Universität Bremen

Fachbereich 3: Mathematik und Informatik

Masterarbeit

Automatische Testfallgenerierung mit realitätsnahen Testfällen durch Code-Instrumentierung

– für Java-Programme in Spock und Groovy –

Vorgelegt von

Anika Bracht

Matrikel-Nr.: 2589692

Abgabedatum:	9. April 2018
Erstgutachter:	Prof. Dr. Rainer Koschke
Zweitgutachter:	Prof. Dr.-Ing. Henrik Lipskoch
Fachliche Betreuung:	Tobias Stöckmann

Vorwort

Die vorliegende Masterarbeit „Automatische Testfallgenerierung mit realitätsnahen Testfällen durch Code-Instrumentierung“ schließt mein Informatikstudium an der Universität Bremen ab.
Die Grundidee des Themas entwickelte sich während meiner täglichen Arbeit am Webshop-Code. Für die Entwicklung der Fragestellung möchte ich Herrn Tobias Stöckmann und Herrn Prof. Dr. Rainer Koschke danken.
Des Weitern danke ich Herrn Prof. Dr.-Ing Henrik Lipskoch für die sehr gute Betreuung. Die Gespräche waren immer sehr aufschlussreich und motivierend.
Ein besonderer Dank geht an die Menschen, die immer ein offenes Ohr für mich hatten und mir mit ihrem Fachwissen zur Seite standen, wenn ich mal nicht weiter wusste. Damit meine ich insbesondere Herrn Carsten Schmidt und Herrn André Schreck.
Für die moralische Unterstützung möchte ich mich bei meinem Freund, meiner Mutter und meinen Schwestern bedanken. Sie haben immer an mich geglaubt und wo es geht unterstützt.

Anika Bracht

Bremen, 9. April 2018

„Du willst ja nicht die Welt retten. Du willst ja nur das Leben etwas erleichtern.“
– Prof. Dr.-Ing Henrik Lipskoch

Abstract

Im Rahmen dieser Masterarbeit wurde ein TestGenerator geschrieben, der für Java-Programme Testfälle in Groovy und Spock generiert und dabei auf realitätsnahe Daten zurückgreift.
Für die Speicherung der realitätsnahen Daten wird AspectJ verwendet. Dies hat den Vorteil, dass der zu testende Code nicht angepasst werden muss. Die Werte für die Testabdeckung werden mittels JaCoCo berechnet.
Eine 100%-ige Testabdeckung kann erreicht werden, wenn die zu testende Methode mit den notwendigen Parametern aufgerufen wird. Durch eine Umsetzung, die an einen genetischen Algorithmus angelehnt ist, wird eine minimale Anzahl von Testfällen generiert. Hierbei gilt, der Stärkere, mit der höheren Testabdeckung, gewinnt. Im Gegensatz zu anderen Testgenerierungs-Tools benötigt der TestGenerator mehr Zeit zur Erstellung der Testfälle, vor allem bei umfangreichen Projekten.
Für kleinere Projekte ist der TestGenerator nicht geeignet, da anstelle der Methodenaufrufe mit verschiedenen Werten, direkt Tests geschrieben werden können.
Durch die Verwendung von Groovy und Spock sind die generierten Tests übersichtlich und verständlich.

Abstract

As part of this master thesis a TestGenerator was developed to generate tests for Java programs, written in Groovy and Spock, using realistic data.
The realistic data will be saved with the use of AspectJ. The advantage of this is, that there is no need to change the code that's to be tested. The values for the test coverage will be provided by JaCoCo.
It is possible to reach a test coverage of 100%, if the method to be tested is called with the correct parameters. A minimum of test cases will be generated by involving a genetic algorithm. The test case with the highest test coverage is the fittest. In comparison to other test generation tools, the TestGenerator needs more time for the generation, especially for complex projects.
The TestGenerator is not suitable for small projects. Instead of calling methods with different values, the test cases could directly be implemented.
Relying on Groovy and Spock make the generated tests easy to read and understand.

Inhaltsverzeichnis

1. Einleitung

Um bei der Softwareentwicklung gewährleisten zu können, dass das Programm den Wünschen des Kunden entspricht und keine Fehler auftreten, wird das Programm während der Entwicklung mehrfach getestet. Neben den manuellen Tests existieren meist auch automatisierte Tests.
Da das Automatisieren von Tests zeitaufwendig ist gibt es zahlreiche Tools, die dabei unterstützen oder komplett selbstständig Testfälle erstellen.
Im Nachfolgenden wird die Motivation, die hinter dieser Arbeit steht, erläutert, die Zielsetzung umrissen und ein kurzer Überblick über existierende Tools zur Testgenerierung gegeben.

1.1. Motivation

Das Ziel ist es, auf Basis von realitätsnahen Daten, automatisch Testfälle generieren zu lassen. Im Rahmen dieser Masterarbeit wird sich an einem großen Gradle[1]-Webshop-Projekt orientiert. Das Backend ist in Java geschrieben und läuft auf einem Tomcat- und einem JBoss-Server. Das Projekt gibt einige Rahmenbedingungen vor und steht zur Verfügung, um den Testgenerator auszuprobieren.
Beim Webshop-Projekt wird darauf geachtet, dass für neuen Code Testfälle existieren und, dass für jeden gefundenen und behobenen Fehlerfall ein Testfall erstellt wird. Für die Tests werden das Framework Spock[2] und die Programmiersprache Groovy[3] verwendet. Das Erstellen der Mocks, das Vorbereiten der Daten, die die Methode benötigt, um aufgerufen werden zu können und ein Ergebnis zu liefern, ist am zeitaufwendigsten.
In Tabelle 1.1 auf Seite 2 sind einige Rahmendaten des Webshop-Projekts aufgeführt, die die Größe des Projekts verdeutlichen sollen. Das Projekt besteht insgesamt aus 20 Modulen. Sechs Module werden für das Backend auf dem Tomcat benötigt, zehn Module für den JBoss. Wichtig zu beachten ist, dass zwei Module von beiden Anwendungen verwendet werden. Aus diesem Grund ist die Summe der Groovy-Klassen der beiden Module größer als die Anzahl der Groovy-Klassen im gesamten Modul.
Die „Non-Comment Lines of Code“ (NCLOC) wurden mittels des IntelliJ-Plugins `Metrics-Reloaded`[4] ermittelt. Dies hat ergeben, dass die zirka 7.300 Java-Klassen (Logik und Tests) knapp 830.000 Zeilen reinen Code beinhalten. Aus ungefähr 11.000 Zeilen bestehen die 94 JUnit[5]-Klassen. Für einige Module existieren noch JUnit-Tests, für neuen Code werden aber Spock-Tests in Groovy geschrieben. Von den über 170.000 Zeilen Groovy-Code befinden sich ungefähr

[1] `https://gradle.org/`
[2] `http://spockframework.org/`
[3] `http://groovy-lang.org/`
[4] `https://plugins.jetbrains.com/plugin/93-metricsreloaded`
[5] `https://junit.org/junit5/`

1.300 Zeilen in Gradle-Dateien. Die restlichen knapp 172.000 Zeilen teilen sich auf zirka 1.000 Testklassen auf.

	Anzahl Module	Anzahl Java Klassen		NCLOC Java	Anzahl Groovy Klassen	NCLOC Groovy
		Logik	Tests			
Gesamt	20	7.300	94	830.000	1.000	170.000
Tomcat	6	3.800	28	350.000	720	110.000
JBoss	10	2.700	67	360.000	380	71.000

Tabelle 1.1.: Rahmendaten des Webshop-Projekts, Stand März 2018

Viele Tools, die Tests generieren, erstellen JUnit-Tests, wodurch sie für das Webshop-Projekt nicht geeignet sind. Zusätzlich könne diese Tools mit der Größe oder der Komplexität des Webshop-Projektes nicht umgehen und können dadurch keine Testfälle generieren. Auf einige Tools wird in Abschnitt 2.2 etwas genauer eingegangen.
Wie bereits geschrieben, wird in dem Projekt darauf geachtet, dass für neuen Code Testfälle geschrieben werden. Laut dem integrierten `SonarQube`[6] des Webshop-Projektes, in dem neun Module überprüft werden (ungefähr 810.000 NCLOC Java-Code), erreichen mehr als 7.600 Tests 22,1% der Zweigabdeckung und 24% der Zeilenabdeckung. Es gibt also genügend existieren Code, der nicht durch Tests abgedeckt ist. Die Problematik hierbei ist, dass einige Methoden sehr unübersichtlich sind und die Fachlichkeit nicht mehr ersichtlich ist. Dafür muss zusätzlich zur Testgenerierung Zeit aufgewendet werden, um die Methode zu verstehen.
Um dieser Problematik entgegenzuwirken, sollen die generierten Tests auf realen Daten aus dem Webshop basieren. Dies wird ermöglicht, indem die Methodenargumente der Methode aufgezeichnet werden, während sich jemand durch den Webshop klickt. Die Hoffnung hierbei ist, dass die Fachlichkeit der Methode einfacher zu verstehen ist.

1.2. Zielsetzung

Ziel dieser Arbeit ist es ein Tool zu entwickeln, welches, ohne zusätzliche Definitionen eines Entwicklers am Code, aussagekräftige und ausführbare Testfälle für Java-Methoden in Groovy und Spock generiert. Die Werte für diese Testfälle werden mittels AspectJ[7] vom Server abgegriffen. Der Ablauf der Testfallgenerierung ist in Abbildung 1.1 auf Seite 3 exemplarisch dargestellt.
Zunächst muss der Entwickler eine Methode auswählen, für die Tests generiert werden sollen. Dies wird in die AspectJ-Datei integriert. Das Vorgehen wird in Abschnitt 4.1 näher beschrieben. Mittels der AspectJ-Datei, dem Projekt und einem Auslöser, der die ausgewählte Methode aufruft, wird eine JSON-Datei generiert. Der Auslöser kann jemand sein, der sich durch den Webshop klickt oder eine `main`-Methode, aus der heraus die Methode aufgerufen wird. Auf weitere Details wird in Abschnitt 4.2 eingegangen.
Aus dem JSON wiederum werden die Testfälle in Groovy und Spock generiert, welche in Abschnitt 4.3 genauer beschrieben werden.

[6] `https://www.sonarqube.org/`
[7] `https://www.eclipse.org/aspectj/`

Anschließend werden die Tests ausgeführt, um sicherzustellen, dass ausführbare Tests erzeugt wurden und, um die Testabdeckung zu ermitteln. Das Ausführen der Tests wird in Abschnitt 4.4 beschrieben.
Konnte der Prozess zum Ausführen der Tests ohne Unterbrechungen durchgeführt werden, werden die Testreports erstellt. Nähre Information dazu sind unter Abschnitt 4.5 zu finden.
Nach der Notierung der Testabdeckung ist es abhängig von der Art der Ermittlung der Testabdeckung, welcher Pfad der Grafik eingeschlagen wird. Die Variante, die in der Grafik mit dem gestrichelten Pfeil dargestellt ist, führt jeden Test einzeln aus und ermittelt somit auch für jeden Test die Testabdeckung. Daher führt der Pfeil zurück zum „Führe Test(s) aus“-Knoten. Die Details zu dieser Variante sind unter Unterabschnitt 4.5.2 zu finden. In Unterabschnitt 4.5.1 ist eine Beschreibung der kurzen Variante (durchgezogener Pfeil) zu finden.
Der Entwickler hat anschließend die Möglichkeit die Bezeichner der Tests anzupassen oder weitere Testfälle hinzuzufügen, falls ihm die generierten nicht ausreichen. Was bei den Verbesserungen der Tests zu beachten ist, wird in Abschnitt 4.6 erläutert.

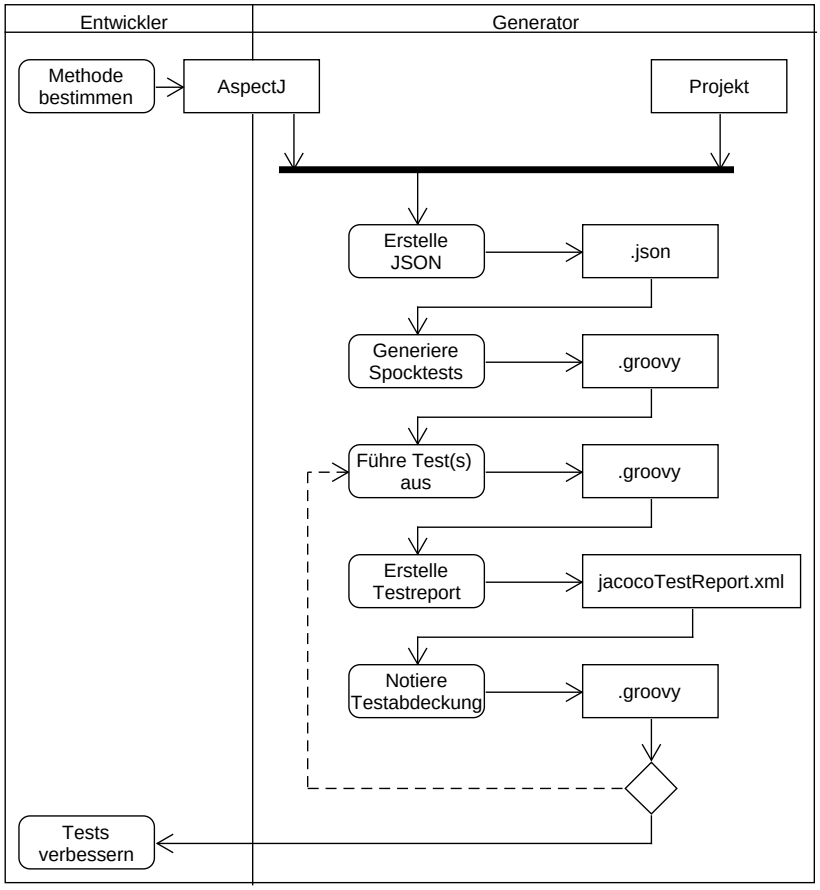

Abbildung 1.1.: Grafische Darstellung des Ablaufs der Testfallgenerierung

Es ist möglich für alle Java-Methoden einer gegebenen Webapplikation, die im Code aufgerufen werden, Testfälle generieren zu lassen, wodurch der Zeitaufwand zur Erstellung von Testfällen reduziert wird. Zusätzlich soll an den Tests erkennbar sein, wie häufig diese Werte beim Durch-

laufen des Webshops vorgekommen sind. Im besten Fall decken die Tests für eine Methode alle Zeilen der Methode ab oder es wird toter Code identifiziert.

