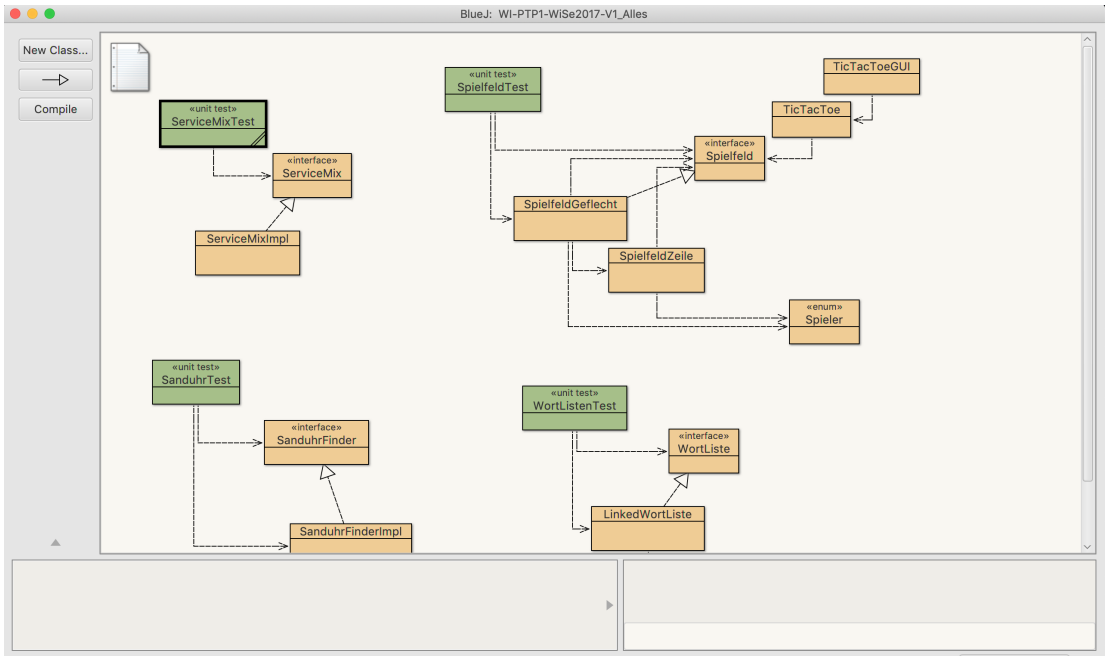
5.2.3. Tests schreiben mit Cutlery

Nachdem Cutlery erfolgreich der Liste der Bibliotheken hinzugefügt wurde, kann die API nun zum Schreiben der Tests verwendet werden. In diesem Proof of Concept wurden die Tests in der Klasse *SpielfeldTest* neu implementiert.

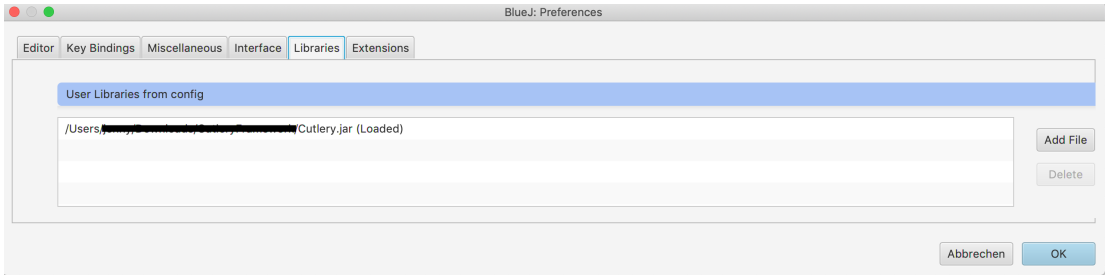
Vorbereitung

In Listing 5.3 ist zu sehen, wie man Cutlery importiert.

[1] Es wurde die BlueJ Version 4.2.0 verwendet.



