HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Musterlösungen in Graja als Mechanismus für Regressionstests

Marc Herschel

Bachelor-Arbeit im Studiengang „Angewandte Informatik“

17. August 2020

Autor Marc Herschel
marc@herschel.io
1476130

Erstprüfer: Prof. Dr. Robert Garmann
Abteilung Informatik, Fakultät IV
Hochschule Hannover
robert.garmann@hs-hannover.de

Zweitprüferin: Prof. Dr. Frauke Sprengel
Abteilung Informatik, Fakultät IV
Hochschule Hannover
frauke.sprengel@hs-hannover.de

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Bachelor-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Marc Herschel

Hannover, den 17. August 2020 Unterschrift

Abstract

Der Autobewerter Graja, der in der Lehre zum Bewerten studentischer Java-Programme verwendet wird, stellt ein komplexes Softwaresystem dar. Aufgrund einer kaum vorhandenen Testabdeckung durch Modul- und Integrationstests, ist die Gewährleistung der fehlerfreien Funktionalität in Hinsicht auf die Weiterentwicklung nicht garantiert. Da sich Graja auf das ProFormA-Aufgabenformat stützt, stellt sich die Frage, inwiefern sich die im ProFormA-Aufgabenformat vorausgesetzten Musterlösungen, für einen automatisierten Regressionstestmechanismus eignen.

Das Ziel dieser Forschung ist es ein Konzept, für einen solchen Regressionstestmechanismus zu erstellen und mithilfe einer Referenzimplementierung als Graja-Erweiterung in die Praxis umzusetzen. Der daraus entstandene Mechanismus operiert durch Verhaltensaufzeichnung und Verhaltensabgleich und konvertiert so das in Graja beobachtete Bewertungsverhalten einer Musterlösung in einen Testfall. In der Testphase findet anschließend ein Abgleich des Soll-Verhaltens eines Testfalls und des beobachteten Ist-Verhaltens einer Musterlösung statt. Die Differenzen dieses Abgleichs sind als potenzielle Regressionen zu behandeln, da diese eine semantische Änderung des Bewertungsergebnisses darstellen.

Um diesen Verhaltensabgleich robust und mit möglichst wenigen Fehlalarme zu realisieren, wurden die in Graja verwendeten Datenmodelle auf Eignung bezüglich einer Verhaltensaufzeichnung untersucht. Außerdem fand eine Datenaufzeichnung mit einer Teilmenge der Musterlösungen statt. Nachfolgend wurde eine Analyse dieser Rohdaten, mit dem Ziel potenzielles Rauschen innerhalb der Aufzeichnungen zu detektieren, durchgeführt. So konnte letztendlich eine Strategie für eine Rauschunterdrückung innerhalb der Verhaltensaufzeichnung entwickelt werden.

Abschließend wurde ein Datenmodell entwickelt, das erlaubt, die durch den Verhaltensabgleich detektierten Regressionen verständlich und lokalisierbar darzustellen. Der durch diese Arbeit entstandene automatisierte Regressionstestmechanismus stellt somit eine Grundlage für die Gewährleistung der korrekten Bewertungsfunktionalität innerhalb des Graja-Entwicklungsprozesses dar. Durch das Detektieren von Regressionen mithilfe der Musterlösungen, lassen sich nun Änderungen an Graja gewissenhaft in eine Produktionsumgebung übernehmen.

Schlüsselwörter: *Graja (Autobewerter)*, *Softwareentwicklung*, *Softwaretests*, *Regressionstests*

Inhaltsverzeichnis

Abbildungsverzeichnis **v**

Tabellenverzeichnis **vi**

Listingverzeichnis **vii**

Glossar **viii**

Abkürzungsverzeichnis **ix**

1 Einleitung **1**
1.1 Motivation 1
1.2 Ziele dieser Arbeit 1
1.3 Aufbau dieser Arbeit 1
1.4 Typografische Konventionen dieser Arbeit 2
1.5 Anhang dieser Arbeit 2

2 Grundlagen **3**
2.1 Graja-Testmodule 3
2.1.1 Modul: Compile 3
2.1.2 Modul: JUnit 4
2.1.3 Modul: Checkstyle 4
2.1.4 Modul: PMD 4
2.2 Sicherheitsaspekte 5
2.3 Bewertungsaspekte und Bewertungsschema 5
2.4 Feedback 7
2.5 Aufgaben 8
2.5.1 Aufbau von Aufgaben 8
2.5.2 Musterlösungen 8
2.5.3 Variable Aufgaben 9
2.5.4 Überführung in das ProFormA-Aufgabenformat 9
2.6 Ausführen von Graja 10
2.6.1 Ausführung durch Graja-CLI 11
2.6.2 Ausführung durch Java-API 11
2.6.3 Interne Ausführungsphasen 12
2.7 Regressionen und Regressionstests 12
2.7.1 Abgrenzung zu Modul- und Integrationstests 12
2.7.2 Testumgebungen für Regressionstests 13
2.8 Problemstellung 14
2.8.1 Musterlösungen in Graja als Mechanismus für Regressionstests . 15

3 Ziele **17**
3.1 Forschungsfragen 17
3.2 Nicht-Ziele 18

4 Methodik **20**
4.1 Literaturrecherche 20
4.2 Quellcodeanalyse 20
4.3 Datenerhebung und Analyse 20

4.4 Entwicklung einer Referenzimplementierung 20

5 Lösungsmöglichkeiten 21
5.1 Verhaltensaufzeichnung anhand von Musterlösungen 21
5.1.1 Integration in den Bewertungsvorgang 21
5.1.2 Erweiterung des Graja-CLI 22
5.1.3 Erweiterung des Graja-Domänenmodells 23
5.1.4 Abspeicherung des aufgezeichneten Verhaltens 25
5.1.5 Validierung der Symmetrie von Aufzeichnung und Testfall . . . 27
5.2 Eignung der durch Graja generierten Daten für Verhaltensaufzeichnungen 28
5.2.1 Eignung der generierten Feedback-Dokumente 28
5.2.2 Eignung der internen Result-Datenstruktur 30
5.2.3 Eignung der Zwischenrepräsentation der Kommentare 33
5.2.3.1 Eignung der verschiedenen Kommentar-Ebenen zum Erkennen von Regressionen 35
5.2.3.2 Lokalisierung der Kommentar-Herkunft im Quellcode . 37
5.2.4 Ein Datenmodell für die Verhaltensaufzeichnung 38
5.3 Behandlung von Fehlalarmen und Rauschen in Aufzeichnungen 40
5.3.1 Generierung von Rohdaten für eine Rauschidentifikation 41
5.3.2 Identifikation und Unterdrückung von Rauschen 42
5.3.2.1 Dateipfad-bedingtes Rauschen 43
5.3.2.2 Zeitangaben bedingtes Rauschen 43
5.3.2.3 *Count*-Attribut einer *ProFormA-Id* 44
5.3.3 Levenshtein-Distanz als Fehlertoleranz 44
5.3.4 Fehlalarme durch unvorhersehbare Abarbeitungsreihenfolgen . . 45
5.3.5 Fehlalarme durch Zufallsgeneratoren innerhalb der Aufgaben . . 45
5.3.6 Fehlalarme durch Zufallsgeneratoren innerhalb der Einreichungen 48
5.3.7 Musterlösungen mit mehreren validen Ausführungssträngen . . . 49
5.4 Verhaltensabgleich: Beobachtetes Ist-Verhalten mit Soll-Verhalten der persistierten Verhaltensaufzeichnung 49
5.4.1 Behandlung von Bewertungsabbrüchen 50
5.4.2 Struktureller Vergleich von Baumstrukturen auf Differenzen . . 51
5.4.2.1 Baum-Isomorphismus und AHU-Algorithmus 51
5.4.2.2 Überprüfung zweier Baumstrukturen auf strukturelle Äquivalenz 53
5.4.2.3 Lokalisierung von Differenzen nach einem strukturellen Baum-Vergleich 55
5.4.2.4 Umwandlung zum Sequenznummer-Pfad 56
5.4.2.5 Umwandlung zum strukturellen Pfad 57
5.4.2.6 Extraktion aller lokalisierbarer Differenzen 58
5.4.3 Behandlung von Kommentarbäumen 60
5.4.4 Behandlung der Bewertungsaspekte und Aspektgruppen 60
5.4.5 Eine Datenstruktur für den Testbericht 61
5.4.5.1 Addressierbarkeit von Regressionen im Testbericht . . 63
5.4.5.2 Kategorisierung von Regressionen im Testbericht . . . 63
5.4.5.2.1 Inhaltliche Regressionen der Kommentare . . 64
5.4.5.2.2 Strukturelle Regressionen der Kommentarbäume 64
5.4.5.2.3 Anderwertige Regressionen 64

5.5 Aufbau einer Test-Infrastruktur um Graja 65
5.5.1 Schnittstelle für den Regressionstestmechanismus 66
5.5.2 Umgang mit variablen Aufgaben 68
5.5.3 Generierung des Testberichts 69

6 Implementierung 70
6.1 Umsetzung der Verhaltensaufzeichnung 70
6.1.1 Umsetzung des Stackdump-Mechanismus 70
6.1.2 Umsetzung der Verhaltensaufzeichnung und inter-JVM Kommunikation 70
6.1.3 Umsetzung der Rauschunterdrückung 72
6.1.4 Umsetzung der Maßnahmen zur Minimierung von Fehlalarmen . 73
6.1.5 Fazit: Umsetzung der Verhaltensaufzeichnung 73
6.2 Umsetzung des Verhaltensabgleichs 74
6.2.1 Grafische Illustration von Kommentarbaum-Differenzen 75
6.2.2 Fazit: Umsetzung des Verhaltensabgleichs 76
6.3 Umsetzung der öffentlichen Schnittestelle 76
6.3.1 Anbindung an Gradle 78
6.3.2 Fazit: Umsetzung der öffentlichen Schnittstelle 79

7 Ergebnisse 80
7.1 Notwendige Einstellungen um Fehlalarme zu vermeiden 81
7.2 Demonstration anhand künstlich erzeugter Regressionen 82
7.2.1 Erzeugung einer Regression im Grader-Code 82
7.2.2 Erzeugung einer Regression in Graja 84
7.3 Fazit: Demonstration des Regressionstestmechanismus 86

8 Ausblick 87

Literaturverzeichnis 89

Anhang 94
Indirekter Aufruf von `gradeRequest` über `WithinJVMStarter` 95
Werkzeug: Inspector 96
Komplette Ausgabe des Inspector-Werkzeugs unter Ausführung der Musterlösung *wrong_fake* der Aufgabe *de.hsh.prog.serialize* 99
Werkzeug: Comment-Dumper 100
Für die Rauschidentifikation verwendeten Aufgaben und Musterlösungen . . 103
Werkzeug: Stacktrace-Dumper 104
Werkzeug: Comment-Dump-Analyzer 105
Statistiken des Werkzeugs: *Comment-Dump-Analyzer* 106
Pseudocode für einen Algorithmus um Baumstrukturen auf Äquivalenz zu überprüfen und die gefundenen Differenzen auszugeben 108

Abbildungsverzeichnis

1 Zwei *RDKindTO*-Domänenobjekte mit unterschiedlichen Einstellungen 7
2 Unterschiede zwischen variablen Aufgaben: Schablone und Instanz . . . 9
3 Überführungsprozess in das ProFormA-Aufgabenformat 10
4 Hierarchie der Graja-Requests 10
5 Interne Graja-Ausführungsphasen vor und während eines Bewertungsvorgangs 12
6 Möglicher Aufbau einer Regressionstest-Pipeline 14
7 Hülle um die Klasse `Core` um eine Verhaltensaufzeichnung zu realisieren 22
8 Erweiterung des Graja-Domänenmodells um eine Anweisung zur Verhaltensaufzeichnung 24
9 Struktur der von Graja generierten Zwischenrepräsentation 30
10 Klassenmodell der programmatischen Kommentar-Abstraktion 33
11 Zwischenrepräsentation-Generat-Mapping der unterschiedlichen Kommentarbäume innerhalb der Klasse `AbstractResultNode` 34
12 Grafische Darstellung der unterschiedlichen Ebenen einer Instanz der in Abbildung 9 gezeigten Klasse `Result` 35
13 Erweiterung der Klasse *Content* um einen *Stackdump* während der Konstruktionsphase durchzuführen 37
14 Vorgeschlagenes Datenmodell um das aufgezeichnets Verhalten einer Musterlösung zu persistieren 38
15 *Factory* um `Random`-Instanzen mit deterministischen *seed* anzubieten . . 47
16 Funktionsweise des Modus *multi sample mode* beim Aufzeichnen 49
17 Beispiel zum Baum-Isomorphismus anhand zweier Bäume von unterschiedlicher Struktur 51
18 Beispiel eines Knuth-Tuples in zwei verschiedenen Schreibweisen 52
19 AHU-Algorithmus um einen Baum-Isomorphismus zu bestimmen 53
20 Algorithmus zur Bestimmung der Äquivalenz einer Baum-Struktur unter Beachtung der Kind-Reihenfolge 54
21 Algorithmus zur Bestimmung der Äquivalenz von Baum-Strukturen unter der Beachtung der Kind-Reihenfolge bei unterschiedlicher Knoten-Anzahl 55
22 Notwendige Backtracking-Informationen um absolute Pfade innerhalb des Baums, in dem eine Differenz detektiert wurde ,zu rekonstruieren . 55
23 Illustrierte Anwendung der Algorithmen aus Listing 19 und 20 57
24 Teilausschnitt des in Listing 22 dargestellten Bewertungsschemas 60
25 Datenstruktur für einen Testbericht 61
26 Abstrahiertes Aktivitätsdiagramm der Operation `Test <Task>` 66
27 Öffentliche Schnittstelle des Regressionstestmechanismus 67
28 Verbindung von Aufgaben-Repositorien mit der Schnittstelle 68
29 Mögliches Layout für den Testbericht des Regressionstestmechanismus . 69
30 Verhaltensaufzeichnung mithilfe von `BehaviorRecordingCore` und Kommunikation zwischen zwei JVMs über die Klasse `TransferService` . . 71
31 Auf regulären Ausdrücken basierende Rauschunterdrückung durch die Klasse `NoiseReductor` 72
32 Verhaltensabgleich mithilfe der Klasse `RegressionDetector` 74
33 Visualisierung der in einem Kommentarbaum-Abgleich gefundenen Differenzen mithilfe der Klasse `TreeVisualizer` 75

34 Öffentliche Gradle-Schnittstelle der Referenzimplementierung mithilfe der Klasse `RegressionTestingOperations` 76
35 Grafische Darstellung von *Gradle Tasks* und *Task Groups* durch das *Eclipse Buildship Plugin* 78
36 Teilausschnitt des Testberichts nach erstmaliger Verhaltensaufzeichnung 80
37 Teilausschnitt des Testberichts nach anschließendem Testdurchlauf . . . 80
38 Erkannte Regression nach unrealistischer Änderung im Grader-Code der Aufgabe *de.hsh.prog.bruch* 82
39 Teilausschnitt der illustrierten Differenzen eines Kommentar-Baums . . 83
40 Ausgabe aller Kinder zweier Knoten mit unterschiedlicher Kind-Anzahl 83
41 Detektierte Regression durch widersprüchliche `Grade`-Statuscodes . . . 84
42 Regression durch Kommentar mit widersprüchlichen Textinhalt 84
43 *Stacktrace* eines Baumknotens mit widersprüchlicher Kinder-Anzahl . . 85
44 Anhang: Indirekter Aufruf von `gradeRequest` über `WithinJVMStarter` 95
45 Enkodierung einer *CommentDumper*-Aufzeichnung 100

Tabellenverzeichnis

1 Relevante Graja-Testmodule dieser Arbeit 3
2 Liste aller verfügbaren ProFormA-Aggregatfunktionen 32
3 Alle Statuscodes bezüglich des Vergleichs von Bewertungsabbrüchen . . 62
4 Alle Statuscodes bezüglich des Ergebnis eines Regressionstests 62
5 Alle optionalen Steuerungsparameter des Regressionstestmechanismus, die pro Aufgabe über die `assignment.properties` gesetzt werden können 67
6 Notwendige Einstellungen in den `assignment.properties`-Dateien der Aufgaben, um einen fehlalarmfreien Testdurchlauf zu garantieren 81
7 Liste aller verfügbaren *Comment-Dump-Analyzer*-Steuerungsparameter 105

Listingverzeichnis

1 Stark vereinfachtes Beispiel einer Konfiguration eines Bewertungsschemas im ProFormA-Aufgabenformat . . . 6
2 Beispiel für in task.zip eingebettete Musterlösungen anhand der task.xml des ProFormA-Aufgabenformats . . . 8
3 Beispiel für Musterlösungen in der task.xml, die auf die in Listing 2 deklarierten Dateien zugreifen . . . 9
4 Graja-Module, die Modultests im Build-Prozess ausführen . . . 14
5 SLOC-Metrik aller derzeit existierenden 31 Graja-Module . . . 15
6 Summe aller Aufgaben und Musterlösungen im Aufgaben-Repositorium, die als Testfälle für einen Regressionstestmechanismus dienen können[1] . 16
7 Beispiel für die Abspeicherung des aufgezeichneten Verhaltens im Wurzelverzeichniss einer Aufgabe . . . 26
8 Ansätze für die Abspeicherung des aufgezeichneten Verhaltens als Anhang in einer task.xml . . . 27
9 Präfixe aller derzeit existierenden Musterlösungen innerhalb des Aufgaben-Repositorium *GrajaAssignments* . . . 27
10 Ausschnitt des generierten HTML eines Feedback-Dokuments . . . 29
11 Verkürzte Ausgabe des *Inspector*-Werkzeugs bei der Musterlösung *wrong_fake* der Aufgabe *de.hsh.prog.serialize* . . . 31
12 Verleich von Kommentaren, die durch eine externe JVM und der JVM des Aufrufers generiert wurden . . . 36
13 Zufällig ausgewählte Aufgaben für die Rauschidentifikation . . . 41
14 Fehlschlagende Rauschunterdrückung trotz Bereinigung durch reguläre Ausdrücke . . . 44
15 Erkannte Differenzen durch `Random`-Nutzung in der Aufgabe *de.hsh.prog.detectzip* . . . 46
16 Erkannte Differenzen durch `Random`-Nutzung in der Aufgabe *de.hsh.prog.serialize* . . . 46
17 Einmalige Verwendung der Klasse `Random` in einem statischen Initialisierer 47
18 Mehrfache Verwendung der Klasse `Random` durch Testmethoden . . . 48
19 Rekronstuktion eines Sequenznummer-Pfads mithilfe von Backtracking 56
20 Konvertierung: Sequenznummer-Pfad in strukturellen Pfad . . . 57
21 Beispielnutzung des *Inspector*-Werkzeugs mit einem *Result*-Objekt . . . 96
22 Anhang: Komplette Ausgabe des *Inspector*-Werkzeugs unter Ausführung der Musterlösung *wrong_fake* der Aufgabe *de.hsh.prog.serialize* . . . 97
23 Beispielnutzung des *CommentDumper*-Werkzeugs . . . 100
24 Von *Comment-Dumper* extrahierte und linearisierte Kommentare . . . 101
25 Beispiel einer von *Comment-Dumper* generierten *metadata.json* . . . 101
26 Für die Rauschidentifikation verwendeten Aufgaben und Musterlösungen 102
27 Beispiel eines abgefangenen *Stacktrace* einer *Content*-Instanz . . . 104
28 *Comment-Dump-Analyzer*-Statistiken nach Sortierung der Checkstyle-Dateien und PMD-Regelsätze und Exklusion der durch die Aufrufer JVM generierten Kommentare . . . 106
29 Pseudocode für einen Algorithmus, um Baumstrukturen auf strukturelle Äquivalenz zu überprüfen und die gefundenen Differenzen auszugeben . 108

Glossar

Checkstyle Werkzeug für die statische Analyse von Java-Quelltexten und zur Überprüfung der Einhaltung des Programmierstils.[2]. 3–5

Grader-Code Softwarepaket, bestehend aus JUnit-Testklassen, das Graja erlaubt die funktionelle Korrektheit von studentisch eingereichten Java-Programmen zu überprüfen. 1, 3–5, 7, 14, 33, 48, 82–87, 100

Gradle Gradle ist ein auf Java basierendes *Build*-Automatisierungs-Werkzeug, das z.B. Abhängigkeiten auflöst und beim Bauen von Softwareprojekten hilft.[3]. 9, 10, 20, 65–69, 77–80, 101

JUnit Modultest Framework zum Testen von Java-Programmen[4]. 3–5, 13, 14, 47, 61, 69, 71, 83, 85, 86

PMD Vielseitiges Werkzeug für die statische Analyse von Quelltexten[5]. Erlaubt sowohl die Überprüfung der Einhaltung von Programmierkonventionen als auch das Auffinden von möglichen Programmier- und Effizienzfehlern. 3–5

ProFormA-Aufgabenformat Standardisiertes, XML-basiertes Aufgabenformat[6], das den Austausch von automatisiert bewertbaren Programmieraufgaben zwischen verschiedenen Lernmanagementsystemen und Autobewertern erlaubt.[7]. ii, 1, 3, 5, 6, 8–10, 15, 19, 20, 25–27, 32, 89

ProFormA-Test Ein Test stellt im ProFormA-Aufgabenformat eine Möglichkeit zur Bewertung einer studentischen Einreichung dar. Im Kontext von Graja ist so ein Test z.B. das Checkstyle, PMD oder JUnit-Modul. Ein Test wird im Bewertungsschema über ein Bewertungskriterium referenziert. 1, 3, 5–7, 25, 29–33, 37, 61, 92, 96

task.xml XML-Konfigurationsdatei des ProFormA-Aufgabenformats. Enthalten sind z.B. die Definitionen der verwendeten ProFormA-Tests, das Bewertungschema, die Aufgabenbeschreibung und die Auflistung aller vorhandener Anhänge der task.zip. Eine detaillierte Beschreibung der Funktionalität findet sich im ProFormA-Whitepaper.[8] . 8, 10, 26, 31

task.zip ProFormA-kompatible ZIP-Datei, die eine Aufgabenkonfiguration, das Bewertungsschema, den Grader-Code und andere von Graja zur Bewertung von studentischen Einreichungen notwendigen Dateien enthält.[9]. 8, 10, 11, 22, 25, 26, 39, 42, 66, 78, 79, 90, 97

---

[2] `https://checkstyle.sourceforge.io/`
[3] `https://gradle.org/`
[4] `https://junit.org/`
[5] `https://pmd.github.io/`
[6] `https://github.com/ProFormA/proformaxml`
[7] [Gar16a] [S. 3-4]
[8] [MGF+07]
[9] [Gara] [1]

Abkürzungsverzeichnis

ACE .... Arbitrary Code Execution
API .... Application Programming Interface
AST .... Abstract syntax tree
CI .... Continuous Integration
CLI .... Command Line Interface
CSS .... Cascading Style Sheets
DOM .... Document Object Model
DoS .... Denial of Service
DSL .... Domain-specific language
Graja .... Grader for java programs
GUI .... Graphical User Interface
HTML .... Hypertext Markup Language
I/O .... Input/Output
ISP .... Interface segregation principle
JAR .... Java ARchive
JAXB .... Java Architecture for XML Binding
JDK .... Java Development Kit
JSON .... JavaScript Object Notation
JVM .... Java Virtual Machine
QA .... Quality Assurance
SLOC .... Source Lines of Code
SRP .... Single-responsibility principle
UML .... Unified Modeling Language
Unix .... Uniplexed Information and Computing System
URI .... Uniform Resource Identifier
URN .... Uniform Resource Name
UUID .... Universally Unique IDentifier
VCS .... Version Control System
XML .... Extensible Markup Language
XSD .... XML Schema Definition

1 Einleitung

1.1 Motivation

Graja[10] ist eine Software zur automatisierten Bewertung von studentischen Java-Programmen. Um Bewertungen durchzuführen, stützt sich Graja auf verschiedene Testmodule. Graja-Aufgaben sind im ProFormA-Aufgabenformat[11] definiert, das pro Aufgabe die auszuführenden ProFormA-Tests und das dazugehörige Bewertungsschema spezifiziert. Momentan stehen Aufgabenautoren fünf Testmodule[12] zur Verfügung um eine Einreichung zu bewerten. Graja ist eine modular aufgebaute Software, die zum Zeitpunkt der Arbeit aus 31 unterschiedlichen Modulen besteht. Eine Analyse mit dem Quellcode-Analysewerkzeug *cloc*[13] hat ergeben, dass alle Module zusammen mit eventuellen Modultests knapp 60000 Zeilen Java-Quellcode[14] akkumulieren. Änderungen innerhalb eines dieser Module kann möglicherweise zu unerwünschten Fehlern, sogenannten Regressionen, an anderen Stellen führen. Momentan existiert keine automatisierte Möglichkeit, solche Regressionen zu erkennen, was die Weiterentwicklung der Software verkompliziert. Demnach besteht stets eine potenzielle Ungewissheit, ob der Bewertungsprozess nach einer Änderung/Ergänzung eines Moduls noch korrekt für alle bisher existierenden Aufgaben durchgeführt wird. Dieser Umstand hält Graja-Nutzer möglicherweise davon ab, neue Versionen in eine Produktionsumgebung zu übernehmen.

1.2 Ziele dieser Arbeit

Musterlösungen[15] sind ein Bestandteil des ProFormA-Aufgabenformats. In der Regel werden sowohl positive als auch negative Musterlösungen erstellt. Diese Musterlösungen können unter der Annahme, dass mehrfache Bewertungen einer Musterlösung durch Graja und dem dazugehörigen Grader-Code zu reproduzierbaren Bewertungsergebnissen führen, einen Regressionstestmechanismus durch Verhaltensaufzeichnung und Verhaltensabgleich ermöglichen. Ein solcher Mechanismus kann Entwicklern dabei assistieren, unerwünschte Regressionen durch Änderungen an Modulen zu erkennen. Ziel dieser Arbeit ist es, die eben genannte Thematik auf Anwendbarkeit, Nutzen und Umsetzungsmöglichkeit zu überprüfen und – falls möglich – ein Konzept für Graja-Regressionstests anhand einer Referenzimplementierung, die in Verbindung mit Graja genutzt werden kann, umzusetzen.

1.3 Aufbau dieser Arbeit

Zu Beginn der Arbeit wird ein Einblick in Graja gegeben. Dabei werden unter anderem die derzeit existierenden Graja-Testmodule, die interne Funktionsweise und das von Graja genutzte Aufgabenformat behandelt. Darauffolgend werden die Ziele und das mit dieser Arbeit behandelte Problem detaillierter beschrieben. Anschließend werden Forschungsfragen formuliert und die gewählte Methodik erläutert. Ab Kapitel 5 gilt es dann Lösungsmöglichkeiten darzulegen und gegeneinander abzuwägen. Falls umsetzbar,

[10] Graja - Grader for java programs – `http://graja.hs-hannover.de/`
[11] [MGF+07]
[12] [Garb] [2.2.1]
[13] `http://cloc.sourceforge.net/`
[14] Metrik: *Source Lines of Code* (SLOC), Listing 5 in Kapitel 2.8.
[15] [MGF+07] [5.8]

wird in Kapitel 6 letztendlich auf die entwickelte Referenzimplementierung eingegangen. In Kapitel 7 dieser Arbeit werden dann die gewonnenen Ergebnisse präsentiert sowie mit Kapitel 8 ein Ausblick auf den neuen Erkenntnisstand gegeben.

Thematisch ist der Inhalt dieser Arbeit den Bereichen der Softwareentwicklung, Software- und Regressionstests zuzuordnen.

1.4 Typografische Konventionen dieser Arbeit

Abkürzungen werden bei der ersten Nennung jeweils einmal im Format: *Lange Abkürzung* (LK) definiert. Im darauffolgenden Text wird nur noch die Abkürzung verwendet. Alle in dieser Arbeit verwendeten Abkürzungen sind außerdem im Abkürzungsverzeichnis zu finden.

Hervorhebungen von z.B. Fachbegriffen, Domänenobjekten, Fragen oder fremdsprachlichen Wörtern erfolgen anhand dieser *kursiven* Schrift.

Für Dateipfade, Shell-Befehle, Klassen-/Methodennamen etc. wird zur Kennzeichnung diese `TrueType-Schriftart` verwendet.

Eine Ausnahme bilden hier Objekte, also Instanzen einer Klasse, die, um den Kontext zu verdeutlichen, *kursiv* dargestellt sind. Beispiel: Die Klasse `HelloWorld` erlaubt es die Welt zu grüßen. *HelloWorld*-Objekte müssen allerdings vor dem Grüßen durch die Methode `greet` in alle Weltsprachen übersetzt werden.

1.5 Anhang dieser Arbeit

Dadurch, dass diese Arbeit das Fachgebiet der Informatik behandelt, und somit digitale Artefakte entstehen, gibt es sowohl einen physischen als auch einen elektronischen Anhang. Der physische Anhang stellt den letzten Abschnitt dieser Arbeit dar und enthält weitere Informationen bezüglich des elektronischen Anhangs, der dieser Arbeit als DVD beiliegt.

2 Grundlagen

Graja ist ein automatischer Bewerter für Java-Programme[16]. Aufgabenautoren erstellen Graja-Aufgaben, die von Lehrpersonen eingesetzt werden, um studentische Einreichungen zu bewerten. Diese Aufgaben stellen ein kleines Softwarepaket dar und erlauben Aufgabenautoren so einen großen Spielraum bei der Definition von Aufgabenstellungen. Graja selber stellt hier nur das Bewertungswerkzeug dar und enthält keine zusätzlichen Werkzeuge für z.B. das Erkennen von Plagiaten unter den Einreichungen. Die gesamte Verwaltungsinfrastruktur muss von einem externen System zur Verfügung gestellt werden. In der Hochschule Hannover ist dies das *Lernmanagementsystem* (LMS) Moodle.[17]. Grajas Vorteile liegen sowohl bei der Entlastung der Kapazitäten des Lehrpersonals, als auch, der Möglichkeit, Studierenden eine sofortige Rückmeldung zu deren Abgaben zu liefern. Graja unterstützt außerdem das ProFormA-Aufgabenformat, das den Austausch von automatisiert bewertbaren Programmieraufgaben zwischen verschiedenen Lernmanagementsystemen und Autobewertern erlaubt.[18]

2.1 Graja-Testmodule

Testmodule stellen Werkzeuge dar, die von Graja genutzt werden, um studentische Einreichungen zu bewerten.[19] ProFormA-Tests, die im weiterem Verlauf des Kapitels behandelt werden, stützen sich auf Module. Momentan existieren fünf Testmodule in Graja, von denen vier in dieser Arbeit von Relevanz sind. Das Testmodul *human* erlaubt das Einfließen eines zeitverzögerten, nicht-automatisierten menschlichen Feedbacks in die Gesamtbewertung[20]. Dieses Modul hat für diese Arbeit allerdings keine Relevanz und wird daher nicht weiter behandelt.

Testmodul	Aufgabe
Compile	Kompilierung und Validierung der studentischen Einreichung
JUnit	Durchführung der vom Aufgabenautor durch Grader-Code definierten JUnit-Modultests
Checkstyle	Statische Analyse des Quellcodes der studentischen Einreichung mit Fokus auf Einhaltung der Programmierkonventionen und des Programmierstils
PMD	Statische Analyse mit ähnlichen Fähigkeiten von Checkstyle und zusätzlich noch Möglichkeiten zur Erkennung von möglichen Programmier- und Effizienzfehlern

Tabelle 1: Relevante Graja-Testmodule dieser Arbeit

2.1.1 Modul: Compile

Das Compile-Modul ist für die Kompilierung des Quellcodes der Einreichung zuständig. Möglicherweise beigelegter Bytecode einer Einreichung wird von Graja ignoriert.[21] Bei der Kompilierung wird sowohl der Bytecode generiert, auf den sich die ProFormA-Tests

[16] [Gar16b] [S. 4-6]
[17] `https://moodle.de/`
[18] [Gar16a] [S. 3-4]
[19] [Garb] [2.2.1]
[20] [Gar16b] [S. 5]
[21] [Gar16b] [S.6]

des JUnit-Moduls beziehen, als auch eine erste Validierung der Einreichung durchgeführt. Quellcode, der keine validen Java-Klassen enthält, kann nicht kompiliert werden und sorgt so automatisch für ein Nichtbestehen aller transitiv vom generierten Bytecode abhängigen Tests des JUnit-Moduls. Momentan muss das Compile-Modul immer mit einbezogen werden und vor dem JUnit-Modul laufen, da es sonst keine Möglichkeit gibt, den Bytecode der Einreichung zu generieren. Eine spezifische Testkonfiguration ist für dieses Modul nicht notwendig.[22]

2.1.2 Modul: JUnit

Das JUnit-Modul stützt sich momentan auf das in Java entwickelte Modultest-Framework JUnit 4[23]. In JUnit werden Tests anhand von `@Test` annotierten Methoden realisiert[24]. Pro Aufgabe existieren also ein oder mehrere Klassen, die ein oder mehrere solcher annotierten Methoden enthalten. Die Menge solcher Klassen pro Aufgabe stellen ein Softwarepaket dar, das im folgenden Verlauf der Arbeit als Grader-Code bezeichnet wird. Das JUnit-Modul bietet Aufgabenautoren alle Freiheiten von JUnit zum Bewerten von Aufgaben an.[25] Ein JUnit-Test kann entweder fehlschlagen oder fehlerfrei verlaufen. Ein Fehlschlag wird durch eine geworfene *Exception* signalisiert und sorgt dafür, dass der Test als nicht bestanden markiert wird. Mithilfe von *Assertions*[26] können verschiedene Bedingungen und Verhaltensweisen des zu testenden Programms überprüft werden. Eine *Assertion* ist eine Aussage über den Zustand des zu testenden Programms, der entweder *wahr* oder *falsch* sein kann. Mit dem JUnit-Modul kann so zum Beispiel überprüft werden, ob die Klasse einer Einreichung alle Felder mit dem Schlüsselwort `private` versehen hat.[27]

2.1.3 Modul: Checkstyle

Das Checkstyle-Modul erlaubt eine statische Analyse des eingereichten Quellcodes mithilfe des Werkzeugs Checkstyle. Der Hauptfokus liegt hier bei der Einhaltung von Programmierkonventionen. Checkstyle-Regeln werden anhand einer *Extensible Markup Language* (XML)-Datei konfiguriert.[28] Über die Aufgabenkonfiguration können diese Regeln dann mit in das Bewertungsschema aufgenommen werden. Checkstyle ermöglicht es zum Beispiel zu überprüfen, ob *Javadoc*[29]-Kommentare über den Klassen/Methoden der Einreichung existieren.

2.1.4 Modul: PMD

Das PMD-Modul basiert auf dem statischen Analysewerkzeug PMD. PMD erlaubt wie Checkstyle das Überprüfen der Einhaltung von vorher definierten Programmierkonventionen. Zusätzlich ist PMD in der Lage, mögliche Programmier- und Effizienzfehler zu erkennen. PMD ist besonders bei Aspekten bezüglich der Wartbarkeit, Effizienz und des Stils relevant. Aufgabenautoren können auch bei PMD eigene Regeln definieren.[30]

[22] [Gar16a] [S. 5]
[23] `https://junit.org/junit4/`
[24] [Gara][3]
[25] [Gar16b] [S. 13]
[26] [JUna]
[27] [Gar16a] [S. 9]
[28] [Gara] [4]
[29] `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`
[30] [Gar16b] [S. 7-8]

2.2 Sicherheitsaspekte

Da zur Bewertung der studentischen Einreichungen nicht vertrauenswürdiger Quellcode auf dem Graja-Hostrechner ausgeführt werden muss, stellt dies einen Angriffsvektor für *Arbitrary Code Execution* (ACE) und *Denial of Service* (DoS)-Attacken dar. Um dem vorzubeugen existieren verschiedene Sicherheitsmechanismen in Graja[31]. Graja selbst stützt sich außerdem auf die in Java eingebaute Sicherheitsarchitektur[32]. Um ACE zu verhindern wird auf den *Java Security Manager* zurückgegriffen und durch eine *Policy*-Datei[33] erlaubte Aktionen für sowohl den Grader-Code als auch die Einreichungen definiert. Die *Policy* stellt hier eine *Whitelist* dar, d.h erlaubte Aktionen müssen explizit freigeschaltet werden. Jede Aufgabe enthält eine solche *Policy*. Sollte eine zu prüfende Einreichung während der Laufzeit nun unerlaubte Aktionen wie z.B. unerwünschte Änderungen an Objekten durch die *Java Reflection API*[34] oder das Ausführen von potenziell schädlichen Konsolenkommandos[35] versuchen, wird der *Security Manager* durch fehlende *Policy*-Einträge eine *SecurityException* werfen. Um Schutz gegenüber DoS-Attacken zu gewährleisten, kann Graja unter Linux-Betriebssystemen den der Einreichung zur Verfügung stehenden Festplattenspeicher begrenzen. Eine Begrenzung des maximal zur Verfügung stehendem Arbeitsspeichers wird durch das Starten des Bewertungsvorgangs in einer separaten *Java Virtual Machine* (JVM) erreicht. Verklemmungen und Endlosschleifen werden mithilfe eines automatischen Abbruchs nach einer Zeitüberschreitung behandelt.

2.3 Bewertungsaspekte und Bewertungsschema

Bewertungsaspekte setzen sich in Graja aus einer zu erreichenden Punktzahl und einem Titel zusammen und können außerdem gewichtet werden.[36] Bewertungsaspekte sind frei definierbar, da diese sich auf die vorhandenen Testmodule (Kapitel 2.1) stützen und einen vom Aufgabenautor definierten ProFormA-Test ausführen. Ein ProFormA-Test ist in diesem Fall ein Domänenobjekt, das eine automatisierte Prüfung der Einreichung anhand eines der Graja-Testmodule spezifiziert.[37]

Im ProFormA-Aufgabenformat werden Bewertungsaspekte in einem Bewertungsschema, den sogenannten *grading-hints* definiert.[38] Das `grading-hints` XML-Element erlaubt es, Tests, Gruppen von Tests und Gruppen von Gruppen in einer Baumstruktur zu spezifizieren. Somit ist es möglich, Bewertungsaspekte hierarchisch zu gruppieren. Der Wurzelknoten wird hier über das XML-Element `root` definiert, der entweder weitere Tests durch das `test-ref` Element oder Gruppen über das `combine-ref` Element referenziert. Gruppen können mithilfe des `combine` Elements erstellt werden. Tests verweisen mit dem `ref` Attribut auf ein Graja-Testmodul, das einen Test ausführt. Das `sub-ref` Attribut erlaubt es, ein Unterergebnis eines Tests abzugreifen[39] (z.B. eine in JUnit mit `@Test` annotierte Methode oder eine PMD/Checkstyle-Regel).

[31] [Gar16b] [S. 10]
[32] `https://docs.oracle.com/javase/8/docs/technotes/guides/security/index.html`
[33] [Garb][3.1.1.9]
[34] `https://docs.oracle.com/javase/tutorial/reflect/index.html`
[35] `https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html#exec(java.lang.String)`
[36] [Gar16b] [S. 5]
[37] [Gar16a] [S. 4]
[38] [MGF+07] [4]
[39] [MGF+07] [4.3]

```xml
<p:grading-hints>
  <p:root>
   <p:description>This is an assignment!</p:description>
   <p:test-ref ref="compile">
    <p:title>Should successfully compile</p:title>
   </p:test-ref>
   <p:combine-ref ref="functionality"></p:combine-ref>
   <p:combine-ref ref="other"></p:combine-ref>
  </p:root>
  <p:combine id="functionality">
   <p:title>Functional correctness</p:title>
   <p:test-ref ref="junit" sub-ref="Grader#methodA">
    <p:title>Method A works accordingly!</p:title>
   </p:test-ref>
  <!-- more references ... -->
  </p:combine>
  <p:combine id="furtherGradingAspects">
    <p:title>Further grading aspects</p:title>
   <p:combine-ref ref="codingStyle"></p:combine-ref>
   <p:combine-ref ref="maintainability"></p:combine-ref>
  </p:combine>
  <p:combine id="codingStyle">
   <p:title>Coding style</p:title>
   <p:test-ref ref="checkstyle" sub-ref="Indentation">
    <p:title>Should use indentation consistently</p:title>
   </p:test-ref>
   <!-- more references ... -->
  </p:combine>
  <p:combine id="maintainability">
   <p:title>Maintainability</p:title>
   <p:test-ref ref="pmd" sub-ref="CommentRequired">
    <p:title>Needs to be commented</p:title>
   </p:test-ref>
   <!-- more references ... -->
  </p:combine>
</p:grading-hints>
```

Listing 1: Stark vereinfachtes Beispiel einer Konfiguration eines Bewertungsschemas im ProFormA-Aufgabenformat

Das oben gezeigte Beispiel verdeutlicht die Konfiguration eines Bewertungsschemas durch das `grading-hints` XML-Element des ProFormA-Aufgabenformats. Es ist anzumerken, dass dies keine valide Konfiguration darstellt. Aggregationsfunktion und die Gewichtung von ProFormA-Tests und Gruppen werden der Übersicht halber weggelassen. Die Beziehung zwischen den Testreferenzen und Gruppen ist im späteren Verlauf der Arbeit von Relevanz. Ebenso spielt die hier eingeführte Baumstruktur des Bewertungsschemas in Kapitel 5 eine Rolle.

2.4 Feedback

Feedback[40] stellt das Ergebnis eines Bewertungsprozesses dar und kann entweder als einfacher Text oder *Hypertext Markup Language* (HTML)-Dokument ausgegeben werden. Das Feedback fungiert als Statusbericht. Es enthält sowohl das ausgefüllte Bewertungsschema mit Informationen über die vom Autor definierten Bewertungsaspekte (Titel und erreichte Punkte der ProFormA-Tests) als auch durch Graja generierte Kommentare, die entweder interne für Lehrpersonal relevante Graja-Informationen loggen oder Studierenden eine Hilfestellung bei nicht bestandenen Teilaufgaben anbieten. Ein Kommentar kann in diesem Zusammenhang aus Überschriften, strukturiertem Text, Tabellen, Listen, Codeausschnitten oder Grafiken bestehen. Kommentare werden für zwei Zielgruppen generiert: Kommentare für Studierende (*S-Feedback*) und Kommentare für Lehrpersonen (*T-Feedback*). Studierende können keinen Einblick in *T-Feedback* erhalten. *S-Feedback* stellt nach Graja-Konventionen eine vereinfachte Version entdeckter Mängel dar und vermeidet so z.B. komplette *Stacktraces*, da diese sicherheitssensitive Informationen enthalten können. *T-Feedback* kann außerdem Protokolle der ausgeführten Testmodule oder Graja selbst enthalten.[41] Kommentare können sowohl von Graja, als auch aus dem Grader-Code der Aufgabe stammen. Es gibt momentan keinen Mechanismus um die eindeutige Herkunft eines Kommentars herauszufinden. Des weiteren ist jedem Kommentar eine Priorität zugeordnet, die das Filtern von Kommentaren nach Zielgruppen erlauben. Mithilfe der Zielgruppen und der Priorität können verschiedene Dokumente mit unterschiedlich detailliertem Informationsgrad generiert werden. [42]

Die Generierung von Feedback wird über das Graja-spezifische Domänenobjekt *RDKindTO* gesteuert.[43]. Ein *RDKindTO*-Objekt enthält ein Level, das als Priorität fungiert. Außerdem wird eine Audienz und Repräsentation konfiguriert. Wie bereits beschrieben gibt es bei der Audienz die Möglichkeit *T-Feedback* oder *S-Feedback* zu generieren. Es können mehrere *RDKindTO*-Objekte pro Bewertungsdurchlauf übergeben und somit mehrere Dokumente für verschiedene Zielgruppen erstellt werden. Ein Level dient als untere Grenze, somit werden nur Kommentare mit einem höheren oder gleichem Level in das generierte Feedback mit übernommen.

RD1: RDKindTO	RD2: RDKindTO
+ level: info	+ level: severe
+ audience: student	+ audience: teacher
+ representation: html	+ representation: html

Abbildung 1: Zwei *RDKindTO*-Domänenobjekte mit verschiedenen Einstellungen

Die in der oberen Abbildung dargestellten *RDKindTO*-Domänenobjekte werden zu zwei HTML-Feedback-Dokumenten führen. Das eine ist für Studierende gedacht und das andere für das Lehrpersonal. Beide unterscheiden sich dadurch, dass das Dokument für Studierende Kommentare mit dem Level *INFO* und höher und der Audienz *STUDENT* oder *BOTH* enthält, während das Dokument für das Lehrpersonal nur Kommentare mit

[40] [Gar16b] [S. 5-6]
[41] [Gar16b] [S. 5-6]
[42] [Gara] [5.2]
[43] [Garb] [3.1.1.33]

mindestens dem Level *SEVERE* und der Audienz *TEACHER* und *BOTH* enthält. Das *T-Feedback* stellt in der gezeigten Konfiguration also eine deutlich verknappte Form, die sich auf Fehlermeldungen beschränkt, dar.

2.5 Aufgaben

Eine Aufgabe ist eine Konfiguration von einem oder mehreren Testmodulen und einem Bewertungsschema.[44] Graja unterstützt neben dem ProFormA-Aufgabenformat[45] auch noch das sogenannte „plain old assignments“-Format. Das „plain old assignments“-Format gilt allerdings als überholt und wird deswegen in dieser Arbeit nicht weiter behandelt.[46]

2.5.1 Aufbau von Aufgaben

Kernstück einer Aufgabe ist die im ProFormA-Aufgabenformat definierte task.xml Datei[47]. Das komplette Domänen-Modell der task.xml ist für diese Arbeit nicht ausschlaggebend und wird nicht weiter erläutert. Es ist anzumerken, dass die task.xml aus Sicht des Aufgabenautors einige Makros enthält, die während der Überführung in das ProFormA-Aufgabenformat ausgeführt und ersetzt werden.[48] Die Definition eines Bewertungsschemas anhand des `grading-hints` XML-Element ist bereits in Kapitel 2.3 behandelt.