1.3. Existierende Tools

Es gibt bereits zahlreiche Tools, die Testfälle generieren und eine hohe Zweigabdeckung bei möglichst wenig Tests als Ziel haben. Sie basieren zum Großteil auf zufällig generierten Testwerten.
Manche Tools lassen eine Einschränkung der Werte durch den Entwickler zu, sodass bestimmte Fälle direkt ausgeschlossen werden. Dies ist bei „Grenzen erschöpfenden“ (`bounded-exhaustive`) Tools wie bei [Boyapati, 2002; Marinov, 2001] der Fall, wobei hier die Zufallswerte die Grenzen nicht überschreiten.
Andere Tools wie in [Păsăreanu, 2008; Sen, 2006] beschrieben setzen auf „symbolische Ausführung“ (`symbolic execution-based`). Sie untersuchen den Code mittels symbolischer Ausführung und sammeln Bedingungen von Abzweigungen, um mit diesen Werten Testfälle zu generieren. Dadurch kann eine hohe Zweigabdeckung erreicht werden.
Beide Ansätze sind für kleinere Klassen gut geeignet, nehmen aber an Qualität ab, sobald eine gewisse Größe überschritten ist oder benötigen sehr viel Zeit [vgl. Zhang, 2011].
Bessere Ergebnisse erzielen evolutionäre Ansätze. Hierbei werden verschiedene Algorithmen angewandt, die eine Evolution simulieren. Bei einem genetischen Algorithmus wird mit Chromosomen gearbeitet. Chromosomen stellen abstrakt einen Testfall für eine Methode dar. Vorher bestimmte Eigenschaften (`Fitness Function`) entscheiden darüber, welches Chromosom sich gegenüber den anderen durchsetzt, wenn sie miteinander kombiniert werden. Des Weiteren können Chromosomen mutieren, um dadurch den Eigenschaften eher zu entsprechen[vgl. Tonella, 2004].
Ein anderer Algorithmus setzt die Partikelschwarmoptimierung (`Particle Swarm Optimization`) ein. Sie orientiert sich an einem Schwarm Vögel auf der Suchen nach Futter. Im Programm „suchen“ die Tests die bestmögliche Abdeckung des Codes. Die Verwendung dieses Algorithmus zur Generierung von Testfällen wurde in [Andalib, 2014] untersucht. Für Software ohne Datenabhängigkeiten lieferte der Algorithmus bessere Ergebnisse als der genetische Algorithmus.
Der imperialistisch kompetitive Algorithmus (`Imperialist Competitive Algorithm`) ist eine weitere Umsetzung des evolutionären Ansatzes. Hierbei wird mit Kolonien und Imperialisten gearbeitet. Das Ziel ist, dass ein Imperialist ein mächtiges Weltreich schafft, indem er andere Kolonien einnimmt [vgl. Saadtjoo, 2018]. Untersuchungen zur Optimierung dieses Algorithmus und Vergleiche mit den anderen evolutionären Algorithmen wurden in [Saadtjoo, 2018] durchgeführt und haben ergeben, dass der optimierte Algorithmus die besten Werte liefert.
Im Rahmen der „Java Unit Testing Tool Competition“[8] werden jährlich Testgenerierungs-Tools für Java-Programme evaluiert. Durch die Festlegung von Wettbewerbsbedingungen können die Tools miteinander verglichen werden und es gibt einen Anreiz die Tools weiterzuentwickeln.
Im Jahr 2017 traten die Tools `EvoSuite`[9] und `JTExpert`[10] gegeneinander an. `EvoSuite` arbei-

[8] `https://github.com/PROSRESEARCHCENTER/junitcontest`
[9] `http://www.evosuite.org/`
[10] `https://sites.google.com/site/saktiabdel/JTExpert`

tet mit einem genetischen Algorithmus [Fraser, 2015] und `JTExpert` setzt auf Zufallswerte, die durch statische Analysen beeinflusst werden [Sakti, 2016]. Zum Vergleich wurden zwei weitere Tools (`T3`[11] , `RANDOOP`[12]), die auf Zufallswerten ohne statische Analysen basieren, auf den ausgewählten Klassen ausgeführt. Des Weiteren stehen Testfälle zur Verfügung, die von Entwicklern geschrieben wurden.
Nach der fünften Runde des Wettbewerbs ist es keinem Tool gelungen die Qualität von manuell geschriebenen Tests zu erreichen. Die generierten Testfälle erreichen nicht die Güte der Zweigabdeckung, die manuell geschriebene Tests abdecken [vgl. Panichella, 2017].
Im Rahmen dieser Arbeit wird sich primär auf die Ansätze konzentriert, die mit zufälligen Werten arbeiten. Zum einen, da diese Ansätze skalierbar sind und zum anderen, da sie insgesamt bessere Ergebnisse liefern als die „statischen“ Ansätze. Zusätzlich wird sich auf den genetischen Algorithmus konzentriert, da `EvoSuite` mit diesem Algorithmus arbeitet und bereits in realen Software-Projekten angewandt wurde.

1.4. Zusammenfassung

Das Testen von Programmen ist ein zeitaufwendiger, aber notwendiger Prozess. Es gibt viele Tools, die automatisiert Testfälle generieren. Häufig werden dafür zufällige Werte und JUnit verwendet. Ein vielversprechender Ansatz ist die Nutzung eines evolutionären Ansatzes, um mit wenigen Testfällen eine hohe Testabdeckung zu erreichen [vgl. Fraser, 2015; Saadtjoo, 2018].
Im Kapitel werden der genetische Algorithmus, die Partikelschwarmoptimierung und der imperialistisch kompetitive Algorithmus vorgestellt. Der genetische Algorithmus wird von `EvoSuite` verwendet und steht auch im Fokus dieser Arbeit.
Im Rahmen dieser Arbeit wird ein Tool entwickelt, welches für Java-Programme Testfälle in Groovy und Spock erstellt. Als Basis werden real vorkommende Daten aus dem Webshop verwendet. Diese werden mittels AspectJ abgegriffen.

[11] `https://git.science.uu.nl/prase101/t3`
[12] `https://randoop.github.io/randoop/`

2. Grundlagen

Nachdem die Motivation und die Zielsetzung im vorherigen Kapitel erläutert wurden, wird im nachfolgenden Kapitel auf die Grundlagen eingegangen. Zum einen ist das Verstehen der Grundlagen notwendig, um das Vorgehen der Arbeit nachvollziehen zu können. Zum anderen geben diese Grundlagen einen Rahmen vor, in dem sich während der Arbeit bewegt wird, welcher aber nicht überschritten werden soll. Zusätzlich werden in diesem Kapitel einige Testgenerierungs-Tools und ein Vorgehen zur Bewertung von Tests vorgestellt.

2.1. Frameworks und Programmiersprachen

Der TestGenerator wird in einem bereits existierenden Projekt Anwendung finden. Dadurch werden das Testframework Spock und die Programmiersprache Groovy für Tests vorgegeben sowie das Build-Management-Tool Gradle. Zusätzlich soll der TestGenerator auch für andere Projekte anwendbar sein, die die Rahmenbedingungen erfüllen.

2.1.1. Groovy

Groovy ist eine JVM-basierte Programmiersprache welche stark an Java erinnert. Zusätzlich zur Kompatibilität zu Java ist Groovy sehr dynamisch und bietet die Verwendung von optionalen Typen an.
Wie man an den Codebeispielen Listing 2.1 und Listing 2.2 sieht, können einige Befehle in Groovy kürzer geschrieben werden als in Java. Vor allem das Initialisieren von `Lists` und `Maps` ist in Groovy übersichtlicher als in Java. Für Platzhalter in `Strings` muss in Java die `String`-Methode `format` verwendet werden. In Groovy wird der Platzhalter automatisch durch den definierten Wert ersetzt.

```
1 List list = []
2
3 Map m = [:]
4 m[1] = 'one'
5
6 String string
7   = "Ein Hase geht nach ${ort}"
8 where: ort << ['Bremen']
```

Listing 2.1: Groovy-Code

```
1 List<T> list = new ArrayList<T>();
2
3 Map<T, T> map = new HashMap<T, T>();
4 map.put(1, "one");
5
6 String string
7   = "Ein Hase geht nach %s!";
8 string = String.format(s, "Bremen");
```

Listing 2.2: Java-Code

Im Zusammenspiel mit Spock lassen sich somit mit Groovy gut lesbare und schnell verständliche Tests schreiben.

2.1.2. Spock

Spock ist ein Spezifikations- und Testsframework mit integrierten Mocking- und Stubbing-Möglichkeiten. Das Besondere an Spock ist die explizite Umsetzung der `arrange-act-assert`-Struktur bei Tests. Hierbei werden zuerst alle verwendeten Klassen aufgeführt und initialisiert (`arrange`). Anschließend wird die zu testende Methode ausgeführt (`act`) und schlussendlich das erwartete Ergebnis mit dem tatsächlichen Ergebnis verglichen (`assert`) [vgl. Kapelonis, 2017]. Durch die Schlagwörter `given`, `when` und `then` sind die Tests übersichtlich und, dass Testfälle mit einem englischen Satz definiert werden, erleichtert das Verstehen eines Tests ungemein. Dies sieht man gut an dem Beispieltest in Listing 2.3 Dies wird sich auch in generierten Tests wiederfinden und macht sie so lesbarer.

```
class TestSpec extends Specification {
    def 'Should order a string array alphabetically' () {
        given: 'a string array'
        String[] stringArray = ['z','e','c','d']

        when: 'the array is ordered'
        Arrays.sort(stringArray)

        then: 'the array is sorted alphabetically'
        stringArray == ['c', 'd', 'e', 'z']
    }
}
```

Listing 2.3: Beispiel Spock-Test

2.1.3. Gradle

Gradle ist ein Build-Management-Tool, welches auf einer Groovy-ähnlichen Sprache basiert. Die Abhängigkeiten und Tasks für Projekte werden in `build.gradle`-Dateien definiert. Hierbei können vorgegebene Standards verwendet oder eigene Skripte geschrieben werden.
In Listing 2.4 ist die `build.gradle` des TestGenerator-Projektes zu sehen. In der ersten Zeile wird ein Versionsname angegeben. Dass es sich um eine Java-Applikation handelt, ist in der dritten Zeile beschrieben. Die Zeilen (`sourceCompatibility` und `targetCompatibility`) darunter definieren, dass mit und für Java 8 entwickelt wurde. Anschließend wird in `repositories` festgelegt aus welchem Repository die notwendigen Libraries verwendet werden. Die Abhängigkeiten (Libraries) sind unter `dependencies` aufgelistet. Hierbei handelt es sich um `Jackson`-Abhängigkeiten, die verwendet werden, um aus Java-Objekten JSON zu erzeugen und es wieder einzulesen.

```
version '1.0-SNAPSHOT'

apply plugin: 'java'
apply plugin: 'application'

mainClassName = 'com.master.Main'

sourceCompatibility = 1.8
```

```
9  targetCompatibility = 1.8
10
11 repositories {
12     mavenCentral()
13 }
14
15 dependencies {
16     compile 'com.fasterxml.jackson.core:jackson-core:2.8.5'
17     compile 'com.fasterxml.jackson.core:jackson-databind:2.8.5'
18     compile 'com.fasterxml.jackson.core:jackson-annotations:2.8.5'
19
20     compile project (':TestGeneratorJson')
21 }
```

Listing 2.4: `build.gradle` des TestGenerator-Projektes

2.1.4. JaCoCo

JaCoCo[13] (Java Code Coverage) ist eine Code-Coverage-Bibliothek, für die es zahlreiche Plugins gibt, sodass sie fast überall eingebunden werden kann.
Die Anleitung zur Einbindung in Gradle ist auf der Gradle-Webseite[14] beschrieben. Unter `JacocoTestReport` kann definiert werden, wo die erzeugten HTML-Dateien abgespeichert werden sollen und ob XML- oder CSV-Dateien erzeugt werden sollen.

Arten der Abdeckung

Es gibt verschiedene Abdeckungsarten oder Arten die Effektivität von Tests zu bewerten. Im Folgenden werden kurz die Arten beschrieben, welche sich im XML von JaCoCo wiederfinden (ein Ausschnitt ist in Abbildung 2.1 zu sehen). Die Beschreibungen sind an die Dokumentation von JaCoCo[15] angelehnt.

```
− <method name="isInteger" desc="(Ljava/lang/String;)Z" line="6">
    <counter type="INSTRUCTION" missed="2" covered="18"/>
    <counter type="BRANCH" missed="3" covered="3"/>
    <counter type="LINE" missed="1" covered="7"/>
    <counter type="COMPLEXITY" missed="3" covered="1"/>
    <counter type="METHOD" missed="0" covered="1"/>
```

Abbildung 2.1.: Auschnitt aus JaCoCo-XML für Methode `isInteger()`

Anweisungsabdeckung oder Instruction-Coverage kann am einfachsten ermittelt werden und richtet sich nach den Anweisungen, wie Zuweisungen oder Methodenaufrufe, im Code. Dabei werden Zeilenumbrüche nicht berücksichtigt. Diese Abdeckung ist einfach zu berechnen, sagt aber nicht sehr viel aus.

Zweigabdeckung oder Branch-Coverage betrachtet die Abzweigungsmöglichkeiten die durch `if`- oder `switch`-Abfragen zustande kommen.

[13] `https://www.jacoco.org/jacoco/trunk/index.html`
[14] `https://docs.gradle.org/current/userguide/jacoco_plugin.html`
[15] `http://www.eclemma.org/jacoco/trunk/doc/counters.html`

Zeilenabdeckung oder Line-Coverage zählt die Zeilen, in der mindestens eine Anweisung steht. Wurde eine Anweisung der Zeile ausgeführt, gilt die Zeile als durchlaufen. Eine Art der grafischen Darstellung ist in Abbildung 2.2 zu sehen.

Zyklomatische Komplexität oder Cyclomatic Complexity bezieht sich auf die Wertigkeit einer Methode nach [McCabe, 1976]. Der Wert gibt an, wie viele Testfälle benötigt werden, um alle Abzweigungen abzudecken.

Methodenaufrufe oder Method executions gelten für eine Methode, sobald eine ihrer Anweisungen ausgeführt wurde. Jede nicht abstrakte Methode besitzt mindestens eine Anweisung.

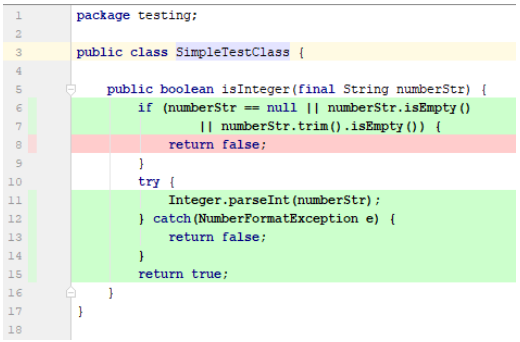

Abbildung 2.2.: Grafische Darstellung der Zeilenabdeckung in IntelliJ

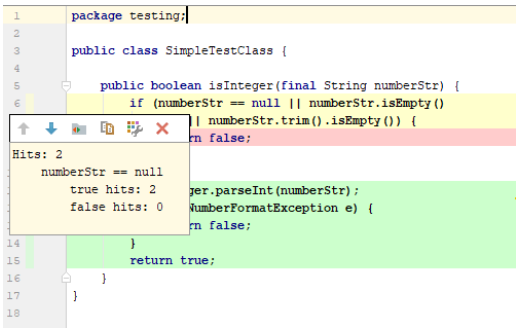

Abbildung 2.3.: Grafische Darstellung der Zeilen- und Branchabdeckung in Kombination in IntelliJ

Meist werden die Zweigabdeckung und die Zeilenabdeckung in Kombination betrachtet, um zu bewerten, ob ein Test eine Methode oder eine Klasse ausreichend abdeckt, so wie es in Abbildung 2.3 zu sehen ist. Und zusätzlich werden in vielen IDEs die Prozentwerte der verschiedenen Abdeckungen ausgegeben (siehe Abbildung 2.4), sodass auf einem Blick erkennbar ist, ob eine Klasse oder eine Methode komplett abgedeckt wurden.

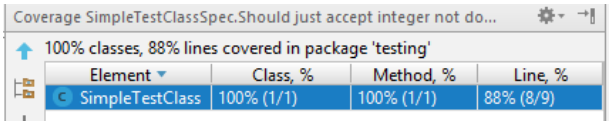

Abbildung 2.4.: Prozentuale Abdeckungsdarstellung in IntelliJ

2.1.5. AspectJ

AspectJ ist eine Erweiterung der Programmiersprache Java, die in der aspektorientierten Programmierung Anwendung findet. Sie ermöglicht es Logging, Sicherheit und Persistenz von der Geschäftslogik zu trennen. Dafür wird die AspectJ-Datei beim kompilieren in die Java-Datei „eingewoben“ [vgl. Böhm, 2006, S. 15 f.].
Um mittels eines Aspects Code in eine Klasse zu integrieren, muss diese Stelle im Code zunächst mithilfe eines `Pointcuts` angesprochen werden. Die `Pointcuts` beziehen sich nicht direkt auf den Code, sondern auf `Joinpoints` im Code. `Joinpoints` bezeichnen Stellen im Code wie Methoden-

und Konstruktoraufrufe, Initialisierungen von Variablen und Zugriffe auf Variablen. Der Beispiel-`Pointcut` in Listing 2.5 reagiert auf den Aufruf der öffentlichen Methode `add()` mit zwei Zahlenwerten. Methoden, deren Sichtbarkeit, Name, Parameteranzahl, -typ oder -bezeichnung sich unterscheiden, werden nicht angesprochen.

```
1 pointcut callAddMethod(int firstNumber, int secondNumber):
2         call(public int add(int, int)) && args(firstNumber, secondNumber);
```

Listing 2.5: Beispiel `Pointcut`

Ein `Pointcut` alleine ändert noch nichts am Code. Er muss über einen `Advice` aufgerufen werden. Dieser bestimmt auch, ob der einzuwebende Code vor dem Ausführen (`before-Advice`) des gewählten `Joinpoints` geschieht, dannach (`after-Advice`) oder währenddessen und ihn beeinflussen kann (`around-Advice`). In dem Beispiel in Listing 2.6 wird vor dem Aufruf der Methode `add()` in der Konsole der Name der Methode ausgegeben, welcher mittels `thisJoinPoint` ermittelt wird. Es gibt noch weitere `Advices`, die auf Exceptions oder Rückgabewerte reagieren.

```
1 before(int firstNumber, int secondNumber): callAddMethod(firstNumber,
    ↪ secondNumber) {
2   System.out.println("Calls: " + thisJoinPoint.getSignature().getName());
3 }
```

Listing 2.6: Beispiel `before-Advice`

Eine Besonderheit, auf die man achten muss, wenn man auf private Methoden zugreifen möchte, ist, dass der Aspect als `privileged` gekennzeichnet sein muss. Dies ist nicht möglich, wenn die Annotations-Schreibweise verwendet wird (`@Aspect` in java-Datei).
Um AspectJ in ein Gradle-Projekt einzubinden, muss der `gradle.build` das AspectJ-Plugin hinzugefügt werden.Der Code dafür ist auf der offiziellen Webseite von Gradle[16] zu finden.