Darst. 5.2.: Geöffnetes Projekt in BlueJ



Darst. 5.3.: Die hinzugefügte Cutlery.jar

```
1 import de.hamburg.haw.cutlery.*;
```

Listing 5.3: Import von Cutlery

Anschließend wird ein `Context`-Objekt erzeugt. Als Parameter wird in diesem Fall der absolute Pfad zu dem Projekt übergeben. Danach werden die benötigten `Tester` erstellt. Wie zuvor erläutert, benötigen einige Tester nicht nur ein `Context`-Objekt als Parameter, sondern auch den Namen der zu untersuchenden Klasse. Zu diesen Testern gehören der `MethodTester` und der `FieldTester`. In Listing 5.4 sieht man, dass diesen Testern nicht

nur der zuvor erstellte `Context`, sondern auch der Name der zu untersuchenden Klasse, übergeben wurde. Nun kann das Schreiben der eigentlichen Tests beginnen.

```
1 Context context = new Context("/Path/To/Project");
2
3 InterfaceTester iTester = new InterfaceTester(context);
4 EnumTester eTester = new EnumTester(context);
5 MethodTester mTester =
6 new MethodTester(context, "SpielfeldZeile");
7 FieldTester fTester =
8 new FieldTester(context, "SpielfeldGeflecht");
```

Listing 5.4: Erstellung des Contexts und der benötigten Tester

Implementation

Die Klasse *SpielfeldTest* hat fünf Testfälle, die es neu zu implementieren gilt. Es gibt noch weitere Tests, diese sind allerdings Blackbox-Tests, die mit Hilfe von *JUnit* implementiert worden sind. Diese brauchen somit nicht mit Cutlery implementiert werden. Im ersten Fall wird getestet, ob die Klasse `SpielfeldGeflecht` das Interface `Spielfeld` implementiert. Dazu wird ein *InterfaceTester* benötigt, der schon erstellt wurde (s. Listing 5.5). Danach wird die Methode `implementsInterface("SpielfeldGeflecht","Spielfeld")` verwendet, die den Namen einer Klasse und den Namen eines Interfaces als Parameter benötigt. In Listing 5.5 sieht man, wie die Implementation mit Cutlery im Detail aussieht. Der Kommentar in Zeile 6 zeigt hierbei die Version, wie es vorher implementiert wurde. Die Parameterliste ist unverändert und erleichtert die Umstellung auf die neue API. Ein Unterschied ist allerdings, dass in diesem Test und auch den Folgenden, `Assertions` verwendet werden. Da dieses Konzept allerdings in den meisten Blackbox-Tests benutzt wird, sollte die Verwendung bekannt sein. Der Name der Test-Methode besagt zusätzlich, dass auch geprüft werden soll, ob die Klasse vorhanden ist. Falls keine Klasse mit den Namen „SpielfeldGeflecht“ gefunden werden kann, würde Cutlery eine *Exception* werfen und es somit dem Anwender mitteilen.

```
1 @Test (timeout = 1000)
2 public void testKlasseVorhandenUndImplementiertInterface()
3 {
4     assertTrue(iTester.implementsInterface("SpielfeldGeflecht",
5                                            "Spielfeld"));
6     //_util.testeKlasseUndInterface("SpielfeldGeflecht",
7                                     "Spielfeld");
8 }
```

Listing 5.5: Klasse implementiert gewünschtes Interface

Im zweiten Testfall wird überprüft, ob ein *Enum* vorhanden ist. Der Name des *Enum*s ist bekannt. Cutlery bietet im *EnumTester* die Methode *hasEnum(String enumName)* an. Diese wird auch in diesem Test verwendet. Außerdem wird überprüft, dass der Aufzählungstyp genau drei Enumkonstanten besitzt. Erneut sieht der Test sehr ähnlich aus. Die Benennung der Methoden in Cutlery fördert allerdings die Lesbarkeit des Quelltextes für Außenstehende(s. Listing 5.6).

```
1 @Test (timeout = 1000)
2 public void testEnumVorhanden()
3 {
4     assertTrue(eTester.hasEnum("Spieler"));
5     assertEquals("Enum_Spieler_nicht_verwendbar",
6                  3, eTester.getEnumConstants("Spieler").size());
7     //Class enumSpieler = _util.gibClass("Spieler");
8     //Object[] valuesSpieler = enumSpieler.getEnumConstants();
9     //assertEquals("Enum Spieler nicht verwendbar",
10    //              3,valuesSpieler.length);
11    }
```

Listing 5.6: Test Enum vorhanden

Der dritte Testfall sieht interessanter aus, da die Implementation etwas unterschiedlich ist. Geprüft werden soll hier, ob die Methode mit den Namen „besetze“ als Parameter den Aufzählungstyp `Spieler` verwendet. Cutlery bietet nicht direkt dieselbe Methode an wie zuvor verwendet wurde (Zeile 5 in Listing 5.7). Allerdings kann dennoch überprüft werden, ob die angegebene Methode mit dem geforderten Typ existiert. Die Aufgabe in der Prüfung ist, die vorgegebene Methode *besetze(int, int)* so umzuschreiben, dass der Aufzählungstyp *Spieler* als zweiter Parameter verwendet wird. Das bedeutet die Methode sieht am Ende so aus: *besetze(int, Spieler)*. Damit dieses sichergestellt ist, wird aus dem *MethodTester* die Methode `hasMethod(String methodName, String... paramTypeNames)` verwendet. In