2.5.2 Musterlösungen

Musterlösungen werden über das `model-solutions` XML-Element eingebunden und sind ein verpflichtender Teil des ProFormA-Aufgabenformats. Über `model-solution` XML-Elemente können ein oder mehrere Musterlösungen einer Aufgabe beigelegt werden. Es ist momentan nicht möglich, eine Musterlösung als korrekt zu markieren.[49] Musterlösungen, die sich aus mehreren Dateien zusammensetzen, werden von Graja als ZIP-Datei in einer überführten Aufgabe erwartet.[50] Das Einbetten von Musterlösungen im Anhang der task.zip erfolgt über Dateireferenzen in der task.xml mithilfe von `file` XML-Elementen (Listing 2).

```
1 <p:files>
2  <p:file id="ms1" used-by-grader="false" visible="no">
3   <p:attached-bin-file>Hello.java</p:attached-bin-file>
4  </p:file>
5  <p:file id="ms2" used-by-grader="false" visible="no">
6   <p:attached-bin-file>Hello.java</p:attached-bin-file>
7  </p:file>
8 </p:files>
```

Listing 2: Beispiel für in task.zip eingebettete Musterlösungen anhand der task.xml des ProFormA-Aufgabenformats

[44] [Garb] [2]
[45] [Gar16a] [S. 2]
[46] [Garb] [3.1.1.29]
[47] [Garb] [3.1.1.1]
[48] [Gara] [1.2]
[49] [MGF+07] [5.8]
[50] [Garb] [3.1.1.13]

Durch das `model-solution` XML-Element können dann eine oder mehrere Musterlösungen definiert werden, die auf die mit dem `files` XML-Element eingebetteten Dateien zeigen (Listing 3).

```
1  <p:model-solutions>
2   <p:model-solution id="correct">
3    <p:filerefs>
4     <p:fileref refid="ms1"></p:fileref>
5    </p:filerefs>
6     <p:internal-description>This is the sample solution.</p:
        internal-description>
7    </p:model-solution>
8    <p:model-solution id="wrong_compileError">
9     <p:filerefs>
10     <p:fileref refid="ms2"></p:fileref>
11    </p:filerefs>
12  </p:model-solution>
13 </p:model-solution>
```

Listing 3: Beispiel für Musterlösungen in der task.xml, die auf die in Listing 2 deklarierten Dateien zugreifen

2.5.3 Variable Aufgaben

Variable Aufgaben stellen einen besonderen Typen der Aufgabenkonfiguration dar. Es wird unterschieden zwischen einem Template, also einer Schablone, die an gewissen Stellen variable Werte zulässt, und einer Instanz dieser Schablone. Nur eine Instanz erlaubt die Bewertung studentischer Einreichungen, da diese konkrete Werte in den Lücken der Schablone aufweist.[51]

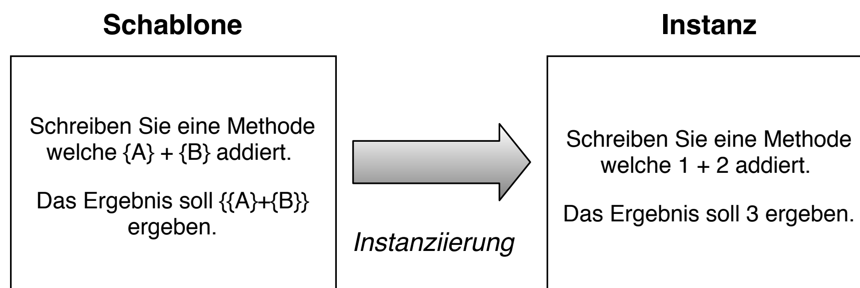


Abbildung 2: Unterschiede zwischen variablen Aufgaben: Schablone und Instanz

Der Vorgang aus einer Schablone eine Instanz zu erstellen ist als Instanziierung definiert und wird in Abbildung 2 veranschaulicht. Momentan findet eine solche Instanziierung standardmäßig im Kapitel 2.5.4 behandelten Überführungsprozess statt.

2.5.4 Überführung in das ProFormA-Aufgabenformat

Graja-Aufgaben stellen ein kleines Softwareprojekt dar. Durch ein Gradle-*Buildscript* wird ein solches Projekt in das ProFormA-Aufgabenformat überführt[52] (Abbildung 3).

[51] [Garb] [5]
[52] [Gar16b] [S. 13-14]

Makros der task.xml Datei werden während des *Build*-Prozesses aufgelöst. Erst nachdem alle Graja-spezifischen Makros verarbeitet worden sind, stellt die task.xml eine valide ProFormA-Aufgabenkonfiguration dar. Als Ergebnis entsteht eine ProFormA-kompatible ZIP-Datei, die sogenannte task.zip[53], die Graja nun erlaubt, studentische Einreichungen zu bewerten. Bei variablen Aufgaben wird derzeit sowohl eine Schablone als auch eine konkrete Instanz der Aufgabe mit Standardwerten generiert. Diese instanziierte Aufgabe ist durch den Dateinamen erkenntlich, statt dem `<identifier>.zip` Namensschema wird `<identifier>.dis.zip` genutzt.[54] Eine task.zip enthält neben einigen notwendigen Dateien wie z.B. der Checkstyle-Konfiguration auch die mit der Aufgabe assoziierten Musterlösungen als Anhang. Der konkrete Überführungsschritt ist in Graja durch die Funktion `buildProforma` der Klasse `BuildProforma` im Paket `de.hsh.graja.devasgmt` implementiert. Das Gradle-*Buildscript* ruft diese Funktion während des *Build*-Prozesses mit den aufgabenrelevanten Pfaden auf.

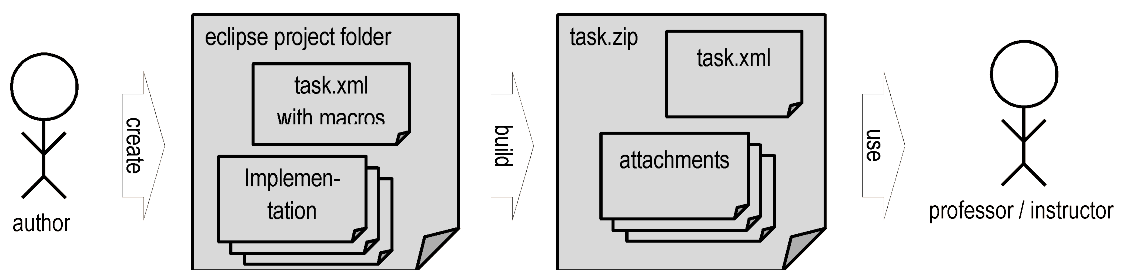


Abbildung 3: Überführungsprozess in das ProFormA-Aufgabenformat[55]

2.6 Ausführen von Graja

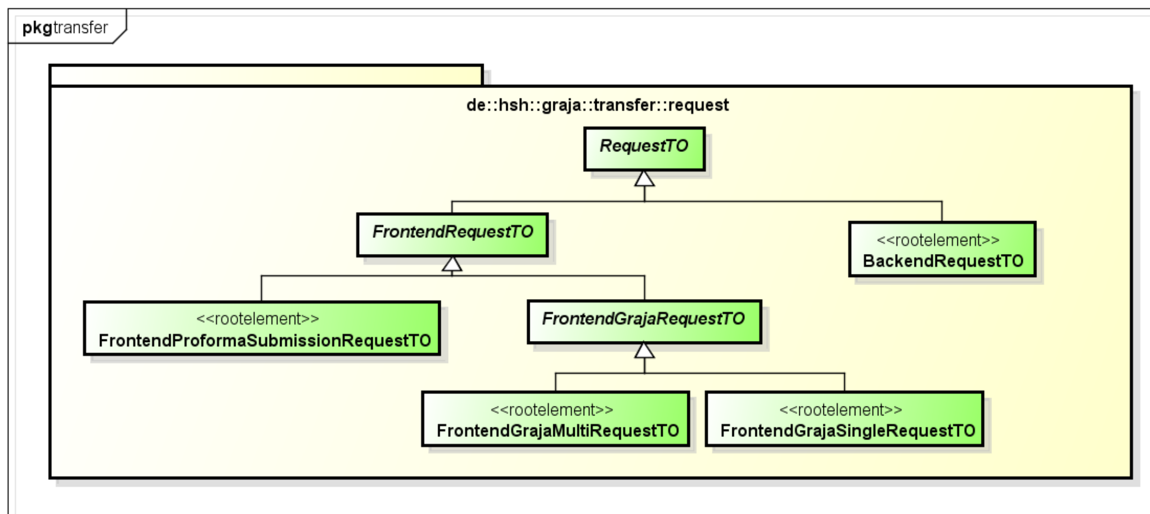


Abbildung 4: Hierarchie der Graja-Requests[56]

[53] [Gara] [1]
[54] [Garb] [5.2]
[55] Bildquelle: [Gara] [1]
[56] Bildquelle: [Garb] [3.1.1.29]

Graja führt *Requests* (Anfragen), die in Abbildung 4 dargestellt sind, aus und produziert *Results* (Ergebnisse).[57] Bei den *Requests* wird zwischen *Frontend-* und *Backendrequest* unterschieden. Es ist außerdem möglich, mithilfe von *Multi Requests* mehrere Abgaben innerhalb einer Graja-Sitzung zu bewerten. *Multi Requests* sind für diese Arbeit allerdings nicht relevant und werden daher nicht weiter behandelt. *Requests*, die ProFormA-Einreichungen [58] darstellen, werden anhand des *FrontendProformaSubmissionRequestTO* repräsentiert. Diese bestehen aus einer einzigen ZIP-Datei[59], die sowohl die Aufgabe als auch die Einreichung beinhaltet. Ein *FrontentGrajaSingleRequestTO* ist dem *FrontendProformaSubmissionRequestTO* ähnlich: Hier liegt die Einreichung allerdings schon in entpackter Form vor, während die Aufgabe als ProFormA-kompatible ZIP-Datei referenziert wird. *Frontend-* und *Backendrequests* unterscheiden sich noch durch bestimmte Merkmale. Ein *Backendrequest* kann nur eine einzige Aufgabe behandeln. Alle relevanten Ressourcen sind bei einem *Backendrequest* außerdem entpackt und existieren auf dem lokalen Dateisystem. Eine Einreichung, die aus mehreren Dateien besteht, ist entpackt und eine dazugehörige task.zip in deren einzelne Bestandteile aufgebrochen. Um einen *Request* auszuführen stehen die in den folgenden Unterkapiteln behandelten zwei Möglichkeiten zur Verfügung.

2.6.1 Ausführung durch Graja-CLI

Das Modul `GrajaCli` bietet eine Befehlszeilenschnittstelle (*Command Line Interface* (CLI)) an. Über diese Schnittstelle kann der Bewertungsvorgang für sowohl *Frontend-* als auch *Backendrequests* gestartet werden.[60] Für Graja-spezifische *Requests* muss hier erst das Dateiformat des *Request* (XML oder JSON) angegeben werden. Danach kann ein Dateipfad zum *Request* und *Result* angegeben werden. ProForma-*Requests* benötigen nur einen Dateipfad zur ZIP-Datei der Anfrage und zur Speicherung des *Result*. Das direkte Ausführen eines *Backendrequest* durch `GrajaCli` garantiert nicht die Einhaltung aller sicherheitsrelevanten Beschränkungen wie z.B. den der Begrenzung des zur Verfügung stehenden Arbeitsspeichers einer Einreichung, da dies nur das Graja-Frontend garantiert.

2.6.2 Ausführung durch Java-API

Graja lässt sich auch programmatisch aus einer bereits gestarteten JVM ausführen[61]. Die Klasse `WithinJVMStarter` im Paket `de.hsh.graja.starter.apcli` bietet dafür ein *Application Programming Interface* (API) mit der Funktion `startGraja` an. Diese Funktion akzeptiert *FrontendRequestTO*-Objekte und garantiert die Einhaltung aller sicherheitsrelevanten Beschränkungen durch das Starten einer zweiten JVM mit speicherlimitierenden Parametern und einen automatischen Abbruch nach einer Zeitüberschreitung der Einreichung. Um einen *Backendrequest* zu bewerten, bietet sich die Funktion `gradeBackendRequest` in der Klasse `WithinJVMBackendAPI` des Pakets `de.hsh.graja.core.apcli` an. Diese Funktion erwartet *BackendRequestTO*-Objekte. Das Aufrufen über diese Schnittstelle garantiert, dass der Bewertungsvorgang in der gleichen JVM des Aufrufers stattfindet. Aufgrund des fehlenden *Frontend*-Durchlaufs können nicht alle sicherheitsrelevanten Restriktionen erzwungen werden. Durch die

[57] [Garb] [3]
[58] [MGF+07] [7]
[59] [Garb] [3.1.1.29]
[60] [Garb] [1.2]
[61] [Garb] [1.3]

gleiche beim Bewertungsvorgang verwendete JVM ist `WithinJVMBackendAPI` vor allem für die Fehlersuche nützlich, sollte aber aufgrund der Sicherheitsaspekte (fehlendes Frontend) nicht in einer Produktionsumgebung verwendet werden.

2.6.3 Interne Ausführungsphasen

Innerhalb von Graja gibt es drei ausschlaggebende Ausführungsphasen, die in Abbildung 5 vereinfacht dargestellt sind. Bei *Frontendrequests* findet erst eine Vorverarbeitung durch das Graja-Frontend statt. Der *Frontendrequest* wird in einen *Backendrequest* umgewandelt. In dieser Vorverarbeitungsphase wird auch sichergestellt, dass alle für die Bewertung notwendigen Dateien existieren und über einen *Backendrequest* referenzierbar sind.

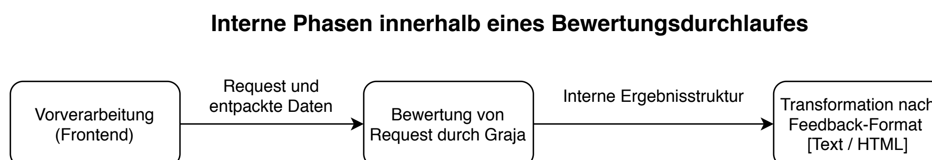


Abbildung 5: Interne Graja-Ausführungsphasen vor und während eines Bewertungsvorgangs

Danach wird ein *Backendrequest* von Graja bewertet. Dies erfolgt, indem der *Backendrequest* in einen `de.hsh.graja.request.Request` umgewandelt wird, der mithilfe der Klasse `Core` bewertet werden kann. In diesem Schritt wird die studentische Einreichung kompiliert und durch die in Kapitel 2.1 vorgestellten Testmodule überprüft. Während des Bewertungsvorgangs wird eine interne Ergebnisstruktur, die das nach der Bewertung ausgefüllte Bewertungsschema enthält, erstellt. Über diese Ergebnisstruktur werden in der letzten Ausführungsphase die in Kapitel 2.4 behandelten Feedback-Dokumente generiert und in die von Graja erstellte *Response* geschrieben. Im Falle der direkten Ausführung eines *Backendrequests* fällt der erste Schritt der Vorverarbeitung mithilfe des Graja-Frontends weg.

2.7 Regressionen und Regressionstests

Regressionen stellen in der Softwareentwicklung einen Fehlertypen dar, der sich dadurch auszeichnet, dass Komponenten der Software nach Änderungen an der Software nicht mehr wie ursprünglich geplant funktionieren.[62][63] Regressionsfehler werden zumeist unbewusst ausgelöst und sind ohne automatisierte Tests schwierig zu detektieren. Um solche Regressionen zu erkennen, wird die Software sogenannten Regressionstests unterzogen.

2.7.1 Abgrenzung zu Modul- und Integrationstests

Modultests (*unit tests*) werden auf die kleinste adressierbare Einheit (Klasse, Funktion) einer Software angewandt.[64] Während eines Modultests werden möglicherweise auch andere komplexe externe Module aufgerufen. Diese werden während eines Modultests durch sogenannte *stubs* emuliert[65]. *Stubs* stellen Platzhalter-Module dar, die

[62] [NTY][S. 218 - 219]
[63] [Cha10][S. 255]
[64] [Cha10][S. 80-81]
[65] [Cha10][S. 215-216]

eine vereinfachte Version des realexistierenden Moduls abbilden. Ohne *stubs* ließen sich Modultests, die auf externe Ressourcen, wie z.B. eine Datenbank zugreifen, in einer simulierten Testumgebung schwer realisieren. In Graja existieren einige Modultests, die mithilfe von JUnit realisiert sind.

Integrationstests stehen eine Stufe über den Modultests. Das Ziel eines Integrationstests ist es, sicherzustellen, dass die Kommunikation zwischen den verschiedenen Modulen über die angebotenen Schnittstellen fehlerfrei abläuft.[66] Externe Module, die bei Modultests durch *stubs* modelliert werden, sind bei Integrationstests in der vollständigen Form vorhanden.

Regressionstests hingegen existieren um das gesamte Softwaresystem nach Änderungen auf die korrekte Funktionalität zu überprüfen. Eine Regression tritt hier auf, falls eine modifizierte Komponente ein ungewollt anderes Verhalten aufweist oder neueingeführte Komponenten zu unerwarteten Seiteneffekten in nichtmodifizierten Komponenten führen.[67]

Im *IEEE Standard Glossary of Software Engineering Terminology* ist der Regressionstest als: *„Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.“* definiert.[68]

2.7.2 Testumgebungen für Regressionstests

Regressionstests werden durch ein *Regression Testing System* ausgeführt. Dieses System führt die Menge der existierenden Testfälle für ein Projekt auf regulärer Basis aus und hilft so dabei, unerwünschte Fehler durch Änderungen an Softwarekomponenten zu detektieren.[69] Das Ergebnis solcher Regressionstests sollte in regelmäßigen Abständen von den an der Entwicklung beteiligten Teilnehmern gesichtet werden. Falls ein Test fehlschlägt, muss entschieden werden, ob es sich um eine Regression oder eine erwünschte Änderung handelt. Im Falle einer erwünschten Änderung müssen die fehlschlagenden Tests abgeändert werden.[70] Regressionstests sollen nach jeder Aktualisierung des Softwaresystems durchgeführt werden.[71] Ein automatisiertes *Regression Testing System* kann die Tests z.B. automatisiert innerhalb der *Build*-Phase der Software ausführen.

Folgende Informationen sollten von einem *Regression Testing System* aufgezeichnet werden.[72]

- Menge der ausgeführten Testfälle
- Anzahl der ausgeführten, fehlschlagenden und erfolgreichen Regressionstests
- Detaillierte Informationen über den fehlschlagenden Testfall, die bei der Lokalisierung der entdeckter Regressionen unterstützen

[66] [Cha10][S. 218-219]
[67] [Cha10][S. 255-256]
[68] [IEE90] [S. 61]
[69] [HK07] [S. 73]
[70] [HK07] [S. 68]
[71] [HK07] [S. 131]
[72] [HK07] [S. 278-279]

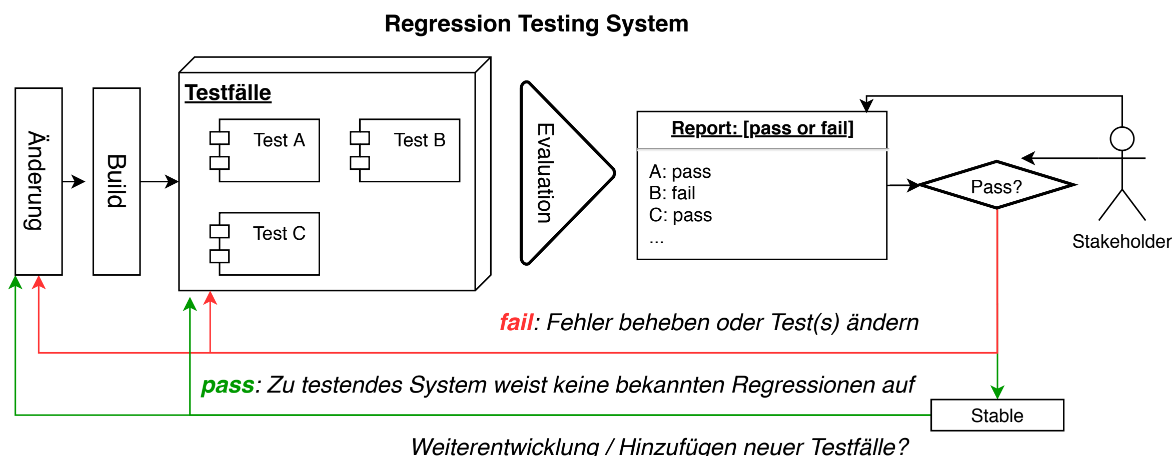


Abbildung 6: Möglicher Aufbau einer Regressionstest-Pipeline (eigene Darstellung)

In Abbildung 6 ist eine mögliche Regressionstest-Pipeline eines *Regression Testing Systems*, das auf dem Konzept *Continuous Integration* (CI)[73] aufbaut, grafisch dargestellt. Eine Änderung an einer Komponente des Softwaresystems startet den *Build*-Prozess. Innerhalb des *Build*-Prozesses wird auf bereits existierende Testfälle zurückgegriffen, um mögliche, durch die Änderung ausgelöste Regressionen zu erkennen. Die geänderte Version der Software wird anschließend diesen Testfällen unterzogen. Mithilfe der Ergebnisse der Testausführung wird schlussendlich ein Bericht generiert. Erkannte Regressionen stellen Fehler dar, die den *Build*-Prozess fehlschlagen lassen. *Stakeholder* wie Entwickler, Projektmanager oder *Quality Assurance* (QA) *Engineers* können bei solchen Fehlern auf den Bericht zurückgreifen, um fehlschlagende Regressionstests zu identifizieren. Bei einem Fehlalarm durch z.B. gewünschte Änderungen müssen die Tests möglicherweise abgeändert oder entfernt werden. Bei unerwünschten Regressionen wird solange eine Fehlersuche und Fehlerbehandlung durchgeführt, bis keiner der Tests mehr fehlschlägt. Falls für neue Komponenten keine Testfälle existieren, müssen diese ergänzt werden, um die Erkennung möglicher Regressionen innerhalb dieser Komponenten durch zukünftige Änderungen zu garantieren.

2.8 Problemstellung

Graja ist ein komplexes modulares Softwaresystem, das momentan aus 31 Modulen besteht. Lediglich drei dieser 31 Module stützen sich im *Build*-Prozess auf Modultests durch JUnit (Listing 4), Integrationstests existieren soweit keine. Es existieren also keine Modultests für einen Großteil der Graja-Module und somit keine Möglichkeit, eine automatisierte Überprüfung des korrekten Bewertungsverhaltens nach Änderungen an Graja selber oder dem Grader-Code einer Aufgabe durchzuführen.

```
1 > find . -path "*src/test"
2
3 ./GrajaModCheckstyle/src/test
4 ./GrajaTransform/src/test
5 ./libvts/src/test
```

Listing 4: Graja-Module, die Modultests im Build-Prozess ausführen

[73] [MSVD07]

Eine Analyse mit dem Quellcode-Analysewerkzeug *cloc*[74] hat außerdem ergeben, dass alle Module zusammen mit eventuellen Modultests ca. 60000 Zeilen Java-Quellcode (Metrik: SLOC) akkumulieren (Listing 5). Das Ergänzen und Warten von fehlenden Modultests, mit dem Ziel eine hundertprozentige Testabdeckung anzustreben, stellt ein zeitaufwändiges Unterfangen dar. Eine hohe Testabdeckung garantiert außerdem keine Fehlerfreiheit und ist in komplexen Systemen schwer zu erreichen.[75] Änderungen innerhalb eines dieser Module kann möglicherweise zu unerwünschten Regressionen an anderen Stellen führen. Momentan gibt es keine automatisierte Möglichkeiten, solche Regressionen zu erkennen, was die Weiterentwicklung der Software verkompliziert. Demnach existiert stets potenzielle Ungewissheit, ob der Bewertungsprozess nach einer Änderung/Ergänzung eines Moduls noch korrekt für alle bisher existierenden Aufgaben durchgeführt wird.

```
1 > cloc $(find . -maxdepth 1 -name "Graja*" | tr '\n' ' ' | awk
     '{print $0" ./libvts ./Proforma"}')
2
3 -------------------------------------------
4 Language   files   blank    comment   code
5 -------------------------------------------
6 Java         905   14160    26605     59827
```

Listing 5: SLOC-Metrik aller derzeit existierenden 31 Graja-Module

Eine Regression kann z.B. durch das Einbinden eines neuen Testmoduls (Kapitel 2.1) oder Refaktorisierungen innerhalb der Grader-API entstehen. Auch eine Erweiterung der Sicherheitsarchitektur um z.B. eine Begrenzung des Festplattenspeichers unter Windows zu ermöglichen, kann zu unvorhersehbaren Regressionen führen. Solche Fehler können ohne Regressionstests für lange Zeit unentdeckt bleiben. Besonders, falls ein Teil der Grader-API betroffen ist, der nur von wenigen Aufgaben genutzt wird. Musterlösungen[76], die Teil des ProFormA-Aufgabenformat sind, können zum Überprüfen des korrekten Bewertungsverhaltens einer Aufgabe genutzt werden. Momentan kann das korrekte Bewertungsverhalten von Aufgaben anhand von Musterlösungen lediglich durch manuelles Testen über die Graja-GUI geprüft werden.[77] Dieser Vorgang ist zurzeit nicht automatisiert und daher zeitaufwändig, da sowohl die richtigen Aufgaben, als auch Musterlösungen ausgewählt und ausgeführt werden müssen. Ein Vorher/Nachher-Vergleich des von Graja generierten Feedbacks, mit dem Ziel mögliche Unterschiede nach Änderungen zu erkennen, müsste außerdem mühsam für jede Aufgaben-Musterlösung-Kombination durch einen Tester durchgeführt werden.

2.8.1 Musterlösungen in Graja als Mechanismus für Regressionstests

Aufgrund der in Kapitel 2.8 beschriebenen Situation bezüglich der Modul- und Integrationstests ist ein Regressionstestmechanismus eine sinnvolle Ergänzung für Graja. Momentan existieren im Aufgaben-Repositorium[78] 57 Aufgaben mit insgesamt 437 Musterlösungen (Listing 6). Alle Aufgaben und Musterlösungen manuell in der Hoffnung, Fehler nach Änderungen des Softwaresystems mithilfe von Graja-GUI zu entdecken,

[74] `http://cloc.sourceforge.net/`
[75] [OW10] [S. 118]
[76] [MGF+07] [5.8]
[77] [Gara] [5.1]
[78] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja-assignments`

stellt also einen erheblichen Arbeitsaufwand dar. Eine Möglichkeit des automatisierten Testens von Graja auf Regressionen anhand der vorhandenen Musterlösungen sollte die zukünftige Entwicklung von Graja vereinfachen. Änderungen können dann mit einer größeren Sicherheit, keine neuen Fehler einzuführen, in eine Produktionsumgebung übernommen werden.

```
1  > ./count_solutions.sh
2
3  All assignments: 57
4  All sample solutions: 437
```

Listing 6: Summe aller Aufgaben und Musterlösungen im Aufgaben-Repositorium, die als Testfälle für einen Regressionstestmechanismus dienen können[79]

Die Musterlösungen bilden bei diesem Ansatz in Verbindung mit den jeweiligen Aufgaben die Testfälle. Dies hat den Vorteil, dass schon eine Menge an Testfällen existiert, die ohne Vorverarbeitung übernommen werden kann. Es geschieht dann eine einmalige Aufzeichnung des Soll-Verhaltens einer Musterlösung, die nach Änderungen an Graja genauso durch ein beobachtetes Ist-Verhalten zu reproduzieren ist[80]. Die Regressionstests operieren somit über einen Verhaltensabgleich von $Soll = Ist$. Es kann also für Graja ein in Abbildung 6 ähnlicher Regressionstestmechanismus realisiert werden. Das aufgezeichnete Soll-Verhalten dient in diesem Fall als Testfall. Der Regressionstestmechanismus kann dann entweder in das Graja oder Graja-Aufgaben-*Buildscript* integriert werden. Bei einem fehlschlagenden Abgleich wird ein Testbericht mit gefundenen Differenzen generiert, der detailliert Informationen bezüglich der potenziell gefundenen Regressionen enthält.

Ein mögliches Problem bei diesem Ansatz stellt unter anderem die Genauigkeit der Aufzeichnung und der Aufbau der Test-Infrastruktur dar. Aufzeichnungen enthalten möglicherweise Abschnitte wie Zeitangaben oder Dateipfade, die zu nicht-reproduzierbarem Verhalten führen. Ein Werkzeug für Graja-Regressionstests soll sich außerdem möglichst nichtinvasiv in Graja integrieren, um die Weiterentwicklung von Graja nicht durch den Mechanismus selber zu erschweren. Bei der Test-Infrastruktur ist außerdem zu untersuchen, wie aufwändig das Ausführen aller Tests ist und ob zusätzliche Metadaten, wie eine Testabdeckung, generiert werden können. Eine Test-Infrastruktur für Graja-Regressionstests soll zudem möglichst einfach aufsetzbar und nutzbar sein. Das automatische Ausführen aller Testfälle, falls vorhanden, kann z.B. in den *Build*-Prozess von Graja eingebunden werden und somit Vorrausetzungen für eine Graja *CI Pipeline* schaffen.

Thematisch ist diese Arbeit in die Bereiche der Softwareentwicklung, Softwaretests und Regressionstests einzuordnen. Im nächsten Kapitel wird die Problemstellung noch einmal detailliert aufgegriffen und es werden sowohl konkrete Forschungsfragen gestellt als auch Nicht-Ziele definiert.

[79] Das verwendete Skript `count_solutions.sh` hängt im elektronischen Anhang an.

[80] Es sei denn, eine grundlegende Funktionsweise ***soll*** so geändert werden, dass diese nicht mehr mit dem aufgezeichnetem Verhalten übereinstimmt. In dem Fall muss natürlich eine Neuaufzeichnung erfolgen, um das neuartige Verhalten als Soll-Verhalten zu definieren.

3 Ziele

Nachdem die Problemstellung aus der Einleitung am Ende des vorherigen Kapitels näher erläutert wurde, gilt es nun, einen wünschenswerten erreichbaren Zustand zu definieren. Idealerweise ist es am Ende dieser Arbeit möglich, durch eine Test-Infrastruktur automatisiert das Verhalten aller Graja-Aufgaben und der dazugehörigen Musterlösungen durch die Ausführung von Graja aufzuzeichnen. Das aufgezeichnete Verhalten stellt dann Testfälle für Regressionstests dar. Ein Abgleich dieser Testfälle (Soll-Verhalten) mit des zum Zeitpunkt der Testausführung beobachteten Verhaltens (Ist-Verhalten) kann dann in den Graja-*Build*-Prozess integriert werden. Somit können potenzielle Regressionen nach Änderungen an Graja-Modulen erkannt werden. Die Test-Infrastruktur soll dem Nutzer auch erlauben eine Neuaufzeichnung des Verhaltens durchzuführen. Dementsprechend können Sonderfälle behandelt werden, bei denen potenzielle Regressionen erwünschtes Verhalten darstellen, indem das Ist-Verhalten zum neuen Soll-Verhalten promoviert wird. Dadurch entsteht ein Regressionstestmechanismus, der unerwünschte Seiteneffekte durch Änderungen an Graja-Modulen entdeckt, bevor diese in eine Produktionsumgebung übernommen werden und somit möglicherweise den Lehrbetrieb stören.

3.1 Forschungsfragen

Um diese Ziele zu erreichen, müssen einige Forschungsfragen gestellt werden, die es in dieser Arbeit zu beantworten gilt. Anhand der Referenzimplementierung geschieht anschließend die Umsetzung der theoretischen Ergebnisse in die Praxis.

Inwiefern können Musterlösungen zum Regressionstest von Graja eingesetzt werden?

Es gilt im Detail zu klären, inwiefern sich Musterlösungen als Testfälle eignen und somit einen Regressionstest von Graja ermöglichen können. Außerdem muss untersucht werden, wie mit variablen Aufgaben umgegangen werden kann und ob es möglich ist, dass Musterlösungen existieren, die zu nicht reproduzierbarem Verhalten führen.

Inwiefern kann eine robuste Aufzeichnung des Bewertungsverhaltens geschehen?

Von Graja produzierte Ausgaben können Variablen enthalten, die für das Verifizieren der korrekten Graja-Ausführung nicht relevant sind. Unter solche Variablen fallen z.B. Zeitstempel oder Dateipfade. Diese sind vom Zeitpunkt der Ausführung oder des ausführenden Systems abhängig und erlauben so keinen Rückschluss auf potenzielle Regressionen, da deren Zustand sich nur in seltenen Fällen nicht verändert. Darunter fallen mitunter auch ausgegebene detaillierte Stacktraces im *T-Feedback*. Unter Java stützen sich I/O-Bezogene APIs wie z.B. im Paket `java.nio.file`[81] vorhanden auf systemspezifische Implementierungen von Dateisystemen. Eine von `java.nio.files` geworfene *Exception* kann also unter Linux zu einem anderen *Stacktrace* als unter Windows führen. Solche variablen Parameter stellen ein Rauschen in den Aufzeichnungen dar und können durch ihre Nicht-Reproduzierbarkeit für Fehlalarme sorgen. Eine robuste Aufzeichnung unterdrückt dieses Rauschen. Es gilt nun herauszufinden, welche Arten von Rauschen existieren und wie eine Rauschunterdrückung geschehen kann.

[81] `https://docs.oracle.com/javase/8/docs/api/java/nio/file/package-summary.html`

Inwieweit können Fehlalarme (false positives) beim Abgleich des Soll-Verhaltens mit dem Ist-Verhalten vermieden werden?

Es gilt, potenzielle Quellen von Fehlalarmen zu identifizieren und eine Strategie aufzustellen, mit diesen umzugehen. Fehlalarme können z.B. auf das Korrigieren eines Tippfehlers[82] zurückzuführen sein. Eine solche Korrektur soll keinen Anlass für eine Neuaufzeichnung geben. Fehlalarme können ebenfalls bei Aufgaben, die sich auf den Pseudozufallszahlengenerator der Klasse `Random`[83] beziehen auftauchen. Trotz des korrekt reproduzierten Bewertungsverhaltens können abweichende, zufallsabhängige Variablen in den Ausgaben der Aufgaben zu Fehlalarmen führen.

Inwiefern kann eine automatisierte Test-Infrastruktur um Graja aufgebaut werden?

Unter einer Test-Infrastruktur versteht sich die Aufzeichnung des Verhaltens der Musterlösungen, als auch das Ausführen der Regressionstests mithilfe der aufgezeichneten Testfälle. Es gilt zu klären, wie eine solche Infrastruktur um Graja aufgebaut werden kann und in den *Build*-Prozess der Aufgaben oder Graja selbst integriert wird.

Wie stark muss Graja verändert werden, um eine solche Infrastruktur zu ermöglichen?

Die Realisierung des Regressionstestmechanismus soll möglichst nichtinvasiv geschehen, um keine neuen Fehlerquellen einzubauen und die Weiterentwicklung von Graja nicht durch eine festverdrahtete Test-Infrastruktur zu erschweren.

Wie kann das Ergebnis eines Regressionstests verständlich dargestellt werden?

Falls eine potenzielle Regression erkannt wird, soll ein Graja-Entwickler über diese informiert werden. Es gilt ein Konzept zu entwickeln, um erkannte Regressionen übersichtlich darzustellen und nützliche Metadaten, die bei der Lokalisierung und Beseitigung dieser hilfreich sind, miteinzubeziehen.

3.2 Nicht-Ziele

Nicht-Ziele stellen Forschungsfragen dar, die in dieser Arbeit definitiv nicht behandelt werden. Das Formulieren von Nicht-Zielen erlaubt eine stärkere thematische Eingrenzung des zu erreichenden Zustands und lässt so einen geringeren Interpretationsspielraum zu.

Testen der Interaktion zwischen Graja und Lernmanagementsystemen

Ziel dieser Arbeit ist die Realisierung eines Regressionstestmechanismus für Graja mithilfe der Nutzung von Musterlösungen als Testfälle. Es geht darum, das korrekte Bewertungsverhalten Grajas zu überprüfen. Die Interaktion zwischen Graja und verschiedenen Lernmanagementsystemen fällt nicht darunter.

Entdeckung von Regressionen, die sich durch Bewertung einer beliebigen Musterlösung einer beliebigen Aufgabe entdecken lassen

[82]Zum Beispiel innerhalb des Titels eines Bewertungsaspekts einer Aufgabe.
[83][Orad]

In Kapitel 2.6.3 sind ahnhand Abbildung 5 die internen Graja-Ausführungsphasen illustriert. Ziel dieser Arbeit ist es, Regressionen innerhalb des Bewertungsvorgangs der Musterlösungen zu erkennen (Phase 2). Das Erkennen von Regressionen innerhalb des Feedback-Dokument generierenden Transformationsschritts (Phase 3) ist daher kein Ziel dieser Arbeit. Dafür reicht ein Testfall mit Beispielergebnissen eines Bewertungsvorgangs aus[84]. Für die Generierung dieser Beispieldaten muss nicht auf alle bereits vorhandene Musterlösungen und Aufgaben zurückgegriffen werden. Daher ist es auch kein Ziel, detailliert auf Regressionen innerhalb des Graja-Frontends (Phase 1) einzugehen. Es ist allerdings davon auszugehen, dass Regressionen im Frontend, welche die Ausführung des Bewertungsvorgangs in Phase 2 verhindern, durch die Tatsache, dass eine Bewertung nicht stattfindet, erkannt werden können. Die Lokalisierung einer solchen Regression im Frontend ist allerdings ebenfalls kein Ziel dieser Arbeit.

Entwicklung eines performanten Regressionstestmechanismus

Ziel der Arbeit ist das Konzept und die Referenzimplementierung eines Graja-Regressionstestmechanismus zu entwickeln. Performanzoptimierungen, welche die Zeit der Abarbeitung aller Regressionstests auf ein Minimum reduzieren, können später auf der Referenzimplementierung aufbauen. Solche Optimierungen erfordern möglicherweise tieferes Eingreifen in Graja selbst und stehen daher im direkten Konflikt mit einem nichtinvasiven Vorgehen.

Genaue Lokalisierung der Fehlerquellen innerhalb des Quellcodes für alle entdeckten Regressionen

Ziel des zu erforschenden Konzepts ist es, zunächst potenzielle Regressionen zu erkennen. Eine genaue Lokalisierung einer Regression, die es erlaubt eine Quelldatei mit Zeilennummer abzugreifen ist, falls nicht trivialerweise lösbar, kein Ziel dieser Arbeit.

Automatische Generierung von Musterlösungen

Musterlösungen müssen weiterhin von den Aufgabenautoren erstellt werden. Eine automatische Generierung von Testfällen stützt sich auf bestehende Musterlösungen, zieht aber die automatische Generierung von Musterlösungen für Aufgaben nicht mit ein.

Fuzzing mithilfe variabler Aufgaben

Variable Aufgaben enthalten, wie in Kapitel 2.5.3 beschrieben, Platzhalter, die nach einer Instanziierung mit konkreten Werten gefüllt sind. Beim Überführungsschritt in das ProFormA-Aufgabenformat, der in Kapitel 2.5.4 beschrieben ist, wird jeweils eine variable Aufgabe mit Standardwerten instanziiert. *Fuzzing*[85] stellt ein Testverfahren für Software dar, bei dem der zu testenden Software unerwartete Eingaben gesendet werden. Es ist in dieser Arbeit kein Ziel, Regressionen in Graja mithilfe von *Fuzzing* über zufällig gewählte Parameterwerte (im Definitionsbereich des Parameters) einer variablen Aufgabe zu entdecken.

[84] Ein Testfall für den Transformationsschritt: *Internes Result-Objekt => Feedback-Dokument* muss nur alle vorhandenen Kommentarbausteine enthalten. Der Inhalt dieser Kommentare ist für den Transformationsschritt irrelevant, da für ein Überprüfen von Phase 3 nur die korrekte HTML-Struktur wiedergegeben werden muss.

[85] [Sch]

4 Methodik

In diesem Kapitel wird die eingesetzte Methodik behandelt, die genutzt wird, um die vorher gestellten Forschungsfragen zu beantworten. Es handelt sich bei dieser Arbeit um eine empirische Arbeit zu einem bereits bestehenden Softwaresystem.

4.1 Literaturrecherche

Da Graja ein Hauptbestandteil dieser Arbeit ist, wird sich in der Literaturrecherche auf die Graja-Dokumentation und Graja-Forschungsberichte bezogen. Aufgrund des von Graja Genutzten ProFormA-Aufgabenformats, ist ebenfalls das ProFormA-Whitepaper miteinbezogen. Um einen Überblick innerhalb der Thematik der Regressions- und Softwaretests zu erhalten, wird sich diesbezüglich auf Fachliteratur gestützt. Eine Literaturrecherche ist ebenfalls im Kapitel der Lösungsmöglichkeiten angewandt, um bereits bestehende Algorithmen, Standards und Lösungsansätze miteinzubeziehen. Bei Java- oder Gradle-spezifischen Aspekten wird zusätzlich die Online-Dokumentation herangezogen.

4.2 Quellcodeanalyse

Da der Regressionstestmechanismus eine Erweiterung von Graja darstellt, müssen interne Graja-Aspekte mit berücksichtigt werden. Diese Aspekte werden anhand einer Quellcodeanalyse des Graja-Quellcodes[86] ermittelt. Der Graja-Quellcode ist zum Zeitpunkt der Arbeit nicht öffentlich einsehbar. Um dem Leser trotz geschlossenem Quellcode bestimmte Aspekte näherzubringen, werden z.B. UML-Diagramme erstellt und erläutert oder Beispiele für bestimmte Funktionsweisen und Softwarekomponenten gegeben.

4.3 Datenerhebung und Analyse

Um eine Aufzeichnung robust zu gestalten, muss Rauschen wie z.B. Dateipfade im Kommentarinhalt, welches bei einem Abgleich auf unterschiedlichen Betriebssystemen zu Fehlalarmen führen kann, unterdrückt werden. Im Rahmen der Identifikation und Klassifikation des Rauschens ist eine Datenerhebung geplant, welche durch das Ausführen von Musterlösungen zu verschiedenen Zeitpunkten unter verschiedenen Betriebssystemen und JDK-Implementationen entsteht. Der gewonnene Datensatz wird dann auf Differenzen untersucht. Die gefundenen Ergebnisse werden analysiert, um eine Klassifikation unterschiedlicher Rauschtypen aufzustellen, die in einer Verhaltensaufzeichnung eines Testfalls unterdrückt werden müssen.

4.4 Entwicklung einer Referenzimplementierung

Um die entwickelten Konzepte umzusetzen, wird ein Prototyp eines Graja-Regressionstestmechanismus in Form einer Referenzimplementierung entwickelt. Dieser Prototyp wird nach softwaretechnischen Prinzipien entwickelt und soll neben den gewonnenen Konzepten als Ausgangspunkt dieser Arbeit dienen.

[86] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja`

5 Lösungsmöglichkeiten

In diesem Kapitel der Arbeit werden verschiedene Lösungsansätze und Konzepte, die zur Beantwortung der genannten Forschungsfragen beitragen, anhand der Methodiken Quellcodeanalyse, Literaturrecherche und Datenerhebung ermittelt und bewertet. Teile der hier behandelten Konzepte basieren auf der im Kapitel 2 behandelten Thematik. Ziel dieses Kapitels ist es, den durch die Forschungsfragen geschaffenen Lösungsraum einzugrenzen und geeignete Verfahren für die Referenzimplementierung des Regressionstestmechanismus im nächsten Kapitel zu finden. Es wird jeweils ein Teilproblem, mit dem Ziel, eine geeignete Lösung zu finden, vorgestellt, analysiert und bewertet. Falls für dieses Teilproblem keine Lösung gefunden werden kann, wird nochmals detaillierter mit einem eigenen Ansatz darauf eingegangen.

Bei Schlussfolgerungen, die aus dem Graja-Quellcode gezogen werden, wird jeweils ein Verweis auf die Quelldatei gegeben. Zum Zeitpunkt der Arbeit befindet sich Graja in der Version 2.1.0. Der Graja-Quellcode ist momentan nicht öffentlich einsehbar und wird über einen Gitlab[87] Server der Hochschule Hannover [88] zur Verfügung gestellt.

5.1 Verhaltensaufzeichnung anhand von Musterlösungen

Aufgezeichnetes Verhalten stellt das von Graja nach einem Bewertungsdurchlauf generierte Feedback dar. Für diese Arbeit wird jeweils eine Musterlösung als alleinstehender Testfall angesehen und getrennt von anderen potenziellen Musterlösungen durch Graja bewertet. Das Feedback eines Bewertungsdurchlaufs stellt dann ein zu erwartendes Soll-Verhalten dar. Dieses sollte, sofern keine gewollten Änderungen am Soll-Verhalten vorgenommen wurden, durch ein beobachtetes Ist-Verhalten zum Zeitpunkt des Regressionstest reproduziert werden. Schlägt eine Reproduktion fehl und die Differenz ist ungewollt, ist davon auszugehen, dass es sich um eine erkannte Regression handelt. Es gilt in diesem Kapitel zu klären, wie eine Verhaltensaufzeichnung zustande kommen kann und welche Änderungen an Graja dafür notwendig sind.

5.1.1 Integration in den Bewertungsvorgang

Ein Aufruf des Bewertungsvorgangs erfolgt über die Funktion `gradeRequest` der Klasse `Core`. Das direkte Aufrufen von `gradeRequest` geschieht in den Klassen `BackendCommandProvider` und `WithinJVMBackendAPI`. Ein indirekter Aufruf geschieht über die Klasse `WithinJVMStarter`, die anhand der Ausführung von Graja-CLI eine zweite JVM startet. Innerhalb der zweiten JVM wird dann über `BackendCommandProvider` der Bewertungsvorgang initiiert. Dieser indirekte Aufruf ist anhand eines UML-Sequenzdiagramms in der Abbildung 44 im Anhang veranschaulicht. Eine Aufzeichnung sollte sowohl über `BackendCommandProvider` als auch `WithinJVMBackendAPI` möglich sein. Im Szenario der Regressionstests sollte, um eine reproduzierbare Umgebung zu gewährleisten, für jeden Testfall eine neue JVM gestartet werden. Um die Entwicklung und Fehlersuche innerhalb des Aufzeichnungsmechanismus zu erleichtern, ist allerdings die Nutzung der Klasse `WithinJVMBackendAPI` von Vorteil. Die Funktion `gradeBackendRequest` erlaubt es einen in Kapitel 2.6 behandelten *Backendrequest* auszuführen. Dies spart Zeit, da kein Transformationsschritt zwischen *Frontendrequest* und *Backendrequest* notwendig ist und erlaubt außerdem, zwecks Fehlersuche einen

[87] `https://about.gitlab.com/`
[88] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja`

direkten Zugriff auf die ausführende JVM ohne Verwendung eines *Remote Debuggers*. Eine Ausführung der Verhaltensaufzeichnung müsste nach dem Aufruf der Funktion `Core.gradeRequest` geschehen.