2.2. Testgenerierungstools

Wie bereits beschrieben, gibt es einige Tools, die Testfälle generieren. Meist wird hierbei auf zufällige Werte gesetzt, um eine hohe Zweig- und Zeilenabdeckung zu erreichen [Pacheco, 2007; Sakti, 2016; Fraser, 2015; Zhang, 2011]. Manchmal wird die Mutationsabdeckung zu Rate gezogen, um zu überprüfen wie stabil die generierten Tests sind [vgl. Panichella, 2017]. Durch die vielen zufällig generierten Werte liegt die Vermutung nahe, dass die Tests Veränderungen, also Mutationen, im Code aufdecken.
Im Folgenden werden einige Vorgehensweisen von Testgenerierungstools näher beschrieben.

2.2.1. RANDOOP

Das Tool `RANDOOP` arbeitet mit Feedback gesteuerten Zufallstestfällen um für Java-Programme Tests zu generieren. Dafür werden Aufrufe der zu testenden Methode mit verschiedenen Werten erstellt, ausgeführt und das Verhalten als Erwartung abgespeichert. Im Rahmen dieser Erwartungen werden anschließend mit Zufallswerten bestückte Tests generiert.

[16] `https://plugins.gradle.org/plugin/aspectj.gradle`

Es ist möglich das Verhalten des Tool mittels Annotationen auf Methodenebene und sogenannten „Verträgen“ zu beeinflussen [Pacheco, 2007]. Dies verändert die Art wie `RANDOOP` mit markierten Klassen umgeht oder wann der Entwickler bei ungewöhnlichen Werten benachrichtigt werden soll.

2.2.2. JTExpert

Das Tool `JTExpert` arbeitet mit zufälligen Testwerten, die durch statistische Analysen beeinflusst werden [vgl. Sakti, 2016]. Für die Analyse wird untersucht, welche Methoden welche Variablen häufig ändern und ob und wann die Methode von der zu testenden Klasse aufgerufen wird. Somit werden Methoden, die keinen Einfluss auf die zu testende Klasse haben, direkt ausgefiltert.
[Sakti, 2016] ist zu entnehmen, dass `JTExperte` nicht für alle Klassen des Wettbewerbs Testfälle in der vorgegebenen Zeit generieren konnte. Dennoch wird angegeben, dass `JTExpert` sich gut für große Klassen eignet. Der Wert für die Mutations-Abdeckung sei geringer als der des Tools `T3`, da wesentlich weniger Testfälle generiert wurden.

2.2.3. Palus

Ein hybrid-automatisiertes Testgenerierungs-Tool für Java ist `Palus`. Es wurde entwickelt um der Problematik entgegenzuwirken, dass zufällig generierte Testfälle auch „illegale“ Testfälle erzeugen [vgl. Zhang, 2011]. Zum Beispiel Methoden in einer Reihenfolge aufruft, die nicht vorgesehen ist und so Fehler erzeugt.
Mittels der dynamischen Analyse wird ein Aufruf-Sequenz-Model erstellt, welche mit Informationen aus der statischen Analyse erweitert werden. Die statische Analyse liefert Daten darüber, wie stark Methoden voneinander abhängig sind und welche Methoden welche Variable modifizieren. Anschließend wird ein Testgenerator mit diesen Daten befüllt, sodass er zufällige Testwerte erzeugen kann, die aber dennoch „legal“ sind, im Sinne des Programms.
Zusätzlich ist es möglich „Test Orakel“ zu definieren, die Theorien enthalten, die immer zutreffen. Diese Theorien werden von `Palus` zusätzlich überprüft.

2.2.4. EvoSuite

`EvoSuite` ist ein Such-basiertes Tool, das unter Verwendung eines genetischen Algorithmus automatisiert Testfälle für Java Klassen generiert. Die Suche bezieht sich auf die Suche von Zweigen im Code, die abgedeckt werden müssen. Die Chromosomen für den genetischen Algorithmus sind komplette Testklassen, die aus mehreren Testfällen bestehen. Es wird die Testklasse genommen, die die größte Codeabdeckung erzielt [vgl. Fraser, 2015].
Das Generieren von Zufallswerten kann dazu führen, dass die Laufzeit der Testgenerierung sehr lang ist. Um dieses Problem einzugrenzen, verwendet `EvoSuite` Timeouts [vgl. Fraser, 2011].
Ein weiterer Nachteil von Tests auf Basis von Zufallswerten sind inkonsistente Tests. Im Durchschnitt generiert `EvoSuite` 0.4 dieser Tests pro Durchlauf [vgl. Fraser, 2011]. Die Frage hierbei ist, ob diese Zahl zustande kommt, nachdem die Optimierungen durchgelaufen sind und falls dem so ist, wie viele dieser Tests werden initial erzeugt?

2.3. Bewertung von Tests

Im Rahmen der Arbeit von [Hu, 2013] wurde die Schätzung von Software-Zuverlässigkeit mittels modifizierten adaptiven Testen untersucht.
Was können Tests über die Software aussagen? Entweder gibt es Tests die fehlschlagen und somit ist die Software fehlerhaft, da sie sich nicht so verhält wie es erwartet wird. Oder die Test laufen ohne Fehler durch, was bestätigt, dass die Software das tut, wofür sie entwickelt wurde oder es fehlen Tests, die die Funktionalität überprüfen.
Um den Grad der Zuverlässigkeit einer Software besser bestimmen zu können als „erfolgreich“ und „fehlerhaft“, werden die Fehler von [Hu, 2013] in zwei Klassen unterteilt: „Kritische Fehler“ und „Unkritische Fehler“. Die Bewertung der Fehler muss von jemandem vorgenommen werden, der die Software und ihre Anforderungen kennt.
Mittels dieser Einteilung kann die Zuverlässigkeit einer Software besser beurteilt werden. Eine Software, die viele unkritische Fehler aufweist ist zuverlässiger als eine, die wenige kritische Fehler wirft.
Eine Art Gewichtung der Testfälle wird auch im Rahmen dieser Arbeit vorgenommen. Dadurch, dass mehrere Durchläufe in der Applikation protokolliert werden, kann an den Testfällen vermerkt werden, wie häufig sie vorkamen und wie viel Testfälle es insgesamt gab. Dadurch kann der Entwickler abschätzen, welche Auswirkung das Fehlschlagen des Testfalls hat.

2.4. Zusammenfassung

Dadurch, dass der TestGenerator an einem existierenden Projekt getestet wird, ergeben sich einige Abhängigkeiten. Die Ausgaben des TestGenerators müssen daher Groovy-Klassen sein, die durch das Spock-Framework zu ausführbaren Tests interpretiert werden. Zusätzlich ist zu beachten, dass es sich um Gradle-Module handelt und somit die Abhängigkeiten mittels einer `build.gradle`-Datei angegeben werden.
Da JaCoCo bereits im existierenden Projekt integriert ist und es sich leicht in Programme integrieren lässt, wird es für die Ermittlung der Code-Abdeckungen verwendet.
Für das protokollieren der Parameter wird AspectJ verwendet. Dadurch ist es nicht notwendig den zu testenden Code anzupassen. Stattdessen werden mittels `Pointcuts` und `Advices` Codestellen angesprochen.
Es existieren zahlreiche Testgenerierungs-Tools, welche alle unterschiedliche Ansätze verfolgen. Zum Erreichen einer hohen Code-Abdeckung haben sich Generierungs-Tools mit Zufallswerten bewährt. Verbesserungsbedarf besteht hinsichtlich der Stabilität der Tests und der Reduzierung der Tests ohne Verluste in der Code-Abdeckung hinzunehmen. Erste Erfolge wurden mit evolutionären Ansätzen erzielt.

3. Entwurfsentscheidungen

Nach der Klärung der Grundlagen wird in diesem Kapitel auf die Entwurfsentscheidungen und deren Begründungen eingegangen, die vor der Implementierung getroffen wurden. Die Entscheidungen grenzen den Rahmen dieser Arbeit weiter ab.

3.1. Code-Instrumentierung

Um das Ziel zu erfüllen, dass die Testfälle realitätsnahe Daten enthalten, müssen die aufgerufenen Methoden und die Parameterwerte ermittelt und abgespeichert werden. Zusätzlich können die aufgerufenen Zeilen dokumentiert werden, um dadurch eine grobe Zeilenabdeckung zu erhalten. Für das Erkennen, welche Zeilen durchlaufen werden, bietet sich die Nutzung eines Profilers an. Profiler sind primär dafür gedacht, dass sie Speicher- und Performance-Defizite aufdecken. Das Auslesen von Parameterwerten ist nur mit Änderungen am zu untersuchenden Code möglich. Dies soll vermieden werden, daher eignet sich diese Variante nicht.
Mittels aspektorientierter Programmierung ist es möglich spezielle Methoden ohne Anpassungen am Code anzusprechen und Daten wie Parameterwerte, Kontext und Methodennamen ausgeben zu lassen. Alle aufgerufenen und durchlaufenen Zeilen können damit nicht dokumentiert werden, da nur Informationen über die angesprochenen Methodenaufrufe verfügbar sind und dazwischen sicher andere Operationen durchgeführt werden. Stattdessen können die Informationen der Methodenaufrufe miteinander verglichen werden. Die Zeilenabdeckung kann nachträglich mit dem generierten Test ermittelt werden.

3.2. JSON

Die mittels Code-Instrumentierung gesammelten Daten werden in einer Datei im JSON-Format abgespeichert. Eine JSON-Datei kann relativ simpel mit Java erstellt und ausgelesen werden. Zusätzlich ist eine Datei im JSON-Format durch die Struktur übersichtlich und kann so schnell von Menschen erfasst und verstanden werden. Außerdem lässt sich die Struktur sehr flexibel erweitern oder reduzieren. Bei einer Datenbank müssen die Tabelle und der Code erweitert werden, um dies umzusetzen.
In Listing 3.1 ist anhand einer Methode `div()` in der Klasse `Calculator` dargestellt, wie die JSON-Struktur für den TestGenerator aussieht. Die Methode bekommt zwei `Integer` als Parameter übergeben und gibt das Ergebnis der Rechnung zurück. Bevor die Operation an den übergebenen Parametern ausgeführt wird, wird die Methode `isZero()` aufgerufen, die überprüft, ob der zweite Parameter der `div()` Methode 0 ist. Dies trifft zu, daher gibt `isZero()` `true` zurück.

```
[
  {
    "selectedMethod": {
      "isStatic": false,
      "packageName": "com.master",
      "className": "Calculator",
      "globalVariables": [
        "variableName": "
            ↪ operationCounter",
        "packageName": "",
        "className": "int",
        "annotation": ""
      ],
      "calledMethod": {
        "methodName": "div",
        "parameters": [
          {
            "type": "Integer",
            "value": "10"
          },
          {
            "type": "Integer",
            "value": "0"
          }
        ],
        "lineNumber": 7
      },
      "returnValue": {
        "type": "Integer",
        "value": "0",
        "errorCode": null,
        "errorMessage": null
      }
    },
    "calledMethods": [
      {
        "isStatic": false,
        "packageName": "com.master
            ↪ ",
        "className": "Calculator",
        "globalVariables": [],
        "calledMethod": {
          "methodName": "isZero",
          "parameters": [
            {
              "type": "Integer",
              "value": "0"
            }
          ],
          "lineNumber": 15
        },
        "returnValue": {
          "type": "Boolean",
          "value": "true",
          "errorCode": null,
          "errorMessage": null
        }
      }
    ],
    "calledLines": [
      7,
      15
    ],
    "moduleName": "Master"
  }
]
```

Listing 3.1: Beispiel zur Verdeutlichung der JSON-Struktur

Zunächst wird die ausgewählte Methode beschrieben. Es wird dokumentiert, ob sie statisch ist und der Paketname sowie der Klassennamen werden notiert. Bei den globalen Variablen werden der Variablen-, der Paket- und der Klassenname sowie die Annotation vermerkt. Von der ausgewählten Methode selbst findet man den Methodennamen, die Liste der übergebenen Parameter und die Zeilennummer, in der die Methode in der Klasse beginnt. Von den Parametern werden der Typ und der Wert als `String` aufgezeichnet. Zum Schluss wird der Rückgabewert dokumentiert. Er enthält, wie bei den Parametern, den Typen und den Wert des Rückgabewertes als `String`. Die zwei zusätzlichen Felder `errorCode` und `errorMessage` sind vorbereitend dafür, dass später Methoden aufgerufen werden, die Exceptions zurückgeben. Da im Beispiel keine Exception geworfen wird, sind die Felder mit dem Wert `null` belegt.
Anschließend werden die Methoden aufgelistet, die aus der ausgewählten Methode heraus aufgerufen werden. Hierbei wird beim Beschreiben der Methoden dieselbe Sturktur verwendet.

Die Liste der aufgerufenen Zeilen (`calledLines`) enthält alle Zeilennummern, die Code enthalten, der aufgerufen wurde. Das JSON wird abgeschlossen mit dem Modulnamen. Dieser wird benötigt, um die generierte `groovy`-Datei am richtigen Ort zu speichern.
Der Übersichtlichkeit wegen ist in Abbildung 3.1 die Struktur der JSON-Klassen als Diagramm dargestellt. Hierbei können mit einem Blick die Abhängigkeiten erkannt werden.