Listing 5.7) kann man sehen, dass die beiden Parameterlisten unterschiedlich aussehen. Zum einen liegt es daran, dass der *MethodTester* schon weiß, dass er in der Klasse *SpielfeldZeile* die gesuchte Methode finden soll (s. Listing 5.4), zum anderen weil `hasMethod` die gesamte Parameterliste benötigt, um sicherzugehen, dass die vorhandene Methode mit der Erwarteten übereinstimmt.

```
1 @Test (timeout = 1000)
2 public void testEnumWirdVerwendet()
3 {
4     assertTrue(mTester.hasMethod("besetze", "int", "Spieler"));
5     //_util.testTypIstParameterVonMethodeInKlasse(
6     //"Spieler", "besetze", "SpielfeldZeile"
7     //);
8 }
```

Listing 5.7: Enum wird als Parameter in Methode verwendet

Im nächsten Fall wird getestet, ob die Klasse *SpielfeldGeflecht* Felder hat und ob eines der Felder *HashMap* verwendet. Hierzu wird der *FieldTester* verwendet, der zuvor erstellt wurde und zusätzlich zum *Context* auch wieder einen Klassennamen als Parameter nimmt. In diesem Fall die Klasse *SpielfeldGeflecht*. Im Vergleich sieht der Test wieder sehr ähnlich aus. Nur die kürzeren Parameterlisten fallen bei den Cutlery-Tests auf. Allerdings ist hier eine Besonderheit aufgefallen, die bisher nicht geklärt werden konnte. Die Methode `usesImplAsField` nimmt, laut Cutlery-Dokumentation, den vollqualifizierten Namen des Typs als Parameter, also `"java.util.HashMap"`, wie auch in der Implementation von Prof. Schmolitzky. Es war allerdings nicht möglich diese Methode in BlueJ, wie im API Vertrag festgelegt, zu benutzen. Die Tests wurden dann Versuchsweise nochmals in einer anderen Entwicklungsumgebung(Eclipse Oxygen Version 4.7.3) durchgeführt. Hier funktionierte alles ohne Probleme, so dass die Methode `usesImplAsField` mit dem Parameter `"java.util.HashMap"` verwendet werden konnte. In BlueJ schlug der gleiche Test mit dem vollqualifizierten Namen fehl. Die Vermutung ist, dass es ein Bug in BlueJ ist oder das Probleme auftreten, wenn das Spoon-Framework mit BlueJ benutzt wird. Leider konnte bisher nicht geklärt werden, warum dieser Fehler auftritt.

```
@Test (timeout = 1000)
public void testKlasseVerwendetMap()
{
    assertTrue(fTester.hasFields());
    assertTrue(fTester.usesImplAsField("HashMap"));
    //_util.testeKlasseHatZustand(_util.gibClass("SpielfeldGeflecht
        "));
    //_util.testKlasseVerwendetImplAlsFeld("SpielfeldGeflecht","java
        .util.HashMap");
}
```

Listing 5.8: Klasse hat Zustand und ein Feld benutzt *HashMap*

Im letztem Testfall werden zwei Methoden verwendet. Als erstes wird überprüft, ob alle Felder einer Klasse *private* sind. Dabei wird wieder derselbe *FieldTester* verwendet. Die zu testende Klasse muss daher in den Parameterlisten der Methoden nicht auftauchen. Dieses macht den Test lesbar und überschaubar. Als zweites wird überprüft, ob es ein Feld vom Typ *Map* gibt. Dieses lässt sich mit der Methode `hasFieldWithType(String typeName)` überprüfen. Allerdings trat hier dasselbe Problem auf wie im vorherigen Beispiel: Bei der Verwendung von vollqualifizierten Namen schlägt der Test in BlueJ fehl. Wird allerdings nur der einfache Name benutzt, funktioniert es. Zum Vergleich wurde der Test wie zuvor in der Eclipse-Entwicklungsumgebung implementiert und funktionierte mit dem vollqualifizierten Namen, wie im API-Vertrag festgelegt. Deswegen wurde, wie in Listing 5.9 dargestellt, wieder mit dem einfachen Namen des Typs gearbeitet, da diese Variante in BlueJ funktioniert und die Tests somit wie gewohnt verwendet werden können.

```
@Test (timeout = 1000)
public void testKlasseHatFeldNachQTK()
{
    assertTrue(fTester.allFieldsPrivate());
    assertTrue(fTester.hasFieldWithType("Map"));
    //_util.testeAlleFelderPrivate(_util.gibClass("SpielfeldGeflecht
        "));
    //_util.testEinFeldInKlasseHatTyp("SpielfeldGeflecht","java.util
        .Map");
}
```

Listing 5.9: Alle Felder in Klasse *private* und ein Feld vom Typ *Map* existiert

6. Schluss

Dieses Kapitel bildet den Abschluss dieser Arbeit. Zunächst wird ein Fazit aus den in dieser Arbeit vorgestellten Inhalten gezogen. Im Ausblick wird dann erläutert, was nicht behandelt werden konnte und welche Möglichkeiten es gibt, diese Arbeit fortzusetzen.