Um eine Modifikation von `gradeRequest` zu vermeiden und eine eindeutige Schnittstelle anzubieten, kann eine abgeänderte Version des *Proxy-Pattern*[89] zum Einsatz kommen. Diese Version verzichtet auf eine Implementierung mittels Vererbung, da die derzeitige API der Klasse `Core` statisch ist und bietet so mithilfe einer neuen Klasse und gleicher Signatur eine Hülle um `Core.gradeRequest` an. Die Funktion `gradeRequest` dieser Klasse kann an `Core` delegieren und anschließend eine Aufzeichnung des observierten Verhaltens durchführen.

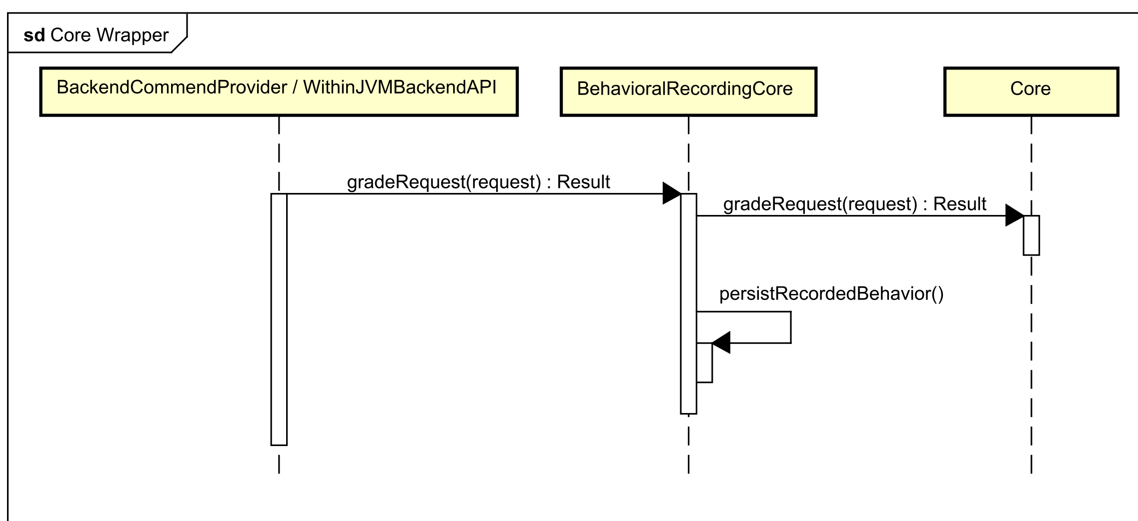


Abbildung 7: Hülle um die Klasse *Core* um eine Verhaltensaufzeichnung zu realisieren

Eine mögliche Realisierung einer solchen Hülle ist anhand des UML-Sequenzdiagramms in Abbildung 7 dargestellt. Durch diesen Lösungsansatz sind sämtlichen zur Aufzeichnung notwendigen Schritte innerhalb der Hülle gekapselt und es ist keine Modifikation der `Core`-Klasse notwendig. Klienten dieser Klasse müssten allerdings statt `Core` nun `BehaviorRecordingCore` aufrufen. Das von `Core` generierte *Result*-Objekt wird weiterhin für mögliche nachfolgende Verarbeitungsschritte an die Klienten zurückgegeben. Somit kann eine Verhaltensaufzeichnung ohne Seiteneffekte für die Klienten `BackendCommandProvider` und `WithinJVMBackendAPI` in den Bewertungsvorgang integriert werden.

5.1.2 Erweiterung des Graja-CLI

Eine Möglichkeit um Graja mitzuteilen, dass eine Aufzeichnung des Bewertungsdurchlaufs erwünscht ist, wäre einen Graja-CLI Befehl um einen Kommandozeilenparameter zu erweitern. Diese Erweiterung würde den Befehl `grade_backend_request` betreffen, der in der Klasse `BackendCommandProvider` implementiert ist. Dieser Befehl veranlasst Graja-CLI einen *Backendrequest* zu bewerten. Grundlage einer erfolgreichen Ausführung des Befehls ist, dass die zu bewertende Einreichung und die dazugehörige task.zip

[89] [GHJV95] [S. 207-217]

bereits in deren Bestandteile zerlegt sind. Nach der Ausführung von Graja den Kommandozeilenbefehl kann dann die Verarbeitung und Persistierung des aufgezeichneten Verhaltens erfolgen.

Das Erweitern des Graja-CLI stellt keine zufriedenstellende Lösung dar. Die Möglichkeit eine Aufzeichnung über die Klasse `WithinJVMBackendAPI` zu starten ist hier nicht gegeben, da diese nicht auf die Funktionalität von Graja-CLI angewiesen ist. Ein weiterer Nachteil liegt darin, dass sich die Implementierung der Klasse `WithinJVMStarter` ebenfalls auf Graja-CLI stützt und somit geändert werden müsste, um den neueingeführten Kommandozeilenparameter zu übergeben. Der Ansatz der Erweiterung von `grade_backend_request` verletzt außerdem das *Interface segregation principle (ISP)*[90], das besagt, dass mehrere spezifische Schnittstellen einer Allzweck-Schnittstelle vorzuziehen sind. In diesem Fall würde der Befehl `grade_backend_request` nicht mehr nur eine Möglichkeit zur Bewertung eines *Backendrequest* zur Verfügung stellen, sondern zusätzlich auch die Aufzeichnung des Verhaltens für den Regressionstestmechanismus steuern. Das *Single-responsibility principle (SRP)* besagt außerdem *„A module should be responsible to one, and only one, actor“*[91]. Der Befehl, der hier ein Modul darstellt, würde nach einer Erweiterung also sowohl Aktoren, die lediglich einen *Backendrequest* zur Bewertung einreichen, als auch der Referenzimplementierung des Regressionstestmechanismus bezüglich der Verhaltensaufzeichnung gegenüber verantwortlich sein.

Eine Einführung eines neuen Graja-CLI Befehls wie z.B. `grade_and_record_backend_request` stellt ebenfalls keine zufriedenstellende Lösung dar. Das *SRP* ist hierbei zwar nicht verletzt, da der Befehl eindeutig nur zum Bewerten und Aufzeichnen dient. Allerdings wäre die Möglichkeit, den Aufzeichnungsvorgang programmatisch durch die Klasse `WithinJVMBackendAPI` einzuleiten dadurch immer noch nicht gegeben. Außerdem fällt somit die `WithinJVMStarter`-API weg, die diesen neuen Befehl nicht aufruft. Somit müsste eine neue Klasse mit ähnlichen Aufgaben wie `WithinJVMStarter` erstellt und gewartet werden. Diese Klasse würde dann den neuen Befehl `grade_and_record_backend_request` aufrufen. Durch ähnliche, aber nicht gleiche Funktionalität der beiden Klassen entsteht duplizierter, nicht geteilter Quellcode, der zukünftige Änderungen am Graja-Frontend erschwert. Eine weitere Verfolgung der Graja-CLI Erweiterung findet in dieser Arbeit wegen der eben genannten Nachteile nicht statt.

5.1.3 Erweiterung des Graja-Domänenmodells

Eine Erweiterung des Graja-Domänenmodells erlaubt es, Anweisungen zur Verhaltensaufzeichnung innerhalb der eingehenden *Requests* (Kapitel 2.6), die durch ein *RequestTO*-Objekt repräsentiert sind, unterzubringen. Für die Referenzimplementierung denkbar wäre die Einführung eines neuen Domänenobjekts namens *BehaviorRecordingTO* innerhalb des `de.hsh.graja.transfer.request` Namensraums, der momentan sämtliche *Request*-Domänenobjekte enthält.[92] Dieses neue Domänenobjekt dient dann als Behälter für die vom Regressionstestmechanismus verwendeten Parameter.

Das *SRP* ist dadurch nicht verletzt, da ein *RequestTO*-Objekt noch immer der Bewertung einer Einreichung dient und die Aktivierung der Verhaltensaufzeichnung durch ein optionales *BehaviorRecordingTO*-Objekt im eingehenden *Request* modelliert ist.

[90] [Mar13] [Kapitel 12]
[91] [Mar17] [S. 62]
[92] [Garb] [3.1.1.29]

Ein Vorteil dieser Lösung ist außerdem, dass keine externen Konfigurationsdateien mit einbezogen werden und das *RequestTO*-Objekt alle zur Aufzeichnung notwendigen Parameter übermittelt.

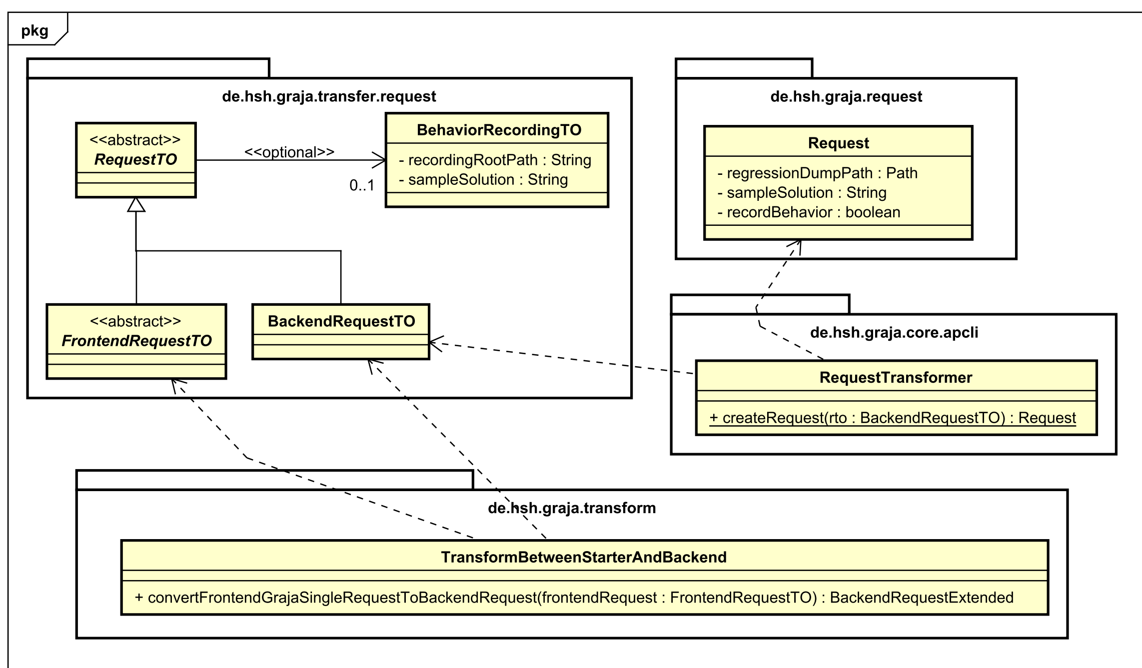


Abbildung 8: Erweiterung des Graja-Domänenmodells um eine Anweisung zur Verhaltensaufzeichnung

In Abbildung 8 ist die vorgeschlagene Erweiterung des Graja-Domänenmodells zu sehen. Die Oberklasse aller *Requests* enthält jeweils einen optionalen Verweis auf das neue Domänenobjekt *BehaviorRecordingTO*. So müssen bereits bestehende Graja-Klienten keine Änderungen am zu übermittelnden *Request* durchführen, da eine Inexistenz eines *BehaviorRecordingTO*-Objekts automatisch eine Verhaltensaufzeichnung ausschließt. Somit ist eine Abwärtskompatibilität mit bereits serialisierten *RequestTO*-Objekten gegeben. Mithilfe des Attributs *recordingRootPath* wird ein Pfad zum Verzeichnis, der das aufgezeichnete Verhalten enthalten soll, übergeben. Um zu identifizieren, für welche Musterlösung die Aufzeichnung stattfindet, ist außerdem das Attribut *sampleSolution* notwending.

Um zu garantieren, dass sowohl *Backendrequests* als auch *Frontendrequests* das initiieren der Verhaltensaufzeichnung unterstützen, wird die Referenz auf das *BehaviorRecordingTO*-Objekt in der Oberklasse aller *Requests* realisiert. Außerdem sind Änderungen in der Klasse `TransformBetweenStarterAndBackend` notwendig. Der Transformationsschritt zwischen *FrontendRequestTO* und *BackendRequestTO* muss um das Überführen des neuen Domänenobjekts angepasst werden, um zu garantieren, dass *Frontendrequests* mit dem Regressionstestmechanismus kompatibel sind.[93]

Die Erweiterung des *Request*-Domänenmodells ist es allerdings noch nicht ausreichend. Um die Integration der Verhaltensaufzeichnung mit dem in Kapitel 5.1.1 beschriebenen *Proxy* durchzuführen, muss die Klasse `Request` des intern in Graja verwendeten

[93] Ansonsten würde der aus einem *Frontendrequest* transformierte *Backendrequest* niemals ein *BehaviorRecordingTO*-Objekt beinhalten und der Regressionstestmechanismus wäre nur unter der Verwendung von *Backendrequests* verwendbar.

Request im Paket `de.hsh.graja.request` um die in Abbildung 8 gezeigten Attribute erweitert werden. Bei einem fehlenden *BehaviorRecordingTO*-Objekt wird das boolesche Attribut *recordBehavior* den Wert *falsch* annehmen und so keine Absicht zur Verhaltensaufzeichnung signalisieren. Hier muss ebenfalls der Transformationsschritt zwischen *BackendRequestTO* und *Request* in der Klasse `RequestTransformer` angepasst werden, um einen Informationsverlust zu vermeiden.

Die Erweiterung des Domänenmodells stellt durch die geringe Anzahl notwendiger Änderungen und der Kompatibilität mit dem *Proxy* aus Kapitel 5.1.1 eine zufriedenstellende Lösung für die für die Verhaltensaufzeichnung dar.

5.1.4 Abspeicherung des aufgezeichneten Verhaltens

Um das aufgezeichnete Verhalten innerhalb einer im Überführungsprozess erstellten task.zip abzuspeichern, kann im ProFormA-Aufgabenformat der Mechanismus des Anhangs verwendet werden, der auch beim Abspeichern der Musterlösungen genutzt wird (Kapitel 2.5.2). Der Regressionstestmechanismus kann dann davon ausgehen, dass falls ein Anhang existiert, bereits das Verhalten der Musterlösung aufgezeichnet wurde. Sollte kein Anhang existieren, muss erst eine Aufzeichnung geschehen. Die task.zip sollte allerdings nicht als primäre Speicherquelle des aufgezeichneten Verhaltens dienen, sondern lediglich dem Bündeln von Aufgaben, Musterlösungen und aufgezeichnetem Verhalten für den Regressionstestmechanismus. Da bei jeder Überführung in das ProFormA-Aufgabenformat die task.zip neu erstellt wird, muss das aufgezeichnete Verhalten auch extern persistiert werden. Dies ist entweder über einen dedizierten Pfad auf dem Hostrechner oder neben dem Verzeichnis für Musterlösungen (*samplesolutions*) innerhalb der Projektstruktur der Aufgabe realisierbar. Im zweiten Fall wäre der Vorteil, dass aufgezeichnetes Verhalten mit in das *Version Control System (VCS)* übernommen wird. Es ist davon auszugehen, dass bei den Testfällen das aufgezeichnete Verhalten öfter gelesen *(Durchführung eines Regressionstests)* als geschrieben *(Neuaufzeichnung des Verhaltens)* wird und so die Versionsgeschichte des *VCS* nicht allzu sehr beeinträchtigt. Mehrere Aufgabenautoren, die innerhalb des gleichen Aufgaben-Repositorium arbeiten, müssten so außerdem beim Klonen keine möglicherweise zeitaufwändige Neuaufzeichnung durchführen

Für die Aufzeichnung sollte das Dateiformat XML[94] gewählt werden. Ein binäres Serialisierungsformat erschwert es Dritten, einen Einblick in das aufgezeichnete Verhalten zu erhalten. Sämtliche Performanzoptimierungen, die ein binäres Serialisierungsformat bietet, können den *Overhead* des Startens einer zweiten *JVM* für die Bewertung einer Musterlösung außerdem nicht negieren. Möglicherweise ergeben sich auch Fälle, in denen das aufgezeichnete Verhalten durch einen Tester manipuliert werden soll. Darunter ´fällt z.B. das Abändern eines referenzierten ProFormA-Testes, bei dem sich lediglich der Name, aber nicht die Funktionalität geändert hat. Neben XML steht auch noch *JavaScript Object Notation* (JSON)[95] zur Auswahl. JSON ist zwar auch menschenlesbar, bietet aber z.B. keine Möglichkeit an, Kommentare mit einzubeziehen. XML erlaubt es außerdem anhand einer *XML Schema Definition* (XSD)[96] ein Schema mit komplexen Typen, die sich aus den XML-Primitiven zusammensetzen, zu definieren. Ein solches Schema kann zur Validierung eines XML-Dokuments verwendet werden. Ein weiterer

[94] [BPSM+]
[95] [Bra14]
[96] [TMB+]

Vorteil von XML ist die Einheitlichkeit mit dem auch XML verwendenden ProFormA-Aufgabenformat. Für das Regressionstestmodul kann außerdem die gesamte bereits bestehende XML-Infrastruktur um Graja herum genutzt werden.

Als Dateiname des aufgezeichneten Verhaltens kann jeweils der Name, der als Testfall gewählten Musterlösung mit dem Präfix `rtrb_` *(Regression Testing Recorded Behavior)* dienen. Bei der Verfolgung dieses Ansatzes ergibt sich eine Ergänzung der Dateistruktur einer Aufgabe, die in Listing 7 abgebildet ist. Das Verzeichnis `rtrb` enthält hier sämtliches aufgezeichnetes Verhalten. Dieser lokale Ansatz ist einem global definierten Verzeichnis vorzuziehen, da eine Aufgabe so ein eigenständiges Modul bildet und ohne externe Abhängigkeit auf einen absoluten, vom Nutzer definierten, systemabhängigen Pfad auskommt. Somit ist auch Portabilität gegeben, da beim Klonen des Aufgaben-Repositorium keine Konfiguration eines solchen Pfads vorgenommen werden muss.

```
1 REPO_ROOT/src/de/hsh/prog/serialize/grader/serialize/grader
2     |-- samplesolutions
3     |    |-- wrong_fake
4     |-- rtrb
5     |    |-- rtrb_wrong_fake.xml
6     |-- task.xml
```

Listing 7: Beispiel für die Abspeicherung des aufgezeichneten Verhaltens im Wurzelverzeichniss einer Aufgabe

Um das aufgezeichnete Verhalten in den Anhang der Aufgabe zu übernehmen, wird das Verzeichnis `rtrb` zur task.zip hinzugefügt. Um die XML-Dateien des aufgezeichneten Verhaltens als gültigen ProFormA-Dateianhang in die task.xml mitaufzunehmen und zu markieren, bieten sich jetzt zwei Möglichkeiten an:

- **Verwendung einer URI**: Um Anhänge als aufgezeichnetes Verhalten zu markieren wird das *id*-Attribut eines Anhangs mit einer *URI*[97] versehen. Als Schema wird hier `rtrb` verwendet. Die *assignmentId* der Aufgabe[98] stellt die *authority* dar und als Pfad wird die jeweilige referenzierte Musterlösung angegeben. Dieses Vorgehen erlaubt eine eindeutige Zuordnung von aufgezeichnetem Verhalten und verwendeter Aufgabe / Musterlösung. Einträge, dessen *id* eine solche valide *URI* darstellen werden automatisch als aufgezeichnetes Verhalten behandelt.
- **Verwendung eines Präfix mit Sequenznummer**: Statt einer *URI* wird für das *id*-Attribut das Präfix `rtrb_` mit einer bei Null startenden Sequenznummer verwendet. Die Sequenznummer dient der Eindeutigkeit der Dateireferenzen. Einträge, deren *id* mit dem Präfix beginnt, werden automatisch als aufgezeichnetes Verhalten behandelt.

Eine Auflösung zwischen Verhalten und Musterlösung geschieht in beiden Fällen über den Dateinamen der angehängten XML-Datei und dem relativen Pfad der task.zip. Eine so eindeutige Zuordnung, wie es eine *URI* erlaubt ist, für das *id*-Attribut allerdings nicht notwendig, da die Aufgabe über den Kontext der task.xml bekannt ist. Das ProFormA-Aufgabenformat spezifiziert bezüglich des `id`-Attributs keinen Standard. Um Komplexität zu vermeiden, wird gemäß dem Paradigma *„convention over*

[97] [BLFM05]
[98] [Garb] [3.1.1.7]

configuration“[99], der zweite Ansatz des Präfixes und der Sequenznummer innerhalb des `id`-Attributs umgesetzt. In Abbildung 8 sind beide Ansätze einmal als XML des ProFormA-Aufgabenformats dargestellt.

```
<p:files>

<!-- id bei Verwendung einer URI -->
<p:file id="rtrb://de.hsh.prog.serialize/wrong_fake" used-
    by-grader="false" visible="no">
  <p:attached-bin-file>rtrb/rtrb_wrong_fake.xml</p:attached
      -bin-file>
</p:file>

<!-- id bei Verwendung eines Praefix -->
<p:file id="rtrb_1" used-by-grader="false" visible="no">
  <p:attached-bin-file>rtrb/rtrb_wrong_fake.xml</p:attached
      -bin-file>
</p:file>

</p:files>
```

Listing 8: Ansätze für die Abspeicherung des aufgezeichneten Verhaltens als Anhang in einer task.xml

5.1.5 Validierung der Symmetrie von Aufzeichnung und Testfall

Es gibt im ProFormA-Aufgabenformat momentan keine Möglichkeit, eine Musterlösung als korrekt zu markieren[100]. Die meisten existierenden Musterlösungen folgen dem Schema, korrekte Musterlösungen mit dem Präfix `correct_` zu markieren und bei negativen Musterlösungen das Präfix `wrong_` zu verwenden. Nach dem Präfix folgt dann zumeist eine Beschreibung, die angibt, warum die Musterlösung korrekt ist oder fehlschlägt. Es gibt diesbezüglich allerdings keinen Standard. In Listing 9 ist die absolute Verteilung aller Präfixe, die mit dem Symbol _ *(Unterstrich)* vom Rest des Verzeichnisnamens der Musterlösung separiert sind. Es existieren allerdings auch Musterlösungen, die keinen Unterstrich zum semantischen Trennen zwischen Präfix und Beschreibung verwenden.

```
> find . -path "*/samplesolutions" -exec ls {} \; | grep -v '.
    html' | cut -d '_' -f 1 | sort | uniq -c | sort -r

    274 wrong
     86 correct
      6 debug
      5 correct2
      4 correct1
      2 korrekt
      2 correct3
      1 test
      1 staticfield
```

[99] [Che]
[100] [MGF+07] [5.8]

```
12         1 leerevorgabe
13         1 korrektMitMap
14         1 iterativ
15         1 fake1
16         1 dummy
17         1 cache
18         1 1wagler
```

Listing 9: Präfixe aller derzeit existierenden Musterlösungen innerhalb des Aufgaben-Repositorium *GrajaAssignments*

Ein überwiegender Teil der Musterlösungen des Aufgaben-Repositorium *GrajaAssignments*[101] hält das inoffizielle Präfix-Schema zwar ein, allerdings gibt es auch eine geringe Anzahl an Musterlösungen, welche ein komplett anderes Schema verfolgen. Diese Musterlösungen müssten für eine Validierung der Symmetrie umbenannt werden. Außerdem müsste ein Standard zur Benennung von Musterlösungen eingeführt werden. Das Ziel einer Validierung der Symmetrie von Aufzeichnung und Testfall (also der Musterlösung, die einen Testfall darstellt), wäre es, schon bei der Aufzeichnung zu überprüfen, ob eine Musterlösung, welche mit dem Präfix `correct_` beginnt auch als korrekt bewertet wird. Dieser Ansatz würde es erlauben sicherzugehen, dass die Bezeichnung der Musterlösung mit dem Verhalten übereinstimmt und somit eine Symmetrie der beiden Faktoren bildet. Eine Weiterverfolgung dieses Ansatzes ist allerdings für den Regressionstestmechanismus nicht notwendig. Der Regressionstestmechanismus operiert anhand von aufgezeichnetem Verhalten und detektierten Abweichungen. Ob eine Musterlösung korrekt von Graja bewertet wird oder nicht, ist irrelevant. Lediglich Abweichungen vom Soll-Verhalten der Aufzeichnung spielen eine Rolle, da diese möglicherweise unbeabsichtigte Regressionen darstellen.

5.2 Eignung der durch Graja generierten Daten für Verhaltensaufzeichnungen

Nachdem der Prozess der Verhaltensaufzeichnung in Kapitel 5.1 behandelt wurde, gilt es in diesem Unterkapitel, die Eignung der von Graja generierten Daten für eine Verhaltensaufzeichnung zu prüfen. Es gilt, Daten zu finden, die das Erkennen möglicher Regressionen bei mehrfacher Ausführung der Musterlösungen nach Änderungen an Graja erlauben. Neben den generierten Feedback-Dokumenten, die z.B. an ein LMS übergeben werden, werden auch interne, von Graja zur Zwischenrepräsentation eines Bewertungsdurchlaufs genutzte Datenstrukturen miteinbezogen. Um auf diese Zwischenrepräsentation zuzugreifen, entstand im Rahmen dieser Arbeit das Werkzeug *Inspector*. Die Funktionsweise des Werkzeugs ist im weiteren Verlauf dieser Arbeit nicht relevant und wird im Anhang unter dem Kapitel *Werkzeug: Inspector* genauer beschrieben.

5.2.1 Eignung der generierten Feedback-Dokumente

Bei Feedback-Dokumenten handelt es sich entweder um HTML-Dokumente oder Textdateien. Bei Textdateien ist davon auszugehen, dass die Baumstruktur des Bewertungsschemas nicht mehr einsehbar ist. Daher sind Textdateien als Basis für das aufgezeichnete Verhalten eines Bewertungsdurchlaufs nicht geeignet. Bei HTML-Dokumenten ist es hingegen mithilfe eines HTML-Parsers möglich mit dem *Document Object Model*

[101] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja-assignments`

(DOM) des HTML-Dokuments zu arbeiten. Druch den Abgleich der Grundstruktur, also dem Aufbau des HTML-Dokuments ohne Beachtung des Texts innerhalb der einzelnen HTML-Knoten, wäre es möglich, die Funktionsweise der Graja-Kommentar-Bibliothek zu überprüfen. Schwieriger hingegen gestaltet es sich, die Ergebnisse der durch die Aufgabe referenzierten ProFormA-Tests zu extrahieren.

```
1  <a id='anchor_7b32...'></a>
2  <span style='...'>
3  Correct. Should use indentation consistently
4  </span>
5  <span style='...'>. Score: 0.15&#47;0.15</span><br>
6  <div>
7  Checks correct indentation of Java code. The expected basic
       indentation offset is 4 spaces.
8  </div><p>
9  <span>
10 Check Indentation (aspect ID: @@!!ProFormA-GradingHints:
      testRef=checkstyle:ProFormA-GradingHints:subRef=
      Indentation:ProFormA-GradingHints:cnt=45!!@@ ) is clean.
11 </span></p>
```

Listing 10: Ausschnitt des generierten HTML eines Feedback-Dokuments

In Listing 10 ist ein Ausschnitt des von Graja generierten HTML-Feedbacks zu sehen. Der Übersicht halber ist der Inhalt von *id* und *style*-Attributen verkürzt dargestellt und durch drei Punkte ersetzt. Es sind im momentanen Generat keine eindeutigen Bezeichner bezüglich der Bewertungsaspekte gegeben. Eine Extraktion dieser Daten würde sich aufgrund der nicht-eindeutigen wiederkehrenden Struktur als nicht robust erweisen. Es müssten in dem Fall verschiedene strukturelle Annahmen getroffen werden, welche die Robustheit des Extraktionsvorgangs einschränken können. Für das HTML in Listing 10 müssten z.B. die Annahmen getroffen werden, dass sich die erreichten Punkte immer in einem `span`-Element befinden und die Referenz auf den dazugehörigen ProFormA-Test nach den `div` und `p`-Elementen in einem `span`-Element eingeschlossen ist. Um diese Limitationen zu umgehen, könnten CSS-Klassenselektoren verwendet werden, um im Transformationschritt Bewertungsaspekte und Aspektgruppen zu markieren. Beim Durchlaufen des DOM ist es damit möglich, Knoten mit z.B. der Klasse `gen-grading-aspect` als transformierte Bewertungaspekt darstellende Knoten zu behandeln. Innerhalb dieser Knoten könnten CSS-Klassenselektoren verwendet werden, um z.B. den Titel des Aspekts, die erreichten Punkte und den beigelegten Kommentar zu kennzeichnen. Eine Verfolgung dieses Ansatzes würde das Themengebiet dieser Arbeit in die Richtung der Thematik *Screen Scraping* verlagern.

Dieser Ansatz ist auch nach einer strukturellen Anpassung des HTML-Generats, der eine eindeutige Identifikation des Bewertungsschemas erlaubt, problematisch. Um herauszufinden, ob alle Tests erfolgreich verlaufen sind, müssten aus dem extrahierten Bewertungsschema die erreichten und erreichbaren Punkte summiert und miteinander verglichen werden. Sollte sich eine Gewichtung oder verwendete Summenfunktion innerhalb des Bewertungsschemas ändern, kann dies zu Fehlalarmen führen. Denn innerhalb des Feedbacks besteht kein Zugriff mehr auf die interne zur Generierung des Feedbacks genutzte Datenstruktur, die es erlaubt, eine relative Erfolgsquote im Intervall [0, 1] zu berechnen. Ebenfalls problematisch gestaltet sich die Zuordnung der

erreichten Punkten und verknüpften ProFormA-Tests. Im oberen Beispiel ist nur durch die angegebene ID darauf zu schließen, dass es sich um einen Test aus dem Testmodul Checkstyle handelt und die Regel *Indentation* referenziert ist.

Abschließend lässt sich feststellen, dass die Feedback-Dokumente verwendet werden können, um die Funktionalität der Graja-Kommentar-Bibliothek zu überprüfen. Zum Auffinden von Regressionen innerhalb der einzelnen ProFormA-Tests oder Graja selber sind diese allerdings aufgrund einer verlustbehafteten Überführung der internen Datenstruktur in ein Feedback-Dokument ungeeignet. Feedback-Dokumente werden durch den Transformationsschritt nach der Bewertung einer Einreichung erstellt und sind so laut dem Nicht-Ziel *„Entdeckung von Regressionen, die sich durch Bewertung einer beliebigen Musterlösung einer beliebigen Aufgabe entdecken lassen“* für diese Arbeit nicht weiter relevant. Außerdem würde ein solcher auf *Screen Scraping* basierender Ansatz die Weiterentwicklung von Graja verhindern. Bei jeder Änderung des Feedback-DOM besteht das Risiko, die Funktionsweise des *Screen Scraping*-Moduls des Regressionstestmechanismus zur Verhaltensaufzeichnung zu beinträchtigen.

5.2.2 Eignung der internen Result-Datenstruktur

Beim Aufruf der `gradeRequest`-Methode der Klasse `Core`[102] wird ein *Result*-Objekt generiert. Dieses *Result*-Objekt stellt eine Zwischenrepräsentation dar, die von Graja genutzt wird um die Feedback-Dokumente zu generieren.

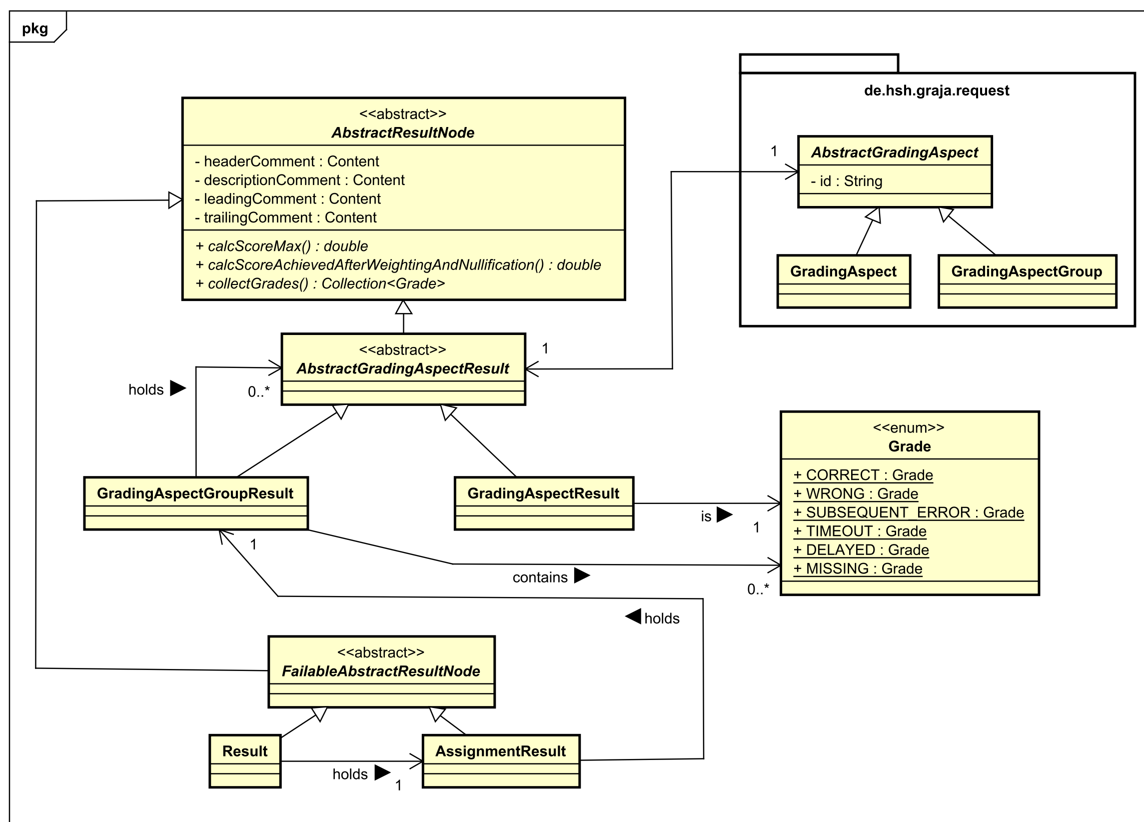


Abbildung 9: Struktur der von Graja generierten Zwischenrepräsentation

Innerhalb dieser Zwischenrepräsentation (UML-Klassendiagram: Abbildung 9) befindet sich die Klasse `AssignmentResult`, die eine Referenz auf eine `GradingAspectGroupRes`

[102] `de.hsh.graja.core.Core`

`ult`-Instanz hält und somit das gesamte ausgefüllte Bewertungsschema und alle von Graja generierten Kommentare repräsentiert. In dem UML-Klassendiagramm in ist die Vererbungshierarchie der Basisklasse `AbstractResultNode` verdeutlicht. Das Klassendiagramm enthält der Übersicht halber nur für diese Arbeit relevante öffentlich zugängliche Methoden und Felder.

Mithilfe der Klasse `AbstractGradingAspect` wird das in Kapitel 2.3 beschriebene Modell des Bewertungsschemas abgebildet. Eine konkrete und ausgefüllte Instanz dieses Bewertungsschemas nach einem Bewertungsdurchlauf wird durch ein *AbstractGradingAspectResult*-Objekt repräsentiert. Dies erlaubt, die gleiche Baumstruktur aus Kapitel 2.3 zu reproduzieren und beinhaltet außerdem die von Graja generierte Kommentare und Informationen bezüglich der erreichten Punktzahl. Jede Instanz der Klasse `AbstractResultNode` ist in der Lage, die maximale eigene erreichte Punktzahl und die derer möglichen Kinderknoten rekursiv zu berechnen. Somit ist es möglich, eine Erfolgsquote im Intervall [0, 1] für sowohl den Wurzelknoten, der Kombinationsgruppen als auch einzelner durch ProFormA-Tests gestützten Bewertungsaspekte, welche die Blätter in dieser Baumstruktur durch die Klasse `GradingAspectResult` modelliert darstellen, zu berechnen. Zusätzlich stellt der Aufzählungstyp `Grade` verschiedene, als Statuscodes für die ausgeführten ProFormA-Tests fungierenden Konstanten zur Verfügung. Diese können anhand der `collectGrades`-Methode rekursiv für alle Kinder eines Knotens im Bewertungsaspektbaums eingesammelt werden. Um die einzelnen Bewertungsaspekte und Aspektgruppen zuzuordnen, halten diese jeweils eine Referenz auf das dazugehörige *AbstractGradingAspect*-Objekt. Über das *id*-Attribut dieses Objekts ist es dann möglich, auf den Namen einer Kombinationsgruppe oder den referenzierten ProFormA-Test des in der task.xml definierten Bewertungsschemas zu schließen. Um aufgrund von Untersuchungszwecken auf den Inhalt des ausgefüllten Bewertungsschemas zuzugreifen, wird das in dieser Arbeit entstandene *Inspector*-Werkzeug verwendet.

```
1  SUCCESS: 0.00 [0.00/3.00]
2  GRADES: [CORRECT, WRONG]
3  ============
4  
5  LEAF: Compile -> Should successfully compile
6  SUCCESS: 0 [0.00/0.00] | [CORRECT]
7  ID: @@!!ProFormA-GradingHints:testRef=compile:ProFormA-
       GradingHints:cnt=37!!@@
8  
9  GROUP: Functional correctness
10 PROFORMA-FUNCTION: sum
11 SUCCESS: 0.00 [0.00/2.70] | [WRONG]
12 ID: @@!!ProFormA-GradingHints:combine=functionality!!@@
13 
14   LEAF: JUnit -> Functional correctness
15   SUCCESS: 0.00 [0.00/2.70] | [WRONG]
16   ID: @@!!ProFormA-GradingHints:testRef=junit:ProFormA-
         GradingHints:subRef=de.hsh.prog.serialize.grader.Grader#
         shouldOutputCorrectResults:ProFormA-GradingHints:cnt=38!!@@
17 
18 GROUP: Further aspects
19 PROFORMA-FUNCTION: sum
20 SUCCESS: 0.00 [0.00/0.30] | [CORRECT, WRONG]
```

```
21 ID: @@!!ProFormA-GradingHints:combine=furtherGradingAspects!!@@
22
23   GROUP: Maintainability
24   PROFORMA-FUNCTION: min
25   SUCCESS: 0.00 [0.00/0.15] | [CORRECT, WRONG]
26   ID: @@!!ProFormA-GradingHints:combine=maintainability!!@@
27
28     LEAF: Checkstyle -> There should be only one statement per
          line
29     SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
30     ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
          GradingHints:subRef=OneStatementPerLine:ProFormA-
          GradingHints:cnt=39!!@@
31
32 ...
```

Listing 11: Verkürzte Ausgabe des *Inspector*-Werkzeugs bei der Musterlösung *wrong_fake* der Aufgabe *de.hsh.prog.serialize*

In Listing 11 ist eine verkürzte Ausgabe aller Kinder des in Kapitel 2.3 beschriebenen `root`-Elements des ProFormA-Aufgabenformat-Bewertungsschemas nach Ausführung aller referenzierten ProFormA-Tests zu sehen. Die ungekürzte Ausgabe befindet sich im Anhang unter Listing 22. In diesem Beispiel wurde die Musterlösung *wrong_fake* der Aufgabe *de.hsh.prog.serialize* aufgrund der Nutzung aller Graja-Testmodule verwendet. Einzelne ProFormA-Tests sind mit `LEAF` gekennzeichnet, während `combine`-Elemente den Bezeichner `GROUP` aufweisen. Die Tiefe des Elements innerhalb des Baums ist mithilfe der Einrückung abgebildet, wobei alle Knoten ohne Einrückung die Tiefe 1 haben, da der Wurzelknoten nicht abgebildet ist. Anhand des *id*-Felds kann nun eine eindeutige Zuweisung zwischen ausgeführtem ProFormA-Test und dessen Ergebnis geschehen. Somit kann mithilfe der internen Datenstruktur des *Result*-Objekts eine Überprüfung der ausgeführten ProFormA-Tests ermöglicht werden. Der ProFormA-Test des Moduls JUnit schlägt hier für die mit `@Test` annotierte Methode `shouldOutputCorrectResults` mit einer Erfolgsquote von 0% fehl. Der ProFormA-Test *OneStatementPerLine* des Moduls *Checkstyle* hingegen besteht mit einer Erfolgsquote von 100%. Die Gruppe *maintainability* beherbergt mehrere ProFormA-Tests. Das Nichtbestehen der Gruppe ergibt sich durch die referenzierte Aggregatfunktionen *min*. Das ProFormA-Aufgabenformat definiert die folgenden Aggregatfunktionen[103] (Tabelle 2), die es Gruppen erlaubt die Ergebnisse ihrer Kinder auszuwerten.

Funktion	Beschreibung
sum	Summation aller Unterergebnisse einer Gruppe. Nützlich, wenn alle Kinder einer Gruppe von Relevanz sind.
min	Minimum aller Unterergebnisse einer Gruppe. Nützlich in einer „alles oder nichts“-Situation, wo nur das Bestehen aller Unterergebnisse zu einem Bestehen der Gruppe führt.
max	Maximum aller Unterergebnisse einer Gruppe. Nützlich bei mehreren Lösungswegen, bei denen bereits einer zum Bestehen der Gruppe führt.

Tabelle 2: Liste aller verfügbaren ProFormA-Aggregatfunktionen

[103][MGF+07] [4.2]

Neben *maintainability* stellt *codingStyle*[104] eine weitere Untergruppe von *furtherGradingAspects* dar. Dadurch, dass *codingStyle* ebenfalls die Aggregatfunktion *min* verwendet und einige derer ProFormA-Tests nicht bestehen, ergibt sich eine finale Erfolgsquote von 0% für die Gruppe *furtherGradingAspects*, welche die beiden Ergebnisse der *sum*-Funktionen aufsummiert.

Das ausgefüllte Bewertungsschema innerhalb der internen *Result*-Datenstruktur ermöglicht es also, das korrekte Verhalten der Aggregatfunktion und referenzierten ProFormA-Tests zu überprüfen und sollte somit in die Verhaltensaufzeichnung einer Musterlösung mit einfließen. Über die Erfolgsquote und den zurückgegebenen Statuscode der *GradingAspectResult*-Objekte können somit Regressionen innerhalb Graja oder dem Grader-Code der Aufgabe festgestellt werden. Sollte z.B. ein ProFormA-Test des Testmoduls JUnit bei der Aufzeichnung fehlschlagen und dann zum Zeitpunkt eines Regressionstests erfolgreich durchlaufen, kann, sofern die Aufgabe und Musterlösung nicht semantisch verändert wurde, so auf eine mögliche Regression geschlossen werden.

Bei der Betrachtung von *Inspector*-Ausgaben weiterer Aufgaben[105] stellt sich heraus, dass bei einer korrekten Musterlösung lediglich der Statuscode *CORRECT* zurückgegeben wird (Aufgabe: *de.hsh.prog.medien*, Musterlösung: *correct*). Außerdem wurde die Musterlösung *constants* der variablen Aufgabe *de.hsh.prog.pointrotate* mit den konkreten Platzhalter-Werten *x=2, y=3* und selbst gewählten Werten *x=8, y=5* ausgeführt. Diese Musterlösung reagiert bei den Parametern *x=8, y=5* in der Gruppe *functionality* mit dem Statuscode *WRONG* und bei *x=2, y=3* mit *WRONG, CORRECT*, da die Werte *x=2, y=3* in der Musterlösung hartkodiert sind. Somit stellt sich zusätzlich heraus, dass bei variablen Aufgaben für den Regressionstestmechanismus immer die Standardwerte der Instanziierung genutzt werden sollten, um die Reproduzierbarkeit des Bewertungsdurchlaufs zu gewährleisten.

5.2.3 Eignung der Zwischenrepräsentation der Kommentare

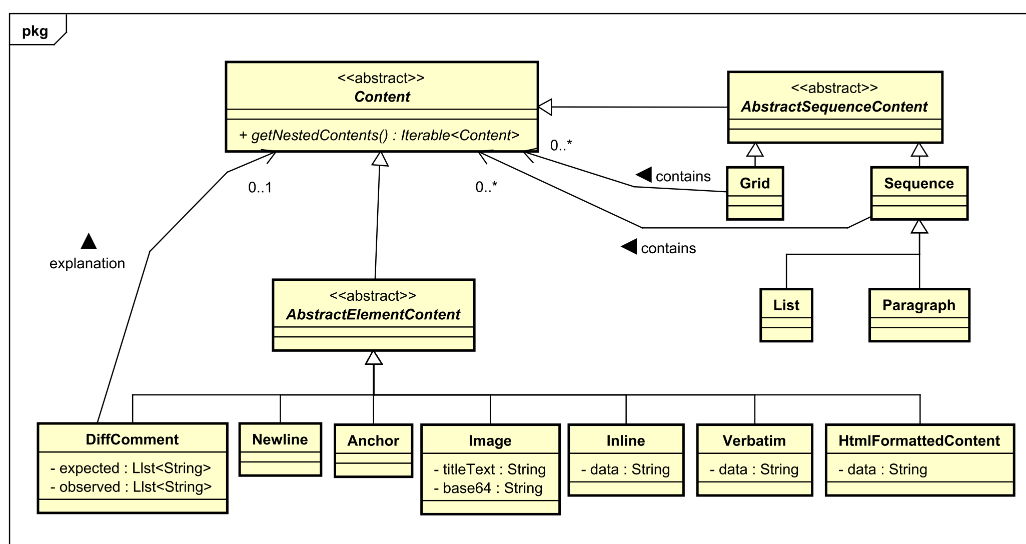


Abbildung 10: Klassenmodell der programmatischen Kommentar-Abstraktion

[104] Im gekürztem Auszug nicht abgebildet. Kompletter Auszug: Anhang Listing 22.
[105] Auffindbar im elektronischen Anhang unter `Daten/Inspector`.

Kommentare werden während des Bewertungsdurchlaufs durch Graja generiert. Um Kommentare zu generieren, wird eine Abstraktion, die sich auf Java-Klassen bezieht, verwendet. Kommentare können so programmatisch zur Laufzeit erstellt werden. Es ist möglich, Kommentare ineinander zu verschachteln und so eine baumartige Zwischenrepräsentation zu generieren, die dem DOM eines HTML-Dokuments ähnelt. Eine Transformation dieser intermediären Datenstruktur in ein HTML- oder Text-Dokument findet mithilfe der Klasse `CommentTransformator` statt. Das Klassenmodell der verwendeten Abstraktion ist in der Abbildung 10 dargestellt. Konkrete Klassen stellen hier einzelne Kommentarbausteine dar, die im finalen Transformationsschritt auf ein HTML- oder Text-Dokument abgebildet werden.