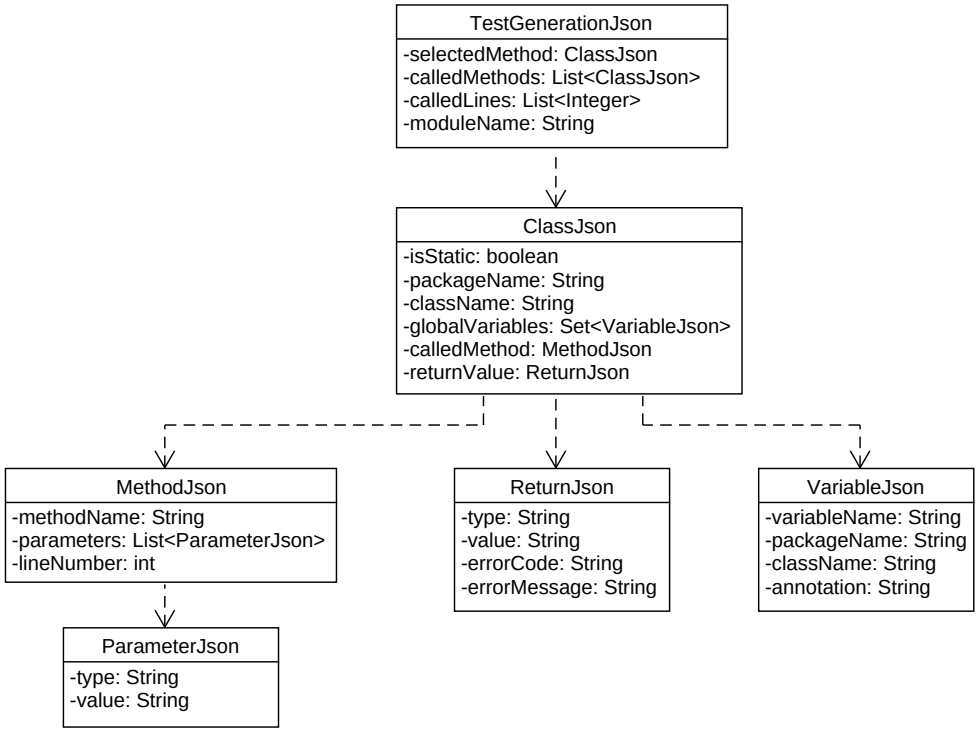

Abbildung 3.1.: Vereinfachte Klassendiagrammdarstellung der JSON-Struktur

3.3. Entwicklungsserver

Der Testgenerator wird auf einer lauffähigen Webshopapplikation getestet und entsprechend dafür entwickelt. Diese Applikation wird auch in der Produktion verwendet, wodurch sich die Möglichkeit ergibt, die Testfälle mit realen Daten von Kunden zu erstellen.
Die Problematik hierbei ist, dass mehrere 1000 Kunden pro Minute auf dem Webshop unterwegs sind. Innerhalb kürzester Zeit würden Unmengen von Daten gesammelt werden, die sich vermutlich sehr ähneln, da die meisten Kunden einem Pfad folgen. Und es kann zu diesem Zeitpunkt nicht sichergestellt werden, dass AspectJ mit diesen Mengen an Daten umgehen kann.
Zusätzlich kann es sein, dass die zu testende Methode etwas spezieller ist und von keinem Kunden aufgerufen wird. Des Weiteren kann nicht vorherbestimmt werden ob und, falls ja, wie stark die Code-Instrumentierung die Performance des Webshops beeinträchtigt.
Abhängig von der zu testenden Methode können personenbezogene Daten protokolliert werden. Diese müssten für die Generierung von Testfällen laut der Datenschutz-Grundverordnung anonymisiert oder pseudonymisiert oder die Verwendung vom Kunden genehmigt werden (§71(1) BDSG neu). Da diese Problematik nicht im Rahmen der Arbeit behandelt wird, wird vorerst

davon abgeraten Methoden in der Produktion zu protokollieren.
Die Alternative dazu ist, dass der Generator auf dem Entwicklungsserver ausgeführt wird. Anstelle von realen Kunden, klicken sich Entwickler oder Tester durch den Webshop. Die dadurch gewonnenen Daten können vielfältiger sein, da verschiedene Buchungspfade ausprobiert werden können.
Bei der daraus resultierenden Statistik muss im Hinterkopf behalten werden, dass Kunden vermutlich häufiger einen Standardfall ausführen und in der Entwicklung primär Bugs aufgedeckt und nachgestellt werden.
Ein weiterer Vorteil der lokalen Verwendung ist, dass die Testabdeckung, welche von Tests und Code im Repository ermittelt wird, nicht verfälscht wird. Mittels AspectJ wird neuer Code in die Klassen integriert, wodurch Zweige entstehen können, die durch Tests nicht abgedeckt werden.

3.4. Zusammenfassung

Vor der Umsetzung der Arbeit wurden einige Entscheidungen getroffen, die das Arbeiten erleichtern oder bei der Implementierung beachtet werden müssen.
Es wurde früh genug erkannt, dass sich, anstelle eines Profilers, aspektorientierte Programmierung eher für das Lösen des Problems eignet. Für das Speichern der Daten wurde sich für Dateien im JSON-Format entschieden.
Zusätzlich wurde festgelegt, dass das Programm nicht auf den Produktivservern laufen wird, da nicht abschätzbar ist, ob und wie die Performance beeinflusst wird. Des Weiteren ist nicht sichergestellt, dass die Datenschutz-Grundverordnung im Bezug auf reale Daten eingehalten wird. Außerdem ist nicht sichergestellt, dass die selektierte Methode im normalen Tagesgeschäft aufgerufen wird.

4. Entwicklung

In diesem Kapitel wird auf die einzelnen Stationen in Abbildung 1.1 näher eingegangen und die Implementierung dahinter erläutert.
Für die Entwickelung wurde primär Eclipse[17] verwendet, da das AspectJ-Plugin für diese Entwicklungsumgebung Verweise zwischen Code und Aspect anzeigt und dies das Entwickeln der passenden `Pointcuts` erheblich vereinfachte. Die Ausführung mit Eclipse ist nicht möglich, da die Projekte Groovy-Tests enthalten und die Plugins für Groovy und AspectJ für Eclipse nicht kompatibel sind. Dies ist bei IntelliJ[18] nicht der Fall. Daher wurde in dieser Entwichlungsumgebung der Code ausgeführt und getestet.

4.1. Methodenbestimmung

Nachdem AspectJ in das Modul integriert wurde, kann das Grundgerüst der AspectJ-Datei in das Modul kopiert werden. In der AspectJ-Datei befinden sich Dummy-`Pointcuts`, welche angepasst werden müssen.
Zuerst muss in der `String`-Variable `MODULE_NAME` der Modulname mit Pfad eingetragen werden, in dem die AspectJ-Datei liegt. Die zweite `String`-Variable, die zu füllen ist, hat den Namen `FILE_NAME` und beinhaltet den Speicherort und den Dateinamen der JSON-Datei. Diese können selbst gewählt werden, damit der Anwender die Datei schnell findet.
Um das JSON akkurat darstellen zu können, muss die JSON-Struktur aus dem TestGenerator-Modul in dem Projekt, in das die AspectJ-Datei hinzugefügt wurde, eingebunden werden.

4.1.1. Methodenauswahl

Listing 4.1 zeigt einen Dummy-`Pointcut`, der dann angesprochen wird, wenn eine Methode mit dem Methodenkopf, welcher unter `$methodHeader` angegeben ist, aufgerufen wird. Er bezieht sich also auf die Methode, die vorher als zu protokollieren ausgewählt wurde. Die Variable `$methodHeader` muss den Zugriffsmodifikator, den Rückgabetypen, den Methodennamen und, falls vorhanden, die Typen der Parameter in Klammern hinter den Methodennamen beinhalten. Es ist möglich mit Platzhaltern zu arbeiten und „*“ anstelle eines Rückgabetypen (notwendig, wenn eigene Objekte zurückgegeben werden) und „..“ anstelle der Parametertypen zu verwenden. `$parameterWithNames` enthält die komplette Parameterliste der Methode mit Typ und Namen in der Reihenfolge, wie sie an der Methode stehen. In `$parameterNames` werden die Namen der Methodenparameter, mit Komma separiert, angegeben.
Zuletzt wird ausgeschlossen, dass der `Pointcut` angesprochen wird, wenn die Spock-Tests ausgeführt werden. Ohne diese Ausnahme würden beim Durchlaufen des TestGenerators JSON-

[17] `https://www.eclipse.org/`
[18] `https://www.jetbrains.com/idea/`

Dateien erstellt werden, die denen sehr ähneln, welche als Basis des TestGenerators dienen. Da alle Spock-Tests auf „Spec“ enden, können alle Tests durch den Befehl `!within(*Spec)` ausgeschlossen werden.

```
42    pointcut callAnyMethod($parameterWithNames):
43            call($methodHeader) && args($parameterNames) && !within(
                 ↪ TestGeneratorAspect) && !within(*Spec);
```

Listing 4.1: Dummy-`Pointcut` für Methodenseletion

4.1.2. Methodenaufrufe

Des Weiteren gibt es einen Dummy-`Pointcut` (siehe Listing 4.2), der auf alle Methoden (`call(* * (..))`) zutrifft, die aus der zuvor selektierten Methode aufgerufen wurden (`withincode`), wobei die AspectJ-Datei selbst ausgeschlossen wird (`!within(TestGeneratorAspect)`), da sonst Schleifen auftreten können. Wie eben beschrieben, muss `$methodHeader` durch den Methodenkopf der selektierten Methode ersetzt werden. Genauso wie beim vorherigen `Pointcut`, werden auch hier die Testfälle ausgeschlossen.

```
163   pointcut anyMethodCalledBy():
164           call(* *(..)) && withincode($methodHeader) && !within(
                 ↪ TestGeneratorAspect) && !within(*Spec);
```

Listing 4.2: Dummy-`Pointcut` für Methodenaufrufe

4.2. JSON-Erstellung

Damit `Pointcuts` wirksam werden, müssen sie von `Advices` aufgerufen werden. Im Nachfolgenden werden die `Advices` für die `Pointcuts`, welche im vorherigen Kapitel erklärt wurden, aufgelistet und beschrieben.

4.2.1. Ausgewählte Methode

Das `before-Advice`, welches sich auf den ersten `Pointcut` aus Listing 4.1 bezieht, ist für das Füllen des JSONs für die ausgewählte Methode zuständig. Dabei wird die JSON-Struktur verwendet, welche in Abschnitt 3.2 beschrieben wurde.

Globale Variablen

Zunächst wird über die Felder der selektieren Klasse iteriert, um die globalen Variablen niederschreiben zu können. Hierbei werden Felder ausgeschlossen, deren Packagename „org.aspectj.lang“ oder deren Feldname „$jacocoData“ ist. Die `jacoco`-Felder existieren für die Testabdeckung, welche für die JSON-Erstellung nicht interessant ist. Die Felder mit dem `aspect`-bezogenen Packagenamen sind für AspectJ wichtig, aber nicht für die JSON-Erstellung, da hierbei die Klasse ohne AspectJ-Einflüsse dargestellt werden soll. Von allen anderen Feldern werden der Variablenname, der Packagename und der Klassenname und sowie die Annotation im JSON gespeichert.

Übergabeparameter

Wie in Listing 4.3 zu sehen ist, stehen die Parameter der Methode direkt am JoinPoint und können mit `getArgs()` aufgerufen werden. Alle Objekte werden mit Klassennamen und Wert als ParameterJson gespeichert. Wobei als Wert eingetragen wird, was bei dem Objekt von der `toString()`-Methode zurückgegeben wird. Wenn der Wert `null` ist, wird als Klassenname „Object“ gespeichert, da der Typ nicht bestimmt werden kann.

```
138         Object[] args = thisJoinPoint.getArgs();
139         if (args != null) {
140             for (Object o : args) {
141                 ParameterJson parameterJson;
142                 if (o == null) {
143                     parameterJson = new ParameterJson("Object", null);
144                 } else {
145                     parameterJson = new ParameterJson(o.getClass().
                        ↪ getSimpleName(), o.toString());
146                 }
147                 parameterJsonList.add(parameterJson);
148             }
149         }
```

Listing 4.3: Parameter aus JoinPoint schreiben

Die somit erstellte Liste aus ParameterJson wird neben dem Namen der Methode (`thisJoinPoint.getSignature().getName()`) und der Zeilennummer `thisJoinPoint.getSourceLocation().getLine()` an ein MethodJson übergeben (siehe Listing 4.4). Um auf einen Blick vergleichen zu können, welche Parameterkombinationen welche Zeilen aufrufen, werden alle Zeilennummern zusätzlich in einer Liste gespeichert, welche zum Schluss zum TestGeneratorJson hinzugefügt wird.

```
150         MethodJson methodJson = new MethodJson(thisJoinPoint.getSignature().
                ↪ getName(), parameterJsonList,
151                 thisJoinPoint.getSourceLocation().getLine());
```

Listing 4.4: MethodJson befüllen

ClassJson-Befüllung

Neben dem Klassennamen, dem Paketnamen und einem `boolean`-Wert für statische Methoden, wird das ClassJson mit dem MethodJson befüllt (siehe Listing 4.5).

```
154         if (thisJoinPoint.getSignature() != null) {
155             className = thisJoinPoint.getSignature().getDeclaringType().
                    ↪ getSimpleName();
156             packageName = thisJoinPoint.getSignature().getDeclaringType().
                    ↪ getPackage().getName();
157         }
158         calledLines.add(thisJoinPoint.getSourceLocation().getLine());
159         return new ClassJson(Modifier.isStatic(thisJoinPoint.getSignature().
                ↪ getModifiers()), packageName, className,
160                 methodJson);
```

Listing 4.5: Speicherung im ClassJson

Rückgabeparameter

Um den Mocks in den Tests die korrekten Rückgabewerte zurückgeben zu lassen, müssen diese mit in das JSON gespeichert werden. Hierfür wird ein `after returning Advice` verwendet. Als Rückgabewert ist alles gültig, solange es von `Object` ableitet. Auch hier wird sich wieder auf den `Pointcut` aus Listing 4.1 bezogen (`callAnyMethod($parameterNames)`). Gespeichert werden der Typ des Rückgabewertes (`objectReturn.getClass().getSimpleName()`) und der Rückgabewert, wobei beim Rückgabewert wieder auf die `toString()`-Methode zurückgegriffen wird.

```
67    after($parameterWithNames) returning(Object objectReturn): callAnyMethod
          ↪ ($parameterNames){
68        ReturnJson returnValue;
69        if (objectReturn == null) {
70            returnValue = new ReturnJson("Object", null, null, null);
71        } else {
72            returnValue = new ReturnJson(objectReturn.getClass().
                  ↪ getSimpleName(), objectReturn.toString(), null, null);
73        }
74        testGenerationJson.getSelectedMethod().setReturnValue(returnValue);
75    }
```

Listing 4.6: Dummy-`Advice` für Rückgabewerte

Speicherung der Werte

Ganz zum Schluss wird der einfache `after-Advice` eines `Pointcuts` aufgerufen. Er wird auch „finally“-`Advice` genannt, da er immer nach dem Durchlaufen einer Methode aufgerufen wird. Hier werden alle befüllten Listen an das TestGenerationJson übergeben und dieses wird schließlich in eine JSON-Datei geschrieben.
Da die Daten für die Methoden über mehrere `Advices` gesammelt werden, erfolgt die Speicherung in globalen Listen im AspectJ. Beim Speichern in das TestGenerationJson muss auf diese Besonderheit geachtet werden. Die Listen können nicht direkt ins JSON gespeichert werden, da sie global sind und alle Werte aller `Advices` enthalten. Um nur die Werte des aktuellen Aufrufs abzuspeichern, werden von den Listen Klone erstellt. Diese werden ins TestGenerationJson geschrieben und anschließend werden die globalen Listen geleert (dargestellt in Listing 4.7), sodass sie im nächsten Aufruf nur die Werte für diesen enthalten und nicht die vom vorherigen Aufruf. Diese Vorgehen wird auch auf die Liste der globalen Variablen angewendet.

```
78        List<ClassJson> copyOfClassList = ((List<ClassJson>) ((ArrayList<
              ↪ ClassJson>) classJsonList).clone());
79        testGenerationJson.setCalledMethods(copyOfClassList);
80        classJsonList.clear();
```

Listing 4.7: Speichern der aufgerufenen Methoden mittels Klonen

4.2.2. Aufgerufene Methoden

Die `Advices` für den `Pointcut` in Listing 4.2 sind dafür zuständig, dass die Methoden, welche aus der selektierten Methode heraus aufgerufen werden, in das JSON geschrieben werden.

Speicherung im ClassJson

Wie auch für den `Pointcut` in Listing 4.1 ist der `before-Advice` für den `Pointcut` in Listing 4.2 dafür zuständig, das ClassJson mit den Werten der Methode zu befüllen. Wie in Listing 4.8 dargestellt, passiert dies über den Aufruf der Methode `createClassJson()`.
Eine Ausnahme gibt es; die Informationen werden nicht hinzugefügt, wenn der Klassenname sich in der Liste der Konstruktoraufrufe befindet. Wie bereits erwähnt, können diese Aufrufe nicht gemockt werden, daher brauchen sie auch nicht im JSON zu stehen.

```
166     before(): anyMethodCalledBy() {
167         if (!constructorList.contains(thisJoinPoint.getSignature().
                ↪ getDeclaringType().getSimpleName())) {
168             classJsonList.add(createClassJson(thisJoinPoint));
169         }
170     }
```

Listing 4.8: `before-Advice` für aufgerufene Methoden

Rückgabewerte

Auch das `after returning-Advice` für den `Pointcut` in Listing 4.2 ist ähnlich zu dem für den `Pointcut` in Listing 4.1. In Listing 4.9 ist das `Advice` dargestellt. Wieder wird nichts gespeichert, wenn der Klassenname sich in der Liste der Konstruktoren befindet. Ins ReturnJson werden der Klassenname und der Wert des Rückgabeobjektes gespeichert und anschließend dem letzten Element in der classJson-Liste hinzugefügt.

```
173     after() returning(Object objectReturn): anyMethodCalledBy(){
174         if (!constructorList.contains(thisJoinPoint.getSignature().
                ↪ getDeclaringType().getSimpleName())) {
175             ReturnJson returnValue = new ReturnJson(objectReturn.getClass().
                    ↪ getSimpleName(), objectReturn.toString(),
176                     null, null);
177             ClassJson classJson = classJsonList.get(classJsonList.size() -
                    ↪ 1);
178             classJson.setReturnValue(returnValue);
179         }
180     }
```

Listing 4.9: `after returning-Advice` für Rückgabeparameter aufgerufener Methoden

JSON abspeichern

Im letzten Schritt werden die gefüllten JSON-Objekte mittels des `Jackson ObjectMappers` in einen validen JSON-String umgewandelt und in eine JSON-Datei mit dem in der Variable `FILE_NAME` angegebenen Pfad und Namen, gefolgt von einem Unterstrich und dem aktuellen Datum, gespeichert. Sollte eine Datei mit diesem Namen bereits existieren, werden die ermittelten Daten in die Datei eingefügt.
Die Umwandlung durch den `ObjectMapper` geschieht problemlos, solange die JSON-Klassen Getter und Setter für alle Variablen enthalten und ein öffentlicher Konstruktor ohne Parameter existiert.