6.1. Fazit

Ziel dieser Arbeit war es eine API zum Schreiben von Whitebox-Tests zu erstellen. Dieses geschah mit Hilfe des Spoon-Frameworks, welches die statische Code-Analyse erleichtert. Des Weiteren sollte die API, die gestellten Anforderungen der Professoren erfüllen. Was bedeutet, dass es eine gemeinsame API gibt, die zum Erstellen von Whitebox-Tests in praktischen Programmierprüfungen verwendet werden kann. Diese API sollte leicht zu benutzen sein und alle Funktionalitäten besitzen, um wie zuvor Tests für die praktischen Prüfungen zu schreiben.
Wie in dem Abschnitt 5.2 gezeigt, kann die erstellte Cutlery-Bibliothek für das Schreiben von Whitebox-Tests, genutzt werden. Die Unterschiede zu den Ursprünglichen Tests waren sehr gering, so dass die Umstellung für den Anwender einfach ist. Allerdings gab es Probleme in BlueJ, so dass der Anwender die Bibliothek nicht so benutzen konnte, wie im API-Vertrag festgelegt. Es wird daher empfohlen Cutlery in der Eclipse IDE zu verwenden, da es dort zu keinen Problemen gekommen ist. Ist allerdings eine Verwendung von BlueJ in der Programmierlehre erwünscht, können als Parameter die einfachen Namen anstatt die Vollqualifizierten, verwendet werden (s. Listing 5.8). Dieses kann jedoch zu noch nicht bekannten Seiteneffekten führen und ist daher nicht empfehlenswert.
Eine weitere Frage in dieser Arbeit war, ob sich das Spoon-Framework eignet eine API für Whitebox-Tests zu erstellen. Statische Code-Analyse ist mit diesem Framework möglich und das Parsen von Quelltexten ist sehr einfach. Jedoch ist die Einarbeitung in das Framework zunächst etwas kompliziert. Die Dokumentation ist an manchen Stellen wenig ausreichend und das Metamodell ist sehr mächtig und umfangreich. Es bedarf einige Zeit, um sich mit dem nötigen Wissen vertraut zu machen. Ein Vorteil ist, dass das Projekt Open-Source ist und man bei Fragen auf die Entwickler zukommen kann. Diese sind gerne und meist auch sehr schnell bereit Unklarheiten in der Dokumentation zu klären. Außerdem wird das Framework

regelmäßig aktualisiert, so dass Fehler beseitigt und neue Funktionen hinzugefügt werden. Abschließend ist zu sagen, dass sich das Spoon-Framework sehr gut geeignet hat die Cutlery-API zu erstellen. Möchte man allerdings nur Spoon nutzen, um Whitebox-Tests zu erstellen, sollte man sich etwas Zeit nehmen, um sich genügend mit dem Framework auseinanderzusetzen.

6.2. Ausblick

In dieser Arbeit lag der Fokus auf die Erstellung einer API zum Schreiben von Whitebox-Tests. Dabei wurden die Anforderungen analysiert und weitestgehend in die Cutlery-API übernommen. Die API ist aber noch nicht vollständig und weitere Anforderungen sind je nach Anwendung denkbar. Somit bietet es sich an Cutlery zu erweitern und weitere Funktionen zu ergänzen. Des Weiteren gibt es auch Anhaltspunkte Cutlery zu verbessern. Fehlermeldungen bei falscher Benutzung können ergänzt werden und neue Randfälle abgedeckt werden, damit eine vollständige Kapselung von dem Spoon-Framework gegeben ist. In Ausnahmefällen kann es nämlich vorkommen, dass eine *Exception* von Spoon nicht ausreichend abgefangen wird. Außerdem wäre es wünschenswert, wenn bessere Fehlermeldungen, bei Fehlschlag eines Tests, erzeugt werden.
Diese Arbeit hat sich ausschließlich mit der Analyse von Quelltexten befasst. Darüber hinaus bietet das Spoon-Framework aber auch Transformation von Quelltexten an (vgl. Pawlak u. a., 2015). Es ist also denkbar, dass sich eine andere Arbeit mit diesem Aspekt des Frameworks befasst.