Alle Kommentarbausteine erben von der Basisklasse `Content`. Die abstrakte Klasse `AbstractSequenceContent` erlaubt eine Schachtelung von Kommentaren, während Unterklassen von `AbstractElementContent` die Blätter innerhalb der Zwischenpräsentation darstellen. Während anhand der von Graja generierten Feedback-Dokumente die korrekte Funktionalität der Graja-Kommentar-Bibliothek gestestet werden kann, eignet sich diese Zwischenrepräsentation, um Regressionen innerhalb des Quellcodes, der den Inhalt der Kommentare generiert, zu entdecken. Der Wurzelknoten eines Kommentarbaums wird innerhalb der Klasse `AbstractResultNode`[106] abgespeichert. Ein *AbstractResultNode*-Objekt kann verschiedene Wurzelknoten mit verschiedene Kommentarbäumen enthalten. `AbstractResultNode` erlaubt momentan vier unterschiedliche Wurzelknoten, die durch die Attribute *headerComment*, *descriptionComment*, *leadingComment* und *trailingComment* definiert sind. Diese Attribute definieren Positionsangaben, die im Transformationsschritt in ein Feedback-Dokument befolgt werden.

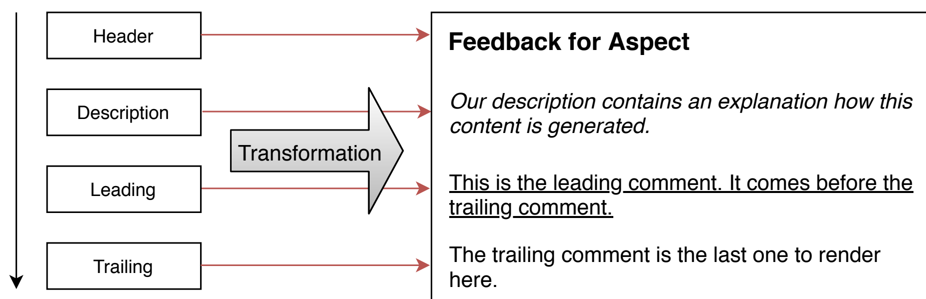


Abbildung 11: Zwischenrepräsentation-Generat-Mapping der unterschiedlichen Kommentarbäume innerhalb der Klasse `AbstractResultNode`

In Abbildung 11 wird die Abarbeitung dieser Reihenfolge und dem daraus resultierenden Mapping genauer abgebildet[107]. Über diese Attribute ist es möglich, verschiedene Kommentarbäume isoliert voneinander zu betrachten. Also nur die Kommentarbäume an der gleichen Position innerhalb zweier Knoten miteinander abzugleichen[108]. Somit müssten nicht erst alle in Abbildung 11 dargestellten Kommentarbäume einer `AbstractResultNode`-Instanz zu einer Entität zusammengefasst werden. Bezüglich Regressionstests wäre es z.B. interessant, ob die Kommentar-Teilbäume der verschiedenen Attribute der Ist-Verhaltens, nach Abgleich mit aufgezeichnetem Soll-

[106] Behandelt in Abbildung 9, Kapitel 5.2.2

[107] Reihenfolge ist in der Klasse `de.hsh.graja.core.apcli.ResultTransformer` innerhalb der Funktion `transformAbstractGradingAspectResultIntoSequenceComment` ersichtlich.

[108] Bei zwei `AbstractResultNode`-Instanzen z.B. *headerComment* des ersten mit dem *headerComment* des zweiten abgleichen.

Verhalten, die gleiche Struktur aufweisen. Um das in der Abbildung beschriebene Verhalten mithilfe eines generierten Feedback-Dokuments zu demonstrieren, liegt im elektronischen Anhang die Datei `Daten/replacedContent.html` bei. Bei der Erstellung dieser Feedback-Datei wurden die Kommentarbäume der Positionsattribute der Klasse `AbstractResultNode` durch Dummy-Elemente ersetzt. Als *RDKindTO*-Konfiguration (Kapitel 2.4) für das Generat wurde der Übersicht halber *student-severe-html* verwendet.

5.2.3.1 Eignung der verschiedenen Kommentar-Ebenen zum Erkennen von Regressionen

Bei der Betrachtung der in Abbildung 9 dargestellten Zwischenpräsentation fällt auf, dass sowohl *Result*-Objekte, *AssignmentResult*-Objekte und Bewertungsaspekte / Aspektgruppen über die Positionsattribute der gemeinsamen Oberklasse `AbstractResultNode` Kommentarbäume beinhalten. Ein *Result*-Objekt stellt also eine Verschachtelung dar, die in der Abbildung 12 dargestellt ist.

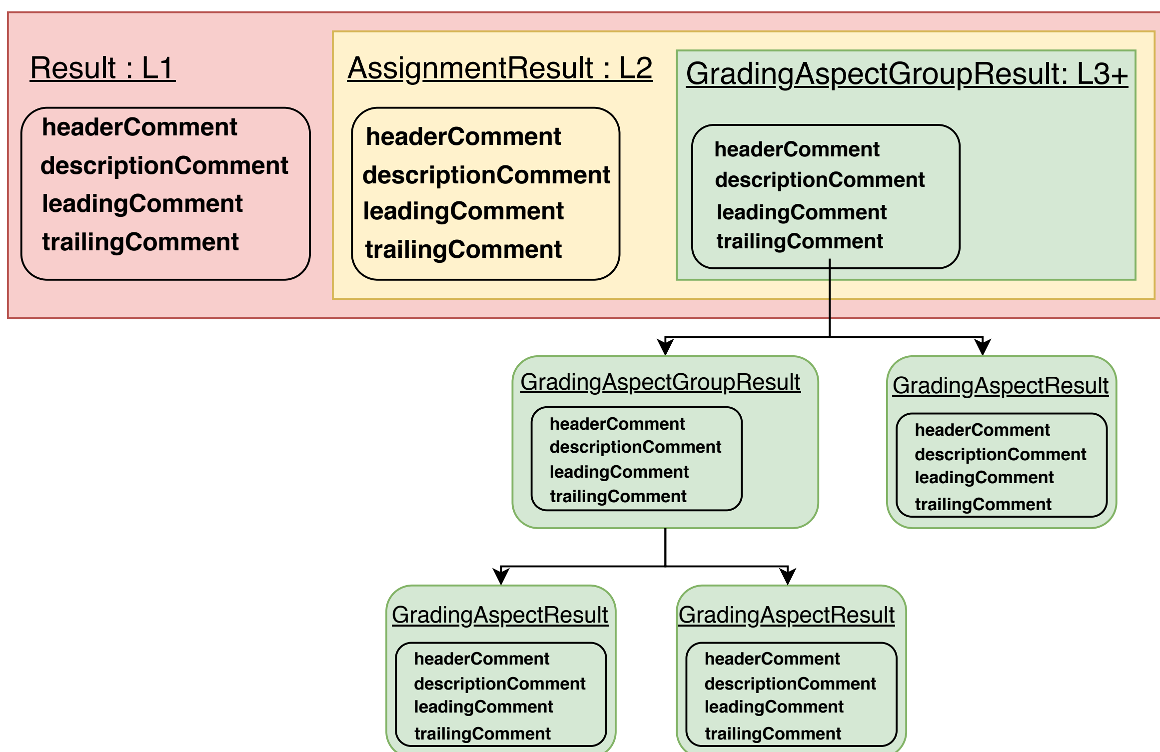


Abbildung 12: Grafische Darstellung der unterschiedlichen Ebenen einer Instanz der in Abbildung 9 gezeigten Klasse `Result`

Auf der ersten Ebene *L1*[109] befindet sich das *Result*-Objekt mit dessen Kommentarbäumen. Die Ebene *L2* enthält die Kommentarbäume des *AssignmentResult*-Objekts und ab Ebene *L3+* beginnt die Baumstruktur des Bewertungsschemas, welche pro Gruppe und Bewertungsaspekt ebenfalls Kommentarbäume enthält. Um nun zu prüfen, inwiefern die Kommentarbäume dieser verschiedenen Ebenen zum Detektieren von Regressionen eine Eignung aufweisen, wurde anhand des *Inspector*-Werkzeugs für die

[109] *L* steht in diesem Fall für *Level*.

Musterlösung *wrong_fake* der Aufgabe *de.hsh.prog.serialize*[110] eine Ausgabe generiert. Diese Ausgabe wurde sowohl einmal durch eine externe JVM generiert[111], als auch über die JVM des Aufrufers. Ziel dieser Ausgabenanalyse ist herauszufinden, ob ein markanter Unterschied zwischen den beiden Ausführungsvarianten (interne JVM / externe JVM) bezüglich der Kommentarebenen besteht.

Das Skript `diff.sh`[112] filtert vor Abgleich mit dem Werkzeug *diff*[113] zuerst alle Zahlen mithilfe von *sed*[114] heraus. Dieser Schritt ist notwendig, um Fehlalarme durch abweichende Zeitangaben zu minimieren. Danach findet jeweils ein Abgleich der gleichen Ebenen der verschiedenen Ausführungsvarianten gegeneinander statt. Die Ausgabe von `diff.sh` unter der Verwendung dieser Daten ist in Listing 12 zu sehen. Es befinden sich keine strukturellen Unterschiede bezüglich der Kommentare in den Ebenen *L2* und *L3+*. Die Unterschiede dieser beiden Ausgaben erklären sich durch Pfadangaben zu temporären Verzeichnissen, die während des Bewertungsvorgangs genutzt wurden. Die Kommentare der Ebene *L1* unterscheiden sich dagegen stark voneinander. Das Starten einer zweiten JVM ergänzt zusätzliche Kommentare mit Informationen bezüglich des zweiten Java-Prozesses, die beim Aufrufen innerhalb derselben JVM fehlen.

```
1 backendFrontend > ./diff.sh
2
3 Differenzen: Kommentare L3+ (Caller JVM/Second JVM)
4 1
5 Differenzen: Kommentare L2 (Caller JVM/Second JVM)
6 14
7 Differenzen: Kommentare L1 (Caller JVM/Second JVM)
8 82
```

Listing 12: Verleich von Kommentaren, die durch eine externe JVM und der JVM des Aufrufers generiert wurden

Durch Analyse aller der vom *Inspector*-Werkzeug generierten Kommentar-Ausgaben, lässt sich darauf schließen, dass die Kommentare der Ebene *L2* (Bewertungsschema), sowie *L3+* (Kommentare der einzelnen ProFormA-Tests) genutzt werden können, um Regressionen innerhalb von Graja zu erkennen. Die Struktur und der meiste Inhalt dieser Kommentare bleibt sowohl durch Ausführung einer zweiten JVM, als auch innerhalb der Aufrufer-JVM gleich. Die Kommentare der Ebene *L1* hingegen erweisen sich hierbei als nicht hilfreich. Beide enthalten Metainformationen bezüglich der Aufgabe und den Parametern des *RequestTO*-Objekts. Die Kommentare des in einer zweiten JVM gestarteten *Result*-Objekts enthalten außerdem noch detaillierte Informationen über des verwendete JDK. Die darin enthaltenen Informationen sind bezüglich möglicher Graja-Regressionen irrelevant und erschweren durch variable Parameter einen einheitlichen Soll-Ist-Abgleich ohne Fehlalarme.

[110] Die hier verwendete Musterlösung und Aufgabe wurde zufällig gewählt, da es erstmal gilt grobe strukturelle Unterschiede zu erkennen. Eine Analyse, die mehrere Aufgaben und Musterlösungen mit einschließt, findet in 5.3 statt.

[111] Diese Ausgabe enthält isoliert voneinander alle Kommentare der Ebenen *L1*, *L2* und *L3+*. Das Präfix *b_* markiert die Ausgaben der innerhalb der Klasse `WithinJVMBackendAPI` generierten Daten.

[112] Die Daten und ein Skript zum Erkennen von Unterschieden sind im elektronischen Anhang unter `Daten/Inspector/backendFrontend` vorhanden.

[113] `https://man7.org/linux/man-pages/man1/diff.1.html`

[114] `https://linux.die.net/man/1/sed`

Mithilfe des in Listing 12 durchgeführten Verhaltensabgleichs haben sich sogar nach einer Filterung aller numerischen Symbole Fehlalarme durch Rauschen ergeben. Eine Untersuchung des Rauschens in Aufzeichnungen durch z.B. Zeitangaben / Pfade und derren möglicher Implikationen bezüglich eines Verhaltensabgleichs findet im Kapitel 5.3 statt. Für eine strukturelle Überprüfung des Kommentarbaums sind alle der in Abbildung 10 genannten Klassen relevant. Bei inhaltlichen Vergleichen kann sich auf alle `Content`-Unterklassen mit konkretem Inhalt (`DiffComment`, `Image`, `Inline`, `Verbatim` und `HtmlFormattedContent`) beschränkt werden.

5.2.3.2 Lokalisierung der Kommentar-Herkunft im Quellcode

Ein Problem stellt letztlich noch eine Lokalisierung der Erstellung eines Kommentars im Quellcode dar. Ist eine mögliche Regression durch einen inhaltlichen Kommentar-Vergleich detektiert, sollte, um eine Behebung zu erleichtern, die Stelle, an dem der Kommentar generiert wurde, lokalisierbar sein. In Graja sind momentan keine Informationen bezüglich der Herkunft eines Kommentars abrufbar. Um dem entgegenzuwirken, kann darauf zurückgegriffen werden, dass die JVM während der Laufzeit Zugriff auf den *Stacktrace* (Aufrufstapel) erlaubt. In Java ist ein Lesezugriff durch der Methode `getStackTrace` der Klasse `Thread` möglich.[115] Die `Content`-Klasse könnte also abgeändert werden, um während der Konstruktion eines *Content*-Objekts über den konstruierenden Thread einen *Stackdump* durchzuführen und den resultierenden *Stacktrace* mit der resultierenden `Content`-Instanz zu assoziieren. Sollte sich beim Abgleich dann eine Differenz mit dem aufgezeichneten Verhalten ergeben, stellt dieser *Stacktrace* eine Lokalisierung der Kommentarquelle im Ist-Verhalten dar. Eine Übernahme des *Stacktrace* in das aufgezeichnete Verhalten ist nicht notwendig, da dieser nur bei möglichen erkannten Regressionen mit dem Ist-Verhalten in der Testphase benötigt wird.

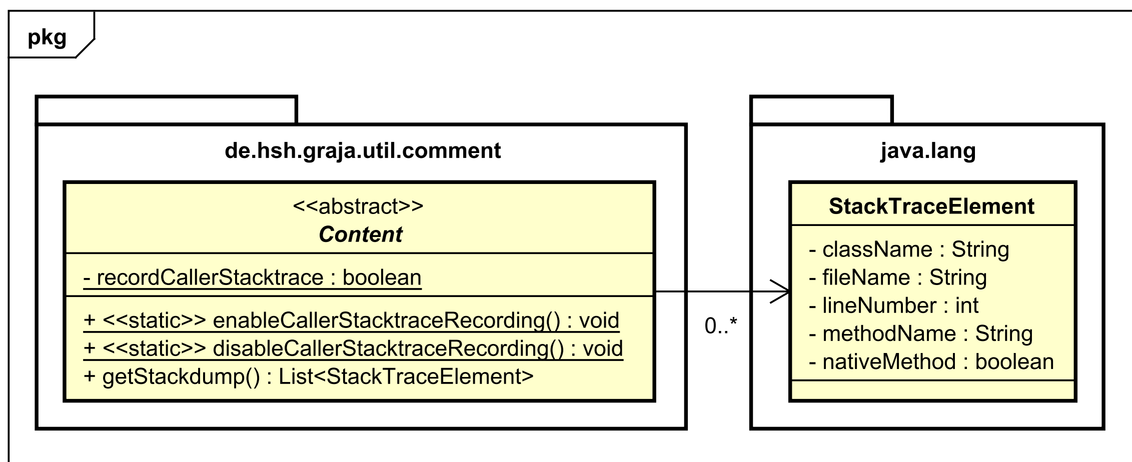


Abbildung 13: Erweiterung der Klasse *Content* um einen *Stackdump* während der Konstruktionsphase durchzuführen

Um einen *Stackdump* zu realisieren, kann die in Abbildung 13 beschriebene Erweiterung der Klasse `Content` durchgeführt werden. Dadurch, dass alle Kommentarbausteine von `Content` erben, ist garantiert, dass für alle Unterklassen die Herkunft des aufrufenden Quellcodes anhand eines *Stackdump* über einen *instance initializer*[116] in-

[115] [Orae]
[116] [Oraa]

nerhalb der `Content`-Klasse ermittelt werden kann. Ein *instance initializer* funktioniert wie ein *static initializer*, nur auf Instanzlevel. Der Java-Compiler kopiert die Instruktionen des *instance initializer* in jeden Konstruktor, somit ist garantiert, dass auch bei der Ergänzung von neuen `Content`-Konstruktoren ein *Stackdump* durchgeführt wird. Ein statisches boolesches Attribut innerhalb der Klasse kann verwendet werden, um eine *Stackdump*-Absicht zu signalisieren. Mit den statischen Funktionen `enableCallerStacktraceRecording` und `disableCallerStacktraceRecording` ist es dann möglich den *Stackdump*-Mechanismus zu steuern. Falls ein *Stackdump* durchgeführt wurde, kann dieser dann mithilfe der Methode `getStackdump` abgerufen werden. Ein Prototyp, der die jeweiligen *Stacktraces* in eine Textdatei schreibt, wird im Anhang unter *Werkzeug: Stacktrace-Dumper* beschrieben. Ein Filtern der ersten zwei *Stackframes* ist allerdings notwendig, da diese den nativen Aufruf von `Thread.getStackTrace` und des `Content`-Konstruktors enthalten, die bezüglich der Kommentarherkunft nicht von Relevanz sind.

5.2.4 Ein Datenmodell für die Verhaltensaufzeichnung

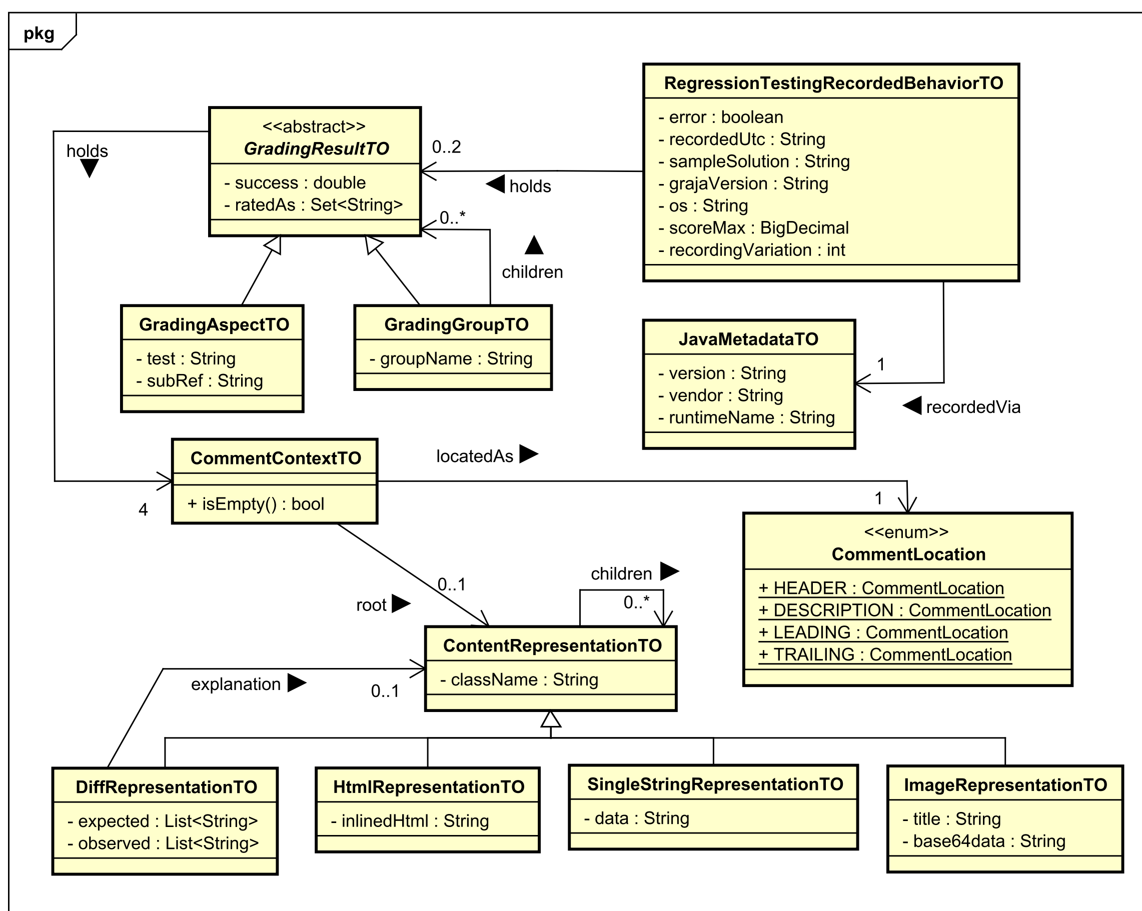


Abbildung 14: Vorgeschlagenes Datenmodell um das aufgezeichnete Verhalten einer Musterlösung zu persistieren

Nachdem die verschiedenen von Graja generierten Daten untersucht wurden, ist ein Datenmodell notwendig, um die Abspeicherung des aufgezeichneten Verhaltens einer Musterlösung umzusetzen. Die generierten Feedback-Dokumente werden wegen schlechter Eignung, Regressionen außerhalb der Graja-Kommentar-Bibliothek zu detektieren, au-

ßen vorgelassen[117]. Das vorgeschlagene Datenmodell, das zum Großteil aus Transfer-Objekten besteht, ist in Abbildung 14 als UML-Klassendiagramm abgebildet.

Auf das Datenmodell wird der Inhalt der internen Zwischenpräsentation des Bewertungsschemas abstrahiert. Enthalten ist die Baumstruktur des Bewertungsschemas mit den Bewertungsaspekten, Aspektgruppen und der dazugehörigen Kommentare. Die Klasse `RegressionTestingRecordedBehaviorTO` stellt in diesem Modell den Einstiegspunkt dar und wird pro Aufzeichnung in einer separaten XML-Datei persistiert. Es werden Metadaten bezüglich der verwendeten Graja-Version, Musterlösung, des Betriebssystems und Aufzeichnungsdatum abgespeichert. Das Aufzeichnungsdatum wird als ISO8601-Zeitstempel mit der Zeitzone UTC persistiert. Im Gegensatz zur Unixzeit stellt das ISO8601-Zeitformat[118] eine menschenlesbare Alternative dar. Die Zeitzone UTC ist gewählt, um eine unkomplizierte Interoperabilität zwischen verschiedenen Zeitzonen zu ermöglichen[119]. Mithilfe des booleschen Attributs *error* wird signalisiert, dass Aufgrund eines Bewertungsfehlers keine interne Zwischenpräsentation in Form eines *Result*-Objektes generiert wurde. Dies ist z.B. der Fall wenn eine Endlosschleife detektiert wird und ein Abbruch des Bewertungsvorgangs stattfindet. Zusätzlich wird die maximale Gesamtpunktzahl der Aufgabe anhand des Attributs *scoreMax* abgespeichert. Dies ist notwendig, um zu garantieren, dass die für den Regressionstest verwendeten task.zip-Dateien mit der gleichen Gesamtpunktzahl wie zum Zeitpunkt der ursprünglichen Verhaltensaufzeichnung erstellt werden und somit gleiche Voraussetzungen bezüglich des Bewertungsschemas existieren. Zusätzliche Metadaten bezüglich des zum Aufzeichnungszeitpunkt verwendeten JDKs werden durch die Klasse `JavaMetadataTO` dargestellt. Das Attribut *recordingVariation* wird später in Kapitel 5.3.7 erläutert.

Der Einstiegspunkt hält außerdem Referenzen auf zwei *GradingResultTO*-Objekte, welche die in Kapitel 5.2.3.1 behandelten Ebenen *L2* und *L3+* abbilden. Anhand der Klasse `GradingResultTO` kann das in Kapitel 5.2.2 dargestellte Bewertungsschema wiedergegeben werden. *GradingResultTO*-Objekte halten vier Referenzen[120] auf *CommentContextTO*-Objekte. Im Falle der Ebene *L2* werden somit die Kommentare, die zum Bewertungsschema gehören, dargestellt, während in der Ebene *L3+* somit die Kommentare der Bewertungaspekte und Aspektgruppen abgebildet sind. Außerdem enthält die Klasse `GradingResultTO` jeweils die Statuscodes der Bewertung[121] und eine relative Erfolgsquote im Intervall [0, 1]. Um Bewertungsaspekte und Aspektgruppen zu unterscheiden, existieren die beiden konkreten Unterklassen `GradingAspectTO` und `GradingGroupTO`. Um das Testmodul und die Unterreferenz des Tests zu identifizieren, wird die in Abbildung 9 vorhandene *ProFormA-Id*[122] geparst. Gleiches geschieht für den Gruppennamen einer Aspektgruppe und dem Wurzelknoten des Bewertungsschemas, welche beide durch die Klasse `GradingGroupTO` dargestellt sind.

Eine `CommentContextTO`-Instanz repräsentiert jeweils eine Position aus dem Aufzählungstypen `CommentLocation`. Somit ist eine Unterscheidung der Kommentarbäume

[117] Kapitel 5.2.1

[118] [ISO]

[119] [NK02] [S. 3]

[120] Alle in Kapitel 5.2.3.1 vorgestellten Positionsattribute.

[121] In diesem Falle als Menge von Zeichenketten um eine Abhängigkeit zum Aufzählungstypen `Grade` zu vermeiden.

[122] Die *ProFormA-Id* ist über die Klasse `AbstractGradingAspect` im Paket `de.hsh.graja.request` abrufbar, da jede `AbstractGradingAspectResult`-Instanz eine Referenz auf diese Klasse hält.

bezüglich der Positionsattribute möglich[123]. Die Methode `isEmpty` erlaubt es zu überprüfen, ob ein leerer Kommentarbaum für das referenzierte Positionsattribut vorliegt.

Eine Referenz eines *CommentContextTO*-Objekts auf ein *ContentRepresentationTO*-Objekt stellt den Wurzelknoten eines Kommentarbaums dar. `ContentRepresentation TO`-Unterklassen werden verwendet, um Spezialfälle der in Abbildung 10 dargestellten `Content`-Klassen zu emulieren. Alle anderen `Content`-Klassen werden über die Basisklasse dargestellt. Das Attribut *className* erlaubt es, die verschiedenen Kommentarbausteine in diesem Datenmodell voneinander zu unterscheiden. Somit lassen sich z.B. `Newline`, `List` oder `Anchor`-Bausteine semantisch unterscheiden ohne unterschiedliche, leere `ContentRepresentationTO`-Unterklassen zu erstellen. Eine direkte Nutzung der `Content`-Klassen findet nicht statt, um eine Trennung der Zuständigkeiten zwischen Graja-Kommentar-Bibliothek und Regressionstestmechanismus zu erreichen. In der unteren Auflistung ist die Beziehung zwischen `ContentRepresentationTO`-Klassen und der jeweiligen speziellen `Content`-Unterklasse dargestellt.

- `DiffRepresentationTO`: Bildet `DiffComment` ab. Enthält den erwarteten und observierten Text. Hält eine Referenz auf einen Kommentar, der als Beschreibung dient.
- `SingleStringRepresentationTO`: Bildet Kommentare ab, die lediglich aus Text bestehen, wie z.B. `Inline` und `Verbatim`. `Inline` und `Verbatim` unterscheiden sich im Verhalten erst bei der Transformation in ein Feedback-Dokument. Ein `Verbatim` garantiert, dass der enthaltene String in einem HTML-Dokument mit der korrekten Formatierung gerendert wird. Da allerdings die Erkennung von Regressionen innerhalb der Graja-Kommentarbibliothek ein Nicht-Ziel darstellt, werden die Rohdaten der `Verbatim` und `Inline` Kommentarbausteine mit der gleichen Klasse repräsentiert.
- `ImageRepresentationTO`: Bildet `Image` ab und enthält den Titel und die in *Base64*[124] encodierten Bilddaten.
- `HtmlRepresentationTO`: Bildet `HtmlFormattedContent` ab und enthält ein HTML-Dokument.

Durch die Verschachtelung von Kommentaren kann somit die in Kapitel 5.2.3 dargestellte Baumstruktur der Kommentar-Zwischenpräsentation wiedergegeben werden.

Der Soll-Ist-Vergleich einer Aufzeichnung und dem zum Testzeitpunkt observierten Verhalten wird auf diesem Datenmodell operieren. Die internen Strukturen des *Result*-Objekts werden dazu erst in dieses Datenmodell überführt. Der Verhaltensabgleich über diese Struktur wird in Kapitel 5.4 behandelt.

5.3 Behandlung von Fehlalarmen und Rauschen in Aufzeichnungen

In Kapitel 5.2.3.1 (Listing 12) wurde anhand einer Musterlösung observiert, dass sich die Kommentare bis auf Rauschen je nach Bewertung innerhalb der Aufrufer-JVM oder durch eine externe JVM nicht unterscheiden. Das dort observierte Rauschen setzte sich aus Zeitangaben und Dateipfaden zusammen. Rauschen wird im Kontext dieser Arbeit

[123] Dies ist wichtig, um nur Kommentarbäume mit den gleichen Positionsattributen miteinander abzugleichen.
[124] [Jos06]

als variable Parameter innerhalb der Kommentare definiert, die keinen Einfluss auf die semantische Korrektheit der Kommentare ausüben und somit keine Relevanz bezüglich des Regressionstestmechanismus haben. Das Beibehalten von Rauschen innerhalb der Aufzeichnungen würde bei einem 1:1-Vergleich mit dem observierten Verhalten schnell zu Fehlalarmen führen. Zeitangaben würden sich beispielsweise pro Bewertungsvorgang ändern und Pfadangaben wären auf unterschiedlichen Systemen selten reproduzierbar. Um mit Rauschen umzugehen, muss erst durch eine Datenerhebung und Datenanalyse herausgefunden werden, welche Arten von Rauschen innerhalb der verschiedenen Musterlösungen auftreten können. Darauffolgend muss dann eine Strategie entwickelt werden, dieses Rauschen zu unterdrücken, um zu garantieren, dass keine bezüglich der Regressionstests irrelevanten Parameter aufgezeichnet und somit abgeglichen werden.

5.3.1 Generierung von Rohdaten für eine Rauschidentifikation

Die im Kapitel 5.2.3.1 durchgeführte Observation ist möglicherweise nicht für alle Musterlösungen und Aufgaben repräsentativ. Daher werden in diesem Kapitel Kommentare für verschiedene Musterlösung-Aufgaben-Kombinationen unter verschiedenen Betriebssystemen und JDK-Versionen gesammelt. Anschließend findet eine Auswertung der gesammelten Rohdaten statt, um Unterschiede und somit Rauschen zu detektieren. Diese Unterschiede werden dann in verschiedene Rausch-Kategorien eingeteilt, die es im Kapitel 5.3.2 zu unterdrücken gilt. Um die Rohdaten zu generieren, wird das für diese Arbeit entwickelte *Comment-Dumper*-Werkzeug verwendet, das im Anhang unter *Werkzeug: Comment-Dumper* genauer beschrieben ist. Momentan existieren in beiden Aufgaben-Repositorien[125] 79 Aufgaben[126]. Da das Ausführen aller Musterlösungen dieser 79 Aufgaben einen großen Aufwand darstellt, werden 15 Aufgaben (*20%*) ausgewählt. Von diesen verschiedene Musterlösungen werden dann mindestens eine korrekte und maximal vier falsche für die Generierung von Rohdaten mithilfe des Werkzeugs *Comment-Dumper* verwendet. Alle dieser 79 Aufgaben der Aufgaben-Repositorien befinden sich außerdem innerhalb des elektronischen Anhangs im task.zip-Format unter dem Verzeichnis `Graja Aufgaben`. Zehn dieser Aufgaben sind zufällig ausgewählt, die restlichen verfolgen bestimmte Kriterien wie z.B. Variabilität der Aufgabe, GUI-Funktionalität oder Dateizugriffe. Die zufällig gewählten Aufgaben wurden mit dem Unix-Kommando *shuf*[127] über die Menge der im elektronischen Anhang vorhandenen Aufgaben bestimmt. Diese sind im Listing 13 zu sehen. Besondere Aufgaben stellen *de.hsh.prog.serialize*, das mit Javas Objekt-Serialisierung[128] arbeitet und *de.hsh.prog.factorsengine*, das Multi-Threading verwendet, dar.

```
1 > ls | shuf -n 10
2
3 dom.grid.zip
4 de.hsh.prog.kunzestatistik.zip
5 de.hsh.prog.serialize.zip
6 dom.charstat.dis.zip
7 de.hsh.prog.innenwinkel.zip
```

[125] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja-assignments` und `development/SampleGrajaAssignments` in `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja`

[126] Aufgaben im *dom* und *de.hsh.prog*-Namensraum

[127] `https://linux.die.net/man/1/shuf`

[128] `https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html`

```
8  de.hsh.prog.doublespace.zip
9  dom.fraction.zip
10 de.hsh.prog.factorsengine.zip
11 de.hsh.prog.verdoppelnelementeinliste.zip
12 dom.student1.zip
```

Listing 13: Zufällig ausgewählte Aufgaben für die Rauschidentifikation

Die Auswahl der restlichen fünf gewählten Aufgaben sind in der folgenden Auflistung begründet.

- `de.hsh.prog.detectzip.zip` : Dateioperationen
- `de.hsh.prog.ballspiele.zip` : Vererbungsbasiert
- `de.hsh.prog.simplegraphic.dis.zip` : Variabilität
- `de.hsh.prog.klassedashedline.zip` : Zeichnung von Linien in einer Grafik
- `de.hsh.prog.maxfromcmdline.zip` : Endlosschleife in den Musterlösungen

Um die Rohdaten zu generieren werden zwei verschiedene Systeme mit unterschiedlichen Betriebssystemen und JDK-Versionen verwendet.

- Betriebssystem: *Windows 10*, JDK: *OracleJDK (1.8.0_231)*[129]
- Betriebssystem: *Ubuntu 18.04 bionic*, JDK: *OpenJDK (1.8.0_252)*[130]

Die Rohdaten wurden mithilfe der im elektronischen Anhang vorhandenen Graja-Version mit Hilfswerkzeugen automatisiert über die Graja-GUI generiert[131] und sind im Anhang unter `Daten/CommentDump.zip` vorhanden. Es wurde für fast alle Aufgaben eine Bewertung über das *Backend* und *Frontend* vorgenommen. Eine Ausnahme bildet die Musterlösung *wrong_endlessLoop* der Aufgabe *de.hsh.prog.maxfromcmdline*. Die dort ausgeführte Endlosschleife würde durch die fehlenden in Kapitel 2.2 beschriebenen Sicherheitsmaßnahmen im *Backend* nicht terminieren und wurde deshalb nur durch das *Frontend* ausgeführt. Eine Liste aller für die Rauschidentifikation verwendeten Aufgaben und Musterlösungen befindet sich im Anhang unter Listing 26.

5.3.2 Identifikation und Unterdrückung von Rauschen

Um Rauschen zu identifizieren wurde das Werkzeug *Comment-Dump-Analyzer* entwickelt. Die interne Funktionalität des Werkzeugs wird im Anhang unter *Werkzeug: Comment-Dump-Analyzer* beschrieben. Die Grundidee bei dem Werkzeug ist es, die in Kapitel 5.3.1 aufgezeichneten Rohdaten zu analysieren. Dabei werden jeweils die unter verschiedenen Betriebssystemen und JDK-Versionen aufgezeichneten Kommentare einer Musterlösung einer Aufgabe miteinander abgeglichen. Ziel eines Abgleichs ist es, Differenzen zu finden. Da jeweils gleiche Musterlösungen und Aufgaben gegeneinander abgeglichen werden, ist davon auszugehen, dass Unterschiede hier Rauschen darstellen. Bezüglich der ausführenden JVM wird jeweils extern gegen extern und intern gegen intern abgeglichen. Ein gemischter Abgleich, also extern gegen intern findet nicht statt, da

[129] `https://www.oracle.com/java/technologies/javase/8u231-relnotes.html`

[130] `https://adoptopenjdk.net/release_notes.html`

[131] Die Nutzung der automatisierten Aufzeichnung des *Commtent-Dumper*-Werkzeugs ist im Anhang unter *Werkzeug: Comment-Dumper* beschrieben. Die hierfür verwendete Konfigurationsdatei befindet sich im elektronischen Anhang unter `Daten/batchDump.txt`.

in der Referenzimplementierung später nur eine Ausführungsvariante verwendet wird. Falls Differenzen erkannt wurden, werden diese in eine Textdatei geschrieben. Am Ende eines *Comment-Dump-Analyzer*-Durchlaufs werden außerdem Statistiken bezüglich der behandelten Fälle und erkannten Differenzen ausgegeben.

Um Rauschen zu unterdrücken findet vor dem Abgleich eine Textersetzung mithilfe Regulärer Ausdrücke[132][133] statt. Reguläre Ausdrücke erlauben es, Muster in einem Text zu erkennen. Erkannte Muster werden dann durch einen neutralen Textausdruck ersetzt. Nach der Ersetzung kann daraufhin der Abgleich geschehen. Dadurch, dass die erkannten Muster durch einen neutralen und immer wiederkehrenden Textausdruck ersetzt wurden, werden keine Differenzen mehr erkannt. Um eine Rauschunterdrückung zu realisieren, werden nun solange neue reguläre Ausdrücke definiert, bis keine Differenzen mehr erkennbar sind. In den folgenden Unterkapiteln wird anschließend Rauschen kategorisiert und jeweils eine passende Methode zur Unterdrückung vorgestellt.

5.3.2.1 Dateipfad-bedingtes Rauschen

Dateipfad bedingtes Rauschen entsteht durch Dateipfad enthaltende Kommentare. Diese Dateipfade folgen je nach Betriebssystem einem anderen Format. Auch auf dem gleichen System kann die mehrmalige Ausführung eines Bewertungsvorgangs durch randomisierte temporäre Arbeitsverzeichnisse zu Rauschen führen. Um solches Rauschen zu behandeln, müssen sowohl Windows-spezifische[134] als auch Linux-spezifische[135] Pfadangaben behandelt werden. Um sämtliches Dateipfad bedingtes Rauschen zu behandeln, werden fünf reguläre Ausdrücke benötigt, die in der Datei `ReplacePatterns.java` des *Comment-Dump-Analyzer*-Werkzeugs im elektronischen Anhang zu finden sind. Einige dieser regulären Ausdrücke sind vergleichsweise umfangreich und daher nicht an dieser Stelle in der Arbeit abgebildet.

Insgesamt wurden, wie in Listing 28 zu sehen ist[136], durch diese fünf regulären Ausdrücke 1044 Dateipfade in dem gesammelten Datensatz unterdrückt.

5.3.2.2 Zeitangaben bedingtes Rauschen

Zeitangaben bedingtes Rauschen entsteht durch Zeitstempel, die sich pro Ausführungszeitpunkt ändern. Graja verwendet hier die Klasse *SimpleDateFormat*[137] mit der Formatierung `yyyy-MM-dd'T'HH:mm:ss.SSS`. Außerdem wird in manchen Kommentaren die Anzahl an Sekunden, nach dem Graja einen Bewertungsvorgang abbricht, referenziert. Diese folgt dem Format `<TimeInSeconds>s`. Die beiden Rauschfaktoren werden ebenfalls durch zwei reguläre Ausdrücke neutralisiert, die auch in der `ReplacePatterns.java`-Datei auffindbar sind.

Insgesamt wurden, wie in Listing 28 zu sehen, durch diese zwei regulären Ausdrücke 1570 Zeitangaben in dem gesammelten Datensatz unterdrückt.

[132] [Kle56]
[133] [HMU07] [Kapitel 3]
[134] [Mic]
[135] [The]
[136] Listing 28 ist im Anhang unter dem Kapitel *Statistiken des Werkzeugs: Comment-Dump-Analyzer* auffindbar.
[137] `https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html`

5.3.2.3 *Count*-Attribut einer *ProFormA-Id*

Um Bewertungsaspekte zu identifizieren vergibt Graja eine eindeutige Identifikationsnummer durch ein *Count*-Attribut innerhalb der *ProFormA-Id*. Sichtbar ist dies z.B. im Listing 22, wo `ProFormA-GradingHints:cnt=39!!@@` ein solches *Count*-Attribut darstellt. Dieses *Count*-Attribut ist durch eine Sequenznummer in der Klasse `GradingHintsConverter` modelliert. Es ist also bei mehrmaliger Ausführung keine eindeutig reproduzierbare Sequenznummer garantiert. In manchen Kommentaren werden diese *ProFormA-Id*s referenziert, was Rauschen zur Folge hat. Um diese Art von Rauschen zu unterdrücken, wird das *Count*-Attribut durch einen regulären Ausdruck neutralisiert.

Insgesamt wurden, wie in Listing 28 zu sehen, durch diesen regulären Ausdruck 584 *Count*-Attribute in dem gesammelten Datensatz unterdrückt.

5.3.3 Levenshtein-Distanz als Fehlertoleranz

Eine besondere Art des Rauschens ist in Listing 14 abgebildet. In beiden zu vergleichenden Datensätzen wurden jeweils zwei Dateipfade neutralisiert. Trotzdem schlägt der Abgleich fehl, da in einem Fall ein Doppelpunkt (`:`) und in dem anderen ein Semikolon (`;`) verwendet wurde, um die beiden Pfade voneinander zu separieren. Einen regulären Ausdruck zu definieren, der diese beiden Symbole rausfiltert, erscheint als unpraktisch, da diese Symbole nur im Kontext von Listing 14 zu Problemen führen. Möglich wäre ein zweiter Durchlauf der Rauschunterdrückung, der versucht, dieses Muster anhand des regulären Ausdrucks `_PATH_(:|;)_PATH_` zu neutralisieren. Dieser Ansatz würde allerdings bei großen Textmengen mindestens ein doppeltes Durchlaufen bedeuten. Des Weiteren müssten für alle ähnlichen Grenzfälle neue reguläre Ausdrücke definiert werden.

```
1 << SOURCE
2 Compiler options: [-encoding, UTF-8, -classpath, _PATH_:_PATH_]
3 >> SOURCE
4
5 << TARGET
6 Compiler options: [-encoding, UTF-8, -classpath, _PATH_;_PATH_]
7 >> TARGET
```

Listing 14: Fehlschlagende Rauschunterdrückung trotz Bereinigung durch reguläre Ausdrücke

Das Problem, das in Listing 14 illustriert wird, ist die Erkennung von Ähnlichkeiten zwischen zwei Zeichenketten. Dieses Problem ist durch reguläre Ausdrücke nicht zufriedenstellend lösbar. Weitere Anwendungsfälle würden z.B. Tippfehler darstellen.

- *Youve solved the exercise correctly!*
- *You've solved the exercise correctly!*

Die beiden oben dargestellten Zeichenketten verdeutlichen einen solchen Tippfehler. Bei einem Verhaltensabgleich wird dies als Differenz erkannt werden. Eine Korrektur von Tippfehlern gilt in diesem Fall allerdings nicht als erkannte Regression. Das Vermeiden von Neuaufzeichnung bei jeder Tippfehler-Korrektur steigert die Benutzerfreundlichkeit des Regressionstestmechanismus. Desweiteren sind solche Fehlalarme zeitaufwändig, da

erst eine Sichtung des Berichts und anschließend eine Neuaufzeichnung durchgeführt werden muss.

Um dies zu verhindern, kann auf den Mechanismus der Levenshtein-Distanz[138] zurückgegriffen werden. Die Levenshtein-Distanz erlaubt es, die minimale Anzahl an Einfüge-, Lösch- und Ersetzungs-Operationen zu akkumulieren, die notwendig sind um die erste Zeichenkette in die zweite umzuwandeln. Falls nun ein Abgleich zu einer Differenz führt, kann die Levenshtein-Distanz dieser Differenz berechnet werden. Daraufhin kann ein Schwellwert definiert werden, dieser wird mit der berechneten Differenz abgeglichen. Bei einer Überschreitung des Schwellwerts kann von einer möglichen Regression ausgegangen werden, ansonsten wird die erkannte Differenz ignoriert. Bei dem Beispiel in Listing 14 und dem des Tippfehlers ergibt sich jeweils eine Levenshtein-Distanz von 1. Diese ergibt sich in Listing 14 dadurch, eine *Ersetzung (':' => ';')* durchzuführen, während bei dem Beispiel mit dem Tippfehler eine *Einfügung (')* notwendig ist.

Um einen Schwellwert auszuwählen, können verschiedene Ansätze verfolgt werden. Ist nur eine Absicherung gegen Tippfehler erwünscht, sollte ein absoluter Schwellwert im Wertebereich von [1, 5] reichen. Möglich ist auch ein relativer Schwellwert, der durch einen prozentualen Anteil des Mittelwerts der Längen beider zu vergleichenden Zeichenketten gebildet werden kann. Es ist in diesem Teil der Arbeit nicht möglich, eine definitive Aussage über die beste Wahl eines Schwellwerts zu treffen. Für die Referenzimplementierung ist sicherlich eine globale und aufgabenbasierte Schwellwertkonfiguration sinnvoll. Somit können Ausreißer, die möglicherweise einen höheren Schwellwert benötigen, behandelt werden, ohne möglicherweise korrekt erkannte Regressionen in anderen Aufgaben zu unterdrücken. Eine Schwachstelle dieser Schwellwertoperation besteht darin, bei einem zu hoch gesetzten Schwellwert potenzielle Differenzen zu übersehen. Es sollte daher also niedriger globaler Schwellwert gewählt werden.