4.2.3. Ausschluss von Konstruktoraufrufen

Zusätzlich gibt es noch einen weiteren Dummy-`Pointcut`, welcher in Listing 4.10 dargestellt ist. Er trifft auf alle Konstruktoraufrufe `call(*.new(..))` in der selektierten Methode zu. Dies ist notwendig, da die Konstruktoraufrufe nicht ins JSON mit aufgenommen werden sollen, da man diese Objekte nicht mocken kann.

```
183     pointcut constructorCalls(): call(*.new(..))
184             && !within(TestGeneratorAspect) && withincode($methodHeader);
```

Listing 4.10: Dummy-`Pointcut` für Konstruktoraufrufe

Mithilfe des in Listing 4.11 zu sehenden `Advice` wird der Objektname in eine Liste geschrieben auf die dann später zugegriffen wird, um diese Aufrufe aus dem JSON auszuschließen.

```
186     before(): constructorCalls() {
187         constructorList.add(thisJoinPoint.getSignature().getDeclaringType().
                ↪ getSimpleName());
188     }
```

Listing 4.11: `before-Advice` für Konstruktoraufrufe

4.3. Spock-Testgenerierung

Das Generieren der Testfälle aus der JSON-Datei übernimmt der TestGenerator. Hier wird das JSON, welches im `resources`-Ordner des TestGenerators liegt und den Namen hat, der im Konstruktor angegeben wird, eingelesen, einzelne Teile ausgefiltert, die Zeilenabdeckung der Objekte verglichen und anschließend die Testfälle generiert und im angegebenen Modul abgespeichert. Alle `TestGenerationJson`-Elemente in dem JSON durchlaufen einzeln die nachfolgend beschriebene Prozedur.

4.3.1. Einlesen

In der `JsonFileReader`-Datei wird das JSON mittels `ObjectMapper` von `Jackson` vom eingelesenen `String` in Objekte umgewandelt. Hierfür sind die Setter in den JSON-Klassen wichtig.
Es werden `ClassJson`-Elemente entfernt, die als „static“ gekennzeichnet sind, da diese nicht gemockt werden können. Zusätzlich werden noch Elemente entfernt, die aus dem „java.lang“- oder dem „java.util“-Paket kommen. Diese fundamentalen Methoden, wie `valueOf()` der `Float`-Klasse („java.lang“) oder `add()` für Listen („java.util“), müssen nicht gemockt werden.

4.3.2. Basis-Dateien

Um das Generieren von Tests etwas zu vereinfachen, wurden gleichbleibende oder sich wiederholende Muster in Textdateien mit Platzhaltern ausgelagert.
Die `TestClassBasis`-Datei, welche in Listing 4.12 zu sehen ist, enthält das Grundgerüst einer Testklasse. `$packagename` wird durch den Paketnamen ersetzt, in dem die Testklasse liegt. Alle Testklassen leiten von `Specification` ab, daher stehen die Ableitung und der Import dafür fest in `TestClassBasis`. Die zusätzlichen Imports, welche noch dazu kommen, werden an die Stelle des `$imports`-Platzhalters geschrieben. Der Name der Testklasse gefolgt von einem Unterstrich

und dem Methodennamen wird in `$classname` geschrieben, sodass `Spec` am Ende steht. Der Name der Testklasse folgt also folgendem Schema: `Klassenname_MethodennameSpec`. Schlussendlich ersetzen die generierten Tests den Platzhalter `$test`.

```
1 package $packagename
2
3 import spock.lang.Specification;
4 $imports
5 // $coverage
6 class $classnameSpec extends Specification {
7 $test
8 }
```

Listing 4.12: TestClassBasis.txt

Auch die Struktur der einzelnen Tests ist gleich. Sie ist in Listing 4.13 dargestellt. Es ist korrekt, dass alles vier Leerzeichen weit eingerückt ist, da die `TestBasis.txt` den `$test`-Platzhalter in der `TestClassBasis.txt` ersetzt. In der Beschreibung des Tests steht meist die Fachlichkeit, die dieser Testfall abdeckt. Bei der Generierung ist nichts über die Fachlichkeit bekannt, daher wird angegeben, dass der Testfall generiert ist, gefolgt vom Methodennamen (`$methodname`) und der Statistik (`$statistic`). Die restlichen Platzhalter stellen die von Spock vorgegebene Struktur da, welche durch die generierten Parts ersetzt werden.

```
1     def 'Generated test for method $methodname $statistic'() {
2         $givenpart
3         $whenpart
4         $thenpart
5         $wherepart
6     }
```

Listing 4.13: TestBasis.txt

4.3.3. Zeilenabdeckung

Ein Ziel der Arbeit ist es, redundante Testfälle zu vermeiden. Also nicht zwei Testfälle, die die selben Zeilen durchlaufen. Damit dies verhindert wird, gibt es zwei Möglichkeiten. Welche Methode verwendet werden soll, kann mit der Variable `USE_EVOLUTIONARY_FILER` gesteuert werden. Die einfachere Variante (`USE_EVOLUTIONARY_FILER = false`), welche nicht viel Zeit in Anspruch nimmt, verwendet die Liste mit durchlaufenen Zeilen, welche im JSON stehen, und vergleicht diese miteinander. Gibt es bereits einen Test, der genau diese Zeilen abdeckt, wird kein weiterer Testfall erstellt.
Werden aus der Methode heraus keine anderen Methoden aufgerufen, steht in der Liste nur die Zeile, aus der die selektierte Methode aufgerufen wurde. Wird die Methode von unterschiedlichen Zeilen aus aufgerufen, ist der JSON-Ansatz nutzlos. Die zweite Variante (`USE_EVOLUTIONARY_FILER = true`) behebt das Problem, indem die Tests einzeln ausgeführt werden und zu jedem einzelnen die JaCoCo-Reports generiert werden. Die Abdeckungs-Werte für die selektierte Methode werden miteinander verglichen und auf Basis dessen, werden doppelte Tests zusammengeführt.

Das Problematische hierbei ist, dass für jeden Testfall das Projekt gesäubert, gebaut und der Test ausgeführt wird. Das Löschen des `build`-Ordners ist wichtig, um zu verhindern, dass alte JaCoCo-Dateien eingelesen werden. Bei einem Projekt, welches die Größenordnung des Projektes hat, auf dem der TestGenerator ausprobiert wird, dauert das Ausführen des Befehls ungefähr vier Minuten[19]. Das Erstellen der JaCoCo-Reports braucht zusätzlich 20 Sekunden. Für zehn Testfälle werden mit diesem Verfahren mehr als 40 Minuten benötigt.
Es könnten weitere Testfälle eingespart werden, indem die durchlaufenen Zeilen auf echte Teilmengen untersucht werden. Somit würden kleinere Testfälle (solche, die eine kleine Anzahl von Zeilen abdecken) in Größere integriert werden. Dadurch ist es aber nicht möglich eine 100%-ige Zweigabdeckung zu erreichen. `if`-Abfragen werden nur einmal durch Tests durchlaufen, wenn sie keinen `else`-Block haben, und zwar mit den Werten, die die Abfrage wahr werden lassen.

4.3.4. Given

Die Klasse `GivenPartGenerator.java` übernimmt den größten Teil der Testfallgenerierung. Hier werden die Mocks aus dem JSON gelesen, in die richtige Reihenfolge gebracht und die nötigen Imports aufgelistet.

StubbedMap

Zum Sammeln der Mocks werden alle `ClassJson`s, welche im JSON unter „calledMethods“ aufgelistet sind, von unten nach oben durchlaufen. Diese Reihenfolge wurde gewählt, da davon ausgegangen wird, dass die Objekte, die zuletzt ins JSON geschrieben wurden, auf die vorherigen Objekte zugreifen.

```
1 CustomElement cElement = session.getCustomer().getElement();
```

Listing 4.14: Verkettetes Beispiel in Java

Der Codeausschnitt in Listing 4.14 sieht in der JSON-Struktur so aus, wie es in Listing 4.15 dargestellt ist. Die Methode `getCustomer()` gibt ein Objekt `Customer` zurück, auf welches wiederum die Methode `getElement()` aufgerufen wird. Es muss zunächst der Mock für `Customer` mit dem Methodenaufruf generiert worden sein, damit der Mock für die Methode `getCustomer()` das gemockte Element zurückgeben kann.

```
1  "calledMethods": [
2        {
3          "isStatic": false,
4          "packageName": "com.master",
5          "className": "Session",
6          "calledMethod": {
7            "methodName": "getCustomer",
8            "parameters": []
9          },
10         "returnValue": {
11           "type": "Cusomer",
12           "value": "com.master.Customer@622fa54e",
13           "errorCode": null,
```

[19] 64-Bit Windows 7 Professional Betriebssystem, 16 GB RAM, 2,5 GHz Prozessor mit zwei Kernen

```
14            "errorMessage": null
15          }
16        },
17        {
18          "isStatic": false,
19          "packageName": "com.master",
20          "className": "Customer",
21          "calledMethod": {
22            "methodName": "getElement",
23            "parameters": []
24          },
25          "returnValue": {
26            "type": "CustomElement",
27            "value": "com.master.CustomElement@642fa74e",
28            "errorCode": null,
29            "errorMessage": null
30          }
31        }
32      ]
```

Listing 4.15: Verkettetes Beispiel in JSON

Daher muss das JSON von unten nach oben durchlaufen werden, damit am Ende ein Mock generiert wird, der so aussieht, wie es in Listing 4.16 gezeigt wird. Das Element, welches schlussendlich durch die Abfrage in Listing 4.14 erzeugt wird, wird als erstes gemockt, sodass es als Rückgabewert für die Methode `getElement()` angegeben werden kann.

```
1  Given: 'A CustomElement'
2  CustomElement customElement = Stub(CustomElement) {}
3
4  and: 'a Customer'
5  Customer customer = Stub(Customer) {
6    getElement() >> customElement
7  }
8
9  and 'a Session'
10 Session session = Stub(Session) {
11   getCustomer() >> customer
12 }
```

Listing 4.16: Verkettetes Beispiel als Mocks in Groovy

In eine `Map` werden die Methoden, mit ihrem Klassennamen als Schlüssel, eingetragen. Zu jedem Schlüssel gibt es eine Liste mit den Methodenaufrufen, die gemockt werden sollen, als `String` im „groovy“-Format (wie `getCustomer() >> customer`).
Ist der Rückgabewert eine Zahl, wird der Typ als Cast angegeben. Groovy arbeitet standardmäßig mit `BigDecimal`, was dazu führt, dass der Compiler eine Codezeile wie `getFirstInt() >> 5` markiert, da `getFirstInt()` eine `Integer`- und keine `BigDecimal`-Zahl zurückgibt. Korrekt müsste die Zeile also `getFirstInt() >> (Integer) 5` lauten.
Methoden, die aus der ausgewählten Klasse aufgerufen wurden und sich in ihr befinden, werden beim Generieren übersprungen. Theoretisch müssen die Aktionen und Manipulationen, die in

den Methoden stattfinden, auch gemockt werden. Doch das Aspekt liefert dafür keine Daten und daher können sie nur übersprungen werden.

Reihenfolge

Nachdem alle `ClassJson`-Objekte durchlaufen wurden, muss die Reihenfolge gegebenenfalls angepasst werden. Dadurch, dass die Objekte von unten nach oben durchlaufen werden, werden die meisten Abhängigkeiten berücksichtigt. Aber für die übergebenen Parameter der selektierten Methode, muss die Mock-Reihenfolge angepasst werden. Sie müssen nach oben rutschen, da andere Methoden direkt darauf zugreifen könnten. Dabei dürfen die anderen Reihenfolgen, die die Abhängigkeiten der anderen Methoden darstellen, nicht verändert werden.
Um das zu erreichen, wird bei den Mocks überprüft, ob sich unter deren Parametern für ihre Methoden, Parameter befinden, die der selektierten Methode übergeben werden. Ist dies der Fall, wird der Mock an die letzte Stelle der `Map` geschoben. Dort ist sichergestellt, dass der Parameter-Mock existiert und die anderen Reihenfolgen werden nicht verändert.

Globale Variablen

Wurde für eine Methode ein Eintrag in der `Map` hinzugefügt, wird überprüft, ob die übergebenen Parameter globale Parameter der Klasse sind. Um zu ermitteln ob dem so ist, werden die Klassennamen der Parameter und der globalen Variablen miteinander verglichen. Sind die Parameter der Klasse global, müssen sie beim Konstruktoraufruf der Klasse als Mocks übergeben werden, damit der Parameter einen gültigen Wert und nicht den Wert `null` hat. Für diesen Zweck werden sie in einer Liste gespeichert und bei der Konstruktoraufruf-Generierung als Parameter-Mocks übergeben.
Zusätzlich werden am Ende der `Map`-Generierung alle Parameter der aufgerufenen Methoden überprüft, ob sie eine globale Variable sind. Existiert für sie noch kein Mock, wird ein leerer erstellt, der `Map` hinzugefügt und die Liste der globalen Variablen um diese Variable erweitert.

Variablennamen

Es ist nicht möglich den Namen einer lokalen Variable über AspectJ zu ermitteln. Daher müssen für die Mocks Variablennamen generiert werden. Für primitive Datentypen werden keine Mocks benötigt, da von ihnen keine Methoden aufgerufen werden, deren Rückgabewert beeinflusst werden muss (zum Beispiel die Methode `toString()`). Daher betrifft dieses Problem nur die komplexen Klassen. Häufig bekommen Variablen ihren Typnamen in CamelCase-Schreibweise als Namen. Dies findet auch beim TestGenerator Anwendung. Damit das funktioniert, wird angenommen, dass jeder komplexe Datentype nur einmal in einer Methode vorkommt.
Sollte es zwei Variablen vom selben Datentypen in einer Methode geben, könnte man hinter den zweiten Namen ein „a“ schreiben und bei der dritten Variable ein „aa“. Dafür müsste zunächst die bis zu diesem Zeitpunkt generierte Zeichenkette nach den bereits vorhandenen Variablennamen durchsucht werden, um zu erkennen, welche Endung die nächste Variable bekommt. Ein größeres Problem ist aber, dass es nicht möglich ist zu unterscheiden, ob eine Methode auf der ersten oder zweiten Variable aufgerufen wurde. Da dieses Problem mittels aspektorientierter

Programmierung nicht lösbar ist, bekommen alle Variablen vom selben Datentypen den gleichen Namen und müssen im Nachhinein vom Entwickler angepasst werden.

Finale Formatierung

Zum Schluss wird die generierte `Map` in einen `String` ungewandelt, sodass sie in der `TestBasis`-Datei den Platzhalter `$givenpart` ersetzen kann. Der String schließt mit dem Konstrukteraufruf der ausgewählten Klasse ab, wobei die zuvor gesammelten globalen Variablen als Mocks übergeben werden.

Importlist

Jedes Mal wenn ein Mock generiert wird, wird vom dazugehörigen `ClassJson` der Paketname mit dem Klassennamen in eine Liste geschrieben. Da die Liste vom Datentyp `HashSet` ist, gibt es keine Duplikate. Anschließend wird über die Liste iteriert und vor jedes Element „import “ gesetzt und jedes Element mit einem Semikolon und einem Zeilenumbruch abgeschlossen.