A. Inhalt der beiliegenden CD

Auf dem Datenträger befinden sich folgende Dateien:

/thesis.pdf	Dieses Dokument im PDF-Format
/Anforderungsanalyse/	Dieser Ordner enthält alle im Kapitel 3 erwähnten Testklassen
/CutleryKomplett/	Dieser Ordner enthält die Implementation(inkl. JUnit-Tests) von Cutlery als ZIP-Datei
/Cutlery/	Dieser Ordner enthält Cutlery als JAR-Datei und das JavaDoc von Cutlery
/ProofOfConcept/	Dieser Ordner enthält den Proof of Concept als ZIP-Datei

B. Benutzung von Cutlery in der Eclipse IDE

Das Einfügen und Benutzen der Cutlery.jar in BlueJ wurde schon im Kapitel 5.2 demonstriert. Da es sich aber empfiehlt Cutlery mit Eclipse zu benutzen, soll dies in diesem Abschnitt kurz erläutert werden. Hierfür wird Eclipse Oxygen Version 4.7.3 verwendet. Voraussetzung ist, dass bereits ein Projekt existiert, in welches die Cutlery-Bibliothek zum Einsatz kommen soll.

1. Mit der rechten Maustaste auf das Projekt klicken. Im Menü unter *Build Path* auswählen und anschließend auf *Add External Archives* klicken.
2. Es öffnet sich das Verzeichnissystem des Computers. Dort kann die JAR-Datei ausgewählt und hinzugefügt werden.
3. Anschließend taucht die Datei im Projekt unter *Referenced Libraries* auf.

Nachdem diese Schritte erledigt wurden, kann die Bibliothek verwendet werden.

Literaturverzeichnis

[Ayewah u. a. 2008] Ayewah, N. ; Pugh, W. ; Hovemeyer, D. ; Morgenthaler, J. D. ; Penix, J.: Using Static Analysis to Find Bugs. In: *IEEE Software* 25 (2008), Sep., Nr. 5, S. 22–29. – ISSN 0740-7459

[Bloch 2006] Bloch, Joshua: How to Design a Good API and Why It Matters. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA : ACM, 2006 (OOPSLA '06), S. 506–507. – URL http://doi.acm.org/10.1145/1176617.1176622. – ISBN 1-59593-491-X

[Bloch 2014] Bloch, Joshua: *A Brief, Opinionated History of the API*. 2014. – URL https://drive.google.com/file/d/0B941PmRjYRpnbWVTajRzaUVCelk/view. – Zugriffsdatum: 2019-03-31

[Bloch 2018] Bloch, Joshua: *Effective Java*. 3. Addison-Wesley, 2018. – ISBN 978-0-13-468599-1

[Inria-Spoon 2019] Inria-Spoon: *Spoon - Source Code Analysis and Transformation for Java*. 2019. – URL http://spoon.gforge.inria.fr/. – Zugriffsdatum: 2019-04-04

[Liskov und Wing 1994] Liskov, Barbara H. ; Wing, Jeannette M.: A Behavioral Notion of Subtyping. In: *ACM Trans. Program. Lang. Syst.* 16 (1994), November, Nr. 6, S. 1811–1841. – URL http://doi.acm.org/10.1145/197320.197383. – ISSN 0164-0925

[Olsen u. a. 2018] Olsen, Klaus ; Parveen, Tauhida ; Black, Rex ; Friedenberg, Debra ; Hamburg, Matthias ; McKay, Judy ; Posthuma, Meile ; Schaefer, Hans ; Smilgin, Radoslaw ; Smith, Mike ; Toms, Steve ; Ulrich, Stephanie ; Walsh, Marie ; Zakaria, Eshraka: *Certified Tester Foundation Level Syllabus*. 2018. – URL https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html. – Zugriffsdatum: 2019-5-3

[Pawlak u. a. 2015] Pawlak, Renaud ; Monperrus, Martin ; Petitprez, Nicolas ; Noguera, Carlos ; Seinturier, Lionel: Spoon: A Library for Implementing Analyses and Transforma-

tions of Java Source Code. In: *Software: Practice and Experience* 46 (2015), S. 1155–1179. – URL https://hal.archives-ouvertes.fr/hal-01078532/document

[Spichale 2017] Spichale, Kai: *API-Design.* dpunkt.verlag, 2017. – ISBN 978-3-86490-387-8

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 6. Mai 2019 Jennifer Momsen