5.3.4 Fehlalarme durch unvorhersehbare Abarbeitungsreihenfolgen

Einige Fehlalarme entstanden durch unvorhersehbare Abarbeitungsreihenfolgen in Graja während der Generierung der Kommentare. So existieren im ersten Datensatz `Daten/CommentDump.zip` für gleiche Musterlösungen unterschiedliche Kommentare des Testmoduls PMD. Diese Kommentare haben den gleichen Inhalt (Auflistung der verfügbaren PMD-Regeln) in einer unterschiedlichen Reihenfolge. Um dies zu verhindern, reicht es, die Regeln vor der Ausgabe durch das PMD Testmodul einmal alphabetisch zu sortieren. Ein ähnliches Problem ergab sich mit dem Modul Checkstyle. In einigen Fällen der Aufgabe *de.hsh.prog.factorsengine* kam es auf Linux und Windows-Systemen zu einer unterschiedlichen Abarbeitungsreihenfolge der Quelldateien und somit zu Kommentaren mit gleichem Inhalt in vertauschter Reihenfolge. Um diesen Fehlalarm zu beheben, reicht es ebenfalls aus, die Liste an zu verarbeitenden Dateien alphabetisch zu sortieren. Anschließend wurde ein zweiter Datensatz angefertigt, der durch diese Änderungen keine Fehlalarme mehr auslöst.[139]

5.3.5 Fehlalarme durch Zufallsgeneratoren innerhalb der Aufgaben

Andere Fehlalarme sind auf die Verwendung von Zufallsgeneratoren zurückzuführen, genauer gesagt auf die Klasse `Random`[140] des JDK, die von einigen Aufgaben verwendet

[138] [Nav01] [S. 5, 7-8]
[139] Unter `Daten/CommentDumpAfterRngChanges.zip` im elektronischen Anhang auffindbar.
[140] [Orad]

wird. In der im Kapitel 5.3.2 durchgeführten Datenanalyse kam es zu solchen Problemen bei den Aufgaben *de.hsh.prog.detectzip* (Listing 15) und *de.hsh.prog.serialize* (Listing 16). Wie in den Listings zu sehen ist, nutzt *detectzip* ein *Random*-Objekt um einen Dateinamen zu generieren. Die Aufgabe *serialize* hingegen generiert zufällige Objektstrukturen, um die studentische Einreichung einem Funktionstest zu unterziehen.

```
1 << SOURCE
2 Optional[Calling 'java DetectZip lctoqdmprflfebotjjoq' where
     the parameter denotes a file starting with the bytes 4B, 50,
     i. e. the opposite byte order.]
3 >> SOURCE
4
5 << TARGET
6 Optional[Calling 'java DetectZip hqtghokljbbfkqomoifj' where
     the parameter denotes a file starting with the bytes 4B, 50,
     i. e. the opposite byte order.]
7 >> TARGET
```

Listing 15: Erkannte Differenzen durch `Random`-Nutzung in der Aufgabe *de.hsh.prog.detectzip*

```
1 << SOURCE
2 Juri mag Hanna
3 >> SOURCE
4
5 << TARGET
6 Juri mag Moritz
7 >> TARGET
8
9 ...
```

Listing 16: Erkannte Differenzen durch `Random`-Nutzung in der Aufgabe *de.hsh.prog.serialize*

Reguläre Ausdrücke und auch die Levenshtein-Distanz stellen hier keine zufriedenstellende Lösung dar. Bei regulären Ausdrücken müsste für die Aufgabe *serialize* z.B. der Ausdruck `[A-Za-z]+ mag [A-Za-z]+` definiert und in die Rauschunterdrückung mitaufgenommen werden. Dieser Ansatz ist aus vielerlei Hinsicht problematisch. Erstens gehen somit Informationen, die mögliche Regressionen aufdecken können, verloren. Außerdem handelt es sich hier nicht um Rauschen, sondern um korrektes aufgabenspezifisches Verhalten. Zuletzt ist das Pflegen von regulären Ausdrücken auf Aufgabenbasis nicht nur aufwändig, sondern stellt auch eine ungewollte Abhängigkeit zwischen Regressionstestmechanismus und Aufgabe dar. Die Verwendung der Levenshtein-Distanz eignet sich auch, um den in Listing 15 dargestellten Fehlalarm zu behandeln. Mit einem Schwellwert von 20 können komplett unterschiedliche randomisierte Dateinamen unterdrückt werden. Dieser Lösungsansatz führt allerdings wieder zu einer engen Kupplung zwischen Levenshtein-Distanz und Aufgabe. Des Weiteren werden nun z.B. keine randomisierten Dateinamen mit mehr als 20 Zeichen behandelt. Es können so außerdem auch mögliche Regressionen wie z.B. ein Nichtübereinstimmen des Inhalts zweier Kommentare mit einer Levenshtein-Distanz kleiner gleich 20 unterdrückt und ungewollt als Fehlalarm markiert werden.

Die hier beschriebene Problematik lässt sich nicht durch die bisher diskutierten Mechanismen behandeln. Daher sollte der Fokus darauf gelegt werden, im Kontext der Regressionstests eine determinierte Generierung von Zufallswerten anzustreben. Dies ist mithilfe des Pseudozufallszahlengenerator der Klasse `Random` möglich, der nach Setzung eines initialen Startwerts (*seed*) eine Folge an zufällig wirkenden aber determiniert berechenbaren Werten erzeugt. Der *Random*-Standardkonstruktor erzeugt einen von der Systemzeit abhängigen *seed*. Es ist allerdings auch möglich, eine `Random`-Instanz mit einem selbst definierten *seed* zu erzeugen.

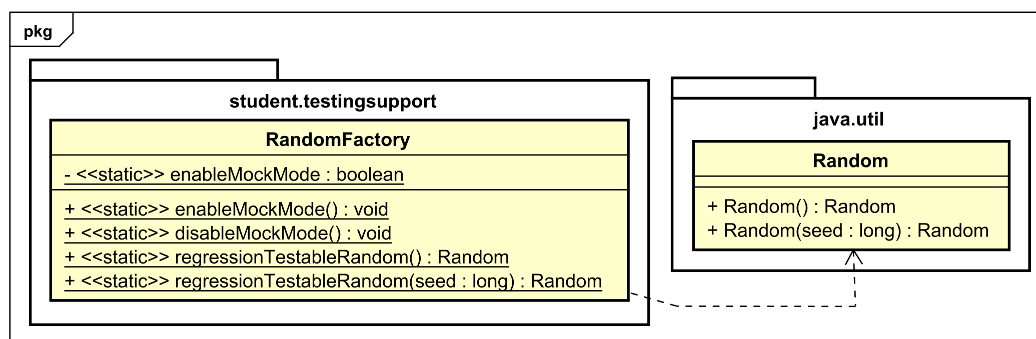


Abbildung 15: *Factory* um `Random`-Instanzen mit deterministischen *seed* anzubieten

Anstatt `Random`-Instanzen mit dem Standardkonstruktor zu erstellen, muss ein Mechanismus entstehen, der einen deterministischen *seed* garantiert. Hierbei bietet sich das *Factory-Pattern*[141] an. Wie in Abbildung 15 sichtbar, wird die Klasse `RandomFactory` neben anderen vom JUnit-Testmodul zum Testen von Einreichungen verwendeten Werkzeuge im Paket `student.testingsupport` eingeführt. Mithilfe der statischen Funktionen `enableMockMode` und `disableMockMode` kann das Verhalten der `RandomFactory` gesteuert werden. Bei aktiviertem *mock mode* werden neue *Random*-Objekte immer mit dem *seed* -1 initialisiert. Außerhalb des Regressionstestmechanismus ist der *mock mode* deaktiviert und garantiert so *Random*-Objekte mit entweder einem eigenen oder dem von der Systemzeit abhängigen *seed*. Dadurch weisen die Aufgaben zum Zeitpunkt eines Regressionstests immer ein gleiches `Random`-Verhalten auf. Es müssten bei diesem Ansatz lediglich alle Aufrufe der `Random`-Konstruktoren im Grader-Code aller Aufgaben durch die der `RandomFactory` ersetzt werden.

Über alle Aufgaben hinweg ergeben sich hier zwei Muster, um diese Änderungen durchzusetzen. Aufgaben wie z.B. *de.hsh.prog.bruch* verwenden die Klasse `Random` einmalig, um durch einen *static initializer* Zufallswerte zu bestimmen (Listing 17). Hier reicht es aus, den Standardkonstruktor mit der *Factory* auszuwechseln.

```
1 // Vorher
2   static {
3       Random random= new Random();
4       ... // stellt weitere Operationen mit random dar
5   }
6 // Nachher
7   static {
8       Random random= RandomFactory.regressionTestableRandom();
```

141 [GHJV95] [S. 107-116]

```
9         ...
10    }
```

Listing 17: Einmalige Verwendung der Klasse `Random` in einem statischen Initialisierer

Andere Aufgaben wie z.B. *de.hsh.prog.detectzip* hingegen nutzen ein *Random*-Objekt mehrmals in unterschiedlichen Testmethoden. Dies ist problematisch, da JUnit 4 standardmäßig keine deterministische Testausführung garantiert[142]. Somit ist bei nur einem *Random*-Objekt nicht garantiert, dass eine Testmethode immer mit dem gleichen Zufallswert arbeitet. Um das Problem zu lösen, sollte eine mit `@Before`[143] annotierte Methode eingeführt werden, die vor jedem Test ein neues *Random*-Objekt erstellt.

```
1 // Vorher (detectzip verwendet z.B. einen eigenen seed)
2   private static Random rnd= new Random(new Date().getTime());
3 // Nachher (fuehrt im mock mode zur Verwendung des seeds -1)
4   private static Random rnd;
5   @Before void initRandom() {
6       rnd = RandomFactory.regressionTestableRandom(new Date().
           getTime());
7   }
```

Listing 18: Mehrfache Verwendung der Klasse `Random` durch Testmethoden

5.3.6 Fehlalarme durch Zufallsgeneratoren innerhalb der Einreichungen

Einige Aufgaben sind so gestellt, dass die studentische Einreichung auf den Pseudozufallszahlengenerator der Klasse `Random` zugreifen muss. So verwendet z.B. die Aufgabe *de.hsh.prog.randomtext* in sämtlichen Musterlösungen ein *Random*-Objekt um zufällige Zeichenketten zu generieren. In solchen Fällen ist es nicht möglich den in Kapitel 5.3.4 vorgestellten Mechanismus der *Factory* zu verwenden.

Eine Möglichkeit, um dieses Problem zu umgehen, wäre es, studentische Einreichungen über den *Abstract Syntax Tree* (AST) des Quellcodes auf Konstruktoraufrufe der Klasse `Random` zu untersuchen. Sollten sich solche Konstruktoraufrufe finden, könnte mithilfe einer Bibliothek zum Manipulieren von *Bytecode* wie z.B. *ByteBuddy*[144] eine Ersetzung des Konstruktoraufrufes durch den Aufruf der *Factory*-Funktionen der Klasse `RandomFactory` geschehen.

Dies ist allerdings eine möglicherweise fehlerträchtige[145], recht aufwändige und komplexe Lösung, da lediglich zwei Aufgaben[146] die Klasse `Random` im Kontext der studentischen Einreichung verwenden. Deshalb sollten, falls solche Aufgaben zu Fehlalarmen führen, diese entweder vom Regressionstestmechanismus ausgeschlossen werden oder ein höherer aufgabenspezifischer Levenshtein-Schwellwert gewählt werden, um solche durch Zufallsgeneratoren erstellten Zeichenketten zu ignorieren.

[142] [JUnc]
[143] [JUnb]
[144] `https://bytebuddy.net/`
[145] Mögliche versteckte Fehler beim Manipulieren von *Bytecode* sind später schwer zu diagnostizieren.
[146] *de.hsh.prog.randomtext* und *de.hsh.prog.zahlenarray*

5.3.7 Musterlösungen mit mehreren validen Ausführungssträngen

Einige Musterlösungen[147] weisen mehrere Ausführungsstränge auf, die zu unterschiedlichen korrekten Endergebnissen führen. Eine Rauschunterdrückung macht bei solchen Musterlösungen keinen Sinn, da es sich nicht um Rauschen handelt. Um mit solchen Musterlösungen umzugehen, wird das Konzept des *multi sample mode* eingeführt. Der Regressionstestmechanismus sollte, sofern erwünscht, eine vom Nutzer für die Musterlösung definierte Anzahl an Aufzeichnungen erstellen und davon nur die Menge der eindeutigen Aufzeichnungen behalten. Bei einem späteren Verhaltensabgleich in Kapitel 5.4 reicht es dann aus, wenn das zum Zeitpunkt eines Regressionstest aufgezeichnete Ist-Verhalten mit mindestens einem Soll-Verhalten aus dieser Menge übereinstimmt.

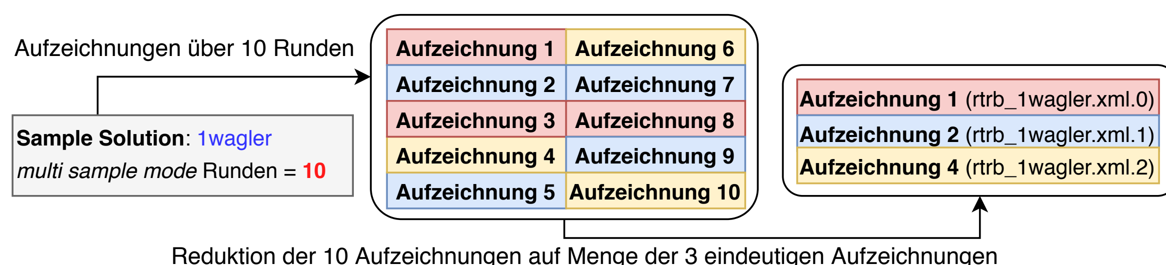


Abbildung 16: Funktionsweise des Modus *multi sample mode* beim Aufzeichnen

In Abbildung 16 ist die Funktionsweise des *multi sample mode* grafisch illustriert. Um die eindeutigen Aufzeichnungen abzuspeichern und als *multi sample mode*-Aufzeichnungen zu markieren, wird die Null indizierte Position der Aufzeichnung innerhalb der Menge eindeutiger Aufzeichnungen an die Dateiendung angehängt. Diese Position wird zusätzlich noch im Attribut *recordingVariation* des in Kapitel 5.2.4 behandelten Datenmodells (Abbildung 14) abgespeichert. Für Aufzeichnungen, die nicht durch den *multi sample mode* entstanden sind, wird dort die Null als Standardwert verwendet. Für den Regressionstestmechanismus ist das *recordingVariation*-Attribut notwendig um das aufgezeichnete Verhalten bei der Generierung des Testberichts auseinanderzuhalten.

Abschließend muss noch erwähnt werden, dass der *multi sample mode* nicht automatisiert eingeschaltet wird. Schlägt ein Regressionstest für eine Musterlösung direkt nach der Aufzeichnung ohne Veränderungen an Graja fehl, ist dies ein guter Indikator dafür, dass der *multi sample mode* aktiviert werden sollte. Bei der Reduktion auf die Menge der eindeutigen Aufzeichnungen sollte außerdem die für die Musterlösung gewählte Levenshtein-Distanz bezüglich des Textinhalts der Kommentare beachtet werden, um die Semantik der eindeutigen Aufzeichnungen beizubehalten.

5.4 Verhaltensabgleich: Beobachtetes Ist-Verhalten mit Soll-Verhalten der persistierten Verhaltensaufzeichnung

Nachdem in Kapitel 5.3 sowohl der Umgang mit Fehlalarmen, als auch die Identifikation und Unterdrückung von Rauschen in Verhaltensaufzeichnungen behandelt wurde,

[147] z.B. die Musterlösung *wrong_concurrentModificationException* der Aufgabe *factorsengine* oder die Musterlösung *1wagler* der Aufgabe *fibonaccizwischenergebnisse* im Namensraum *de.hsh.prog*. Bei der Musterlösungen der Aufgabe *factorsengine* gibt es zwei mögliche Endergebnisse, einmal mit und ohne Zeitüberschreitung. Bei der Musterlösung *1wagler* gibt es drei mögliche Endergebnisse, hier ist jeweils die Anzahl der *„Runtime-Factor ...“* Kommentare unterschiedlich. Auffindbar sind die unterschiedlichen Feedback-Dokumente der Ausführungsstränge unter dem Verzeichnis `Sonstiges` des elektronischen Anhangs.

gilt es nun festzulegen, wie der Verhaltensabgleich des aufgezeichneten Soll-Verhaltens mit dem zum Zeitpunkt eines Regressionstests beobachteten Ist-Verhaltens mithilfe des Datenmodells geschehen kann[148]. Verletzungen dieses Abgleichs werden im weiteren Verlauf der Arbeit als potenzielle Regressionen behandelt. Ob solche durch Ungleichheiten im Verhaltensabgleich erkannten Verletzungen letztendlich Regressionen darstellen, kann allerdings nur durch einen Tester, der den später in Kapitel 5.5.3 beschriebenen Testbericht sichtet, entschieden werden. Wie in Kapitel 5.3.7 beschrieben, werden sämtliche Verhaltensabgleiche auch auf alle durch den *multi sample mode* entstandenen Aufzeichnungen angewandt. Bezüglich des *multi sample mode* ist die Bedingung zum fehlerfreien Durchlaufen der Regressionstests dadurch gegeben, dass mindestens ein Verhaltensabgleich ohne potenziell erkannte Regressionen geschieht.

Eine notwendige Grundvoraussetzung des Verhaltensabgleichs ist außerdem, dass sowohl die Aufzeichnung des Soll-Verhaltens, als auch die des Ist-Verhaltens unter der gleichen maximal erreichbaren Punktzahl der Aufgabe entstanden ist. Dies wird durch des Attribut *scoreMax* der Klasse `RegressionTestingRecordedBehaviorTO` des Datenmodells geprüft. Bei einer Nichtübereinstimmung muss der Regressionstest automatisch abgebrochen werden, da sonst mögliche Fehlalarme durch Kommentare, welche die absolute Punktzahl enthalten, entstehen können. Dadurch wäre keine reproduzierbare Testumgebung gegeben und der Regressionstest nicht-deterministisch.

Die wichtigsten Verhaltensabgleiche des Regressionstestmechanismus sind die der Kommentare und des Bewertungsschemas, die in Kapitel 5.4.3 und 5.4.4 behandelt werden. Vorher muss allerdings noch der Vergleich von Baumstrukturen in Kapitel 5.4.2 behandelt werden. Ein geringerer Fokus wird auf den Abgleich von Bewertungsabbrüchen in Kapitel 5.4.1 gelegt.

5.4.1 Behandlung von Bewertungsabbrüchen

In der im Kapitel 5.3.1 durchgeführten Datenerhebung führte das Fehlschlagen der Musterlösung *wrong_endlessLoop* der Aufgabe *de.hsh.prog.maxfromcmdline* zu einer Textdatei im Megabyte-Bereich. Diese Textdatei unterschied sich in der Größe außerdem massiv zwischen Windows und Linux[149]. Eine Suche nach Differenzen innerhalb dieser Dateien gestaltet sich als ineffizient, da sie zum einen sehr rechenaufwändig ist und zum anderen zu Fehlalarmen, die nicht trivialerweise behandelt werden können führt. Um bei einem Bewertungsabbruch einen genaueren Vergleich durchzuführen, müsste ein neues Datenformat eingeführt werden, das nur notwendige Informationen wie z.B. den Bewertungsabbruch in Form der geworfenen *Exception* enthält. Dieser Ansatz wird allerdings in der Arbeit nicht näher verfolgt, da Musterlösungen, die in Graja zu einem Bewertungsabbruch führen, nur eine kleine Menge aller Musterlösungen ausmachen. Außerdem steht dieser Ansatz mit dem in Kapitel 3.2 formulierten Nicht-Ziel *„Genaue Lokalisierung möglicher Fehlerquellen innerhalb des Quellcodes für alle entdeckten Regressionen“* im Konflikt. Sollte sich der Wert des *error*-Attributs der Klasse `RegressionTestingRecordedBehaviorTO` des Datenmodells unterscheiden, ist klar, dass hier wahrscheinlich eine potenzielle Regression erkannt wurde, da eine Aufzeichnung beim Bewertungsvorgang fehlschlägt, während die andere ohne Probleme

[148] Abbildung 14 in Kapitel 5.2.4

[149] 32 Megabyte (Windows) gegenüber 9 Megabyte (Linux). Vermutlich dadurch zu Erklären, dass der Windows-Hostrechner die Endlosschleife schneller ausführte als die Linux-VM.

durchläuft. Somit existiert ein binärer auf *wahr* und *falsch* operierender Erkennungsmechanismus für Bewertungsabbrüche durch Graja.

5.4.2 Struktureller Vergleich von Baumstrukturen auf Differenzen

Da die interne Repräsentation der Kommentare[150] eine baumartige Struktur aufweist, ist es für die Funktionsweise des Regressionstestmechanismus notwendig, einen Vergleich zweier Kommentarbäume auf Differenzen anhand des in Kapitel 5.2.4 behandelten Datenmodells durchzuführen.[151]. Für so einen Vergleich reicht es zum Finden der Differenzen nicht aus, rekursiv eine korrekt implementierte `equals`-Methode[152] des Kommentarbaum-Wurzelknotens aufzurufen, da diese nur einen booleschen Wert zurückgibt. Eine Detektion von Unterschieden, ohne einen Ansatzpunkt zu liefern, ist allerdings kaum hilfreich. Kommentarbäume können in bestimmten Fällen eine gewisse Größe (z.B. 2000 Knoten) erreichen. Ein einfaches *„zwischen diesen Bäumen wurden Unterschiede erkannt“* hilft bei der Erkennung von potenziellen Regressionen nicht, da die Informationen bezüglich der erkannten Unterschiede fehlen.

Es sollte möglich sein, strukturelle Unterschiede, also Unterschiede in der Anordnung der Knoten ohne Sicht auf den Knoten-Inhalt der beiden Bäume, zu detektieren und auszugeben. Unterschiede in der Struktur weisen darauf hin, dass redundante oder fehlende Knoten in einem Kommentarbaum des Soll/Ist-Verhaltens existieren. Dies kann ein Indikator dafür sein, dass der Quellcode, der die Kommentare generiert, zum Zeitpunkt des Regressionstests nicht das gleiche Verhalten wie zum Zeitpunkt der initialen Verhaltensaufzeichnung aufweist. Durch den in Kapitel 5.2.3.2 beschriebenen Mechanismus des *Stackdumps* kann so die Herkunft von Kommentarknoten im Quellcode, die innerhalb der Bäume einen gleichen relativen Pfad und eine widersprüchliche Anzahl von Kindern aufweisen, zwecks der Fehlersuche herausgefunden werden. Denkbar wäre so zum Beispiel bezüglich des Testberichts eine grafische Darstellung des Baums, der die fehlenden/redundanten Knoten des Soll/Ist-Verhaltens visualisiert. Ein inhaltlicher Vergleich bezüglich des Knoteninhalts, der in Kapitel 5.4.3 behandelt wird, kann außerdem nur geschehen, falls die zu vergleichenden Knoten in beiden Bäumen vorhanden sind.

5.4.2.1 Baum-Isomorphismus und AHU-Algorithmus

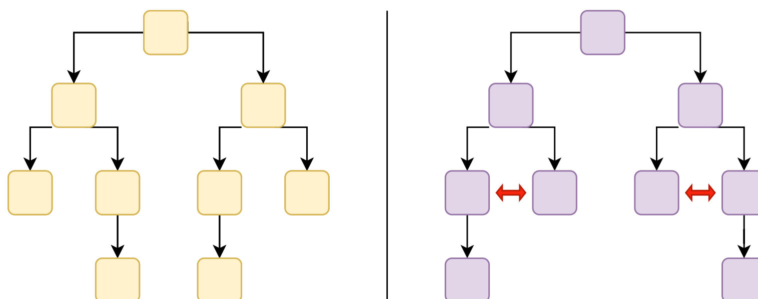


Abbildung 17: Beispiel zum Baum-Isomorphismus anhand zweier Bäume von unterschiedlicher Struktur

[150] Abbildung 10 in Kapitel 5.2.3
[151] In Falle des Regressionstestmechanismus z.B. ein Vergleich eines Kommentarbaums des Soll-Verhaltens mit dem des Ist-Verhaltens.
[152] [Orab]

Um einen für diesen Anwendungsfall spezifischen Algorithmus zu entwickeln, wird auf bereits existierende Forschung bezüglich des Themengebiets des Baum-Isomorphismus zurückgegriffen. Zwei Bäume gelten als isomorph, wenn ein Baum auf den anderen durch das Permutieren der Reihenfolge der Kinder der einzelnen Knoten abgebildet werden kann.[153]

Abbildung 17 enthält zwei Bäume, die sich isomorph zueinander verhalten. Durch das Permutieren der Reihenfolge der Kinder bestimmter Knoten (in der Abbildung mit einem roten Pfeil markiert), erreicht der zweite Baum eine Struktur, die äquivalent zu der des ersten Baums ist. Einen Baum-Isomorphismus zu erkennen ist bezüglich des Vergleichs von Baumstrukturen für den Regressionstestmechanismus allerdings nicht hilfreich. Zum Erkennen von möglichen Regressionen in der Baumstruktur des Kommentarbaums ist die Einhaltung der Reihenfolge der Kinder pro Knoten bedeutsam. Diese Eigenschaft erfüllt der Baum-Isomorphismus nicht, da hier die Reihenfolge der Kinder pro Knoten nicht beachtet wird. Der Isomorphismus gilt, solange pro Knoten eine Permutation der Kinder-Reihenfolge existiert, die eine Äquivalenz der Struktur der beiden abzugleichenden Bäume ermöglicht.

Interessant ist allerdings ein Algorithmus, der anhand zwei gegebener Bäumen einen Baum-Isomorphismus detektieren kann. Eine abgeänderte Art dieses Algorithmus kann genutzt werden, um Differenzen zu detektieren und auszugeben. Ein bereits existierender Algorithmus zum Erkennen von Baum-Isomorphismen ist der AHU Algorithmus[154]. Dieser operiert mithilfe von Knuth-Tupeln, die jedem Knoten des Baums zugeordnet werden. Es gibt für Knuth-Tupel zwei Schreibweisen, die in Abbildung 18 dargestellt sind. Die linke Schreibweise verwendet das Tupel *(0)* um Blätter zu markieren und Klammern, um die jeweiligen Tupel der Kinder eines Knotens zusammenzuführen. Die Klammern markieren den Anfang und das Ende eines Tupels. Die rechte Schreibweise, die im folgenden Verlauf der Arbeit verwendet wird, verzichtet bei Blättern auf die 0 in der Mitte des Tupels bildet die Klammern wie folgt ab:

- Öffnende Klammer: *(* $\mapsto$ *0*
- Schließende Klammer: *)* $\mapsto$ *1*

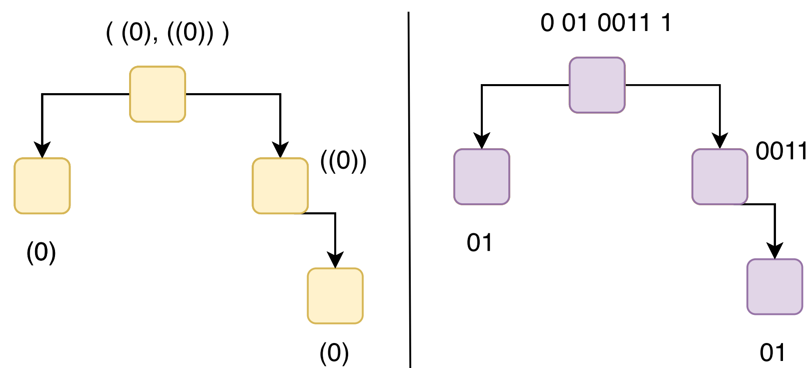


Abbildung 18: Beispiel eines Knuth-Tuples in zwei verschiedenen Schreibweisen

Die Tupel werden dann im Baum pro Knoten von unten nach oben vergeben. Angefangen wird an den Blättern des Baums, diese erhalten das Tupel *01*. Danach wird

[153] [AHU74] [3.2: S.84]
[154] [CR91] [S. 256-261]

pro Vorgänger-Knoten aller bereits markierten Knoten ein Tupel vergeben, indem die Tupel aller Kinder eingesammelt und mit Klammern umschlossen werden. Dies wird solange durchgeführt, bis der Wurzelknoten erreicht ist. Bei jeder Zusammenführung der Kinder-Tupel müssen diese vor der Konkatenation lexikographisch sortiert werden[155], um arbiträre Kinder-Reihenfolgen durch eine feste Tupel-Reihenfolge als äquivalent zu behandeln. Um nun herauszufinden, ob sich zwei Bäume isomorph zueinander verhalten, werden nach der Abarbeitung aller Knoten der Bäume die beiden Tupel der Wurzelknoten miteinander verglichen. Eine Äquivalenz derer bestätigt einen Baum-Isomorphismus.

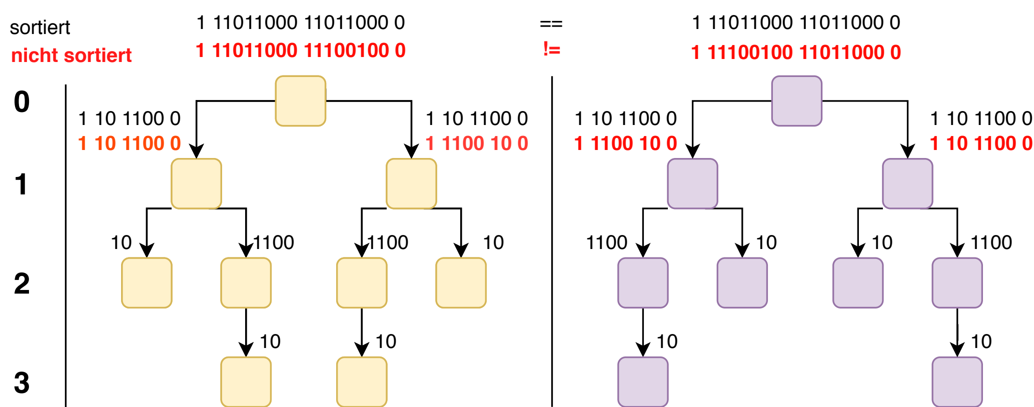


Abbildung 19: AHU-Algorithmus um einen Baum-Isomorphismus zu bestimmen

Abbildung 19 illustriert die praktische Anwendung des AHU-Algorithmus anhand zweier Beispiel-Bäume. Die rot markierten Tupel stellen jeweils eine Konkatenation ohne vorherige lexikographische Sortierung dar. Der hier vorgestellte Algorithmus eignet sich also auch, um Bäume unter der Beachtung der Kind-Reihenfolge miteinander abzugleichen. Ein Vergleich der roten Tupel des Wurzelknotens schließt so eine Äquivalenz aus. Problematisch an diesem Ansatz ist, dass nur die strukturelle Äquivalenz überprüft werden kann. Es ergibt sich allerdings kein Ansatzpunkt um die für eine Ungleichheit sorgenden Knoten innerhalb der Baumstruktur zu identifizieren.

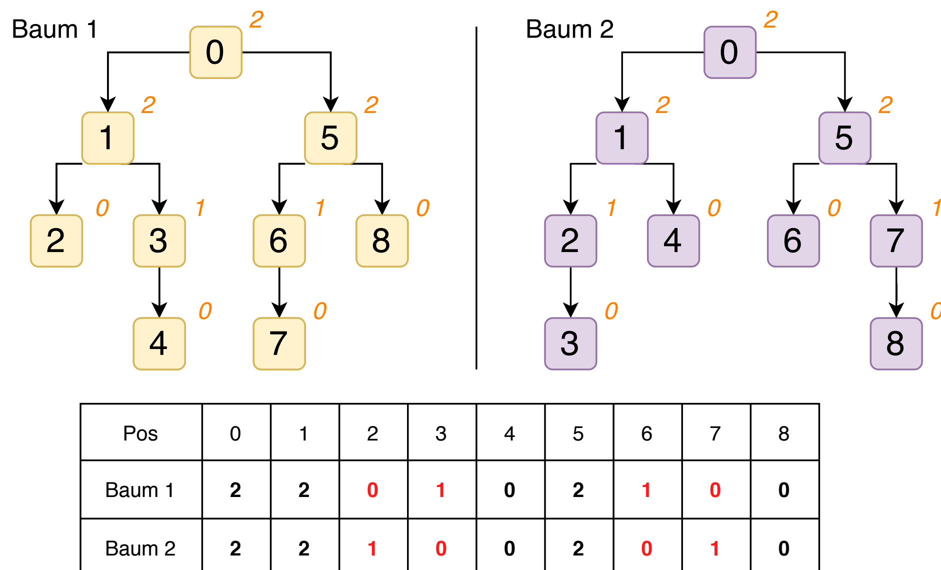
5.4.2.2 Überprüfung zweier Baumstrukturen auf strukturelle Äquivalenz

Durch einen ähnlichen Ansatz ist es allerdings möglich, die im AHU-Algorithmus verlorenen Daten abzugreifen und eine strukturelle Überprüfung unter Beachtung der Kind-Reihenfolge aller Knoten durchzuführen. In Abbildung 20 ist die Funktionsweise dieses Algorithmus an dem gleichen Beispiel-Baum verdeutlicht.

Bei diesem für die Arbeit entwickelten Algorithmus, wird ähnlich wie im AHU-Algorithmus pro Knoten die Anzahl der Kinder gespeichert. Anders als im AHU-Algorithmus existiert pro Kind allerdings keine rekursive Historie aller Kindes-Kinder-Knoten. Dieser Algorithmus arbeitet alle Knoten beider Bäume durch eine *Preorder*-Traversierung ab. Dabei existieren zwei mit Null startende Sequenzen, jeder Knoten wird bei der Traversierung mit einer eindeutigen Sequenznummer assoziiert[156]. Außerdem existiert pro Baum eine Liste aus Ganzzahlen[157]. Während der Traversierung des Baums wird

[155] Eine lexikographisch Sortierung fand in Abbildung 18 nicht statt, da diese nur dazu dient die beiden Schreibweisen vorzustellen.
[156] Dargestellt durch die schwarze Nummern in Abbildung 20.
[157] Dargestellt durch die Tabelle in Abbildung 20.

die Sequenznummer als Listenindex verwendet, um an dieser Stelle den Grad der ausgehenden Kanten (Kindern) des betrachteten Knotens[158] zu schreiben. Nachdem beide Bäume traversiert wurden, kann anhand der Listenlänge sofort festgestellt werden, ob die Struktur der Bäume äquivalent zueinander ist oder nicht[159]. Bei einer gleichen Listenlänge wird nun über alle Werte der Liste iteriert. Sollten sich die Werte der beiden Listen bei einem gleichen Index unterscheiden, stellen die Knoten mit dieser Sequenznummer Kandidaten für einen Unterschied in der Struktur der beiden Unterbäume dar[160]. Mittels dieser Kandidaten kann dann geprüft werden, ob an einer Position im Baum unterschiedliche Unterbäume vorliegen. Bezüglich der Kommentarabstraktion kann dies eine Regression darstellen, da an dieser Stelle entweder ein Kind-Kommentar fehlt oder redundant ist.



Pos	0	1	2	3	4	5	6	7	8
Baum 1	2	2	0	1	0	2	1	0	0
Baum 2	2	2	1	0	0	2	0	1	0

Abbildung 20: Algorithmus zur Bestimmung der Äquivalenz einer Baum-Struktur unter Beachtung der Kind-Reihenfolge

Dieser Algorithmus stützt sich auf drei Invarianten, um einen korrekten Vergleich zweier Bäume auf strukturelle Unterschiede zu gewährleisten:

- **Invariante 1:** Zwei Bäume B_1, B_2 mit gleicher Struktur haben die gleiche Anzahl an Knoten ($K \in \mathbb{N}_0$). Zwei Bäume sind nicht strukturell äquivalent wenn $B_{1_K} \neq B_{2_K}$.
- **Invariante 2:** Die pro Baum geführte Liste stellt aufgrund der *Preorder*-Traversierung und Vergabe eindeutiger Sequenznummern die surjektive Abbildung $g\colon S\ (Sequenznummer) \to G\ (Grad\ ausgehender\ Kanten\ pro\ Knoten) : s \mapsto g(s)$ mit dem Definitionsbereich $D_S = [0; K-1] \in \mathbb{N}_0$ und Wertebereich $W_G = \mathbb{N}_0$ dar.
- **Invariante 3:** Sei die Abbildung des ersten Baums A_1 und die Abbildung des zweiten Baums A_2, so gelten beide Bäume als strukturell äquivalent, falls für alle $s \in [0; K-1]$, gilt $A_1(s) = A_2(s)$.

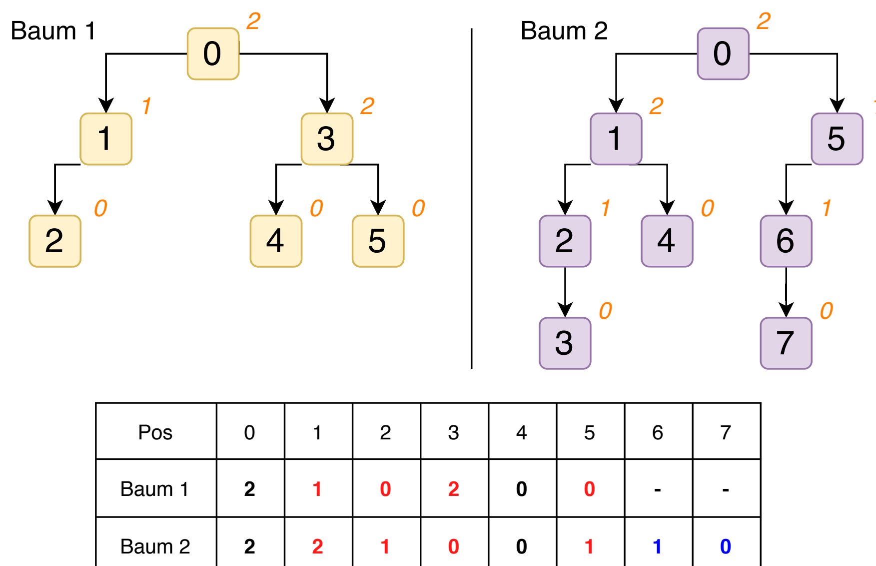
[158] Dargestellt durch die orangenen Nummern in Abbildung 20.

[159] Sollten die beiden Listen von widersprüchlicher Länge sein, können die beiden Bäume nicht von gleicher Struktur sein, da eine unterschiedliche Anzahl an Knoten vorliegt.

[160] Betonung auf Kandidaten, da nicht garantiert ist, dass zwei Knoten an der gleichen Position in beiden Bäumen mit den gleichen Sequenznummern adressiert sind.

Die in Abbildung 20 dargestellten Bäume verletzen die dritte Invariante an den Positionen *2, 3, 6, 7*.

Um das Verhalten des Algorithmus unter Bäumen mit ungleicher Knoten-Anzahl zu illustrieren, ist ein Beispiel durch Abbildung 21 gegeben. Die erste Invariante ist dadurch verletzt, dass beide Listen eine unterschiedliche Länge aufweisen. Zusätzlich findet an den Positionen *1, 2, 3, 5* eine Verletzung der dritten Invariante statt.



Pos	0	1	2	3	4	5	6	7
Baum 1	**2**	**1**	**0**	**2**	**0**	**0**	**-**	**-**
Baum 2	**2**	**2**	**1**	**0**	**0**	**1**	**1**	**0**

Abbildung 21: Algorithmus zur Bestimmung der Äquivalenz von Baum-Strukturen unter der Beachtung der Kind-Reihenfolge bei unterschiedlicher Knoten-Anzahl

5.4.2.3 Lokalisierung von Differenzen nach einem strukturellen Baum-Vergleich

Pos	0	1	2	3	4	5	6	7
Baum 1	**-1**	**0**	**1**	**0**	**3**	**3**	**-**	**-**
Baum 2	**-1**	**0**	**1**	**2**	**1**	**0**	**5**	**6**

Abbildung 22: Notwendige Backtracking-Informationen um absolute Pfade innerhalb des Baums, in dem eine Differenz detektiert wurde, zu rekonstruieren

Um im Testbericht eine strukturelle Differenz durch ein Verletzen der in Kapitel 5.4.2.2 definierten Invarianten zu visualisieren, muss ein relativer Pfad[161] zum verletzenden Knoten aus den generierten Daten des Algorithmus extrahiert werden. Um eine solche Extraktion durchzuführen, müssen zusätzliche Backtracking-Informationen gespeichert werden. Ein Beispiel für solche Backtracking-Informationen ist in Abbildung 22 gegeben. Hier wird pro Sequenznummer eines Knotens[162] jeweils die Sequenznummer des

[161] Ein relativer Pfad ist nur notwendig, um Knoten mit einer unterschiedlichen Anzahl von Kindern in beiden Bäumen zu adressieren. Somit können dann die Unterbäume ausgegeben werden, um zu verstehen, welche Kindes-Kinder fehlen oder redundant sind.

[162] In diesem Falle wieder als Listenindex gespeichert.

Vorgängerknotens gespeichert. Somit kann bei einer detektierten Differenz der absolute Pfad vom verletzenden Knoten bis zum Wurzelknoten des Baums durch ein Aufrollen rekonstruiert werden. Um einen Wurzelknoten zu markieren, wird ein Wert außerhalb des Definitionsbereichs der Sequenznummern wie z.B. *-1* verwendet.

Um die folgenden Kapitel bezüglich des Algorithmus zu beschreiben, muss zwischen zwei Arten von Baumpfaden unterschieden werden:

- **Sequenznummer-Pfad:** Pfad aus Knoten, der anhand derer Sequenznummern adressiert wird. Kann nicht verwendet werden, um einen Invarianten-verletzenden Pfad in einem anderen Baum zu adressieren, da die Sequenznummern möglicherweise pro Baum unterschiedlich vergeben sind und dieser Pfad somit einen absoluten Pfad darstellt.

  Schreibweisen-Beispiel: `1 -> 3 -> 4` (Starte beim Knoten mit der Sequenznummer `1`, rufe das Kind mit der Sequenznummer `3` auf, rufe das Kind mit der Sequenznummer `4` auf und lande somit beim adressierten Knoten des Sequenznummer-Pfads).

- **struktureller Pfad:** Pfad aus Knoten, der anhand der Kind-Positionen im Elternknoten angegeben ist. Stellt eine Wegbeschreibung zu einem Knoten dar, die erlaubt, einen Invarianten-verletzenden Knoten in beiden Bäumen zu adressieren. Ein struktureller Pfad ist somit als relativer Pfad zu betrachten.

  Schreibweisen-Beispiel: `[-1, 0, 1]` (Starte beim Wurzelknoten, der keine Position im Elternknoten besitzt. Besuche das erste Kind[163], besuche von diesem Kind aus dann das Kind an zweiter Position und lande somit beim adressierten Knoten des strukturellen Pfads).

5.4.2.4 Umwandlung zum Sequenznummer-Pfad

Mit einem gegebenen Startknoten[164] kann nun mit den im vorherigen Unterkapitel vorgestellten Backtracking-Informationen solange rückwärts über die Vorgängerknoten iteriert werden, bis der vorgängerlose Wurzelknoten erreicht ist (Listing 19).

```
1  fun backtrackPath(start: backtrack,
2                    backtracks: list<backtrack>) : list<node>
3
4      stack: stack<node>
5      stack.push(start.node)
6
7      while(start.previous != -1)
8          start = backtracks[start.previous]
9          stack.push(start.node)
10
11     return stack.asList()
```

Listing 19: Rekronstuktion eines Sequenznummer-Pfads mithilfe von Backtracking

[163] Wert: `0`, da Null indizierte Abspeicherung.
[164] Im Kontext dieses Algorithmus immer ein Knoten, der die dritte Invariante verletzt.

Die Anwendung des in Listing 19 aufgeführten Backtracking-Algorithmus ergibt für den Knoten mit der Sequenznummer *6* des zweiten Baums (Abbildung 21), der die dritte Invariante durch eine widersprüchliche Kinder-Anzahl verletzt, den Sequenznummer-Pfad `0 -> 5 -> 6` zurück. Im jetzigen Format kann dieser Sequenznummer-Pfad allerdings noch nicht genutzt werden, um in einem Testbericht strukturelle Unterschiede aufzuzeigen, da die Sequenznummern unterschiedlich vergeben sein können und in den beiden Bäumen nicht zwingend auf den gleichen strukturellen Pfad abbilden. Der Sequenznummer-Pfad `0 -> 5 -> 6` ist im ersten Baum der Abbildung 21 z.B. durch den widersprüchlichen Sequenznummer-Pfad `0 -> 3 -> 4` modelliert.

5.4.2.5 Umwandlung zum strukturellen Pfad

Um nun einen solchen Sequenznummer-Pfad in einen strukturellen Pfad zu konvertieren, muss für jeden Knoten zusätzlich die Indexposition im Elternknoten gespeichert werden.

```
1 fun calcOffsets(path: list<node>) : list<int>
2     offsets: list<int>
3 
4     for(node : path)
5         offsets.add(node.positionInParent())
```

Listing 20: Konvertierung: Sequenznummer-Pfad in strukturellen Pfad

Mit dieser Vorrausetzung kann dann anhand des Pseudocodes in Listing 20 ein Sequenznummer-Pfad in einen strukturellen Pfad konvertiert werden. Ein solcher struktureller Pfad wird als eine Liste aus Indexpositionen abgebildet und erlaubt so eine Adressierbarkeit des referenzierten Knotens in einem beliebigen Baum[165]. Die Anwendung von `calcOffsets` auf den im vorherigen Kapitel berechneten Sequenznummer-Pfad `0 -> 5 -> 6` führt zum strukturellen Pfad `[-1, 1, 0]`. Durch diesen strukturellen Pfad kann nun vom Wurzelknoten eines anderen Baums aus überprüft werden, ob der durch die Liste von Indexpositionen abgebildete Pfad in diesem Baum existiert. Falls vorhanden, kann eine Auflösung des Pfads auf den durch den strukturellen Pfad adressierten Knoten innerhalb eines beliebigen Baums geschehen[166]. Für den strukturellen Pfad `[-1, 1, 0]`, der aus dem Sequenznummer-Pfad des zweiten Baums errechnet wurde, ergibt sich der Sequenznummer-Pfad `0 -> 3 -> 4` in Baum 1 (Abbildung 21). Dadurch ist die Problematik der widersprüchlichen Sequenznummer-Pfade bei gleicher Struktur aus dem vorherigen Kapitel durch den strukturellen Pfad behandelt.