4.3.5. Kompletter Testfall

Wie der `$givenpart`-Platzhalter befüllt wird, wurde eben beschrieben. Der Platzhalter für den `when`-Part wird durch den Aufruf der gewählten Methode ersetzt. Bei `then` wird bei primitiven Datentypen überprüft, ob das zurückgegebene Objekt dem aufgezeichnetem Rückgabewert entspricht. Handelt es sich nicht um einen primitiven Datentypen, wird das Objekt auf `!= null` geprüft. Der Platzhalter für `where` ist leer.
Wie ein generierter Testfall aussehen kann, ist in Listing 4.17 zu sehen. Der Testfall ist sehr kurz und daher übersichtlich. Es gibt jeweils nur ein Element in den einzelnen Testschritten.

```
1  def 'Generated test for method isInteger_1 (2/6)'() {
2      given: 'a StringCheck'
3      StringCheck stringCheck = new StringCheck()
4
5      when: 'call method isInteger'
6      def result = stringCheck.isInteger('Test')
7
8      then: 'the result is as expected'
9      result == false
10
11
12 }
```

Listing 4.17: Generierter Testfall für die Methode `isInteger()`

4.3.6. Statistik

Für die Variante, welche nicht für jeden Test die JaCoCo-Reports erstellt, wird die Statistik vor dem Abspeichern in eine Datei ermittelt. Hierbei wird die Anzahl der `TestGenerationJsons` in der JSON-Datei den tatsächlich generierten Testfällen gegenübergestellt und an der Beschreibung der Tests ergänzt (zum Beispiel `def 'Generated test for method add (4/6)'()`).

4.4. Ausführung der Tests

Nachdem die Tests generiert wurden, führt der TestGenerator sie aus, um zu prüfen, ob die Generierung erfolgreich war und die Tests ohne Probleme durchlaufen.
Zum Ausführen der Tests wird auf den Gradle-Befehl zurückgegriffen. Dafür wird aus dem Java-Programm heraus ein Terminal-Prozess gestartet, in dem der Befehl, welcher in Listing 4.18 dargestellt ist, aufgerufen wird. Der Befehl, der der Methode `runCommand()` übergeben wird, löscht den `build`-Ordner (`clean`), baut das Projekt neu (`build`) und ruft den Befehl zur Ausführung der Spock-Tests auf (`TESTING_COMMAND`). Um alle Tests auszuführen, muss die Variable `TESTING_COMMAND` mit „`tests`“ belegt sein. Sollte es im Modul noch andere Tests außer der Spock-Tests geben, wird es vermutlich einen Gradle-Befehl geben, der nur die Tests ausführt. Dieser muss dann als `TESTING_COMMAND` eingetragen werden. Mittels `--tests` wird definiert, welche Tests ausgeführt werden sollen. Im TestGenerator wird explizit auf die generierte Testklasse verwiesen, die den Klassennamen, gefolgt vom Schlüsselwort `Spec`, als Namen bekommen hat.

```
36          int returnValue = runCommand(
37                  "gradle clean build " + TESTING_COMMAND + " --tests *." +
                        ↪ TEST_CLASS_NAME + "Spec");
```

Listing 4.18: Befehl zur Ausführung der Tests

In der Methode `runCommand()` wird zunächst überprüft, ob das Betriebssystem ein Windows-System ist. Dafür wird vom System die Eigenschaft `os.name` auf das Enthalten des Wortes „`windows`“ untersucht. Ist das der Fall, wird vor dem auszuführenden Befehl „`cmd /c`“ gesetzt. Dies ist notwendig, damit der Befehl unter Windows ausgeführt werden kann.
Anschließend wird der Prozess gestartet, wobei der Befehl und das Verzeichnis, in dem der Befehl ausgeführt werden soll, übergeben werden. Das Verzeichnis ist das Modul in dem sich der generierte Test befindet.
Mittels einer Variable (`PRINT_STREAM`) ist es dem Entwickler möglich zu entscheiden, ob ihm die Ausgaben des Befehls ausgegeben werden sollen oder der Prozess nur mittels Punkten dargestellt werden soll.
Liefen die Tests erfolgreich durch, wird der gestartete Prozess mit 0 beendet und dem Entwickler wird ein `BUILD SUCCESSFUL` ausgegeben. Sollte ein Compilerfehler auftreten oder die Tests nicht „grün“ sein, wird ein `BUILD FAILED` ausgegeben, mit dem Hinweis `Test failed`.

4.5. Ermittlung der Testabdeckung

Das Ausführen der Tests ist, neben dem Ermitteln, ob die Tests durchlaufen und keine Fehler aufweisen, zusätzlich wichtig für die Ermittlung der Testabdeckung. Denn bevor mittels JaCoCo der Testreport erstellt werden kann, muss der Test ausgeführt worden sein.
Der Befehl für das Generieren des Reports (`gradle jacocoTestReport`) wird, wie der Befehl zum Ausführen der Tests, über die Methode `runCommand()` ausgeführt. Wichtig ist, dass im `gradle`-Plugin angegeben ist, dass eine XML-Datei erzeugt werden soll, damit die nachfolgenden Schritte ausgeführt werden können.
Nachdem der Prozess beendet ist, befindet sich im `build`-Ordner des Moduls, unter dem Pfad `reports/jacoco/test`, eine `jacocoTestReport.xml`-Datei. Auch wenn nur eine Testklasse aus-

geführt wurde, erzeugt JaCoCo einen Report für das gesamte Modul. Daher wird die Datei umfangreicher, je größer das Modul ist.
Diese Datei wird im TestGenerator mittels `DocumentBuilderFactory` und `DocumentBuilder` eingelesen und als XML-Datei erkannt. Erkennen bedeutet in diesem Fall, dass es möglich ist die einzelnen Knoten zu traversieren und deren Eigenschaften abzufragen.
In dem erzeugten XML von JaCoCo wird am Anfang auf eine `report.dtd` referenziert, welche Style-Informationen enthält. Dies ist für das Einlesen mittels Java nicht notwendig. Damit `DocumentBuilderFacotry` keinen Fehler wirft, müssen dafür zwei Features deaktiviert werden. Der Befehl und die Namen der Features sind in Listing 4.19 zu sehen.

```
69          dbFactory.setFeature("http://apache.org/xml/features/
                ↪ nonvalidating/load-dtd-grammar", false);
70          dbFactory.setFeature("http://apache.org/xml/features/
                ↪ nonvalidating/load-external-dtd", false);
```

Listing 4.19: Feature-Einstellung für DocumentBuilderFactory

Wurde die Datei erfolgreich eingelesen, kann nach dem Knoten für die Informationen der zu testenden Methode gesucht werden.

4.5.1. Zweig- und Zeilenabdeckung

Für das Auslesen der Abzweigungs- und Zeilenabdeckung muss zunächst der Knoten mit der korrekten Klassenbezeichnung gefunden werden. Dabei ist auch die Ordnerstruktur des Paketes wichtig. Dafür werden alle Knoten mit dem `Tag Name` „class“ genommen und nach dem passenden Knoten gesucht. Für das XML, welches in Abbildung 4.1 dargestellt ist, würde nach „testing/SimpleTestClass“ gesucht werden. Anschließend wird in den Kinderknoten des Klassenknotens nach der Methode gesucht, die zu Anfang mittels des Aspects ausgewählt wurde. In Abbildung 4.1 wäre es „isInteger“. Damit wurde der Knoten gefunden, welcher die Informationen für die Testabdeckung bereithält. Von den fünf Werten, die JaCoCo generiert, werden die `BRANCH`- und die `LINE`-Abdeckung verwendet.[20]

```
- <report name="TestingProject">
    <sessioninfo id="DESKTOP-2RNS224-eb03e92d" start="1519920998263"
    dump="1519921000824"/>
  - <package name="testing">
    - <class name="testing/SimpleTestClass">
      - <method name="<init>" desc="()V" line="3">
          <counter type="INSTRUCTION" missed="0" covered="3"/>
          <counter type="LINE" missed="0" covered="1"/>
          <counter type="COMPLEXITY" missed="0" covered="1"/>
          <counter type="METHOD" missed="0" covered="1"/>
        </method>
      - <method name="isInteger" desc="(Ljava/lang/String;)Z" line="6">
          <counter type="INSTRUCTION" missed="2" covered="18"/>
          <counter type="BRANCH" missed="3" covered="3"/>
          <counter type="LINE" missed="1" covered="7"/>
          <counter type="COMPLEXITY" missed="3" covered="1"/>
          <counter type="METHOD" missed="0" covered="1"/>
        </method>
        <counter type="INSTRUCTION" missed="2" covered="21"/>
```

Abbildung 4.1.: Anfang einer von JaCoCo generierten XML-Datei

[20] Wofür die Abdeckungen stehen, kann in Unterabschnitt 2.1.4 nachgelesen werden.

Aus den `missed`- und `covered`-Werten wird für beide Abdeckungsarten ein Prozentsatz ermittelt. Dafür dient die Summe der beiden Werte als Basis und das was der `covered`-Wert einnimmt, wird als Prozentwert für die Abdeckungsart niedergeschrieben.
Es wird ein Prozentwert dargestellt, da damit auf einen Blick klar wird, ob die Methode durch die Tests abgedeckt wird oder nicht. Würden alle Zeilen und deren Abdeckungswerte als Kommentar über dem ersten Test stehen, wäre die Informationsmenge zu groß und würde dadurch unübersichtlich.
Zur Ermittlung der Testabdeckung hätten auch die aufgerufenen Zeilen im JSON zu Rate gezogen werden können. Die Problematik hierbei ist, dass nicht sichergestellt ist, dass das alle Zeilen der Methode sind, da sich dort nur Zeilen wiederfinden, die ausgeführt wurde.

4.5.2. Detaillierte Zeilenabdeckung

Wie bereits beschrieben, werden in der zweiten Variante der Testfilterung zu jedem Testfall die JaCoCo-Reports ermittelt. Um Tests einzeln ausführen zu können, muss der Befehl so angepasst werden, dass nach dem Testklassennamen der Methodenname steht (zu sehen in Listing 4.20).

```
46          int returnValue = runCommand(
47                  "gradle clean build " + TESTING_COMMAND + " --tests \"*" +
                        ↪ TEST_CLASS_NAME + "Spec." + methodName + "\"");
```

Listing 4.20: Befehl zur Ausführung eines Tests mit der Beschreibung `methodName`

Anschließend werden aus der JaCoCo-XML die Daten unter dem Knoten „sourcefile“ mit dem Namen der Klasse inklusive „.java“ ermittelt. Ein Knoten ist exemplarisch in Abbildung 4.2 dargestellt. Neben den Abdeckungszählern (`countern`) ist hier für jede Zeile mit Inhalt angegeben wie viele Anweisungen aufgerufen wurden (`ci` steht für „Covered Instruction“) und welche verfehlt wurden (`mi` bedeutet „Missed Instruction“) sowie die gleichen Angaben für die Zweigabdeckung („Missed Branch“ (`mb`) und „Covered Branch“ (`cb`)).

```
−<sourcefile name="SimpleTestClass.java">
    <line nr="1" mi="0" ci="2" mb="0" cb="0"/>
    <line nr="6" mi="0" ci="3" mb="0" cb="0"/>
    <line nr="9" mi="4" ci="0" mb="0" cb="0"/>
    <line nr="17" mi="4" ci="0" mb="0" cb="0"/>
    <line nr="18" mi="45" ci="0" mb="0" cb="0"/>
    <line nr="19" mi="4" ci="0" mb="0" cb="0"/>
    <line nr="20" mi="4" ci="0" mb="0" cb="0"/>
    <line nr="21" mi="45" ci="0" mb="0" cb="0"/>
    <line nr="22" mi="1" ci="0" mb="0" cb="0"/>
    <line nr="25" mi="0" ci="9" mb="0" cb="6"/>
    <line nr="26" mi="0" ci="2" mb="0" cb="0"/>
    <line nr="29" mi="0" ci="3" mb="0" cb="0"/>
    <line nr="30" mi="0" ci="2" mb="0" cb="0"/>
    <line nr="31" mi="0" ci="2" mb="0" cb="0"/>
    <line nr="33" mi="0" ci="2" mb="0" cb="0"/>
    <line nr="37" mi="23" ci="0" mb="4" cb="0"/>
    <line nr="38" mi="2" ci="0" mb="0" cb="0"/>
    <line nr="40" mi="2" ci="0" mb="0" cb="0"/>
    <counter type="INSTRUCTION" missed="134" covered="25"/>
    <counter type="BRANCH" missed="4" covered="6"/>
    <counter type="LINE" missed="10" covered="8"/>
    <counter type="COMPLEXITY" missed="4" covered="6"/>
    <counter type="METHOD" missed="2" covered="3"/>
    <counter type="CLASS" missed="0" covered="1"/>
```

Abbildung 4.2.: `sourcefile`-Knoten einer von JaCoCo generierten XML-Datei

Alle Daten werden, zuordenbar zum Testfall, in einem Objekt gespeichert. Nachdem alle Testfälle durchlaufen wurden, werden die Objekte miteinander vergleichen. Nur wenn alle Werte gleich sind wird vermerkt, dass die Testfälle doppelt sind. Der zweite Testfall wird dann entfernt und beim ersten wird die Statistik um eins inkrementiert.
Ganz am Ende wird die gesamte Testklasse ausgeführt und die JaCoCo-Reports so eingelesen, wie es im vorherigen Kapitel beschrieben wurde, um die Zweig- und Zeilenabdeckung zu ermitteln.

4.6. Testverbesserung

Auch wenn es für den Entwickler eine Arbeitserleichterung ist, dass Tests generiert werden, sind sie dennoch nicht perfekt. Daher sollte nach der Generierung definitiv noch ein prüfender Blick auf die Tests geworfen werden. Zu beachten ist, dass der TestGenerator die Tests immer überschreibt, die er generiert. Es wird nicht überprüft, ob eine Datei mit diesem Namen bereits existiert. Falls Änderungen an einem generierten Test vorgenommen wurden, sollte man ihn umbenennen, sodass er beim nächsten Start des TestGenerators nicht überschrieben wird.

4.6.1. Testbeschreibungen

Die Beschreibungen der generierten Tests sind allgemein gehalten, da sie für alle Tests gelten müssen. Ist der Entwickler mit den erstellten Tests zufrieden, sollte er die Beschreibungen anpassen, damit später einfacher nachvollzogen werden kann, was der Test im Code prüft.

4.6.2. Konstanten

Durch den `Aspect` ist es nicht möglich den Namen einer lokalen Variable zu ermitteln. Dafür können der Typ und der Wert der Variable identifiziert werden. So ist es auch bei Konstanten. Es ist möglich, dass im Code für einen Text oder für eine Zahl eine Konstante angelegt wurde, doch durch den `Aspect` steht im Testfall nur der Wert. Darauf sollte ein Entwickler achten und den Konstantennamen eintragen, da der Wert geändert werden kann und damit der Test mit dem festen Wert fehl schlagen würde.

4.6.3. Rückgabewert

Die Prüfung des Rückgabewertes kann spezifiziert werden. Für primitive Datentypen muss nichts verändert werden, da auf den Wert abgefragt wird. Bei komplexen Datentypen wird auf `!= null` abgefragt, wenn sie nicht `null` sind. Dies kann dahingehend angepasst werden, dass die Anzahl der Elemente in einer Liste abgefragt werden oder der Inhalt der Elemente.

4.6.4. Nicht ausführbarer Test

Wenn bei der Ausführung der Tests ein Fehler ausgegeben wurde, muss der Entwickler einen Blick auf die Tests werfen. Dies kann vorkommen wenn die Methode nicht testbar ist und somit keine korrekten Tests generiert oder etwas fehlt und Variablen nicht korrekt gemockt werden konnten. So wie die Beschreibung der Testfälle ist auch der TestGenerator allgemein gehalten, sodass er für möglichst viele Projekte funktioniert. Spezielle Inferfaces oder Ableitungen für

Testfälle werden nicht beachtet, was zu Fehlern führen kann, die im besten Fall mit wenig Handgriffen eines Entwicklers behoben werden können.