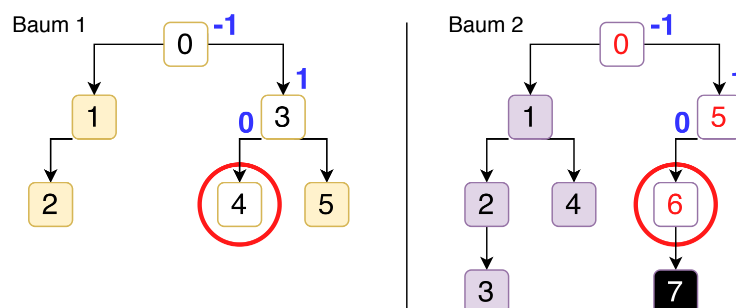


Abbildung 23: Illustrierte Anwendung der Algorithmen aus Listing 19 und 20

[165] Im Kontext dieses Algorithmus nur für den verglichenen Baum von Bedeutung.
[166] Der ungültige Index -1 signalisiert, dass dieser strukturelle Pfad vom Wurzelknoten aus startet.

In Abbildung 23 werden die beiden Umwandlungen aus Kapitel 5.4.2.4 und 5.4.2.5 nochmals grafisch illustriert. Die hier aufgrund einer widersprüchlichen Kinder-Anzahl betrachteten Knoten sind mit einem roten Kreis umrandet. Es ergeben sich die beiden Sequenznummer-Pfade `0 -> 3 -> 4` und `0 -> 5 -> 6`, die auf einen Knoten an der gleichen Position in den beiden zu vergleichenden Bäumen zeigen. Um diesen Knoten nun in beiden Bäumen zu adressieren, ist eine Umwandlung in einen strukturellen Pfad notwendig. Für beide Sequenznummer-Pfade ergibt sich der strukturelle Pfad `[-1, 1, 0]`, der in Abbildung durch die blaue Indexposition der Knoten im Elternknoten dargestellt ist.

5.4.2.6 Extraktion aller lokalisierbarer Differenzen

Knoten, die nun die dritte Invariante verletzen, stellen potenzielle Kandidaten für eine strukturelle Differenz zwischen den beiden Bäumen dar. Um solche potenziellen Kandidaten in eine Menge aus Differenzen zwischen den beiden Bäumen zu übersetzen, ist der in diesem Kapitel besprochene Algorithmus notwendig. In der folgenden Aufzählung werden einige notwendige Variablen und Definitionen eingeführt.

- Seien B_1 und B_2 die beiden zu vergleichenden Bäume.
- Seien L_1 und L_2 die Listen, welche die in Kapitel 5.4.2.2 besprochene Abbildung der Sequenznummern auf die Anzahl der Kinder pro Knoten, die für jeden Baum B geführt werden, darstellen.
- Seien l_1 und l_2 die Längen der beiden Listen L_1 und L_2.
- Definiere das Tupel $T_i = (S, K)$, bestehend aus der Sequenznummer eines Knotens S und der Anzahl der Kinder K, am Index i einer Liste.
- Sei D die Menge aller strukturellen Pfade, die auf Knoten mit unterschiedlicher Kinder-Anzahl in den beiden Bäumen B_1 und B_2 zeigen.

Der Algorithmus ist nun in der folgenden Auflistung beschriebenen (Einrückung der Tiefe 0 mit `-` und der Tiefe 1 mit `+` gekennzeichnet):

Für alle $i \in [0; \max(0, \min(l_0, l_1)\text{-}1)]$:

- Seien TB_1 und TB_2 die Tupel an der Position i in den für die beiden Bäume B_1 und B_2 geführten Listen L_1 und L_2.
- Backtracking: Sequenznummer-Pfade SP_1 und SP_2 für das S der beiden Tupel TB_1 und TB_2.
- Umwandlung: Strukturelle Pfade P_1 und P_2 aus den beiden Sequenznummer-Pfaden SP_1 und SP_2.
- P_1 zu D hinzufügen, falls die beiden strukturellen Pfade P_1 und P_2 übereinstimmen und das K der beiden Tupel unterschiedlich ist.[167]

[167] Bei den gleichen strukturellen Pfaden handelt es sich um die gleichen Knoten in den beiden Bäumen. Falls einer der beiden Bäume stark unterschiedlich balanciert ist, kann es dazu kommen, dass die Knoten mit der gleichen Sequenznummer nicht immer die gleichen strukturellen Pfade aufweisen und somit keine valide Differenz darstellen, da hier komplett unterschiedliche Knoten miteinander verglichen werden.

- Bei Nichtübereinstimmung[168] von P_1 und P_2 für beide strukturellen Pfade als P_i:
  + P_i zu D hinzufügen, falls der strukturelle Pfad in beiden Bäumen existiert und die Anzahl der Kinder beider referenzierter Knoten unterschiedlich ist.[169]
  + Ansonsten überspringen und nicht weiter behandeln.[170]

Mit dem oben beschriebenen Algorithmus lassen sich nun also alle Knoten zweier Bäume, die an einem gleichen strukturellen Pfad eine unterschiedliche Anzahl von Kindern aufweisen, ausgeben. Redundante oder fehlende Knoten werden allerdings nicht als Differenz behandelt. Das liegt daran, dass ein redundanten oder fehlender Knoten in einem Baum, bei einem Vergleich mit einem anderen Baum, automatisch zu einem Unterschied bezüglich der Kinder-Anzahl in einem der in beiden Bäumen existierenden Knoten mit gleichem strukturellen Pfad führt. Redundante oder fehlende Knoten wären sowieso auf diesen Knoten mit unterschiedlicher Kinder-Anzahl zurückzuführen. Außerdem ist durch dieses Verhalten garantiert, dass sich die Anzahl der erkannten Differenzen auch bei großen fehlenden/redundanten Teilbäumen in Grenzen hält. Bezüglich des Regressionstestmechanismus ist für Kommentarbäume von Bedeutung, an welchem Knoten ein Unterschied in der Struktur nachweisbar ist. Wenn z.B. zwei Kommentarbäume, einer mit 5 Knoten und der andere mit 3000 Knoten, miteinander verglichen werden, und sich herausstellt, dass in einem dieser Kommentarbäume ein großer Teilbaum fehlt, ist es übersichtlicher, nur den Knoten, an dem dieser fehlende Teilbaum anhängt, auszugeben. Ansonsten würden möglicherweise Tausende von Pfade, die alle in diesem Teilbaum beinhaltenden Knoten adressieren und auf den Knoten mit unterschiedlicher Kinder-Anzahl zurückzuführen sind, zu einer erschwerten Lesbarkeit des Testberichts beitragen. Im Rahmen des Regressionstestmechanismus ist es allerdings sinnvoll, den Inhalt aller Knoten eines solchen fehlenden oder redundanten Teilbaums in Textform auszugeben. Dies kann über die Menge, der durch strukturellen Pfade repräsentierten Differenzen geschehen.

Eine Implementierung, aller ab Kapitel 5.4.2.2 vorgestellten Algorithmen, ist im Abschnitt *„Pseudocode für einen Algorithmus um Baumstrukturen auf Äquivalenz zu überprüfen und die gefundenen Differenzen auszugeben“* des Anhangs unter Listing 29 vorzufinden. Diese Implementierung löst das in Kapitel 5.4.2 angesprochene Problem des strukturellen Vergleichs zweier Baumstrukturen durch das Ausgeben aller bestehenden Differenzen bezüglich der unterschiedlichen Anordnung beliebiger Kinder-Knoten in Form von strukturellen Pfaden.

[168] Dadurch, dass die strukturellen Pfade der beiden Kandidaten nicht übereinstimmen, müssen nun die Knoten der strukturellen Pfade innerhalb des anderen Baums herausgesucht werden, um, falls diese existieren, zu entschieden, ob es sich hier um eine Differenz durch eine unterschiedliche Anzahl von Kindern handelt.

[169] Dadurch, dass die beiden ursprünglichen Kandidaten nicht die gleichen strukturellen Pfade aufweisen und somit nicht vergleichbar sind, kann durch die einzelne Auflösung der beiden strukturellen Pfade in beiden Bäumen eine gleichwertige Überprüfung auf Differenzen durchgeführt werden.

[170] Falls ein struktureller Pfad eines solchen Kandidaten nicht in beiden Bäumen existiert oder die Anzahl der Kinder in den beiden Knoten übereinstimmt, stellt dies keine übernommene Differenz dar. Dadurch, dass ein Pfad nicht existiert, wird in einem der im Pfad enthaltenden Knoten ebenfalls eine widersprüchliche Anzahl an Kindern detektiert werden. Dies führt dazu, dass dieser Knoten in einem anderen Durchlauf als Differenz markiert wird.

5.4.3 Behandlung von Kommentarbäumen

Um zwei Kommentarbäume zu vergleichen, wird nun zuerst der in Kapitel 5.4.2 behandelte strukturelle Vergleich durchgeführt. Abschließend, findet ein zweiter Vergleich statt, der den Inhalt der einzelnen Knoten der in Abbildung 14 (Kapitel 5.2.4) definierten Kommentar-Abstaktion miteinander vergleicht. Dieser inhaltliche Vergleich kann natürlich nur zwischen Knoten durchgeführt werden, die in beiden Bäumen präsent sind. In allen Fällen wird das Attribut *className* verglichen, um sicherzustellen, dass Knoten mit einem gleichen strukturellen Pfad vom gleichen Typen sind. Für alle `ContentRepresentationTO`-Unterklassen werden außerdem noch spezielle klassenspezifische Vergleiche erstellt, die alle Attribute der Unterklassen mit einschließen. Da in der Aufzeichnungsphase bereits die in Kapitel 5.3.2 behandelte Rauschunterdrückung stattfindet, werden diese Vergleiche durch einen 1:1 Abgleich durchgeführt. Sollten Differenzen erkannt werden, wird auf diese die in Kapitel 5.3.3 vorgestellte Levenshtein-Distanz angewandt. Falls die berechnete Levenshtein-Distanz der beiden Zeichenketten kleiner-gleich dem definierten Schwellwert ist, gilt der Vergleich trotz eines fehlschlagenden 1:1 Abgleichs als bestanden. Unterschiede zwischen den beiden Baumstrukturen und Ungleichheiten bezüglich der Knoten-Inhalte stellen potenzielle Regressionen dar, die in den Testbericht mit einfließen. Ebenso kann auf den in Kapitel 5.2.3.2 besprochenen Mechanismus zur Lokalisierung eines Kommentars zurückgegriffen werden, um einen Ansatzpunkt bezüglich der Fehlersuche zu liefern.

5.4.4 Behandlung der Bewertungsaspekte und Aspektgruppen

Um Bewertungsaspekte und Aspektgruppen zu vergleichen, findet kein Abgleich der Baumstruktur durch den in Kapitel 5.4.2 vorgestellten Algorithmus statt. Stattdessen wird für jede Aufzeichnung die Menge aller Pfade der Knoten des Bewertungsschemas durch das *Parsen* der pro Knoten assoziierten *ProFormA-Id* generiert. Somit kann für den Wurzelknoten der Bezeichner *root* vergeben werden, während Aspektgruppen jeweils mit dem jeweiligen Gruppennamen adressiert werden. Bei Bewertungsaspekten wird das jeweilige Graja-Testmodul und falls vorhanden, die Test-Unterreferenz als Bezeichner vergeben[171].

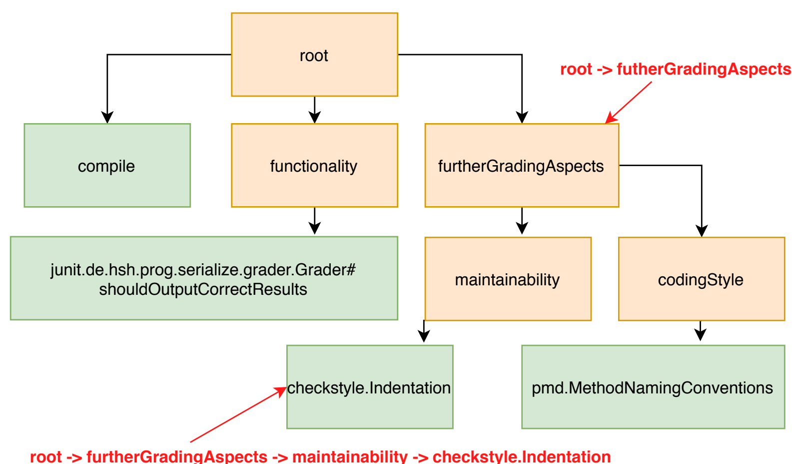


Abbildung 24: Teilausschnitt des in Listing 22 dargestellten Bewertungsschemas

[171]Einige Testmodule wie Compile verwenden momentan keine Unterreferenz, während beim Testmodul JUnit z.B. der jeweilige Methodenname des Tests als Unterreferenz verwendet wird.

In Abbildung 24 ist eine Baumstruktur aus Bewertungsaspekten und Aspektgruppen abgebildet. Gruppen sind orange eingefärbt, während Bewertungsaspekte, die einen ProFormA-Test referenzieren, die Farbe grün tragen. Zwei Beispiele für einen Pfad der Pfadmenge sind in rot dargestellt.

Als erste Voraussetzung muss nun für die Pfadmengen der Aufzeichnung P_1 und des zum Testzeitpunkt observierten Verhaltens P_2 gelten: $P_1 = P_2$. Danach werden die einzelnen Knoten des Datenmodells, die das Bewertungsschema abstrahieren[172] miteinander abgeglichen. Überprüft wird dabei pro Knoten jeweils die Erfolgsquote des Attributs *success* und die Menge der zurückgegebenen `Grade`-Statuscodes, die durch das Attribut *ratedAs* dargestellt sind. Da jeder Knoten außerdem noch vier Kommentarbäume enthält, werden diese durch den in Kapitel 5.4.3 behandelten Mechanismus behandelt. Leere Kommentarbäume werden allerdings nicht miteinander verglichen.

Sollte einer der genannten Abgleiche fehlschlagen oder die Mengen aller Pfade nicht übereinstimmen, ist eine potenzielle Regression detektiert, die in den Testbericht aufgenommen wird.

5.4.5 Eine Datenstruktur für den Testbericht

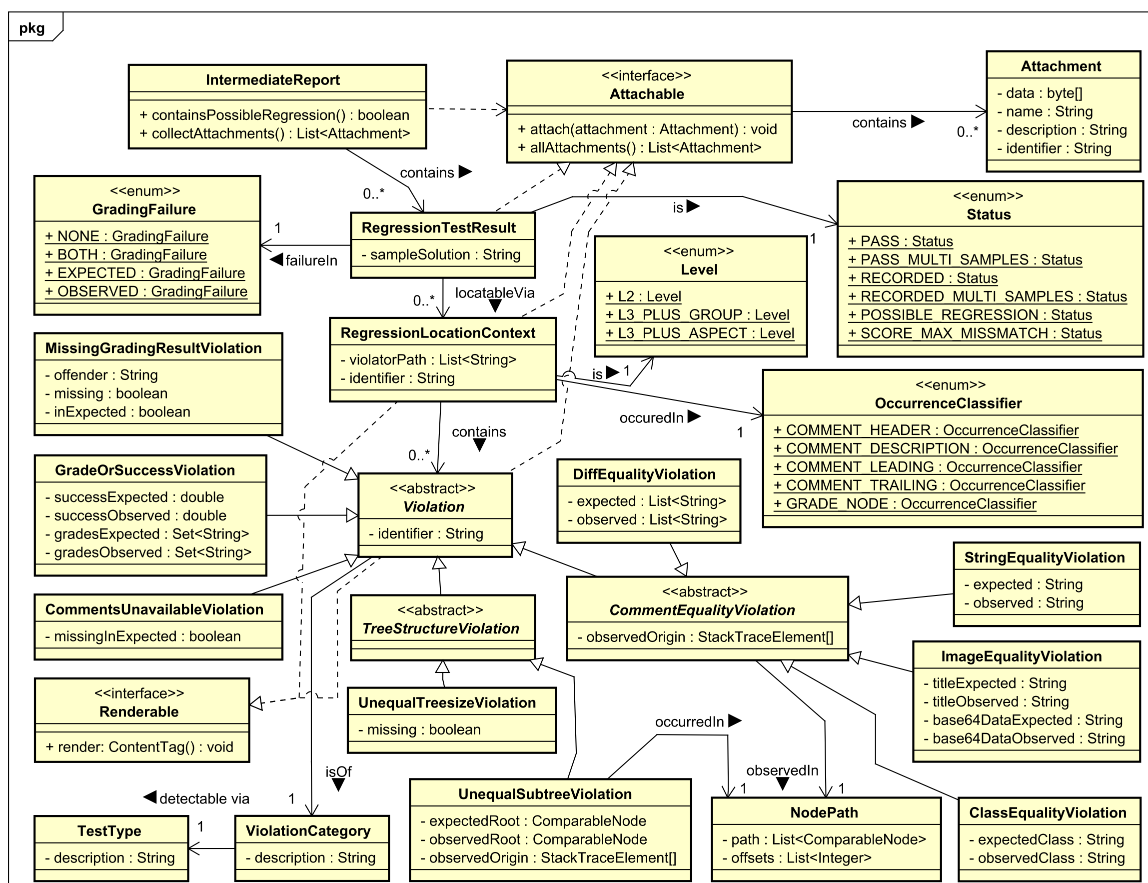


Abbildung 25: Datenstruktur für einen Testbericht

Um in Kapitel 5.5.3 einen übersichtlichen Testbericht zu generieren, ist eine Datenstruktur notwendig, die es erlaubt, die potenziell endeckten Regressionen zu gruppie-

[172] Abbildung 14 in Kapitel 5.2.4.

ren. Durch die Anwendung der in den vorherigen Kapiteln beschriebenen Vergleiche, werden so potenziell endeckte Regressionen für das Generieren eines Testberichts abgespeichert. Die für den Testbericht verwendete Datenstruktur ist in Abbildung 25 als UML-Klassendiagramm dargestellt und wird im folgenden Verlauf des Kapitels genauer erläutert und begründet.

Den Einstiegspunkt für diese Datenstruktur bietet die Klasse `IntermediateReport`, welche die Ergebnisse sämtlicher durchgeführten Regressionstests einer Aufgabe enthält. Die Klasse `RegressionTestResult` bildet einen abgeschlossenen Regressionstest ab. Das Attribut *sampleSolution* der Klasse stellt hier den Namen der im Regressionstest verwendeten Musterlösung dar.

Um den im Kapitel 5.4.1 beschriebenen Mechanismus des Abgleichs von Bewertungsabbrüchen zu realisieren, werden die im Aufzählungstypen `GradingFailure` definierten Statuscodes, die in Tabelle 3 näher beschrieben sind, eingeführt.

Statuscode	Beschreibung
`NONE`	Beide Aufzeichnungen wurden ohne Abbruch bewertet.
`BOTH`	Ein Bewertungsabbruch liegt bei beiden Aufzeichnungen vor.
`EXPECTED`	Differenz: Ist-Verhalten weißt keinen Bewertungsabbruch vor.
`OBSERVED`	Differenz: Soll-Verhalten weißt keinen Bewertungsabbruch vor.

Tabelle 3: Alle Statuscodes bezüglich des Vergleichs von Bewertungsabbrüchen

Um zusätzlich den Ausgang eines Regressionstests zu bestimmen, werden die in Tabelle 4 beschriebenen Statuscodes des Aufzählungstypen `Status` eingeführt. Mithilfe des `IntermediateReport` können so die Ergebnisse mehrerer Regressionstests in einem Testbericht zusammengefasst werden. Erstmalige Verhaltensaufzeichnungen werden der Einfachheit halber auch durch die Klasse `RegressionTestResult` und zwei spezifischen Statuscodes modelliert.

Statuscode	Beschreibung
`PASS_PRINT`	Erfolgreicher Testabschluss.
`PASS_MULTI_SAMPLES`	Erfolgreicher Testabschluss nach Übereinstimmung des beobachteten Verhaltens mit mindestens einer durch den *multi sample mode*[173]entstandenen Aufzeichnungen.
`RECORDED`	Erfolgreiche Aufzeichnung des Soll-Verhaltens.
`RECORDED_MULTI_SAMPLES`	Erfolgreiche mehrfache Aufzeichnung des Soll-Verhaltens unter der Verwendung des *multi sample mode*.
`POSSIBLE_REGRESSION`	Fehler, da potenzielle Regressionen erkannt.
`SCORE_MAX_MISSMATCH`	Fehler, da Änderung der maximalen Punktezahl[174].

Tabelle 4: Alle Statuscodes bezüglich des Ergebnis eines Regressionstests

Testberichte sollten als HTML-Dokument generiert werden, da so die größtmöglichste Freiheit bezüglich des Layouts und der Portabilität des Testberichts besteht. Die Schnittstelle `Renderable` gibt die Methode `render` vor, die ein Objekt vom Typen

[173] Behandelt in Kapitel 5.3.7.

[174] Grund: Keine reproduzierbare Testumgebung, da hartkodierte absolute Punktezahlen in z.B. Kommentaren als potenzielle Regressionen erkannt werden könnten. (Kapitel: 5.4)

`ContentTag` zurückgibt. `ContentTag` gehört zur Java-Bibliothek *j2html*[175], die eine *domain-specific language* (DSL) innerhalb Javas für das Erstellen von HTML-Dokumenten anbietet.

Mithilfe der Schnittstelle `Attachable` können außerdem Anhänge in Form von `Attachment`-Instanzen an implementierende Klassen angehängt werden. Diese sollten im HTML-Dokument des Testberichts durch *Hyperlinks* aufrufbar sein. Anhänge eignen sich z.B. um eine grafische Darstellung des Vergleichs zweier Kommentarbäume auf Differenzen in den Testbericht einzubetten.

5.4.5.1 Addressierbarkeit von Regressionen im Testbericht

Um die Herkunft einer entdeckten Regression zu adressieren, wird die Klasse `RegressionLocationContext` verwendet. Eine solche Instanz wird automatisch angelegt, sobald mindestens eine potenzielle Regression erkannt wurde. Die Menge aller Regressionen ist also über `RegressionLocationContext`-Instanzen in einem `RegressionTestResult` gruppiert. Eine Addressierbarkeit der enthaltenen Regressionen ist nun durch die Aufzählungstypen `Level` / `OccurrenceClassifier` und dem Attribut `violatorPath` gegeben.

Durch `Level` werden die in Kapitel 5.2.3.1 beschriebenen Ebenen dargestellt. Die Ebene *L2* stellt somit also z.B. das *AssignmentResult*-Objekt dar, während die beiden *L3+*-Ebenen für Knoten innerhalb des Bewertungsschemas verwendet werden. Um die in Kapitel 5.4.4 (Abbildung 24) behandelten Pfade der einzelnen Knoten des Bewertungsschemas darzustellen, wird das Attribut *violatorPath* verwendet. Im Falle einer *L2*-Regression würde dieses allerdings leer bleiben, da die Ebene *L2* nicht zum in Kapitel 2.3 behandelten Bewertungsschema gehört. Der Aufzählungstyp `OccurrenceClassifier` wird letztendlich dafür verwendet, die Regression innerhalb eines durch die Kombination der Ebene und des Pfads bestimmten Elements einzuordnen. Die ersten vier Konstanten beziehen sich auf die Positionsattribute der Kommentare (Kapitel: 5.2.3), während die Konstante `GRADE_NODE` auf eine erkannte Regression innerhalb eines *AbstractResultNode* hindeutet (Kapitel: 5.2.2, 5.4.4).

Das Attribut *identifier* ist bei der Generierung des Testberichts relevant, um durch *Hyperlinks*[176] auf verschiedene gerenderte `RegressionLocationContext`-Elemente zu verweisen. Für den Bezeichner selber wird, um Eindeutigkeit zu garantieren eine *UUID*[177] verwendet.

5.4.5.2 Kategorisierung von Regressionen im Testbericht

Um erkannte Regressionen zu kategorisieren und übersichtlich mit Kontextinformationen darzustellen, wird die Basisklasse `Violation` eingeführt. Eine erkannte Regression wird also als Verletzung der Gleichheit des Soll- und Ist-Verhaltens modelliert. Hier ist ebenfalls wie in Kapitel 5.4.5.1 für die Unterstützung von *Hyperlinks* ein *identifier*-Attribut vorhanden. Für jede Regression wird außerdem noch eine Beschreibung (`ViolationCategory`) und der erkennende Mechanismus (`TestType`) abgespeichert, um diese im Testbericht einfließen zu lassen.

[175] https://j2html.com/
[176] [RHJ] [Kapitel 12: Links]
[177] [LSM05]

5.4.5.2.1 Inhaltliche Regressionen der Kommentare

Um inhaltliche Regressionen bezüglich der Kommentare (Kapitel: 5.4.3) darzustellen, werden Unterklassen von `CommentEqualityViolation` verwendet. Durch das Attribut *observedOrigin*, kann nach Anwendung des im Kapitel 5.2.3.2 beschriebenen Mechanismus die Herkunft des Ist-Kommentars im Quellcode durch einen *Stacktrace* herausgefunden werden. Die Position im Kommentarbaum wird durch die Klasse `NodePath` gespeichert. Es gibt nun die folgenden Unterklassen:

- `ClassEqualityViolation`: Klassenname der Kommentarbausteine stimmen nicht überein, weitere Vergleiche können nicht durchgeführt werden.
- `DiffEqualityViolation`: Ein `DiffComment` stimmt inhaltlich nicht mit dem anderen `DiffComment` überein. Die Anwendung der Schwellwertoperation mittels der Levenshtein-Distanz schlägt außerdem fehl.
- `StringEqualityViolation`: Der Inhalt der Zeichenketten der Kommentare stimmt nicht überein. Die Anwendung der Schwellwertoperation mittels der Levenshtein-Distanz schlägt außerdem fehl.
- `ImageEqualityViolation`: Entweder stimmen die Titel nicht überein oder die *Base64* enkodierten Bilddaten unterscheiden sich. Die Anwendung der Schwellwertoperation mittels der Levenshtein-Distanz auf die Titel schlägt außerdem fehl.

Inhaltliche Vergleiche werden nur auf Knoten, die in beiden Kommentarbäumen existieren, angewandt.

5.4.5.2.2 Strukturelle Regressionen der Kommentarbäume

Um strukturelle Regressionen bezüglich der Kommentarbäume darzustellen, werden Unterklassen von `TreeStructureViolation` verwendet.

- `UnequalTreesizeViolation`: Ein Baum ist entweder größer/kleiner als der andere.
- `UnequalSubtreeViolation`: Es wurden Knoten endeckt, die am gleichen strukturellen Pfad im Baum eine unterschiedliche Anzahl an Kindern aufweisen (Kapitel: 5.4.2). Zusätzlich wird der Inhalt der beiden Unterbäume in Textform angegeben und als Anhang angehängt.

5.4.5.2.3 Anderwertige Regressionen

Alle verbleibenden Regressionen haben keine andere spezifische Oberklasse als `Violation`.

- `MissingGradingResultViolation`: Ein Knoten des Bewertungsschemas ist nicht im anderen Bewerungsschema auffindbar.

- `GradeOrSuccessViolation`: Entweder ist der Erfolgswert eines Knotens des Bewertungsschemas ungleich[178], oder die Menge der `Grade`-Statuscodes stimmt nicht überein.

- `CommentsUnavailableViolation`: Es ist nicht möglich zwei Kommentarbäume miteinander abzugleichen, da ein Kommentarbaum komplett leer ist, während der andere mindestens aus einem Knoten besteht.

5.5 Aufbau einer Test-Infrastruktur um Graja

Das letzte Unterkapitel behandelt den Aufbau einer Test-Infrastruktur, um den Regressionstestmechanismus als eine Erweiterung Grajas zu integrieren. Da der Regressionstestmechanismus mithilfe der Musterlösungen der einzelnen Aufgaben operiert, ist es sinnvoll, diese Integration durch das Erweitern des Gradle-*Build*-Skripts der Aufgaben zu realisieren. Es werden für die Test-Infrastruktur die folgenden Operationen definiert:

- `Test <Task>` - Führe für alle nicht-ignorierten Musterlösungen der Aufgabe aus:

  - Falls kein aufgezeichnetes Soll-Verhalten existiert: Ist-Verhalten aufzeichnen und als neues Soll-Verhalten persistieren.

  - Falls aufgezeichnetes Soll-Verhalten existiert: Ist-Verhalten aufzeichnen und mit dem Soll-Verhalten abgleichen.
- `Tests All` - Wie `Test <Task>`, aber auf alle Aufgaben angewandt.
- `Record <Task>` - Zeichne das Ist-Verhalten aller nicht-ignorierten Musterlösungen auf und persistierte dies als neues Soll-Verhalten.
- `Record All` - Wie `Record <Task>`, aber auf alle Aufgaben angewandt.
- `Clean <Task>` - Lösche sämtliches aufgezeichnetes Soll-Verhalten einer Aufgabe.
- `Clean All` - Wie `Clean <Task>`, aber auf alle Aufgaben angewandt.

Um diese Operationen im *Build*-Skript umzusetzen, werden *Gradle Tasks*[179] verwendet. Da die Aufgaben für den Regressionstestmechanismus schon im task.zip-Format vorliegen müssen, stehen die *Gradle Tasks* der Operationen `Record` und `Test` in Abhängigkeit zu den jeweiligen aufgabenspezifischen *Build-Tasks*. Somit ist automatisch garantiert, dass beim Aufruf eine ProFormA-konforme task.zip im *Build*-Ordner unter `build/distributions` vorhanden ist. Durch Gradles *Incremental Build*-Feature[180] entsteht außerdem kein Overhead durch diese *Gradle Task*-Abhängigkeit. Aufgaben werden nur neu überführt, falls Änderungen an den Quelldaten derer detektiert wurden.

In Abbildung 26 ist ein abstrahiertes[181] UML-Aktivitätsdiagramm der Operation `Test <Task>` zu sehen. Falls in der task.zip der Aufgabe keine Aufzeichnungen des Soll-

[178] Aufgrund des Datentyps `double` muss hier ein Vergleich geschehen, der die Ungenauigkeit von Gleitkommazahlen berücksichtigt.

[179] [Graa]

[180] [Grab] Abschnitt: Incremental build

[181] Der *multi sample mode* wird z.B. bereits im Bewertungsvorgang vorausgesetzt. Außerdem ist nicht gegenzeichnet, dass während des Transformationsvorgangs möglicherweise eine Kommunikation zwischen zwei JVMs stattfindet. Die Anwendung der einzelnen in Kapitel 5.4 behandelten Vergleiche ist auch nicht weiter beschrieben.

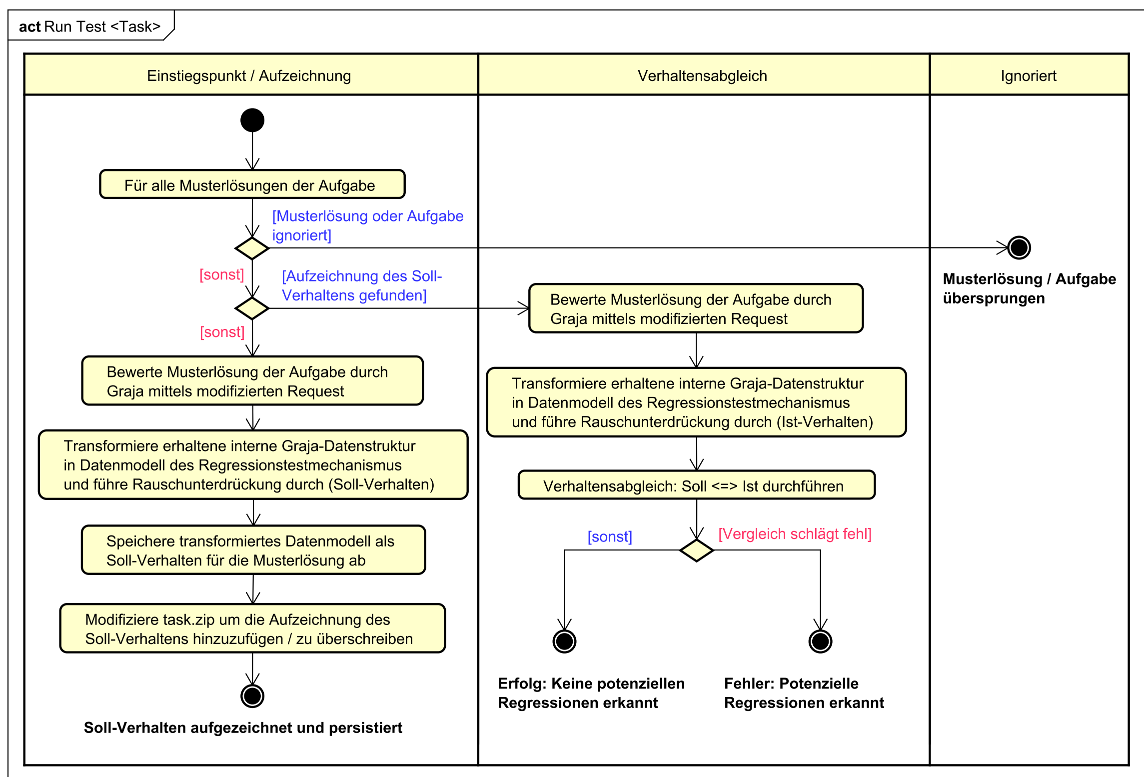


Abbildung 26: Abstrahiertes Aktivitätsdiagramm der Operation `Test <Task>`

Verhaltens auffindbar sind, werden diese in der *Swimlane „Einstiegspunkt / Aufzeichnung“* erstellt, persistiert und im Anhang der task.zip (Kapitel 2.5.2 und 5.1.4) angehängt. Bei einer bestehenden Aufzeichnung wird in der *Swimlane „Verhaltensabgleich“* ein Ist-Verhalten aufgezeichnet. Dieses durchläuft während der Verhaltensaufzeichnung ebenso wie das Soll-Verhalten die in Kapitel 5.3.2 vorgestellte Rauschunterdrückung und ist im Format des in Abbildung 14[182] vorgestellten Datenmodells. Anschließend findet der Soll-Ist-Abgleich statt. Bei erkannten potenziellen Regressionen gilt der Regressionstest als nicht bestanden. Für die Operation `Record <Task>` werden die nach den beiden Entscheidungspunkten anfallenden Schritte in der *Swimlane „Einstiegspunkt / Aufzeichnung“* ausgeführt.

Anschließend wird pro Ausführungsstrang ein Testbericht generiert. Für die Referenzimplementierung ist es notwendig, beim Fehlschlagen eines Regressionstests eine *Exception* zu werfen, um die Gradle-Instanz abstürzen zu lassen. So signalisiert die Referenzimplementierung dem Anwender, dass mögliche Regressionen detektiert wurden.

5.5.1 Schnittstelle für den Regressionstestmechanismus

Um den Regressionstestmechanismus aus Gradle heraus aufzurufen, sollte eine möglichst einfach gehaltene Schnittstelle angeboten werden. Dies lässt sich durch das *Facade-Pattern*[183] umsetzen. Um eine Feineinstellung des Regressionstestmechanismus auf Aufgabenbasis anzubieten, können optionale Parameter über die Aufgabenkonfigurationsdatei `assignment.properties`, die jeder Aufgabe beiliegt, gesetzt werden. Um diese

[182] Kapitel: 5.2.4
[183] [GHJV95] [S. 185-193]

Steuerungsparameter hervorzuheben, sollten diese mit dem Präfix *reg* adressiert werden, da das *Properties*-Format[184] keine Namensräume unterstützt.

Optionaler Parameter	Beschreibung
regGlobalLevenshtein `int >= 0`[185]	Überschreiben des globalen Schwellwerts der Levenshtein-Distanz für alle Musterlösungen dieser Aufgabe.
regLocalLevenshtein `String (eigenes Format)`[186]	Überschreiben des globalen Schwellwerts der Levenshtein-Distanz für spezifische Musterlösungen dieser Aufgabe.
regTimeout[187] `int > 0`	Überschreiben des globalen *Timeout*-Werts, der nach Überschreiten automatisch einen Bewertungsabbruch durch Graja zur Folge hat.
regIgnoreSampleSolutions `String im CSV-Format`[188]	Bestimmte Musterlösungen einer Aufgabe vom Regressionstestmechanismus ausschließen.
regIgnore `boolean`	Gesamte Aufgabe vom Regressionstestmechanismus ausschließen.
regRecordedSampleSizeRounds `int > 0`	Rundenzahl für Aufzeichnungsversuche des *multi sample mode*[189]definieren.
regRecordedSampleSizeRoundsFor `String im CSV-Format`	*multi sample mode* nur bei bestimmten Musterlösungen anwenden.

Tabelle 5: Alle optionalen Steuerungsparameter des Regressionstestmechanismus, die pro Aufgabe über die `assignment.properties` gesetzt werden können

Mit den in Tabelle 5 beschriebenen Parametern können nun alle Standardeinstellungen des Regressionstestmechanismus überschrieben werden. Der *multi sample mode* ist standardmäßig immer durch den Wert *1* deaktiviert. Ein Wert von *3* wird für die globale Levenshtein-Distanz verwendet, da dieser in der Mitte des in Kapitel 5.3.3 beschriebenen sinnvollen Intervalls von [1;5] liegt. Für einige Aufgaben oder Musterlösungen ist später möglicherweise ein höherer Wert notwendig. Ein Bewertungsabbruch findet außerdem ohne Überschreiben des globalen Werts nach 20 Sekunden statt.

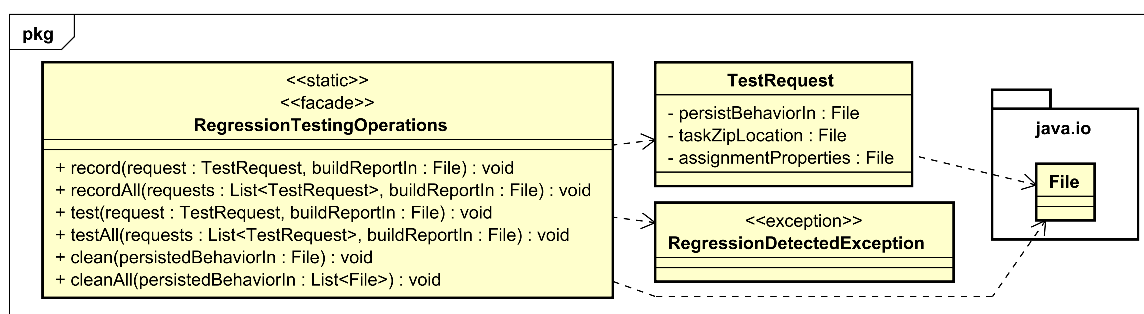


Abbildung 27: Öffentliche Schnittstelle des Regressionstestmechanismus

[184] [Orac]
[185] Ein Wert von 0 sorgt für ein Nichtanwenden der Levenshtein-Distanz-Schwellwertoperation.
[186] Mischung aus CSV und *Key => Value* Schreibweise. Beispiel: `solutionA => 4, solution_B => 30`. Musterlösung `solutionA` hat den Wert 4, `solution_B` hingegen 30.
[187] Wert in Sekunden.
[188] [Sha05]
[189] Behandelt in Kapitel 5.3.7.

Im UML-Klassendiagramm der Abbildung 27 ist die Realisierung einer solchen öffentlichen Schnittstelle mithilfe des *Facade-Patterns* in Form der Klasse `RegressionTesting Operations`, die hier die Fassade darstellt, abgebildet. Die Klasse `TestRequest` kapselt alle notwendigen Pfade, die für die Operationen `Record` und `Test` notwendig sind. Im Falle einer entdeckten Regression wird eine `RegressionDetectedException` geworfen, damit das *Build*-Skript abstürzt. Statt der *Path*-API[190] wird die *File*-API[191] verwendet, da diese in Gradle den empfohlenen Weg der Dateibehandlung darstellt[192].

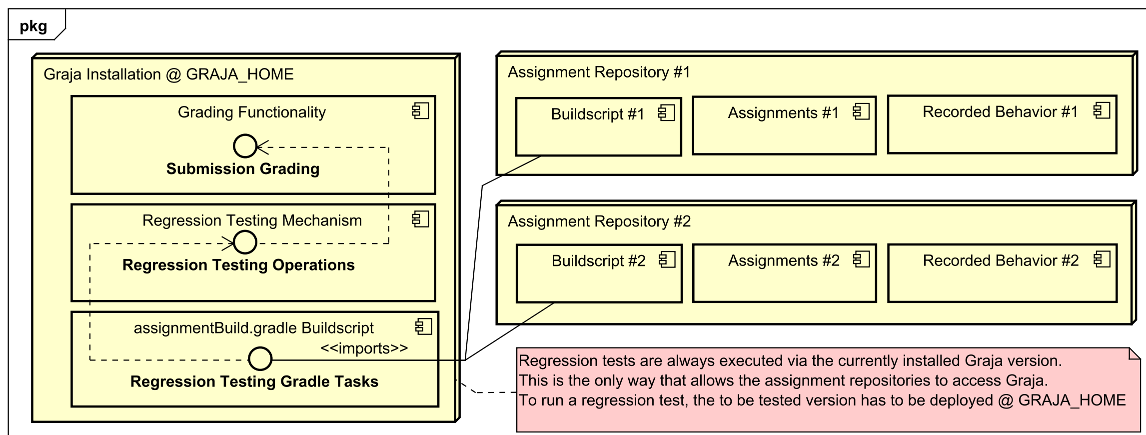


Abbildung 28: Verbindung von Aufgaben-Repositorien mit der Schnittstelle

Eine Verbindung dieser Schnittstelle mit den Aufgaben und Musterlösungen eines Ausgaben-Repositoriums, wie in Abbildung 28 dargestellt, ist nun anhand des `assignment Build.gradle` *Build*-Skripts möglich. Dieses wird durch die *Build*-Skripte aller Aufgaben-Repositorien importiert[193]. Das `assignmentBuild.gradle`-*Build*-Skript verweist immer auf die im *Graja Home*-Verzeichnis[194] befindliche Graja-Installation. Es ist für den Regressionstestmechanismus also notwendig, dass ein Tester die zu testende Version im *Graja Home*-Verzeichnis bereitstellt. Der Regressionstestmechanismus stellt somit also kein externes Werkzeug dar, sondern ein Graja-Untermodul, das durch eine öffentliche Schnittstelle angesprochen wird. Dadurch, dass aufgezeichnetes Soll-Verhalten in das VCS aufgenommen wird, ist dieses mit den enthaltenen Aufgaben des Aufgaben-Repositorium gekoppelt. Somit findet eine saubere Trennung zwischen dem Regressionstestmechanismus und der Regressionstests (Musterlösungen der Aufgaben und aufgezeichneten Soll-Verhalten) statt.

5.5.2 Umgang mit variablen Aufgaben

Bei variablen Aufgaben wird auf die Instanz, die mit Platzhalter-Standardwerten instanziiert wird, zurückgegriffen (Kapitel 2.5.4). Aufgrund des Nicht-Ziels *„Fuzzing mithilfe variabler Aufgaben“* findet keine spezielle Behandlung der zu variablen Aufgaben gehörenden Schablonen statt. Die Instanz mit Standardwerten gilt als äquivalent zu einer gewöhnlichen Aufgabe und kann deshalb für Regressionstests verwendet werden.

[190] `https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html`
[191] `https://docs.oracle.com/javase/8/docs/api/java/io/File.html`
[192] [Grac]
[193] [Gara] [2.1]
[194] [Garc] [2.1]

5.5.3 Generierung des Testberichts

Der abschließende Testbericht wird durch die Überführung der in Kapitel 5.4.5 behandelten Datenstruktur in das HTML-Format generiert. Bezüglich des Layouts wird sich an den durch Gradle erzeugten JUnit-Testberichten, zu sehen in Abbildung 29, orientiert. Das dort verwendete CSS und die DOM-Struktur wird als Grundgerüst für den Testbericht übernommen. Der Testbericht wird jeweils im *Build*-Verzeichnis des Aufgaben-Repositorium unter `build/reports/regression_test_report` generiert.

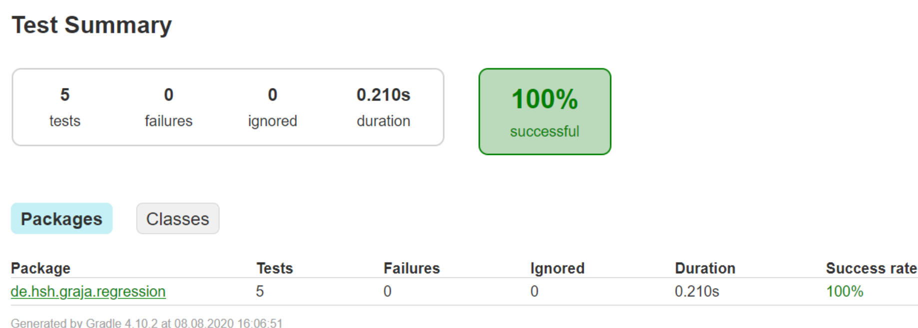


Abbildung 29: Mögliches Layout für den Testbericht des Regressionstestmechanismus

Die folgende Ordnerstruktur wird verwendet, um den Testbericht darzustellen:

+ `assignments` - Gruppierung nach Aufgabe.
    + `<assignment>` - Gruppierung nach Musterlösung.
        – `index.html` - Übersicht über die Aufgabe und alle Musterlösungen.
        – `<sampleSolution>` - Detaillierter Bericht bzgl. d. Regressionstests.
+ `attachments` - Liste aller Anhänge
    – `<identifier>` - *identifier* um Eindeutigkeit des Anhangs zu garantieren.
        – `<name.ext>` - Name und Dateiendung des Anhangs.
– `index.html` - Alle abgearbeiteten Aufgaben.
– `report.js / style.css` - Um Wiederverwendbarkeit zu ermöglichen.

Somit kann ein menschenlesbarer, bei der Fehlersuche assistierender Testbericht für alle gefundenen potenziellen Regressionen generiert werden.

6 Implementierung

Der Abschnitt der Implementierung behandelt das Umsetzen der in Kapitel 5 vorgestellten Konzepte in Form einer Referenzimplementierung. Für die Referenzimplementierung werden die beiden neuen Module `GrajaRegressionTesting`[195] und `GrajaRegressionTestingRunner`[196] neben den bereits bestehenden Graja-Modulen eingeführt. In einigen Fällen ist es allerdings möglich, dass Änderungen innerhalb anderer Module durchgeführt werden müssen, um den Regressionstestmechanismus zu realisieren. Um eine korrekte Funktionalität zu garantieren, wird die Referenzimplementierung durch Modultests des Test-Frameworks JUnit 4[197] unterstützt.