4.7. MiniProjekt

Während der Entwicklung wurde primär darauf geachtet, dass der TestGenerator mit dem Webshop-Projekt, welches anfangs beschrieben wurde und einige Rahmenbedingungen vorgibt, funktioniert. Nichtsdestotrotz soll der TestGenerator auch Anwendung in anderen Projekten finden, die die Rahmenbedingungen erfüllen. Aus diesem Grund wurde er zum einen am Webshop-Projekt ausprobiert und zum anderen an einem MiniProjekt, welches, im Gegensatz zum umfangreichen Code des Webshops, simple Methoden enthält.

4.7.1. Aufbau

Das Gradle-Projekt besteht aus einer ausführbaren Klasse mit dem Namen `SimpleTestClass`. In der `main`-Methode werden die Methoden, für die Tests generiert werden sollen, mit verschiedenen Werten aufgerufen.
Als Abhängigkeit ist in der `build.gradle` die JSON-Struktur des TestGenerators angegeben, da für die Erstellung des JSONs darauf zugegriffen werden muss.
Für jede Methode existiert eine eigene AspectJ-Klasse, sodass es nicht notwendig ist die Methoden mittels angepasster AspectJ-Datei nacheinander auszuführen. Wird eine Methode nicht aufgerufen, werden die `Pointcuts` im AspectJ nicht angesprochen und es wird kein JSON für die Methode erstellt.
Der Fokus der Methoden lag darauf, möglichst verschiedene Anwendungsfälle darzustellen. Die Lösungsansätze können sicherlich vereinfacht werden, doch für den TestGenerator sind Methoden mit mehreren Zeilen interessanter, um die Testabdeckung zu untersuchen.

StringCheck

Die Klasse `StringCheck` enthält Methoden, die einen `String` als Parameter übergeben bekommen.
Die Methode `isInteger()` gibt `true` zurück, wenn der übergebene String einen ganzzahligen Wert im `Integer`-Bereich darstellt.
Das Besondere an der Methode `removeFirstLetter()` ist, dass es sich hierbei um eine statische Methode handelt. Die Methode entfernt den ersten Buchstaben des übergebenen Parameters, sofern er nicht leer oder `null` ist.

ListCheck

In der Klasse `ListCheck` befinden sich Methoden, die mit Listen arbeiten.
Die Methode `hasElements()` überprüft, ob eine übergebene Liste Elemente enthält und gibt `true` oder `false` zurück.
Eine Liste der Objekte `Element` wird in der Methode `createListWithElement()` erzeugt. Dafür wird eine weitere Methode `createElement()` aufgerufen, die ein `Element` erstellt.

Die Klasse `Element` an sich enthält nur einen Standard-Konstruktor, einen `Integer` als Index, einen `String` als Inhalt (value) und Getter und Setter dafür.

4.7.2. Ausführung

Zur Ausführung der `main`-Methode muss das Terminal verwendet werden, damit die AspectJ-Klassen integriert und angewendet werden. Im Verzeichnis des Mini-Projektes `gradle clean build run -x test` ausführen. Mit `-x test` werden bereits vorhandene Tests nicht ausgeführt. Anschließend die generierte JSON-Datei aus dem Verzeichnis des Mini-Projektes in den `resources`-Ordner der TestGenerators kopieren und im TestGenerator-Projekt die `main`-Methode der Klasse `Main` ausführen. Das funktioniert über die IDE oder das Terminal (`gradle clean build run`).

4.7.3. Erkenntnisse

Dadurch, dass aus den Methoden heraus keine anderen Methoden aufgerufen wurden, ist aufgefallen, dass die Reduzierung der Testfälle mithilfe der im JSON gespeicherten Zeilen nicht ganz durchdacht war. Hierbei werden Testfälle gelöscht, deren Zeilen im JSON identisch sind. Es wurden aber nur die Zeilen gespeichert, in der die Methode aufgerufen wurde und da die Aufrufe untereinander in der `main`-Methode stehen, ist jede Zeile unterschiedlich.
Dieses Problem wurde dadurch behoben, dass die Tests einzeln ausgeführt und anschließend die Zeilenangeben in der jacocoTestReport.xml miteinander verglichen werden. Dies ähnelt eher einem genetischen Algorithmus, der Testfälle aufgrund von Zweig- und Zeilenabdeckungen aussortiert. Die Chromosomen sind hierbei die Daten von JaCoCo.

4.8. Zusammenfassung

Vom Code bis zum generierten Test müssen einige Arbeitsschritte durchlaufen werden. Mittels `Pointcuts` und `Advices` in AspectJ werden Methoden selektiert und die Werte in einer JSON-Struktur abgespeichert. Die JSON-Struktur wird durch Java-Objekte vorgegeben, wodurch es durch den `ObjectMapper` einfach ist das JSON in Objekte zu wandeln und umgekehrt.
Mithilfe der gespeicherten Werte im JSON-Format und Basis-Dateien, die die Struktur einer Spock-Klasse und eines Testfalls beinhalten, werden Testfälle generiert. Für die Ausfilterung von doppelten Testfällen wurden zwei Varianten entwickelt. Die erste und schnellere Variante sortiert Testfälle aufgrund der gespeicherten durchlaufenen Zeilen im JSON aus. Die zeitintensive Variante führt jeden Tests einzeln aus und ermittelt doppelte Tests auf Basis der Zeileninformationen des JaCoCo-Reports. Anschließend wird die Test-Klasse mit dem Gradle-Befehl über ein Terminal ausgeführt. Mittels JaCoCo wird daraufhin eine XML-Datei erstellt, aus der die Testabdeckung für die selektierte Methode ausgelesen und in die Testklasse geschrieben wird.
Da es sich um einen Generator handelt, der für möglichst viele Projekte funktionieren soll, gibt es viele Texte, die allgemeingültig sind und einige Sonderfälle, die nicht berücksichtigt werden. Aus diesem Grund wird es unerlässlich sein, dass ein Entwickler auf die generierten Tests schaut und sie anpasst.

5. Evaluation

Nachdem erläutert wurde, wie der TestGenerator aufgebaut ist und funktioniert, wird in diesem Kapitel darauf eingegangen welche Erkenntnisse während der Arbeit gewonnen wurden. Hierfür wurden stichprobenartig mit dem TestGenerator Tests für das Webshop-Projekt und für das MiniProjekt erstellt.

5.1. Methodenauswahl

Für die Auswahl der Methode muss die AspectJ-Datei korrekt abgelegt und angepasst werden. Um dem Entwickler entgegenzukommen, gibt es eine Template-Datei in der lediglich die Parameter mit voranstehendem „$“ angepasst werden müssen.

Das Ziel, dass der zu testende Code nicht angepasst werden muss, ist durch die AspectJ-Datei erfüllt. Es können von privaten bis zu statischen Methoden alle Arten von Methoden angesprochen werden.

Werden statische Methoden getestet, werden für den Testfall Objekte der Klasse erstellt und darauf die statische Methode aufgerufen. Die Erstellung eines Objektes ist nicht notwendig, behindert den Test aber auch nicht.

Eine Ausnahme zur Unveränderlichkeit des Codes kann durch die Testbarkeit des Codes erforderlich sein. Diese Änderungen würden aber auch am Code vorgenommen werden, wenn ein Entwickler Tests schreibt. Hierbei handelt es sich zum Beispiel um Variablen, die im Konstruktor initiiert werden, somit aber nicht im Test gemockt werden können, da ein Zugriff auf diese Variable nicht möglich ist. Dies ist in Listing 5.1 dargestellt. Eine Möglichkeit das Problem zu lösen, ist die Übergabe eines Parameters im Konstruktor. Dies ist in Listing 5.2 zu sehen.

```
1  public class Example() {
2    private Manager manager;
3
4    public Example() {
5      manager = Manager.getInstance();
6    }
7
8    public void methodToTest(int cId) {
9      Customer customer = manager.getCustomer(
           ↪ cId);
10     customer.isLoggedIn(true);
11   }
12 }
```

Listing 5.1: Beispiel einer nicht testbaren Methode

```
1  public class Example() {
2    private Manager manager;
3
4    public Example(final Manager manager) {
5      if(manager != null) {
6        this.manager = manager;
7      } else {
8        manager = Manager.getInstance();
9      }
10   }
11
12   public void methodToTest(int cId) {
13     Customer customer = manager.getCustomer(
           ↪ cId);
14     customer.isLoggedIn(true);
15   }
16 }
```

Listing 5.2: Angepasste Methode, sodass sie testbar ist

5.2. Lesbarkeit der Tests

Ein Testgenerierungs-Tool kann noch so gut sein, wenn die generierten Tests nicht verständlich sind, wird es sich nicht durchsetzen oder verwendet werden. Daher sollte bei diesen Tools ein prüfender Blick auf die entstandenen Tests geworfen werden.

Auf den ersten Blick wirken die Tests, die vom TestGenerator erstellt wurden, recht übersichtlich. Die einzelnen Testfälle sind gut voneinander zu unterscheiden und die `arrange-act-assert`-Struktur ist erkennbar.

Bei genauer Betrachtung fällt auf, dass für jedes Objekt ein Mock erstellt wird, sei es auch nur ein Übergangs-Objekt. Dies wirkt sich nachteilig auf die Länge der Testfälle aus. Mittels Verkettungen in den Testfällen können die Übergangs-Objekte weggelassen werden. Dabei liegt es im Auge des Betrachters welche Variante übersichtlicher und lesbarer ist. Für den Beispielcode `CustomElement cElement = session.getCustomer().getElement();` sind in Listing 5.3 die generierten Mocks dargestellt und in Listing 5.4 ein verketteter Mock.

```
1  Given: 'A CustomElement'
2  CustomElement customElement = Stub(
       ↪ CustomElement) {}
3
4  and: 'a Customer'
5  Customer customer = Stub(Customer) {
6    getElement() >> customElement
7  }
8
9  and 'a Session'
10 Session session = Stub(Session) {
11   getCustomer() >> customer
12 }
```

Listing 5.3: Generierte Mocks in Groovy

```
1 Given 'a Session with a Customer and a
      ↪ CustomElement'
2 Session session = Stub(Session) {
3   getCustomer() >> Stub(Customer) {
4     getElement() >> Stub(CustomElement) {}
5   }
6 }
```

Listing 5.4: Verketteter Mock in Groovy

Zusätzlich sind die Abfragen am Ende des Testfalls sehr einfach gehalten. Für Listen können zusätzlich die Länge sowie bei primitiven Datentypen die einzelnen Listen-Elemente ermittelt und überprüft werden. Die `Strings` der Klasse können ausgelesen und interpretiert werden, um damit die Abfrage der Tests zu spezifizieren.

Für komplexere Objekte kann auf andere wesentliche Eigenschaften eingegangen werden. Hierbei kann nicht auf die `toString()`-Methode zurückgegriffen und interpretiert werden, da nicht sichergestellt ist, dass die Methode überschrieben wurde und einen aussagekräftigen Wert zurück gibt. Außerdem müsste der `String` für jedes Objekt anders interpretiert werden. AspectJ kann auf die Methoden der Klasse des Return-Elements zugreifen und darüber die Getter ermitteln. Durch `Reflection` kann die Methode ausgeführt und der Rückgabewert ermittelt werden. Anschließend muss noch ein passendes Format zur Speicherung im JSON entwickelt werden. Aufgrund der Anpassungen im JSON, der Interpretation, dem damit Verbunden Aufwand an Anpassungen und dem Mangel an Zeit, wurde der Ansatz so detailliert beschrieben, aber nicht umgesetzt.

Weitere Übersichtlichkeit kann dadurch gewonnen werden, dass der `where`-Block verwendet wird. Damit können mehrere Testfälle in einem Testfall dargestellt werden, wobei die unterschiedlichen Eingaben im `where` stehen. Technisch gesehen reduziert es die Testfälle nicht, aber ein Test ist übersichtlicher als drei. Vor allem wenn ohne scrollen direkt erkannt wird, worin sich die Testfälle unterscheiden. Beim `where` stehen die Werte direkt untereinander oder nebeneinander.

5.3. Dauer der Testgenerierung

Der TestGenerator wurde auf verschiedenen Methoden aus dem Webshop-Projekt ausprobiert. Die Durchführung fand auf einem 64-Bit Windows 7 Professional Betriebssystem mit 16 GB Arbeitsspeicher und einem 2.5 GHz Prozessor mit zwei Kernen bei normaler Auslastung statt.
Im Durchschnitt werden für die Testausführung des Hauptmoduls des Tomcats vier Minuten benötigt. Dies liegt vor allem an der Größe und den Abhängigkeiten des Moduls. Anschließend folgen 20 Sekunden für die Erstellung des Test-Reports durch JaCoCo.
Für den Ansatz, der jeden Test einzeln durchläuft multipliziert sich die Zeit mit der Anzahl der Tests. Für zehn Tests benötigt der TestGenerator damit über 40 Minuten.
Je kürzer die Generierung dauert, desto besser. Es ist zu beachten, dass wenn ein Entwickler Tests schreibt, er die Tests auch ausführt. Um Zeit zu sparen wird das Aufräumen und Bauen des Projektes nicht durchgeführt. Im Webshop-Projekt werden für das initiale Ausführen einer Testklasse 25 Sekunden benötigt. Jede weitere Ausführung dauert 6 Sekunden. Die Generierung eines Tests sollte nicht länger dauern als ein Entwickler zum Schreiben der Tests benötigt.
`EvoSuite` wurde beispielsweise für die selbe Klasse nach zwei Minuten beendet, konnte aber keine Testfälle erzeugen.
Für eine einfache Klasse mit Gettern und Settern hat `EvoSuite` nach dem definierten Maximum von drei Minuten Tests generiert. Für die Liste mit komplexen Objekten wurde in allen Testfällen `null` übergeben. Für den `String`-Parameter und den `ENUM`-Parameter wurden verschiedene Werte übergeben. Zusätzlich wurde ein Testfall generiert, der eine `NullPointerException` testet.
Die Namen der Testmethoden folgen dem Schema `textX()`, wobei `X` eine fortlaufende Nummer ist und die Werte für den `String` wirken kryptisch (z.B. „8M /]3>p1T%ˆmz=e)[q“).

5.4. Testabdeckung

Ein Ziel der Arbeit war es, eine 100%-ige Zweig- und Zeilenabdeckung mit möglichst wenig Testfällen zu erreichen.
Es ist möglich mit dem TestGenerator eine Zweig- und Zeilenabdeckung von 100% zu erreichen. Die einzige Bedingung dafür ist, dass die Methode mit den Parameterkombinationen aufgerufen werden, die dafür benötigt werden. Und dies ist nicht immer gegeben. Damit können Codestellen, die nicht aufgerufen werden, nicht als ungenutzter Code identifiziert werden.
Tritt eine Codestelle auf, die nicht durch die generierten Tests abgedeckt werden, kann der Entwickler die Stelle prüfen. Es kann sein, dass der Code durch Anpassungen (Feature oder Refactoring) nicht mehr erreicht wird und daher entfernt werden kann. Ist das nicht der Fall, können die Inhalte des Webshops so angepasst werden, dass die Codestelle aufgerufen werden und neue Tests generiert werden. Oder der Entwickler passt die bereits generierten Tests an oder fügt auf Basis dessen einen neuen Testfall hinzu, sodass der Code durchlaufen wird. Dies ist insgesamt weniger Aufwand als alle Tests vom Entwickler schreiben zu lassen.
Um solche Änderungen an den Tests durch Entwickler zu vermeiden, kann in Betracht gezogen werden einen Ansatz der „symbolischen Ausführung“ zu untersuchen oder Mutationen zu verwenden.

Mittels „symbolischer Ausführung“ können die Abzweigungen einer Methode untersucht und vermerkt werden. Sollte für eine Abzweigung kein Testfall existieren, können durch Zufallswerte ein Testfall für die fehlende Abzweigung generiert werden [vgl. Păsăreanu, 2008].
Bei einem Ansatz mit Mutationen, im Bezug auf genetische Algorithmen, können Testfälle als Chromosomen verwendet werden [vgl. Fraser, 2015] und minimale Veränderungen an ihnen vorgenommen werden, bis der komplette Code der Methode abgedeckt ist. Dabei muss darauf geachtet werden, dass nur Veränderungen als Testfall behalten werden, wenn sie eine Verbesserung der Testabdeckung darstellen und die Anzahl der Tests minimal bleibt.
Solche Lösungsansätze verfälschen die realitätsnahen Daten und es ist zu klären, ob weiterhin Statistiken an den Tests notiert werden und was sie aussagen. Zusätzlich kann es durch diese Ansätze passieren, dass Methoden mit ungültigen Werten aufgerufen werden.

5.5. Statistik

Der TestGenerator reduziert Testfälle, die dieselben Zeilen, entweder die im JSON oder die der JaCoCo-XML, abdecken. Wird ein Testfall entfernt, wird an dem Testfall, der den entfernten Testfall mit abdeckt, die Statistik hochgezählt. So ist erkenntlich, dass dieser Testfall im Durchlauf zweimal vorkam.
Die Werte repräsentieren nicht das Kundenverhalten, da sie den Werten des Entwicklungsservers entsprechen und dort primär Pfade für neue Features durchgeklickt werden. Mittels dieser Werte kann abgelesen werden mit welchen Parametern die Methode häufig oder nicht so häufig aufgerufen wird. Meist wird eine Methode mehrfach aufgerufen, auch wenn der Anwender im Webshop nur einen Pfad durchläuft.
Zur Zeit kann nicht beurteilt werden, ob die Statistik hilfreich oder gar hinderlich für den Entwickler ist. Dies wird sich während der Arbeit mit dem TestGenerator zeigen oder es müssen Studien dazu durchgeführt werden. Vielleicht können im Zuge dessen auch andere Statistiken ermittelt werden, die wertvolle Informationen für den Entwickler liefern.

5.6. Performance

Auf einem Entwicklungssystem wurden mittels `Apache JMeter`[21] exemplarische Web Tests durchgeführt. Als mit AspectJ zu protokollierende Methode wurde eine ausgewählt, die direkt mit dem Homepage-Aufruf angesprochen wird. Somit muss nur diese Seite aufgerufen und nicht weiter durch den Webshop manövriert werden.
`Apache JMeter` wurde so konfiguriert, dass 50 Nutzer in ständiger Wiederholung die Homepage eine halbe Minute lang aufrufen. Die Ergebnisse sind in Tabelle 5.1 aufgeführt. Es kann davon ausgegangen werden, dass der Server nicht großartig durch andere Requests beschäftigt wurde.
Es ist zu erkennen, dass der Server mit AspectJ weniger Requests bearbeitet, die Latenz etwas steigt und der Verbindungsaufbau länger braucht, gegenüber dem Server ohne AspectJ. Da der Test nur für eine halbe Minute durchgeführt wurde, sind die Abweichungen so geringfügig.

[21] `https://jmeter.apache.org/`

	mit AspectJ	ohne AspectJ
Anzahl Requests	1.100	1.000
Latenz	1.300 ms	1.500 ms
Verbindungsaufbau	145 ms	164 ms

Tabelle 5.1.: Ergebnisse des Web Tests mit 100 Nutzern in einer Minute

Wenn man die Werte hochrechnet, ist der Server mit AspectJ deutlich in der Performance beeinträchtigt. Damit ist es nicht möglich AspectJ auf einem Server zu nutzen, der produktiv verwendet wird. Um sicher zu gehen, sollten größer aufgestellte Tests durchgeführt werden.

5.7. Zusammenfassung

Die Generierung der Tests mit dem TestGenerator funktioniert. Bei der Auswahl der Methoden gibt es keine Einschränkungen, selbst erstellte Objekte werden im Test gemockt, wodurch damit gearbeitet werden kann, und auch die Lesbarkeit der Tests ist annehmbar. Im Vergleich zu Tests, die von Entwicklern geschrieben wurden, gibt es noch Verbesserungspotential, bezogen auf die Größe der Testklasse.
Bei großen Projekten benötigt die Generierung mehrere Minuten, um das Projekt aufzuräumen und anschließend die Tests zu erstellen. Dies bedarf definitiv Optimierungsbedarf, wobei dafür noch kein Ansatz existiert.
Eine 100%-ige Testabdeckung ist möglich, dafür müssen aber die passenden Parameter übergeben werden. Es ist fraglich, ob hier Verbesserungen notwendig sind. Der TestGenerator greift explizit nicht auf zufällige Werte zu, da die Möglichkeit besteht reale Daten zu verwenden. Um immer eine 100%-ige Testabdeckung gewährleisten zu können, müssten Ansätze implementiert werden, die die protokollierten Daten verändern.
Aussagen zur Nützlichkeit der Statistik an den Testfällen können nicht getroffen werden. Objektiv betrachtet gibt sie Auskunft darüber, wie oft die Methode mit den Parameterwerten im Test, im Vergleich zu allen Aufrufen, ausgeführt wurde.
Die exemplarischen Performance-Tests haben ergeben, dass AspectJ den Server beeinträchtigt. Innerhalb einer halben Minute benötigte der Server 20 ms länger für den Verbindungsaufbau als der Server ohne AspectJ. Für genauere Aussagen müssen umfangreichere Tests durchgeführt werden.

6. Fazit

Um die Qualität eines Programmes sicherzustellen, wird es getestet. Software kann manuell oder automatisiert getestet werden. Beides ist zeitaufwendig, aber notwendig.
Zur Unterstützung der Entwickler gibt es Testgenerierungs-Tools, die automatisiert Testfälle für ein Programm erstellen. Viele Tools für Java-Programme arbeiten hierfür mit Zufallswerten und JUnit.
Im Rahmen dieser Arbeit wurde ein Testgenerierungs-Tool entwickelt, welches für Java-Programme Testfälle in Groovy und Spock generiert, wobei nicht auf zufällig Werte, sondern auf realitätsnahe Daten zurückgegriffen wird.
Dank der `arrange-act-assert`-Struktur von Spock und dem syntaktischem Zucker von Groovy sind die generierten Tests übersichtlich und lesbar.
Durch die Verwendung von JaCoCo werden verschiedene Abdeckungen der Tests generiert, die in den Testklassen notiert werden. Damit ist direkt sichtbar wie viel Code die Tests abdecken.

6.1. Testfallminimierung

Mittels zwei verschiedenen Ansätzen zur Testfallminimierung, die an einem genetischen Algorithmus angelehnt sind, hält sich die Anzahl der Testfälle in Grenzen. Beide Ansätze verfolgen die Strategie Testfälle zu löschen, wenn die Zeilen oder die protokollierten Abdeckungen identisch sein.
Der Nachteil des Ansatzes, der auf die JaCoCo-Abdeckungen prüft ist, dass diese für jeden generierten Testfall erstellt wird und somit für große Projekte viel Zeit benötigt wird bis alle Abdeckungen ermittelt wurden. Dies gilt es in einer Weiterentwicklung des TestGenerators zu verbessern.
Der Ansatz, der die Zeilen miteinander vergleicht funktioniert nicht immer, da dies nur Zeilen sind, die mittels AspectJ ermittelt wurden. Somit sind es Zeilen der Methodenaufrufe die AspectJ protokolliert, aber keine Operationen die dazwischen stattfinden. Ruft eine Methode keine andere Methode auf, enthält die Liste der Zeilen nur die Aufrufzeile der einen Methode. Damit unterscheiden sich alle Methodenaufrufe dieser Methode, solange sie von einer anderen Zeile aus aufgerufen werden. Auch hier gibt es Verbesserungsbedarf.

6.2. Testabdeckung

Es ist möglich, mit durch den TestGenerator erstellten Testfällen, eine 100%-ige Zweig- und Zeilenabdeckung zu erreichen. Die Bedingung dafür ist, dass die Methode mit den dafür notwendigen Parameterkombinationen aufgerufen wird. Soll immer eine 100%-ige Testabdeckung erreicht werden, müssen die protokollierten Daten verändert werden.

Für kleine Projekte ist der TestGenerator nicht sehr gewinnbringend, da die zu testende Methode mit den Parameterwerten, die für das Testen verwendet werden sollen, aufgerufen werden muss, damit das Generieren der Tests funktioniert. Stattdessen könnten direkt Tests geschrieben werden.
Größere Projekte, wie das Webshop-Projekt, profitieren mehr vom TestGenerator. Für Methoden müssen meisten mehrere Mocks aufgeschrieben werden, damit sie im Test ausführbar ist. Und nicht immer sind die Werte der Mocks einfach zu ermitteln. Durch das Klicken im Webshop werden die Testfälle mit den Mocks automatisch generiert und ersparen dem Entwickler Einarbeitungszeit in den Code und Zeit, die zum Ermitteln und zum Schreiben der Mocks investiert worden wäre. Je länger der zu testende Code existiert, desto schwieriger ist es die fachliche Anforderung zu verstehen. Außerdem geben die realitätsnahen Daten, die in den Tests verarbeitet werden, dem Entwickler Hinweise für weitere Tests. Dies ist bei zufällig generierten und damit meist kryptischen Werten nicht der Fall.
Stichprobenartig wurden für das Webshop-Projekt mit `EvoSuite` Testfälle generiert. Für einfache Klassen mit Gettern und Settern wurden Testfälle erzeugt, die eine sehr gute Testabdeckung aufweisen. Bei komplexeren Klassen brach die Generierung frühzeitig ab und es kamen keine Tests zustande. Die Vermutung ist, dass die Abhängigkeiten der Klasse zu komplex waren oder `EvoSuite` nicht mit `CDI` injizierten Variablen[22] umgehen kann. Der TestGenerator hingegen kann für solche Klassen Testfälle erstellen.

6.3. Ausblick

Wie gerade beschrieben gibt es Klassen im Webshop-Projekt für die `EvoSuite` keine Testfälle generieren kann. Auch der TestGenerator kann nicht für alle Klassen Tests generieren, die durchlaufen. Einige Fehler müssen am TestGenerator selbst behoben werden. Andere Fehler wiederum werden dadurch verursacht, dass es Abhängigkeiten gibt, die nicht beachtet werden, da der TestGenerator allgemein gültig sein soll. Würde der Code des TestGenerators dafür angepasst werden, kann es sein, dass er nicht mehr für andere Projekte verwendbar ist. Oder der Code muss für jedes Projekt angepasst werden. Eine elegantere Variante stellt die Möglichkeit von personalisierten Konfigurationen dar. Mittels Pattern, die in der Konfiguration niedergeschrieben sind, weiß der TestGenerator welche Abhängigkeiten gelten und geht darauf ein. Hier muss untersucht werden, ob und wie dieser Ansatz realisierbar ist.
Sollten umfangreichere Tests ergeben, dass die Server-Performance mit AspectJ nur minimal beeinträchtigt wird, bleibt immer noch die Problematik der realen Daten bezogen auf die Datenschutz-Grundverordnung. Kundendaten dürfen ohne Einwilligung oder triftigen Grund nicht verwendet werden. Ohne die Einwilligung müssen Personenbezogene Daten anonymisiert oder pseudonymisiert werden. Die Notwendigkeit und Machbarkeit des Themas kann in nachfolgenden Arbeiten näher untersucht werden.

[22] `https://docs.oracle.com/javaee/6/tutorial/doc/giwhl.html`

A. Anhang

Literaturverzeichnis

[Andalib, 2014] A. Andalib und S. M. Babamir. "A new approach for test case generation by discrete particle swarm optimization algorithm". In: *2014 22nd Iranian Conference on Electrical Engineering (ICEE)*. Mai 2014, S. 1180–1185. DOI: 10.1109/IranianCEE.2014.6999714.

[Böhm, 2006] Böhm. *Aspektorientierte Programmierung mit AspectJ 5*. 1. Aufl. Heidelberg: dpunkt.verlag, 2006. ISBN: 3-89864-330-1.

[Boyapati, 2002] Chandrasekhar Boyapati, Sarfraz Khurshid und Darko Marinov. "Korat: Automated Testing Based on Java Predicates". In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '02. New York, NY, USA: ACM, 2002, S. 123–133. ISBN: 978-1-58113-562-6. DOI: 10.1145/566172.566191. URL: http://doi.acm.org/10.1145/566172.566191 (besucht am 26. 03. 2018).

[Fraser, 2011] Gordon Fraser und Andrea Arcuri. "EvoSuite: automatic test suite generation for object-oriented software". en. In: ACM Press, 2011, S. 416. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025179. URL: http://dl.acm.org/citation.cfm?doid=2025113.2025179 (besucht am 19. 03. 2018).

[Fraser, 2015] Gordon Fraser u. a. "EvoSuite at the SBST 2015 Tool Competition". In: IEEE, Mai 2015, S. 25–27. ISBN: 978-1-4673-7079-0. DOI: 10.1109/SBST.2015.13. URL: http://ieeexplore.ieee.org/document/7173586/ (besucht am 19. 03. 2018).

[Hu, 2013] Hai Hu u. a. "Enhancing software reliability estimates using modified adaptive testing". In: *Information and Software Technology*. Special Section: Component-Based Software Engineering (CBSE), 2011 55.2 (Feb. 2013), S. 288–300. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.08.012. URL: http://www.sciencedirect.com/science/article/pii/S0950584912001796 (besucht am 27. 09. 2017).

[Kapelonis, 2017] Kostis Kapelonis. *Spock testing framework versus JUnit · Codepipes Blog*. Apr. 2017. URL: http://blog.codepipes.com/testing/spock-vs-junit.html (besucht am 13. 03. 2018).

[Marinov, 2001] D. Marinov und S. Khurshid. "TestEra: a novel framework for automated testing of Java programs". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. Nov. 2001, S. 22–31. DOI: 10.1109/ASE.2001.989787.

[McCabe, 1976] T. J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dez. 1976), S. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[Pacheco, 2007] Carlos Pacheco und Michael D. Ernst. “Randoop: Feedback-directed Random Testing for Java”. In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA ’07. New York, NY, USA: ACM, 2007, S. 815–816. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297902. URL: http://doi.acm.org/10.1145/1297846.1297902 (besucht am 19. 03. 2018).

[Panichella, 2017] A. Panichella und U. R. Molina. “Java Unit Testing Tool Competition - Fifth Round”. In: *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. Mai 2017, S. 32–38. DOI: 10.1109/SBST.2017.7.

[Păsăreanu, 2008] Corina S. Păsăreanu u. a. “Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software”. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA ’08. New York, NY, USA: ACM, 2008, S. 15–26. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390635. URL: http://doi.acm.org/10.1145/1390630.1390635 (besucht am 26. 03. 2018).

[Saadtjoo, 2018] M. A. Saadtjoo und S. M. Babamir. “Optimizing Cost Function in Imperialist Competitive Algorithm for Path Coverage Problem in Software Testing”. In: *Journal of AI and Data Mining* 6.2 (Juli 2018), S. 375–385. ISSN: 2322-5211. DOI: 10.22044/jadm.2017.5015.1603. URL: http://jad.shahroodut.ac.ir/article_1013.html (besucht am 19. 03. 2018).

[Sakti, 2016] Abdelilah Sakti, Gilles Pesant und Yann-Gaël Guéhéneuc. “JTExpert at the fourth unit testing tool competition”. en. In: ACM Press, 2016, S. 37–40. ISBN: 978-1-4503-4166-0. DOI: 10.1145/2897010.2897021. URL: http://dl.acm.org/citation.cfm?doid=2897010.2897021 (besucht am 19. 03. 2018).

[Sen, 2006] Koushik Sen und Gul Agha. “CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools”. In: *Proceedings of the 18th International Conference on Computer Aided Verification*. CAV’06. Berlin, Heidelberg: Springer-Verlag, 2006, S. 419–423. ISBN: 978-3-540-37406-0. DOI: 10.1007/11817963_38. URL: http://dx.doi.org/10.1007/11817963_38 (besucht am 26. 03. 2018).

[Tonella, 2004] Paolo Tonella. “Evolutionary Testing of Classes”. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’04. New York, NY, USA: ACM, 2004, S. 119–128. ISBN: 978-1-58113-820-7. DOI: 10.1145/1007512.1007528. URL: http://doi.acm.org/10.1145/1007512.1007528 (besucht am 27. 09. 2017).

[Zhang, 2011] S. Zhang. “Palus: a hybrid automated test generation tool for java”. In: *2011 33rd International Conference on Software Engineering (ICSE)*. Mai 2011, S. 1182–1184. DOI: 10.1145/1985793.1986036.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

B. Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich diese Masterarbeit selbstständig, ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle, den benutzten Quellen wörtlich oder sinngemäß, entnommenen Stellen sind als solche einzeln kenntlich gemacht.
Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt worden und auch nicht veröffentlicht worden.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Bremen, den 9. April 2018

_____

Anika Bracht