6.1 Umsetzung der Verhaltensaufzeichnung

Das in Kapitel 5.2.4 angesprochene Datenmodell wird dem UML-Diagramm in Abbildung 14 gemäß umgesetzt[198]. Um das in Kapitel 5.1.4 gewählte XML-Serialisierungsformat zu verwenden, werden die jeweiligen Klassen mit passenden *Java Architecture for XML Binding*[199] (JAXB) Annotationen versehen. JAXB erlaubt auch die Autogenerierung eines XSD Schemas, das im elektronischen Anhang unter `graja_regression_testing_recorded_behavior.xsd` auffindbar ist. Die Generierung der XSD ist in den *Build*-Prozess integriert, um das abgeleitete Schema in der resultierenden *Java Archive* (JAR)-Datei einzubetten. Das Schema kann dann zur Validierung des Datenmodells durch JAXB verwendet werden. Ein gemeinsamer XML-Namensraum für alle Domänenobjekte ist über einen *Uniform Resource Name*[200] (URN) mit dem Wert `urn:graja:regressiontesting:v1.0.0` und dem Präfix *rtrb* definiert. Die einzige notwendige Änderung gegenüber dem UML-Diagramm besteht darin, dass die Klasse `ContentRepresentationTO` um das Integer-Attribut *stackdumpId* ergänzt wird. Diese Änderung ist für die Implementierung des *Stackdump*-Mechanismus notwendig.

6.1.1 Umsetzung des Stackdump-Mechanismus

Der in Kapitel 5.2.3.2 besprochene Mechanismus zur Lokalisierung der Kommentarherkunft im Graja-Quellcode ist gemäß des in Abbildung 13 dargestellten UML-Diagramms umgesetzt. Um *ContentRepresentationTO*-Objekte mit *Stackdumps* zu assoziieren, werden die *Stackdumps* in einer Liste gespeichert (Liste einer Liste aus *Stackframes*). Das *stackdumpId*-Attribut der Klasse `ContentRepresentationTO` korrespondiert dann mit einer Indexposition in dieser Liste und erlaubt so ein Auflösen der Beziehung: *Kommentar => Stackdump.*

6.1.2 Umsetzung der Verhaltensaufzeichnung und inter-JVM Kommunikation

Um eine Aufzeichnung des Verhaltens zu realisieren, wird der Proxy-Mechanismus, der in Kapitel 5.1.1 beschrieben ist, durch die Klasse `BehaviorRecordingCore` implementiert. Die beiden Klienten `BackendCommendProvider` und `WithinJVMBackendAPI` verweisen nun also auf diesen Proxy und nicht mehr direkt auf die Klasse `Core`.

[195] Beinhaltet Untermodule des Regressionstestmechanismus und interne Funktionalität.

[196] Enthält die öffentliche in Kapitel 5.5.1 behandelte API und den Regressionstestmechanismus, der sich auf die Funktionalität des Moduls `GrajaRegressionTesting` stützt.

[197] `https://junit.org/junit4/`

[198] Umsetzung im Paket: `de.hsh.graja.regression.recorded`

[199] `https://docs.oracle.com/javase/tutorial/jaxb/intro/index.html`

[200] [SAK17]

Zusätzlich wird die in Kapitel 5.1.3 besprochene Erweiterung des in Abbildung 8 dargestellten *Request*-Domänenmodells mit einer Änderung[201] übernommen. Um ein von der Klasse `Core` erstelltes *Result*-Objekt in das Datenmodell für aufgezeichnetes Verhalten zu konvertieren, ist im Modul `Core` die Klasse `ConverterService` ergänzt[202]. Der Rückgabewert des `ConverterService`, eine `TransformedEntity`-Instanz enthält außerdem, falls vorhanden, den *Stackdump* der Kommentare durch die Klasse `StackdumpTransfer`. Über die Methode `getStackdump(int)` kann dann mithilfe des *stackdumpId*-Attributs der Klasse `ContentRepresentationTO` auf den assoziierten *Stackdump* zugegriffen werden. In Abbildung 30 ist die Implementierung des `ConverterService` und der Kommunikationsschnittstelle `TransferService` (nächste Seite) anhand eines UML-Klassendiagramms visualisiert[203].

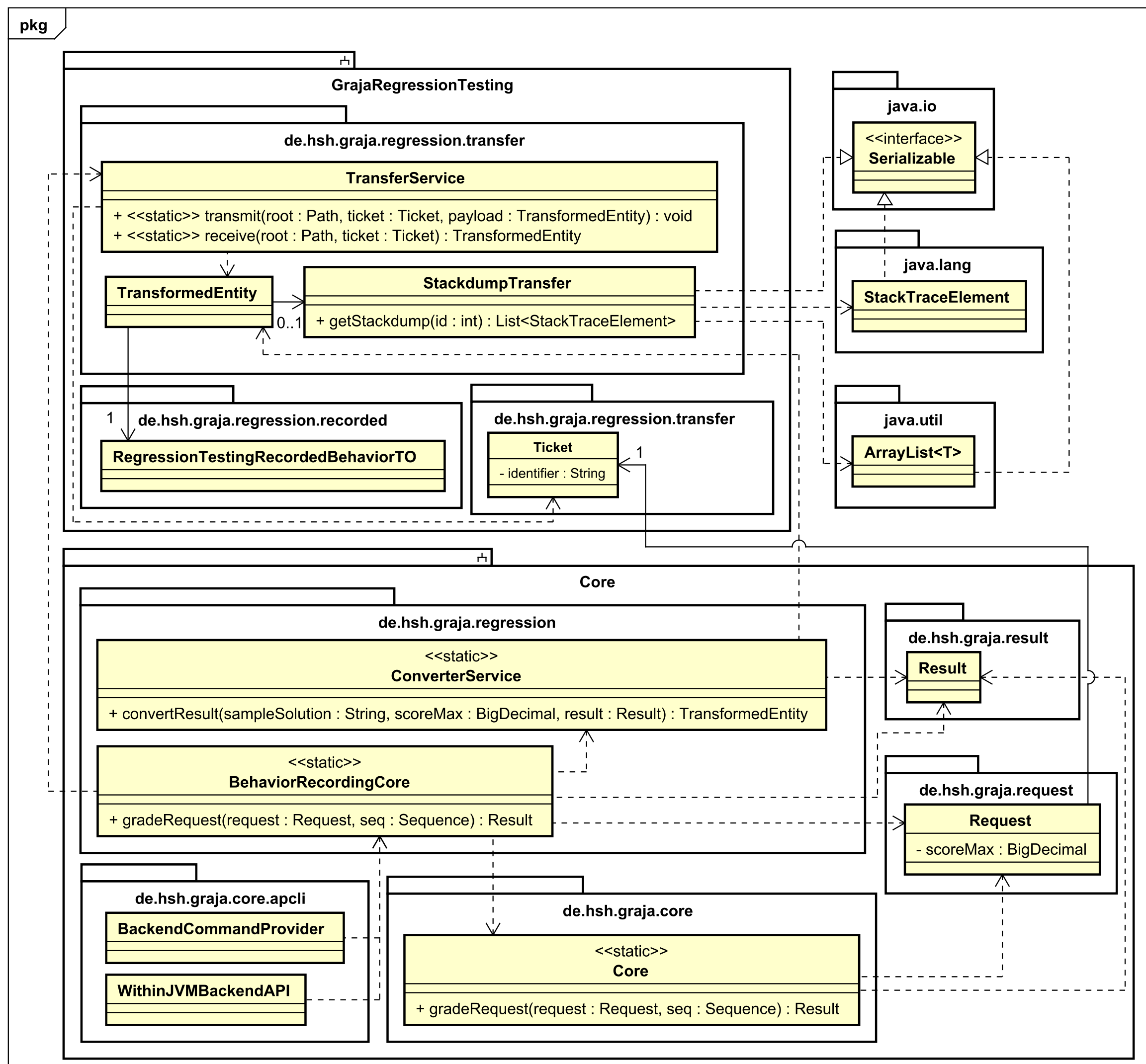


Abbildung 30: Verhaltensaufzeichnung mithilfe von `BehaviorRecordingCore` und Kommunikation zwischen zwei JVMs über die Klasse `TransferService`

[201] Die Klasse `Request` muss wie in Abbildung 30 dargestellt, um das Attribut *scoreMax* erweitert werden. *scoreMax* wird für die Transformation durch die Klasse `ConverterService` benötigt.

[202] Um die Metainformationen *Name der Musterlösung* und *maximale Punktzahl* im Verhalten abzuspeichern, existieren die Parameter *scoreMax* und *sampleSolution* in der Funktion `convertResult`.

[203] Die in Kapitel 5.1.3 beschriebenen Attribute der Klasse `Request` gelten als vorrausgesetzt und sind nicht abgebildet.

Da der Regressionstestmechanismus voraussichtlich Verhaltensaufzeichnungen durch das Aufrufen einer externen JVM durchführen wird, muss eine Kommunikation zwischen den beiden JVMs gewährleistet werden. Dafür wird die Klasse `Ticket` eingeführt, dessen Instanz einen Abholschein[204] darstellt. Dieser Abholschein enthält als Bezeichner eine eindeutige *UUID* in Form einer Zeichenkette. Falls die externe JVM nun den Bewertungsvorgang ohne Unterbrechung abschließt, kann das aufgezeichnete Verhalten der Behälter-Klasse `TransformedEntity` mithilfe der Funktion `transmit` der Klasse `TransferService` und des Abholscheins in einem temporären Verzeichnis auf dem Hostrechner serialisiert werden. Um auch von der externen JVM aus auf diesen Abholschein zuzugreifen, wird die Klasse `Request` um eine `Ticket`-Referenz erweitert.

In der ursprünglichen JVM wird das aufgezeichnete Verhalten der externen JVM dann mithilfe des übergebenen Tickets und dem temporären Verzeichnis *„abgeholt“*, sprich deserialisiert. Für das transformierte Datenmodell (Abgebildet durch die Klasse `RegressionTestingRecordedBehaviorTO`) wird der Serialisierungsprozess von JAXB angewandt. Die Klasse `StackdumpTransfer` hingegen wird durch Javas internen Serialisierungsmechanismus persistiert. Da *Stackdumps* nur für das Generieren des Testberichts notwendig sind, stellen diese kurzlebige Objekte dar. Die Probleme, welche mit Javas Serialisierungsmechanismus existieren, wie z.B. das der Versionierung[205] oder der Sicherheitsaspekte (DoS-Attacken[206]), sind in diesem Fall vertretbar, da diese Objekte nicht aus fremden Quellen stammen und die Klasse `StackdumpsTransfer` nicht mehr verändert wird. Bei den verwendeten JDK-Klassen ist eine Kompatibilität mit dem Serialisierungsmechanismus durch die Implementierung von `Serializable`[207] garantiert.

6.1.3 Umsetzung der Rauschunterdrückung

Der in Kapitel 5.3.2 behandelte Mechanismus zur Rauschunterdrückung wird implementiert, um im Transformationsschritt der Klasse `ConverterService` das Unterdrücken von Rauschen in den Kommentaren der Aufzeichnung zu ermöglichen. In Abbildung 31 sind die Details der Implementierung anhand eines UML-Klassendiagramms illustriert.

Abbildung 31: Auf regulären Ausdrücken basierende Rauschunterdrückung durch die Klasse `NoiseReductor`

[204] Da in der Zukunft eine Umstellung des Regressionstestmechanismus auf Nebenläufigkeit möglich ist, wird hier direkt eine potenzielle *race condition* durch das Überschreiben einer Datei mit immer gleichbleibenden Namen eliminiert.

[205] [Goe]

[206] [Svo] [S. 15-17]

[207] `https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html`

Die im Kapitel 5.3.2 durch die Datenanalyse angefertigten regulären Ausdrücke zur Rauschunterdrückung werden direkt aus dem Werkzeug *Comment-Dump-Analyser* übernommen. Mithilfe des Aufzählungstypen `ReplacePattern` können jederzeit zusätzliche Muster durch reguläre Ausdrücke definiert werden. Muster werden einem Alias (`ReplaceAlias`) zugeordnet. Dieses wird verwendet, um einen mithilfe eines Musters erkannten *substring* durch einen neutralen Ausdruck zu ersetzen. Dabei wird die Java-API für reguläre Ausdrücke verwendet[208]. Alle regulären Ausdrücke der `ReplacePattern`-Konstanten werden zu einem einheitlichen, durch *Regex*-Gruppen aufgeteilten, regulären Ausdruck zusammengefasst. Die Rauschunterdrückung der Klasse `NoiseReductor` funktioniert so, dass anhand der Gruppe eines gefundenen Musters auf das damit assoziierte Alias zurückgeschlossen werden kann und anschließend eine Textersetzung durch den neutralen Ausdruck des Alias erfolgt. Da der gruppierte, vereinigte reguläre Ausdruck aller Muster der Konstanten *greedy matching* ist, wird dieser der Reihenfolge der Konstanten nach erstellt. Um auch `DiffComments` zu behandeln, wird eine API für `String`-Listen angeboten.

6.1.4 Umsetzung der Maßnahmen zur Minimierung von Fehlalarmen

Um Fehlalarme zu vermeiden, werden die in Kapitel 5.3.4 und 5.3.5 besprochenen Maßnahmen umgesetzt. Insgesamt werden in 11 verschiedenen Aufgaben die Aufrufe von `Random`-Konstrukturen auf die der *Factory*-Methoden der Klasse `RandomFactory` migriert.

6.1.5 Fazit: Umsetzung der Verhaltensaufzeichnung

Es ist durch die Implementierung des Datenmodells (Kapitel 5.2.4), `Core`-Proxies (Kapitel 5.1.1), der Rauschunterdrückung (Kapitel 5.3.2) und der Erweiterung des *Request*-Domänenmodells (Kapitel 5.1.3) in Verbindung mit dem Konverter und der Kommunikationsschnittstelle nun möglich,

- eine optionale Verhaltensaufzeichnung durch das Setzen eines *BehaviorRecordingTO*-Objekts in der Basisklasse `RequestTO` aller *Requests* zu erwirken.
- eine Verhaltensaufzeichnung unter der Verwendung der Proxy-Klasse `Behavior RecordingCore` durchführen, welche nur geschieht, falls vorher ein *Request* mit gesetzten *BehaviorRecordingTO*-Objekt eingeht.
- das von der Klasse `Core` generierte *Result*-Objekt in das Datenmodell für aufgezeichnetes Verhalten zu konvertieren und die in Kapitel 5.3.2 besprochene Rauschunterdrückung auf sämtlichen Textinhalt der Kommentare anzuwenden.
- einen flexiblen und erweiterungsfreundlichen Mechanismus der Rauschunterdrückung bereitzustellen.
- anhand der Klasse `TransferService` eine Kommunikationsschnittstelle für inter-JVM Kommunikation anzubieten.

Dadurch, dass die Implementierung der Verhaltensaufzeichnung durch minimale Änderungen in bereits existierenden Graja-Klassen auskommt und ein optionales Feature darstellt, ist das Ziel des nichtinvasiven Eingriffs für dieses Teilsystem erfüllt. Die Änderungen an den Aufgaben selber sind ebenfalls vertretbar und absolut notwendig, um Fehlalarme durch Pseudozufallsgeneratoren zu vermeiden.

[208] `https://docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html`

6.2 Umsetzung des Verhaltensabgleichs

Für die Umsetzung des Verhaltensabgleichs zwischen Soll- und Ist-Verhalten wird die Datenstruktur für einen Testbericht aus Kapitel 5.4.5 in Java-Klassen übersetzt[209]. Die in Kapitel 5.4.1, 5.4.3 und 5.4.4 besprochenen Verhaltensabgleiche sind gemäß des UML-Diagramms in Abbildung 32 anhand der Klasse `RegressionDetector` umgesetzt. Der in Kapitel 5.4.2 besprochene Mechanismus zum Erkennen von strukturellen Differenzen zwischen zwei Bäumen wird von der Klasse `RegressionDetektor` verwendet und ist mithilfe der Klasse `TreeCompare` gemäß des Pseudocodes aus Listing 29 im Anhang umgesetzt. Um die Funktionsweise des *multi sample modes* aus Kapitel 5.3.7 zu implementieren, wird die Klasse `ExpectedEntity` eingeführt, die entweder ein aufgezeichnetes oder N aufgezeichnete Soll-Verhalten enthalten kann.

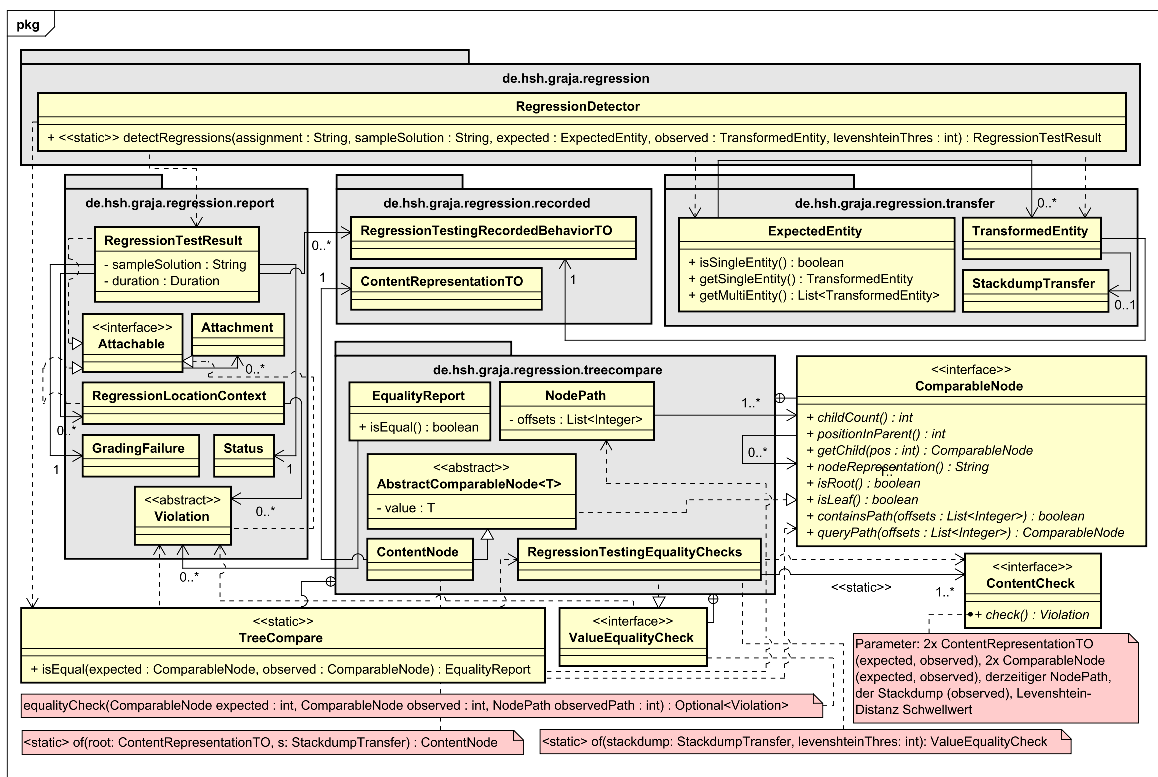


Abbildung 32: Verhaltensabgleich mithilfe der Klasse `RegressionDetector`

Mithilfe der statischen `of`-Methode der Klasse `ContentNode` kann ein Kommentarbaum des aufgezeichneten Verhaltens in eine auf Differenzen untersuchbare Datenstruktur umgewandelt werden. Die Schnittstelle `ValueEqualityCheck` erlaubt die Implementierung inhaltlicher Vergleiche zweier Knoten. Sämtliche inhaltlichen Vergleiche der in Kapitel 5.2.4 beschriebene Kommentar-Abstraktion sind durch die Schnittestelle `ContentCheck` innerhalb der Klasse `RegressionTestingEqualityChecks` implementiert. Durch die statische Funktion `of` ist es möglich, eine `RegressionTestingEqualityChecks`-Instanz für einen gewählten Levenshtein-Distanz-Schwellwert zu erstellen. Für die Levenshtein-Distanz wird auf die bereits bestehende Implementierung der Bibliothek *Apache Commons Text*[210] zurückgegriffen.

Die Anwendung durch die Klasse `RegressionDetector` zur Verfügung gestellten Funk-

[209] Umsetzung im Paket: `de.hsh.graja.regression.report`
[210] `https://commons.apache.org/proper/commons-text/`

tionalität auf ein Ist- und Soll-Verhalten führt zu einem Ergebnis der Klasse `RegressionTestResult`. Dieses enthält, falls vorhanden, die in Kapitel 5.4.5.2 beschriebenen Kandidaten potenzieller Regressionen, die durch den in Kapitel 5.4.5.1 behandelten Mechanismus zur Adressierbarkeit von Regressionen gruppiert sind und von einem `RegressionDetector`-Aufrufer in einen Testbericht mit übernommen werden können.

6.2.1 Grafische Illustration von Kommentarbaum-Differenzen

Um die in einem Kommentarbaum-Abgleich gefundenen Differenzen im Testbericht übersichtlich darzustellen, wird eine grafische Illustration der strukturellen Vereinigung der beiden Bäume mit unterschiedlicher Knoten-Einfärbung im *Scalable Vector Graphic* (SVG)-Format[211] durch die Klasse `TreeVisualizer` generiert[212]. Zur korrekten Anordnung der Knoten des darzustellenden Baums wird die Bibliothek *treelayout*[213] verwendet. Die Klasse `SVGGenerator` entsteht durch die Modifikation der *treelayout*-Beispielklassen[214]. Um die Differenzen zweier Kommentarbäume zu visualisieren, müssen diese erst in eine `RenderableNode`-Instanz umgewandelt werden. Beim Umwandlungsschritt findet eine Vereinigung der Struktur statt, außerdem werden anhand der detektierten Differenzen pro Knoten `RenderFlag`-Attribute bestimmt, welche die Knotenfarben abbilden. Bei mehreren `RenderFlag`-Attributen werden mehrere Farben als Knotenfarbe verwendet. Da SVG *Hyperlinks* unterstützt, kann mithilfe des *identifier*-Attributs der Klasse `Violation` aus der Testbericht-Datenstruktur ein Verweis auf den mit einem Knoten assoziierte Regression geschehen. Somit können Nutzer den Testbericht auch mithilfe der Visualisierung des Kommentarbaums navigieren.

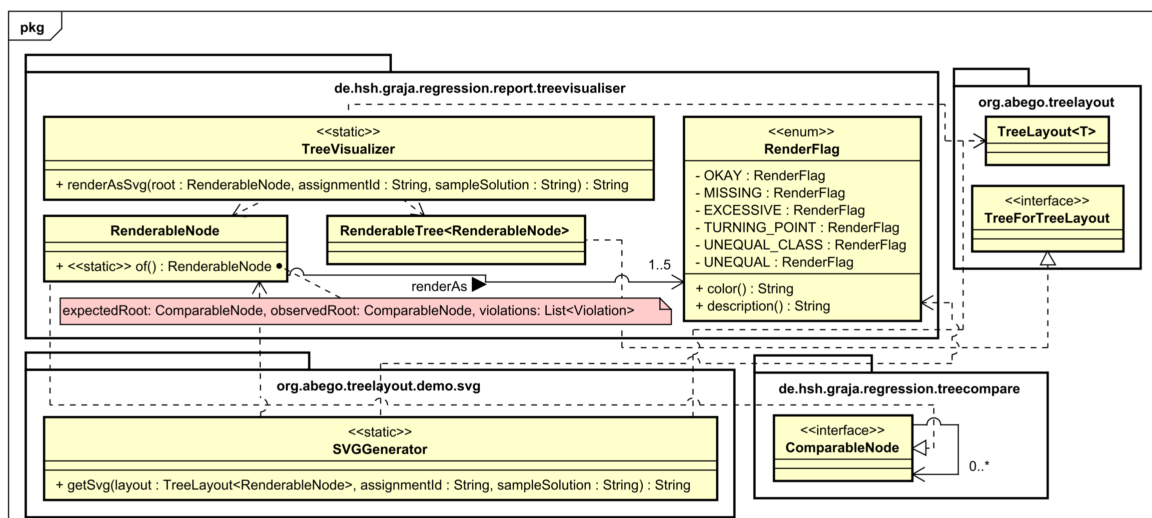


Abbildung 33: Visualisierung der in einem Kommentarbaum-Abgleich gefundenen Differenzen mithilfe der Klasse `TreeVisualizer`

Eine solche Baumvisualisierung wird über die Schnittstelle `Attachable` der Testbericht-Datenstruktur an eine `RegressionLocationContext`-Instanz angehängt und kann somit vom generierten Testbericht aus betrachtet werden.

[211] [BRBL+]

[212] Das SVG-Format wurde gewählt, da so im Gegensatz zu Rastergrafiken, auch bei Bäumen mit hoher Knotenzahl ein verlustfreies Vergrößern der generierten Illustration möglich ist.

[213] `https://github.com/abego/treelayout`

[214] `https://github.com/abego/treelayout/tree/master/org.abego.treelayout.demo/src/main/java/org/abego/treelayout/demo/svg`

6.2.2 Fazit: Umsetzung des Verhaltensabgleichs

Es ist durch die Implementierung der Testbericht-Datenstruktur (Kapitel 5.4.5) und des Verhaltensabgleichs (Kapitel 5.4.1, 5.4.3 und 5.4.4) nun möglich,

- einen Verhaltensabgleich zwischen Soll- und Ist-Verhalten gemäß Kapitel 5.4 durchzuführen, der auch aufgezeichnetes Verhalten des *multi sample mode* korrekt behandelt.
- mittels Levenshtein-Distanz einen schwellwertbasierten, unscharfen Vergleich für den Textinhalt der Kommentare in den Verhaltensabgleich mit aufzunehmen.
- Ungleichheiten des Verhaltensabgleichs anhand der Klasse `RegressionLocationContext` zu gruppieren und lokalisieren.
- Differenzen zweier Kommentarbäume grafisch im SVG-Format zu visualisieren und über *Hyperlinks*, die auf den eindeutigen Bezeichner einer entdeckten Regression verweisen, eine alternative Navigation für den Testbericht anzubieten.
- eine erweiterbare Architektur für inhaltliche Kommentar-Abgleiche innerhalb der Klasse `RegressionTestingEqualityChecks` mithilfe der Schnittstelle `ContentCheck` zur Verfügung zu stellen.

6.3 Umsetzung der öffentlichen Schnittestelle

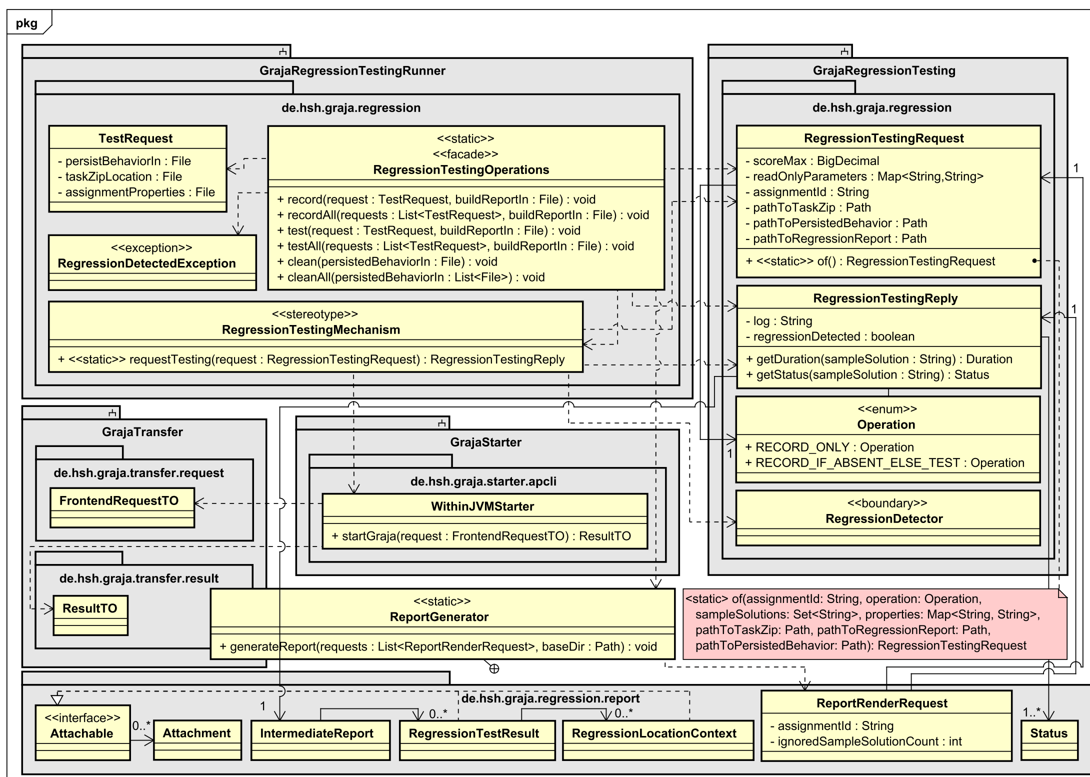


Abbildung 34: Öffentliche Gradle-Schnittstelle der Referenzimplementierung mithilfe der Klasse `RegressionTestingOperations`

Die Umsetzung der in Kapitel 5.5.1 angesprochenen öffentlichen Gradle-Schnittstelle über die Fassade der Klasse `RegressionTestingOperations` geschieht gemäß des

UML-Diagramms in Abbildung 34. Der Regressionstestmechanismus ist durch die Schnittstelle der Klasse `RegressionTestingMechanism` implementiert. Diese Schnittstelle führt Anfragen vom Typen `RegressionTestingRequest` aus und produziert anschließend `RegressionTestingReply`-Instanzen als Ergebnisse. Die in Kapitel 5.5 definierten Basisoperationen `Test` und `Record` sind auf den Aufzählungstypen `Operation` abgebildet.

Die Schnittstelle `RegressionTestingMechanism` übernimmt die folgenden Teilaufgaben:

- Einmaliges Aufrufen der `register`-Funktionen aller Graja-Testmodule, um diese zu initialisieren.
- Einrichten eines temporären Verzeichnis, um die Inter-JVM Kommunikation zu gewährleisten.
- Falls vorhanden: Extraktion des innerhalb der eingereichten task.zip enthaltenen Soll-Verhaltens.
- Erstellen der `FrontendRequestTO`-Instanzen, die zur programmatischen Verhaltensaufzeichnung durch die Graja-Schnittstelle `WithinJVMStarter` genutzt werden.
- `Test (Soll-Verhalten gefunden)`: Ist-Verhalten aufzeichnen, dann Abgleich mit Soll-Verhalten mithilfe der Klasse `RegressionDetector`.
- `Record` / `Test (Kein Soll-Verhalten gefunden)`: Entweder einfache oder mehrfache (*multi sample mode*) Aufzeichnung des Ist-Verhaltens. Danach Abspeicherung des Ist-Verhaltens als Soll-Verhalten und Aktualisierung des Soll-Verhaltens der task.zip.
- Transformation des *ResultTO*-Objekts in ein im Kapitel 2.4 besprochenes HTML-Feedback-Dokument. Dieses Feedback-Dokument wird dann durch die Schnittstelle `Attachable` an die pro Musterlösung generierte `IntermediateReport`-Instanz angehängt.
- Löschen aller temporären Dateien und Rückgabe einer `RegressionTestingReply`-Instanz, die pro behandelter Musterlösung den Statuscode in From des in Kapitel 5.4.5 behandelten Aufzählungstypen `Status`, die Bearbeitungsdauer (inklusive Graja-Aufruf) und den damit assoziierten Testbericht enthält.

Ein Testbericht wird nicht innerhalb der Klasse `RegressionTestingMechanism` generiert. Dafür ist die Schnittstelle der Klasse `ReportGenerator` verantwortlich. Um den Bericht zu generieren, werden Anfragen vom Typen `ReportRenderRequest` und ein Basis-Verzeichnis übergeben. Diese Anfragen erlauben das Bündeln der Testberichte mehrerer Testergebnisse, um so einen gemeinsamen zusammengefügten Testbericht gemäß der Verzeichnisstruktur aus Kapitel 5.5.3 zu generieren. Um das DOM des HTML-Dokuments programmatisch innerhalb von Java zu konstruieren, wird auf die Bibliothek *j2html*[215] zurückgegriffen.

Da die Schnittstelle der Klasse `RegressionTestingMechanism` für die Gradle-Integration zu komplex ist, wird anhand der Klasse `RegressionTestingOperations` eine Fassade angeboten, welche vereinfachte Anfragen vom Typen `TestRequest` verarbeitet. Diese Fassade übernimmt die folgenden Teilaufgaben:

[215] `https://j2html.com/`

- Ausführung der `Clean`-Operationen durch das Löschen des persistierten Soll-Verhaltens innerhalb eines angegebenen Pfads.
- Ausführung der `Test`- und `Record`-Operationen durch die Umwandlung eines `TestRequest` in einen `RegressionTestingRequest` und dem anschließenden Aufrufen der Schnittstelle `RegressionTestingMechanism`.
- Generierung eines Testberichts durch die zurückgegebenen `RegressionTesting Reply`-Ergebnisse über die Testbericht-Schnittstelle `ReportGenerator` und einer `ReportRenderRequest`-Anfrage.

Eine Aufteilung in diese drei Komponenten (Gradle-Fassade, Regressionstestmechanismus-Schnittstelle und Testbericht-Schnittstelle) ist durch das Prinzip *„seperation of concerns“*[216] begründet. Hierdurch ist die Aufteilung der Funktionalität klar definiert. Nachfolgende Änderungen an den Schnittstellen können somit unkompliziert umgesetzt werden. Momentan stellt die Gradle-Fassade einen Klienten der Regressionstestmechanismus- und Testbericht-Schnittstelle dar. Falls nun in der Zukunft ein weiterer Klient nur auf die Schnittstelle des Regressionstestmechanismus zugreifen muss, sind keine weiteren Änderungen notwendig, da der Testmechanismus und die Generierung von Testberichten voneinander separiert sind.

6.3.1 Anbindung an Gradle

Um die in Kapitel 5.5.1 (Abbildung 28) vorgestellte Verbindung der Aufgaben-Repositorien mit dem Regressionstestmechanismus zu realisieren, muss das standardmäßig von allen Repositorien genutzte *Build*-Skript `assignmentBuild.gradle` um eine dynamische Generierung von *Gradle-Tasks*, die pro Aufgabe auf die Operationen `Record`, `Test` und `Clean` verweisen, erweitert werden. Zusätzlich werden noch die Operationen `Record All`, `Test All` und `Clean All` durch drei *Gradle-Tasks* implementiert. Abschließend findet eine Erweiterung der Klasse `BuildProforma` statt, um das persistierte Soll-Verhalten im Überführungsschritt der Aufgabe in die von Gradle generierte task.zip als einen ProFormA-konformen Anhang mitaufzunehmen.

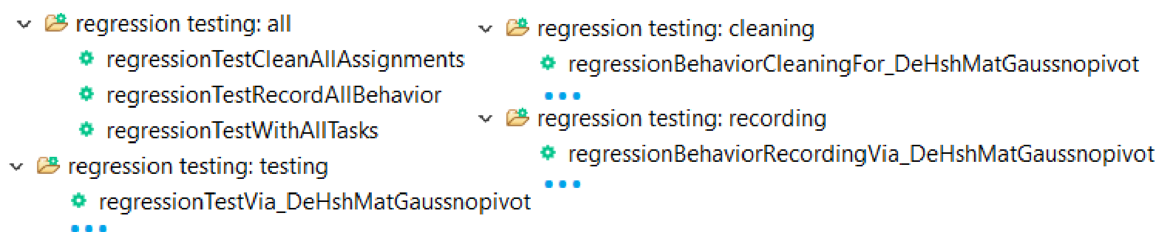


Abbildung 35: Grafische Darstellung von *Gradle Tasks* und *Task Groups* durch das *Eclipse Buildship Plugin*

Um die automatisch generierten *Gradle Tasks* semantisch voneinander zu trennen, werden *Task Groups*[217] verwendet. Entwicklerwerkzeuge wie z.B. *Eclipse*[218] mit *Buildship-Plugin*[219] erlauben, wie in Abbildung 35 illustriert, die grafische Darstellung solcher *Gradle Tasks*. Nutzer der Referenzimplementierung können somit, falls gewollt, ohne

[216] [Mit90] [S. 5]
[217] [Graa]
[218] `https://www.eclipse.org/`
[219] `https://projects.eclipse.org/projects/tools.buildship`

Einstiegshürden mithilfe einer grafischen Benutzeroberfläche mit dem Regressionstestmechanismus interagieren. Die Möglichkeit *Gradle Tasks* durch ein CLI auszuführen bleibt dabei bestehen. Dies ist notwendig, um ein automatisiertes Ausführen der Regressionstests bei Änderungen an Graja durch z.B. eine *Deployment Pipeline*[220] zu implementieren.

6.3.2 Fazit: Umsetzung der öffentlichen Schnittstelle

Durch das Umsetzen der Gradle-Fassade aus Kapitel 5.5.1 und dem anschließenden Verbinden mit dem *Build*-Skript der Aufgaben-Repositorien ist die Referenzimplementierung des Regressionstestmechanismus nun implementiert und einsatzbereit. Eine Test-Infrastruktur ist mithilfe der dynamisch generierten *Gradle Tasks* realisiert. Diese Infrastruktur erlaubt sowohl eine manuelle Testausführung als auch eine zukünftige Automatisierung der Regressionstests.

Zusätzlich wird eine komplexere Schnittstelle durch die Klasse `RegressionTesting Mechanism` angeboten, die verwendet werden kann, falls die Gradle-spezifische Fassade zu restriktiv ist oder kein anschließender HTML-Testbericht benötigt wird.

Eine Beispielanwendung der Referenzimplementierung mit anschließender Bewertung findet im nächsten Kapitel statt.

[220] [HF10] [Kapitel: 5]

7 Ergebnisse

Nach der Umsetzung der Konzepte aus Kapitel 5 durch die in Kapitel 6 beschriebene Referenzimplementierung, gilt es nun, den Regressionstestmechanismus praktisch einzusetzen. Als Testfälle werden hierfür alle 437 Musterlösungen der 57 Aufgaben des *graja-assignments*[221] Aufgaben-Repositoriums verwendet. Die erstmalige Aufzeichnung des Verhaltens aller Musterlösungen über den *Gradle Task* `regressionTestRecordAllBehavior` wurde innerhalb von 40 Minuten und 58 Sekunden abgeschlossen[222] (Abbildung 36). Der komplette Testbericht dieser Aufzeichnungsphase ist im elektronischen Anhang unter `Ergebnisse/report_aufzeichnung.zip` auffindbar.

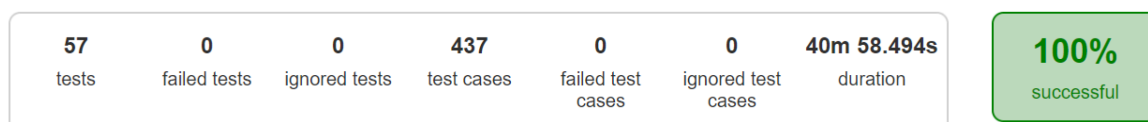
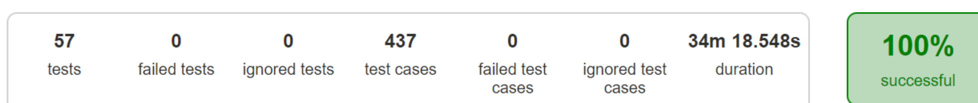


Abbildung 36: Teilausschnitt des Testberichts nach erstmaliger Verhaltensaufzeichnung

Anschließend wurden durch den *Gradle Task* `regressionTestWithAllTasks` sämtliche durch aufgezeichnetes Soll-Verhalten gestützte Regressionstests ausgeführt. Dadurch, dass Graja seit der Verhaltensaufzeichnung nicht modifiziert wurde, sollten keine Fehlalarme auftreten. In Abbildung 37 ist ein Teilausschnitt des Testberichts abgebildet, auf dem zu sehen ist, dass sämtliche Regressionstests fehlerfrei verlaufen sind. Die Differenz von 6 Minuten und 40 Sekunden zur Erstaufzeichnung, ist dadurch zu erklären, dass der *multi sample mode* für einige Musterlösungen in der initialen Aufzeichnungsphase aktiviert war. Der komplette Testbericht dieser Testphase ist im elektronischen Anhang unter `Ergebnisse/report_testall.zip` auffindbar.

Graja Regression Testing Summary



Assignments Test Cases

Assignments	Testcases	Failures	Ignored	Duration	Success rate
de.hsh.mat.gaussnopivot	8	0	0	15.727s	100%
de.hsh.mat.gausspivot	8	0	0	25.794s	100%
de.hsh.mat.gausspivotlu	10	0	0	23.112s	100%
de.hsh.mat.gausspivotround5	9	0	0	16.441s	100%
de.hsh.mat.iterationgaussseidel	6	0	0	13.983s	100%
de.hsh.mat.iterationjacobi	7	0	0	16.343s	100%

Abbildung 37: Teilausschnitt des Testberichts nach anschließendem Testdurchlauf

Durch die erfolgreiche Testphase ist davon auszugehen, dass der Regressionstestmechanismus korrekt funktioniert[223]. Einige Musterlösungen führten allerdings zu Fehlalarmen, die erst mithilfe der in Kapitel 5.5.1 beschriebenen Steuerungsparameter durch Anwendung dieser im folgenden Kapitel 7.1 unterdrückt werden konnten.

[221] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja-assignments`

[222] System des Autors: Prozessor: *AMD Ryzen 5 3600*, Arbeitsspeicher: *16 GB*, Speichermedium: *Solid State Disk*, OS: *Windows 10*

[223] Zumindest für die Menge der Aufgaben des Aufgaben-Repositorium *graja-assignments*.

7.1 Notwendige Einstellungen um Fehlalarme zu vermeiden

Um den fehlerfreien in Abbildung 37 vorhandenen Testbericht zu generieren, mussten zwei zuvor unbehandelte Rauschmuster durch regulärer Ausdrücke in die Rauschunterdrückung mitaufgenommen werden:

- `toString()`-Aufrufe, welche an die durch `Object`[224] implementierte `toString()`-Methode delegieren. Diese Methode stützt sich auf die Methode `hashCode()`, die standardmäßig, falls nicht überschrieben einen Hashwert mithilfe der Speicheradresse des Objekts innerhalb der JVM generiert. Dies führt zu nicht reproduzierbaren Hashwerten, da eine gleichbleibende Speicheradresse für ein beliebiges Objekt innerhalb einer beliebiger JVM unwahrscheinlich ist.
- Zeitstempel innerhalb der Aufgabe *de.hsh.prog.spieler*, da diese in einem vorher unbehandelten Format auftreten.

Aufgabe	**de.hsh.prog.factorsengine**
Parameter	*regRecordedSampleSizeRounds*=`7` *regRecordedSampleSizeRoundsFor*=`wrong_secondTFEThread,wrong_concurrentModificationException`
Begründung	Aktivierung des *multi sample mode*, da mehrere durch *multi threading* bedingte Ausführungsstränge für zwei Musterlösungen existieren.
Aufgabe	**de.hsh.prog.fibonaccizwischenergebnisse**
Parameter	*regRecordedSampleSizeRounds*=`10` *regRecordedSampleSizeRoundsFor*=`1wagler,cache` *regLocalLevenshtein*=`1wagler => 30,cache => 30`
Begründung	Aktivierung des *multi sample mode*, da unterschiedliche Reihenfolgen zweier Kommentare durch *multi threading* im Grader-Code zweier Musterlösungen der Aufgabe existieren. Lokale Levenshtein-Distanz um den inhaltlichen Vergleich dieser Vertauschungen fehlerfrei durchzuführen.
Aufgabe	**de.hsh.prog.klasseline**
Parameter	*regLocalLevenshtein*=`wrong_differentConstructorChain => 15`
Begründung	Lokale Levenshtein-Distanz, da Zeichenkette eines Kommentars, einer Musterlösung je nach Bewertungslauf in vertauschter Reihenfolge ist.
Aufgabe	**de.hsh.prog.nachricht1000**
Parameter	*regRecordedSampleSizeRounds*=`10` *regRecordedSampleSizeRoundsFor*=`wrong_recursion`
Begründung	Aktivierung des *multi sample mode*, da zwei unterschiedliche `StackOverflowError` *Stacktraces* für die Musterlösung einer Aufgabe möglich sind.
Aufgabe	**de.hsh.prog.randomtext**
Parameter	*regGlobalLevenshtein*=`10`
Begründung	Höherer globaler Levenshtein-Schwellwert für Behandlung von Zufallswerten.
Aufgabe	**de.hsh.prog.zahlenarray**
Parameter	*regGlobalLevenshtein*=`10`
Begründung	Höherer globaler Levenshtein-Schwellwert für Behandlung von Zufallswerten.

Tabelle 6: Notwendige Einstellungen in den `assignment.properties`-Dateien der Aufgaben, um einen fehlalarmfreien Testdurchlauf zu garantieren

[224] [Orab]

Außerdem mussten für einige Aufgaben die in Tabelle 6 aus Kapitel 5.5.1 eingeführten Steuerungsparameter angewandt werden, um einen fehlalarmfreien Testdurchlauf zu garantieren.

7.2 Demonstration anhand künstlich erzeugter Regressionen

Um das Erkennen einer Regression zu simulieren, werden in diesem Kapitel künstliche Regressionen durch unrealistische Änderungen erzeugt[225]. Es wird sowohl eine Änderung im Grader-Code einer Aufgabe durchgeführt als auch das Bewertungsverhalten des Graja-Testmoduls JUnit modifiziert.

7.2.1 Erzeugung einer Regression im Grader-Code

Für die Erzeugung dieser Regression wird das Verhalten des Grader-Codes der Aufgabe *de.hsh.prog.bruch* verändert. In der Testmethode `gettersShouldReturnValuesPassedToSetters` findet die Modifikation einer enthaltenen *Assertion* von `ZAEHLER==zaehlerReceived` zu `ZAEHLER!=zaehlerReceived` statt. Dadurch kann das aufgezeichnete Soll-Verhalten nicht reproduziert werden, da eine Veränderung der Semantik des Grader-Codes stattfindet.

↑ - Location Context - 1 violation(s)

Context Classifier: **GRADE_NODE** - 1 violation(s)

Violation(s) occured in Level L3+ (Nodes of ProFormA-Grading Aspects and Combine-Groups)

Violation(s) occured within a node of the grading schema.

Path in grading schema:

`root -> functionality -> junit@de.hsh.prog.bruch.grader.Grader#gettersShouldReturnValuesPassedToSetters`

Attachments

No attachments exist.

↑ - Violation: **EQUALITY_CLASH_GRADES**

Both success ratings or grade sets do not match when being compared with each other.

Detected by: **Content Equality Test (CONTENT)**

Expected Grades:

`CORRECT`

Observed Grades:

`WRONG`

Expected Success:

`0,000`

Observed Success:

`1,000`

Abbildung 38: Teilausschnitt des Testberichts nach anschließendem Testdurchlauf

In Abbildung 38 ist ein Teilausschnitt des resultierenden Testberichts dargestellt, in dem eine Regression gemäß Kapitel 5.4.5.1 adressiert ist. Es handelt sich bei dieser Regression um einen fehlschlagenden Abgleich von Erfolgswert und `Grade`-Statuscode.

[225] Unrealistische Änderungen, die in der Praxis so wahrscheinlich nicht vorkommen, da es das Ziel ist, den Regressionstestmechanismus anschlagen zu lassen. Auf eine Erzeugung einer Regression innerhalb einer Musterlösung wird bewusst verzichtet, da Musterlösungen in der Regel nach Erstellung aufgrund der dadurch entstehenden semantischen Änderungen nicht mehr bearbeitet werden sollten.

Durch den im Testbericht der Abbildung 38 ausgegebenen Pfad im Bewertungsschema, lässt sich darauf schließen, dass diese Regression innerhalb der Methode `gettersShouldReturnValuesPassed` des Grader-Codes der Aufgabe verursacht wurde.

Da die erkannte Regression auch strukturelle Differenzen innerhalb der Kommentarbäume erzeugt, enthält Abbildung 39 eine Ausgabe der in Kapitel 6.2.1 behandelten grafischen Illustration von Kommentarbaum-Differenzen[226]. Die Unterschiede zwischen den Kommentarbäumen in Abbildung 39 sind durch die widersprüchlichen Bewertungsschemas des Soll- und Ist-Verhaltens aufgrund des geänderten Grader-Codes zu erklären.

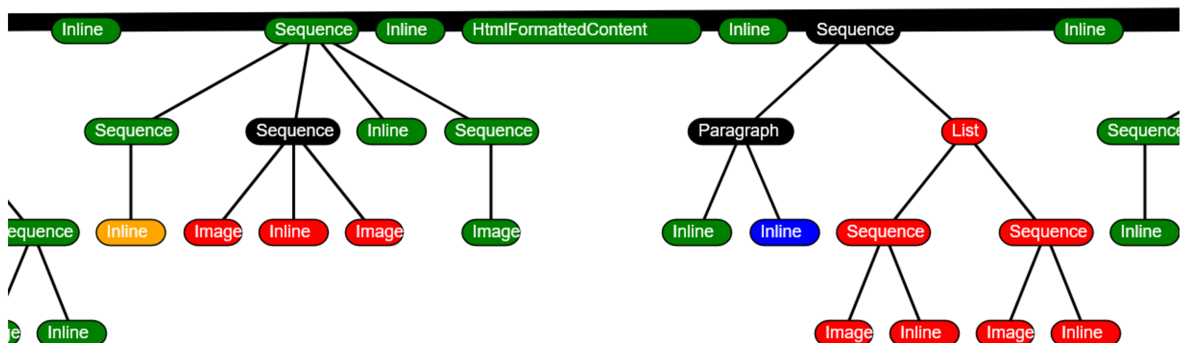


Abbildung 39: Teilausschnitt der illustrierten Differenzen eines Kommentar-Baums

Um die inhaltlichen und strukturellen Unterschiede einer solchen Kommentarbaum-Differenz besser zu verdeutlichen, ist in Abbildung 40 eine Ausgabe sämtlicher Kinder-Knoten des Soll- und Ist-Verhaltens eines Knotens mit abweichender Kinder-Anzahl illustriert. In diesem Falle findet sich durch die veränderte Semantik des Grader-Codes ein `AssertionError` im observierten Ist-Verhalten, der nicht im Soll-Verhalten vorhanden ist.

```
<< IN EXPECTED (children=0) >>


<< IN OBSERVED (children=1) >>

Sequence [3 children]
  Sequence [2 children]
    Paragraph [1 children]
      Inline [0 children] -> Data: [ getZaehler should return the value that was passed to setZaehler.]
    Verbatim [0 children] -> Data: [ AssertionError (getZaehler should return the value that was passed to setZaehler)
    org.junit.Assert.fail(Assert.java:88)
    org.junit.Assert.assertTrue(Assert.java:41)
    de.hsh.prog.bruch.grader.Grader.gettersShouldReturnValuesPassedToSetters(Grader.java:87)
    sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    java.lang.reflect.Method.invoke(Method.java:498)
    org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
    org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
    org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
    org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
    org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)
    org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:27)
    de.hsh.graja.graderapi.GrajaRunner$2.evaluate(GrajaRunner.java:54)
    org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)
    org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:70)
    org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:50)
    org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)
    org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:63)
    org.junit.runners.ParentRunner.runChildren(ParentRunner.java:236)]
  Sequence [0 children]
  Inline [0 children] -> Data: [finished gettersShouldReturnValuesPassedToSetters(de.hsh.prog.bruch.grader.Grader)]
```

Abbildung 40: Ausgabe aller Kinder zweier Knoten mit unterschiedlicher Kind-Anzahl

[226] Legende der Farben: *Orange* - Abweichender Kommentarinhalt, *Rot* - Fehlend im Ist-Verhalten, *Blau* - Fehlend im Soll-Verhalten, *Schwarz* - Knoten mit unterschiedlicher Kind-Anzahl - *Grün* - Differenzfreier Abgleich

Die künstliche, im Grader-Code erzeugte Regression ist detektierbar und mithilfe des Testberichts auf die korrekte Herkunft im Quellcode zurückzuführen. Die Illustration des Kommentarbaums gibt visuelle Hinweise auf Differenzen zwischen dem Soll- und Ist-Verhalten. Der vollständige Testbericht ist im elektronischen Anhang unter `Ergebnisse/rr_bruch.zip` vorhanden.

7.2.2 Erzeugung einer Regression in Graja

Für die Erzeugung einer Regression innerhalb Graja selber, wird das Verhalten des Testmoduls JUnit durch die Methode `createGradingAspectResult` der Klasse `JunitEvaluator` so modifiziert, dass immer der Statuscode `WRONG` des Aufzählungstypen `Grade` in das Bewertungsschema übernommen wird.

↑ - Violation: **EQUALITY_CLASH_GRADES**

Both success ratings or grade sets do not match when being compared with each other.

Detected by: **Content Equality Test (CONTENT)**

Expected Grades:

`CORRECT`

Observed Grades:

`CORRECT, WRONG`

Expected Success:

`1,000`

Observed Success:

`1,000`

Abbildung 41: Detektierte Regression durch widersprüchliche `Grade`-Statuscodes

↑ - Violation: **EQUALITY_CLASH_STRING**

Both string values of a node do not match when being compared with each other.

Detected by: **Content Equality Test (CONTENT)**

```
=== Expected ===

Correct. Score: 2.00/2.00. Assignment de.hsh.prog.referenztypenswap.

=== Observed ===

Error. Score: 2.00/2.00. Assignment de.hsh.prog.referenztypenswap.
```

Origin of observed comment:

```
de.hsh.graja.util.comment.AbstractElementContent.<init>(AbstractElementContent.java:17)
de.hsh.graja.util.comment.Inline.<init>(Inline.java:50)
de.hsh.graja.util.comment.Inline.<init>(Inline.java:39)
de.hsh.graja.util.comment.Inline.<init>(Inline.java:62)
de.hsh.graja.util.comment.Inline.format(Inline.java:124)
de.hsh.graja.util.comment.Paragraph.formatInlineItem(Paragraph.java:82)
de.hsh.graja.core.Core.gradeAssignment(Core.java:238)
de.hsh.graja.core.Core.gradeRequest(Core.java:104)
de.hsh.graja.regression.BehaviorRecordingCore.gradeRequest(BehaviorRecordingCore.java:41)
de.hsh.graja.core.apcli.BackendCommandProvider.gradeBackendRequest(BackendCommandProvider.java:84)
de.hsh.graja.core.apcli.BackendCommandProvider.executeCommand(BackendCommandProvider.java:50)
de.hsh.graja.util.cli.Command.execute(Command.java:342)
de.hsh.graja.util.cli.Main.main(Main.java:61)
de.hsh.graja.Cli.main(Cli.java:62)
```

Abbildung 42: Regression durch Kommentar mit widersprüchlichen Textinhalt

Dies führt, wie in Abbildung 41 sichtbar, bei einem Verhaltensabgleich mit dem Soll-Verhalten zu detektierten Regressionen, da die erwarteten Statuscodes nun nicht mehr mit den observierten Statuscodes übereinstimmen. Außerdem entstehen, wie in Abbildung 42 sichtbar, Widersprüchlichkeiten bezüglich des Kommentarinhalts eines Kommentars im Bewertungsschema. Die Levenshtein-Distanz zwischen den beiden Zeichenketten *Correct. Sco..* und *Error. Sco..* ist groß genug, um nicht durch den globalen Schwellwert von 3 unterdrückt zu werden.

Eine Lokalisierung dieser Regression durch den *Stacktrace* der Kommentare erweist sich als schwierig, da dieser zu einem Zeitpunkt generiert wurde, in dem sich die interne Graja-Ergebnisstruktur schon in einem inkonsistenten Zustand befand. Der *Stacktrace* des Kommentars aus Abbildung 42 lässt sich somit nicht auf das Testmodul JUnit zurückführen.

Allerdings enthält der *Stacktrace* eines Kommentarbaum-Knotens mit widersprüchlicher Kinder-Anzahl, wie in Abbildung 43 in Rot markiert zu sehen ist, einen Hinweis auf die im Rahmen des Kapitels modifizierte Methode.

Same path structure is shared until this node. Observed node has several excessive child nodes that are not within the expected node.

Detected by: **Structural Tree Equality Test (STRUCTURAL)**

Expected children:

0

Observed children:

1

Path until violation:

Sequence

Origin of violating node in observed behavior:

```
de.hsh.graja.util.comment.AbstractSequenceContent.<init>(AbstractSequenceContent.java:18)
de.hsh.graja.util.comment.Sequence.<init>(Sequence.java:27)
de.hsh.graja.util.comment.Sequence.<init>(Sequence.java:47)
de.hsh.graja.core.Helper.createGradingAspectResultWithoutHeader(Helper.java:142)
de.hsh.graja.modules.junit.JunitEvaluator.createGradingAspectResult(JunitEvaluator.java:60)
de.hsh.graja.core.Core.createGradingAspectGroupResult(Core.java:464)
de.hsh.graja.core.Core.runAllModules(Core.java:444)
de.hsh.graja.core.Core.gradeAssignment(Core.java:227)
de.hsh.graja.core.Core.gradeRequest(Core.java:104)
de.hsh.graja.regression.BehaviorRecordingCore.gradeRequest(BehaviorRecordingCore.java:41)
de.hsh.graja.core.apcli.BackendCommandProvider.gradeBackendRequest(BackendCommandProvider.java:84)
de.hsh.graja.core.apcli.BackendCommandProvider.executeCommand(BackendCommandProvider.java:50)
de.hsh.graja.util.cli.Command.execute(Command.java:342)
de.hsh.graja.util.cli.Main.main(Main.java:61)
de.hsh.graja.Cli.main(Cli.java:62)
```

Abbildung 43: *Stacktrace* eines Baumknotens mit widersprüchlicher Kinder-Anzahl

Die künstlich erzeugte Regression innerhalb des JUnit Testmoduls ist detektierbar und mithilfe des Testberichts auf die Herkunft im Quellcode zurückführen. Im Gegensatz zum Beispiel einer Regression im Grader-Code gestaltet sich die Suche nach der Herkunft der Regression allerdings komplizierter. Ob die Herkunft der künstlich herbeigeführten Regression auch in der Praxis, ohne einen konkreten Ansatzpunkt[227], so schnell auf eine Stelle im Quellcode zurückzuführen wäre, bleibt eine offene Frage. Der vollständige Testbericht ist im elektronischen Anhang unter `Ergebnisse/rr_junit.zip` vorhanden.

[227]Da der Autor dieser Arbeit die Regression künstlich herbeigeführt hat, wusste er bereits wonach er suchen musste.

7.3 Fazit: Demonstration des Regressionstestmechanismus

Anhand zweier künstlich herbeigeführter Beispielregressionen, wurde eine exemplarische Demonstration des in dieser Arbeit entwickelten Regressionstestmechanismus durchgeführt. Während die Regression innerhalb des Grader-Codes anhand des Pfads im Bewertungsschema eindeutig lokalisierbar ist, stellt sich die Lokalisierung einer Regression innerhalb eines von Graja verwendeten Moduls als komplizierter heraus. Im Falle des zweiten Beispiels konnte die Regression nur durch den in Abbildung 43 gezeigten *Stacktrace* auf die Herkunft im Quellcode zurückgeführt werden, da der Autor der Arbeit diese bewusst verursachte.

Bei einer praktischen Anwendung des Regressionstestmechanismus ist allerdings davon auszugehen, dass bei einer erkannten Regression, die Historie der Änderungen seit der letzten regressionsfreien Version im Repositorium, gepaart mit dem Testbericht einen ähnlichen Hinweis auf die möglichen Ursachen geben können.

8 Ausblick

Durch diese Arbeit wurde gezeigt, dass die Verwendung von Musterlösungen in Graja als Mechanismus für Regressionstests umsetzbar ist. Der in dieser Arbeit entwickelte, auf Verhaltensaufzeichnung und Verhaltensabgleich basierende Regressionstestmechanismus, ist grundlegend funktionsfähig, wenn auch noch verbesserungswürdig. Trotz der Graja-Spezialisierung, konnten bereits existierende Ansätze wie der in Kapitel 2.7 beschriebene Regressionstest-Arbeitsablauf in den konzipierten Mechanismus übernommen werden.

Anhand der Referenzimplementierung ist eine einsatzbereite, konkrete Implementierung der erarbeiteten Konzepte entstanden, die keine tiefen Eingriffe in die grundlegende Funktionsweise Grajas erfordert und eine optionale, weitgehend von Graja entkoppelte Erweiterung darstellt. Bezüglich der Performanz sind Verbesserungen durch z.B. *multi threading* möglich, da die sequenzielle Abarbeitung aller Testfälle, wie in Kapitel 7 zu sehen, ineffizient ist. Auch eine Verlegung des Bewertungsvorgangs in die Aufrufer-JVM und das Überspringen des Graja-Frontends kann die Laufzeit eines Regressionstests verringern.

Während die Erkennung von Regressionen durch ein vom Testfall abweichendes Ist-Verhalten möglich ist, stellt sich deren verständliche Darstellung und Lokalisierung noch immer als problematisch heraus. Es ist nicht ohne Weiteres möglich, in allen Fällen die Herkunft einer potenziellen Regression im Quellcode aufzuspüren oder auf ein Modul zu reduzieren. Der *Stackdump*-Mechanismus ist, wie in Kapitel 7.2.2 beschrieben, in den Fällen des inhaltlichen Kommentar-Abgleichs zum Teil nicht bei der Lokalisierung unterstützend. Nur anhand des durch strukturellen Unterschieden lokalisierten *Stackdumps* in Abbildung 43 konnte letztendlich, mithilfe des Wissens die Regression selbst künstlich erzeugt zu haben, eine Vermutung bezüglich der Herkunft im Quellcode getroffen werden.

Ob sich der Regressionstestmechanismus in der Praxis als nützlich erweist und fehlalarmfrei verwendet werden kann, lässt sich in dieser Arbeit nicht beantworten. Durch die in Kapitel 5.5.1 behandelten Steuerungsparameter und der erweiterbaren Rauschunterdrückung kann allerdings adäquat mit Fehlalarmen umgegangen werden. Es bleibt abzusehen, inwieweit eine Robustheit gegenüber Plattform- und JDK-abhängigen *Stacktraces*[228] in den Kommentaren einer Soll-Aufzeichnung durch die Schwellwertoperation der Levenshtein-Distanz gegeben ist. Ein hoch angesetzter Schwellwert kann mitunter für eine verringerte Nützlichkeit des inhaltlichen Vergleichs der Kommentare sorgen. Daher sollte der Regressionstestmechanismus im momentanen Stadium nur auf einer Plattform und JDK-Kombination verwendet werden. Diese Einschränkung ist in der heutigen Zeit durch Container-basierte Virtualisierungstechnologien wie z.B. *Docker*[229] oder *LXC*[230] trivialerweise umgehbar. Ungewiss und ebenfalls nur durch praktische Verwendung klärbar ist, inwiefern die mit einer erkannten Regression verknüpften Informationen letztendlich bei der Lokalisierung eines Fehlers in der Graja-Entwicklung helfen.

[228]Plattform und JDK abhängige *Stacktraces* in dem Sinn, dass sich die mitausgegebenen Zeilennummern in den Klassen der Standardbibliothek ändern oder Plattform-abhängige I/O-Implementierungen unter Windows und Linux zu unterschiedlichen *Stacktraces*, welche sich durch die enthaltenen Klassen unterscheiden, führen.

[229]https://www.docker.com/

[230]https://linuxcontainers.org/

Eine praktische Anwendung der Referenzimplementierung kann sich am Konzept der kontinuierlichen Integration orientieren und wäre z.B. durch einen *webhook*[231] im *master branch* des Graja-Repositoriums realisierbar. Dadurch können nach einem *commit* in den *master branch* automatisch, das aktuelle *graja-assignments*-Repositorium geklont werden und eine Ausführung aller durch Verhaltensaufzeichnungen dargestellten Regressionstests geschehen. Ein Fehlschlagen eines Regressionstests, sollte dann zu einer Benachrichtigung, aller an der Entwicklung beteiligten Personen, führen.

Inwiefern sich die Konzepte des in dieser Arbeit entstandene Regressionstestmechanismus auf andere ProFormA-Aufgabenformat-basierte Autobewerter übertragen lässt, bleibt erst mal eine offene Frage. Das Konzept und die grundlegende Architektur ist sicherlich annähernd übertragbar, sämtliche Graja-spezifischen Datenstrukturen und Sonderfälle wahrscheinlich eher nicht. Als Endergebnis dieser Arbeit ist also eine Grundlage für zukünftige Graja-Regressionstests geschaffen, sowie eine exemplarische Anwendung eines verhaltensbasierten Regressionstestmechanismus aufgezeigt.

[231] `https://docs.gitlab.com/ee/user/project/integrations/webhooks.html`

Literaturverzeichnis

[AHU74] Aho, Alfred V. ; Hopcroft, John E. ; Ullman, Jeffrey D.: *The Design and Analysis of Computer Algorithms.* Pearson Education (US), 1974. – ISBN 0201000296

[BLFM05] Berners-Lee, Tim ; Fielding, Roy T. ; Masinter, Larry M.: *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986. `http://dx.doi.org/10.17487/RFC3986`. Version: Januar 2005 (Request for Comments)

[BPSM+] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C. M. ; Maler, Eve ; Yergeau, François ; Cowan, John: *Extensible Markup Language (XML) 1.1 (Second Edition).* `https://www.w3.org/TR/2006/REC-xml11-20060816/`, Abruf: 01.08.2020. – Version vom 29.09.2006

[Bra14] Bray, Tim: *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 7159. `http://dx.doi.org/10.17487/RFC7159`. Version: März 2014 (Request for Comments)

[BRBL+] Bellamy-Royds, Amelia ; Brinza, Bogdan ; Lilley, Chris ; Schulze, Dirk ; Storey, David ; Willigers, Eric: *Scalable Vector Graphics (SVG) 2.* `https://www.w3.org/TR/2018/CR-SVG2-20181004/`, Abruf: 12.08.2020. – Version vom 04.10.2018

[Cha10] Chauhan, Naresh: *Software Testing - Principles and Practices.* Oxford University Press, 2010. – ISBN 9780198061847

[Che] Chen, Nicholas: *Convention over Configuration.* `http://softwareengineering.vazexqi.com/files/pattern.html`, Abruf: 08.06.2020. – Version vom 29.11.2006

[CR91] Campbell, Douglas M. ; Radford, David: Tree Isomorphism Algorithms: Speed vs. Clarity. In: *Mathematics Magazine* 64 (1991), oct, Nr. 4, S. 252–261. `http://dx.doi.org/10.1080/0025570x.1991.11977616`. – DOI 10.1080/0025570x.1991.11977616

[Gara] Garmann, Robert: *Develop assignments.* `http://graja.hs-hannover.de/doc/developassignments/index.html`, Abruf: 08.08.2020. – Version 2.1.0

[Garb] Garmann, Robert: *Execute Graja.* `http://graja.hs-hannover.de/doc/executegraja/index.html`, Abruf: 08.06.2020. – Version 2.1.0

[Garc] Garmann, Robert: *Installation.* `http://graja.hs-hannover.de/doc/installation/index.html`, Abruf: 08.08.2020. – Version 2.1.0

[Gar16a] Garmann, Robert: Bewertungsaspekte und Tests in Java-Programmieraufgaben für Graja im ProFormA-Aufgabenformat.

(2016). http://dx.doi.org/10.25968/OPUS-834. – DOI 10.25968/OPUS–834

[Gar16b] Garmann, Robert: Graja - Autobewerter für Java-Programme. (2016). http://dx.doi.org/10.25968/OPUS-941. – DOI 10.25968/OPUS–941

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph E. ; Vlissides, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. – ISBN 0201633612

[Goe] Goetz, Brian: *Towards Better Serialization*. https://cr.openjdk.java.net/~briangoetz/amber/serialization.html, Abruf: 07.07.2020. – Version vom Juni 2019

[Graa] Gradle: *Gradle DSL - Task*. https://docs.gradle.org/6.5.1/dsl/org.gradle.api.Task.html, Abruf: 01.08.2020. – Version: 6.5.1

[Grab] Gradle: *Improving the Performance of Gradle Builds*. https://guides.gradle.org/performance, Abruf: 01.08.2020

[Grac] Gradle: *Working With Files*. https://docs.gradle.org/6.5.1/userguide/working_with_files.html, Abruf: 01.08.2020. – Version: 6.5.1

[HF10] Humble, Jez ; Farley, David: *Continuous Delivery - Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison Wesley, 2010. – ISBN 0321601912

[HK07] Huizinga, Dorota ; Kolawa, Adam: *Automated Defect Prevention*. John Wiley & Sons, Inc., 2007. http://dx.doi.org/10.1002/9780470165171. http://dx.doi.org/10.1002/9780470165171

[HMU07] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation*. Boston : Pearson/Addison Wesley, 2007. – ISBN 0321455363

[IEE90] IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std 610.12-1990* (1990), S. 1–84. http://dx.doi.org/10.1109/IEEESTD.1990.101064. – DOI 10.1109/IEEESTD.1990.101064

[ISO] ISO: *ISO 8601 - Date and time format*. https://www.iso.org/iso-8601-date-and-time-format.html, Abruf: 01.08.2020. – Version: Standard von 2019

[Jos06] Josefsson, Simon: *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. http://dx.doi.org/10.17487/RFC4648. Version: Oktober 2006 (Request for Comments)

[JUna] JUnit: *Assertions (JUnit 5.0.1 API)*. https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html, Abruf: 01.08.2020. – Version: 5.1

[JUnb] JUnit: *Before (JUnit API).* https://junit.org/junit4/javadoc/4.12/org/junit/Before.html, Abruf: 01.08.2020. – Version: 4.0

[JUnc] JUnit: *Test execution order.* https://github.com/junit-team/junit4/wiki/Test-execution-order, Abruf: 01.08.2020. – Version vom 28.03.2018

[Kle56] Kleene, S. C.: Representation of Events in Nerve Nets and Finite Automata. Version: dec 1956. http://dx.doi.org/10.1515/9781400882618-002. In: Shannon, C. E. (Hrsg.) ; McCarthy, J. (Hrsg.): *Automata Studies. (AM-34).* Princeton University Press, dec 1956. – DOI 10.1515/9781400882618–002, S. 3–42

[LSM05] Leach, Paul J. ; Salz, Rich ; Mealling, Michael H.: *A Universally Unique IDentifier (UUID) URN Namespace.* RFC 4122. http://dx.doi.org/10.17487/RFC4122. Version: Juli 2005 (Request for Comments)

[Mar13] Martin, Robert C.: *Agile Software Development - Principles, Patterns, and Practices.* Pearson, 2013. – ISBN 1292025948

[Mar17] Martin, Robert C.: *Clean Architecture - A Craftsman's Guide to Software Structure and Design.* Prentice Hall, 2017. – ISBN 0134494164

[MGF+07] Müller, Oliver ; Garmann, Robert ; Fricke, Peter ; Reiser, Paul ; Borm, Karin ; Priss, Uta ; Rod, Oliver: *An XML exchange format for (programming) tasks - ProFormA-XML format.* https://github.com/ProFormA/proformaxml/blob/master/Whitepaper.md. Version: 2007, Abruf: 08.06.2020. – Version 2.0

[Mic] Microsoft: *File path formats on Windows systems.* https://docs.microsoft.com/en-us/dotnet/standard/io/file-path-formats, Abruf: 01.07.2020. – Version vom 06.06.2019

[Mit90] Mitchell, Dr. R. J.: *Managing Complexity in Software Engineering.* Institution of Engineering and Technology, 1990. – ISBN 0863411711

[MSVD07] Matyas, Stephen M. ; Schneider, Nicholas ; Voit, Mark ; Duvall, Paul: *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison Wesley, 2007. – ISBN 0321336380

[Nav01] Navarro, Gonzalo: A guided tour to approximate string matching. In: *ACM Computing Surveys (CSUR)* 33 (2001), mar, Nr. 1, S. 31–88. http://dx.doi.org/10.1145/375360.375365. – DOI 10.1145/375360.375365

[NK02] Newman, Chris ; Klyne, Graham: *Date and Time on the Internet: Timestamps.* RFC 3339. http://dx.doi.org/10.17487/RFC3339. Version: Juli 2002 (Request for Comments)

[NTY] Nir, Dor ; Tyszberowicz, Shmuel ; Yehudai, Amiram: Locating Regression Bugs. http://dx.doi.org/10.1007/978-3-540-77966-7. In:

Hardware and Software: Verification and Testing. Springer Berlin Heidelberg. – DOI 10.1007/978–3–540–77966–7, Kapitel 18, S. 218–234

[Oraa] ORACLE: *Learning the Java Language: Initializing Fields*. `https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html`, Abruf: 01.08.2020. – Version: Geschrieben für Java SE 8

[Orab] ORACLE: *Object (Java Platform SE 8)*. `https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`, Abruf: 01.08.2020. – Version: JavaDoc für Java SE 8

[Orac] ORACLE: *Properties (Java Platform SE 8)*. `https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html`, Abruf: 07.07.2020. – Version: JavaDoc für Java SE 8

[Orad] ORACLE: *Random (Java Platform SE 8)*. `https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`, Abruf: 07.07.2020. – Version: JavaDoc für Java SE 8

[Orae] ORACLE: *Thread (Java Platform SE 8)*. `https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`, Abruf: 07.07.2020. – Version: JavaDoc für Java SE 8

[OW10] ORAM, Andy ; WILSON, Greg: *Making Software - What Really Works, and Why We Believe It*. O'Reilly Media, Inc, USA, 2010. – ISBN 0596808321

[RHJ] RAGGETT, Dave ; HORS, Arnaud L. ; JACOBS, Ian: *HTML 4.01 Specification*. `https://www.w3.org/TR/2018/SPSD-html401-20180327`, Abruf: 08.08.2020. – Version vom 27.03.2018

[Riv92] RIVEST, Ronald L.: *The MD5 Message-Digest Algorithm*. RFC 1321. `http://dx.doi.org/10.17487/RFC1321`. Version: April 1992 (Request for Comments)

[SAK17] SAINT-ANDRE, Peter ; KLENSIN, Dr. John C.: *Uniform Resource Names (URNs)*. RFC 8141. `http://dx.doi.org/10.17487/RFC8141`. Version: April 2017 (Request for Comments)

[Sch] SCHLOSSER, Hartmut: *Dynamische Software-Tests: Wie Sie mit Fuzzing mehr Bugs aufspüren*. `https://jaxenter.de/devops/software-tests-fuzzing-90723`, Abruf: 08.06.2020. – Version vom 10.01.2020

[Sha05] SHAFRANOVICH, Yakov: *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. `http://dx.doi.org/10.17487/RFC4180`. Version: Oktober 2005 (Request for Comments)

[Svo] SVOBODA, David: *Exploiting Java Serialization for Fun and Profit*. `https://static.rainfocus.com/oracle/oow16/sess/1461174451300001tAQ7/ppt/Exploiting%20Deserialization.pdf`, Abruf: 07.07.2020. – Version vom Juni 2019

[The] The Linux Information Project: *Path Definition.* http://www.linfo.org/path.html, Abruf: 01.07.2020. – Version vom 15.06.2006

[TMB+] Thompson, Henry S. ; Mendelsohn, Noah ; Beech, David ; Maloney, Murray ; Gao, Shudi ; Sperberg-McQueen, C. M.: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/, Abruf: 01.08.2020. – Version vom 05.04.2012

Anhang

Im Anhang befinden sich Listings, Grafiken und andere Dokumente, die für die Arbeit relevant sind, aber den Lesefluss stören. Im physischen Anhang dieser Arbeit werden sich nur Grafiken und ausgewählte Listings befinden. Um auf den Quellcode der in dieser Arbeit entstandenen Werkzeuge und der Referenzimplementierung zuzugreifen, wird auf den elektronischen Anhang, der als DVD beiliegt, verwiesen. Dadurch ist es einfacher für Leser auf den Quellcode zuzugreifen, da so kein Abtippen aus dem Anhang erfolgt und eine freie Wahl bezüglich der *Integrated Developer Environment* (IDE) oder des Texteditor besteht.

Für entstandene Werkzeuge sind im Anhang jeweils Beschreibungen und Instruktionen zur Anwendung vorhanden. Im elektronischen Anhang liegt diese Arbeit außerdem im *Portable Document Format (PDF)* vor. Um die Vollständigkeit und Korrektheit des elektronischen Anhangs zu gewährleisten, finden sich unter `hashes.md5` MD5[232]-Hashes zu sämtlichen Dateien. Diese können zur Verifizierung des korrekten Dateiinhalts verwendet werden. Der *Master Hash* für die `hashes.md5`-Datei steht unter diesem Absatz.

`e5da01319f56107bd2105a918cee2558`

Im elektronischen Anhang befindet sich außerdem unter `graja_mit_werkzeugen.zip` eine Graja-Installation, die alle in dieser Arbeit erstellten Hilfswerkzeuge enthält. Eine Konfiguration der Hilfswerkzeuge findet anhand von Umgebungsvariablen statt. Bevor diese Version genutzt werden kann, müssen Pfade in der `installation.settings` an die Umgebung des auszuführenden Rechners angepasst werden. Die Umgebungsvariablen werden innerhalb des Skripts `runGrajaGuiWithBaFeatures.sh` gesetzt. Das Skript startet dann Graja-GUI mit den gesetzten Umgebungsvariablen, die das Bewerten von Musterlösungen und somit auch das Verwenden der in dieser Arbeit entstandenen Hilfswerkzeugen erlaubt. Aufgaben mit Musterlösungen liegen als task.zip-Dateien im Verzeichnis `Graja Aufgaben` vor.[233]

Der Quellcode der im Rahmen dieser Arbeit entstandenen Referenzimplementierung befindet sich mit dem gesamten Graja-Quellcode im Verzeichnis `Referenzimplementierung`. Sämtlicher Quellcode aller Aufgaben, mit für den Regressionstestmechanismus notwendigen Änderungen befindet sich im Verzeichnis `Aufgaben Quellcode`. Es handelt sich bei diesen beiden Ordnern um eine Spieglung des `dev/#0069_herschel_selftest` *branches* im Graja-[234] und Graja-Aufgaben-Repositorium[235].

Sollte sich die DVD des elektronischen Anhangs nicht in einer an der Vorderseite der Arbeit festgeklebten Hülle befinden, bitte umgehend den Autor der Arbeit über die auf Seite 2 angegebenen E-Mail-Adresse kontaktieren!

[232] [Riv92]

[233] Es handelt sich hier um die Aufgaben, die vor der Arbeit vorgefunden wurden. Die für die Referenzimplementierung notwendigen Änderungen sind hier noch nicht vorhanden.

[234] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja`

[235] `https://lab.it.hs-hannover.de/f4-informatik/forschung/graja/graja-assignments`

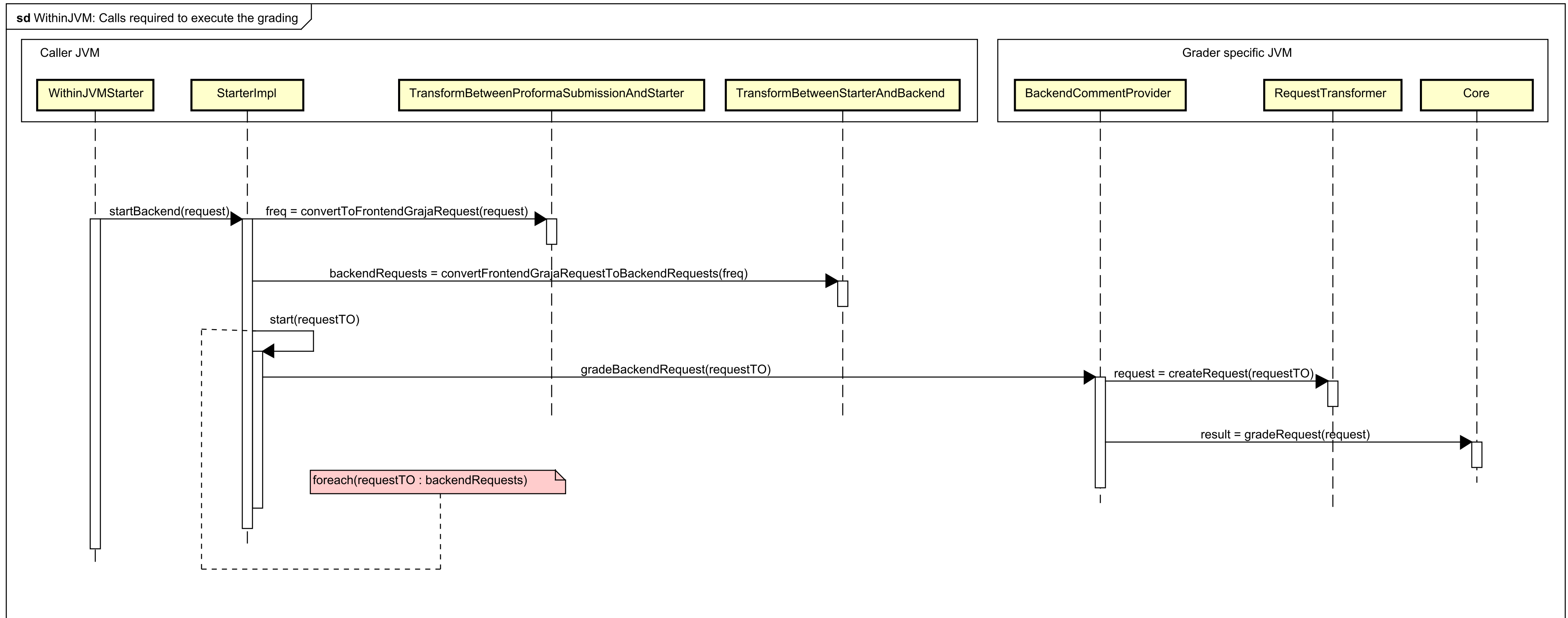


Abbildung 44: Indirekter Aufruf von `gradeRequest` über `WithinJVMStarter`

Werkzeug: Inspector

Das Werkzeug *Inspector* erlaubt es das ausgefüllte Bewertungsschema eines *Result*-Objekts auszugeben. Des Weiteren können alle Kommentare der *Result* und *AssignmentResult*-Objekte ausgegeben werden. Eine isolierte Betrachtung der Kommentare der einzelnen Bewertungsaspekt-Gruppen und Bewertungsaspekte ist ebenfalls möglich.

Inspector wurde innerhalb dieser Arbeit verwendet, um die Verwendbarkeit von Grajas interner *Result*-Datenstruktur für die Verhaltensaufzeichnung zu überprüfen. Um dieses Werkzeug zu verwenden, muss auf das *Result*-Objekt zugegriffen werden, das durch die Funktion `gradeRequest` der Klasse `de.hsh.graja.core.Core` generiert wird. Möglich ist das z.B. über die Methode `gradeBackendRequest` der Klasse `BackendCommandProvider`. Ein weiterer Punkt zum Einharken stellt die Klasse `WithinJVMBackendAPI` dar.

Ein *Inspector*-Objekt wird jeweils mit einem Pfad zu einem Verzeichnis, in dem die generierten Textdateien abgespeichert werden und einem *Result*-Objekt instanziiert. Es ist dann möglich mit der Methode `dumpAssignmentComments` die Baumstruktur der Kommentare des dazugehörigen *AssignmentResult*-Objekts zu visualisieren. Es werden sowohl die *Content*-Objekte der Graja-Kommentar-Bibliothek angezeigt als auch die Zielgruppe und das dazugehörige Level (falls vorhanden). Für alle Knoten des Baums wird außerdem noch die Anzahl der Kinder angezeigt. Es werden jeweils die *header*, *description*, *leading* und *trailing* *Content*-Objekte verarbeitet. Die Methoden `dumpResultComments` und `dumpCommentsGraded` bewirken die gleiche Ausgabe, allerdings für das *Result*-Objekt und die jeweiligen *AbstractGradingAspectResult*-Objekte des ausgefüllten Bewertungsschemas. Anhand der Methode `dumpScores` kann die Baumstruktur des ausgefüllten Bewertungsschemas und die errichten Punkte der ProFormA-Tests und `combine`-Gruppen visualisiert werden.

```
1 Result res = Core.gradeRequest(request, commentCollector);
2 Inspector i = new Inspector(Paths.get("/tmp/"), res);
3 i.dumpAssignmentComments("assignment_comments.txt");
4 i.dumpResultComments("result_comments.txt");
5 i.dumpCommentsGraded("grading_comments.txt");
6 i.dumpScores("scores.txt");
```

Listing 21: Beispielnutzung des *Inspector*-Werkzeugs mit einem *Result*-Objekt

Ein Beispiel zur Nutzung ist im oberen Listing gegeben. Der Quellcode des Werkzeugs befindet sich im elektronischen Anhang unter `Hilfswerkzeuge/Inspector`. Die `dumpScores`-Methode wurde genutzt, um den folgenden Anhang zu generieren.

Um *Inspector* mit der im elektronischen Anhang vorhandenen Graja-Version zu nutzen, müssen die Umgebungsvariablen `BA_USE_INSPECTOR` und `BA_INSPECTOR_DUMP_LOCATION` des Skripts `runGrajaGuiWithBaFeatures.sh` an das lokale System angepasst werden.[236]

[236] Version: `graja_mit_werkzeugen.zip`

```
1  SUCCESS: 0.00 [0.00/3.00]
2  GRADES: [CORRECT, WRONG]
3  ============
4
5  LEAF: Compile -> Should successfully compile
6  SUCCESS: 0 [0.00/0.00] | [CORRECT]
7  ID: @@!!ProFormA-GradingHints:testRef=compile:ProFormA-
     GradingHints:cnt=37!!@@
8
9  GROUP: Functional correctness
10 PROFORMA-FUNCTION: sum
11 SUCCESS: 0.00 [0.00/2.70] | [WRONG]
12 ID: @@!!ProFormA-GradingHints:combine=functionality!!@@
13
14   LEAF: JUnit -> Functional correctness
15   SUCCESS: 0.00 [0.00/2.70] | [WRONG]
16   ID: @@!!ProFormA-GradingHints:testRef=junit:ProFormA-
       GradingHints:subRef=de.hsh.prog.serialize.grader.Grader#
       shouldOutputCorrectResults:ProFormA-GradingHints:cnt=38!!@@
17
18 GROUP: Further aspects
19 PROFORMA-FUNCTION: sum
20 SUCCESS: 0.00 [0.00/0.30] | [CORRECT, WRONG]
21 ID: @@!!ProFormA-GradingHints:combine=furtherGradingAspects!!@@
22
23   GROUP: Maintainability
24   PROFORMA-FUNCTION: min
25   SUCCESS: 0.00 [0.00/0.15] | [CORRECT, WRONG]
26   ID: @@!!ProFormA-GradingHints:combine=maintainability!!@@
27
28     LEAF: Checkstyle -> There should be only one statement per
         line
29     SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
30     ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
         GradingHints:subRef=OneStatementPerLine:ProFormA-
         GradingHints:cnt=39!!@@
31
32     LEAF: Checkstyle -> Should avoid duplicate string literals
33     SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
34     ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
         GradingHints:subRef=MultipleStringLiterals:ProFormA-
         GradingHints:cnt=40!!@@
35
36     LEAF: PMD -> Comments needed in front of methods and
         classes
37     SUCCESS: 0.00 [0.00/0.15] | [WRONG]
38     ID: @@!!ProFormA-GradingHints:testRef=pmd:ProFormA-
         GradingHints:subRef=CommentRequired:ProFormA-GradingHints
         :cnt=41!!@@
39
40     LEAF: PMD -> Fields (attributes) should be at the start of
         the class
```

```
    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
    ID: @@!!ProFormA-GradingHints:testRef=pmd:ProFormA-GradingHints:subRef=FieldDeclarationsShouldBeAtStartOfClass:ProFormA-GradingHints:cnt=42!!@@

    LEAF: PMD -> Should adhere to variable naming conventions
    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
    ID: @@!!ProFormA-GradingHints:testRef=pmd:ProFormA-GradingHints:subRef=VariableNamingConventions:ProFormA-GradingHints:cnt=43!!@@

    LEAF: PMD -> Should adhere to method naming conventions
    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
    ID: @@!!ProFormA-GradingHints:testRef=pmd:ProFormA-GradingHints:subRef=MethodNamingConventions:ProFormA-GradingHints:cnt=44!!@@

  GROUP: Coding style
  PROFORMA-FUNCTION: min
  SUCCESS: 0.00 [0.00/0.15] | [CORRECT, WRONG]
  ID: @@!!ProFormA-GradingHints:combine=codingStyle!!@@

    LEAF: Checkstyle -> Should use indentation consistently
    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-GradingHints:subRef=Indentation:ProFormA-GradingHints:cnt=45!!@@

    LEAF: Checkstyle -> Left curly brace should be at end of line
    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-GradingHints:subRef=LeftCurly:ProFormA-GradingHints:cnt=46!!@@

    LEAF: Checkstyle -> Right curly brace should be placed correctly
    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-GradingHints:subRef=RightCurly:ProFormA-GradingHints:cnt=47!!@@

    LEAF: Checkstyle -> Tabs should be replaced by soft tabs
    SUCCESS: 0.00 [0.00/0.15] | [WRONG]
    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-GradingHints:subRef=FileTabCharacter:ProFormA-GradingHints:cnt=48!!@@

    LEAF: Checkstyle -> Lines should not exceed maximum length
    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
```

```
        GradingHints:subRef=LineLength:ProFormA-GradingHints:cnt
        =49!!@@
76
77    LEAF: Checkstyle -> Should avoid unnecessary parentheses
78    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
79    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
        GradingHints:subRef=UnnecessaryParentheses:ProFormA-
        GradingHints:cnt=50!!@@
80
81    LEAF: Checkstyle -> Code blocks should have braces
82    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
83    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
        GradingHints:subRef=NeedBraces:ProFormA-GradingHints:cnt
        =51!!@@
84
85    LEAF: Checkstyle -> Should use whitespace around operators,
         keywords, and curly braces
86    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
87    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
        GradingHints:subRef=WhitespaceAround:ProFormA-
        GradingHints:cnt=52!!@@
88
89    LEAF: Checkstyle -> There should be no whitespace before
        certain symbols and operators
90    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
91    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
        GradingHints:subRef=NoWhitespaceBefore:ProFormA-
        GradingHints:cnt=53!!@@
92
93    LEAF: Checkstyle -> Semicolon should be placed at end of
        line
94    SUCCESS: 1.00 [0.15/0.15] | [CORRECT]
95    ID: @@!!ProFormA-GradingHints:testRef=checkstyle:ProFormA-
        GradingHints:subRef=SeparatorWrap#semicolon:ProFormA-
        GradingHints:cnt=54!!@@
```

Listing 22: Anhang: Komplette Ausgabe des *Inspector*-Werkzeugs unter Ausführung der Musterlösung *wrong_fake* der Aufgabe *de.hsh.prog.serialize*

Werkzeug: Comment-Dumper

Das Werkzeug *Comment-Dumper* erlaubt es, die Kommentare eines *AssignmentResult*-Objekts und die der einzelnen Bewertungsaspekte und Aspektgruppen abzuspeichern. In dieser Arbeit wird *Comment-Dumper* dafür verwendet, die Rohdaten bezüglich der in Kapitel 5.3 durchgeführten Rauschidentifikation zu generieren.

Der Quellcode befindet sich im elektronischen Anhang unter `Hilfswerkzeuge/CommentDumper/CommentDumper.java`. Um *CommentDumper* zu verwenden, kann die Funktion `dump` aufgerufen werden. Im folgendem Listing ist eine Beispielnutzung verdeutlicht. Es wird ein *AssignmentResult*-Objekt, die *assignmentId* und ein boolescher Wert übergeben, der signalisiert, ob der Bewertungsvorgang in einer zweiten JVM stattfindet.

```
1 CommentDumper.dump(gradeResult.getAssignmentResult(), requestTO
    .getAssignment().getAssignmentSettings().
    getAssignmentMetaData().getAssignmentId(), true);
```

Listing 23: Beispielnutzung des *CommentDumper*-Werkzeugs

Um auf die zu bewertende Musterlösung zuzugreifen, wird innerhalb der Graja-GUI die ausgewählte Musterlösung in eine temporäre Datei geschrieben. Zur Verwendung von *Comment-Dumper* wird empfohlen, die im elektronischen Anhang beigelegte Graja-Version[237] zu nutzen. Bevor das Werkzeug einsatzbereit ist, müssen die Umgebungsvariablen `BA_COMMENT_DUMPER_TMP_CTX`, `BA_INSPECTOR_DUMP_LOCATION` und `BA_USE_COMMENT_DUMPER` des Skripts `runGrajaGuiWithBaFeatures.sh` angepasst werden.

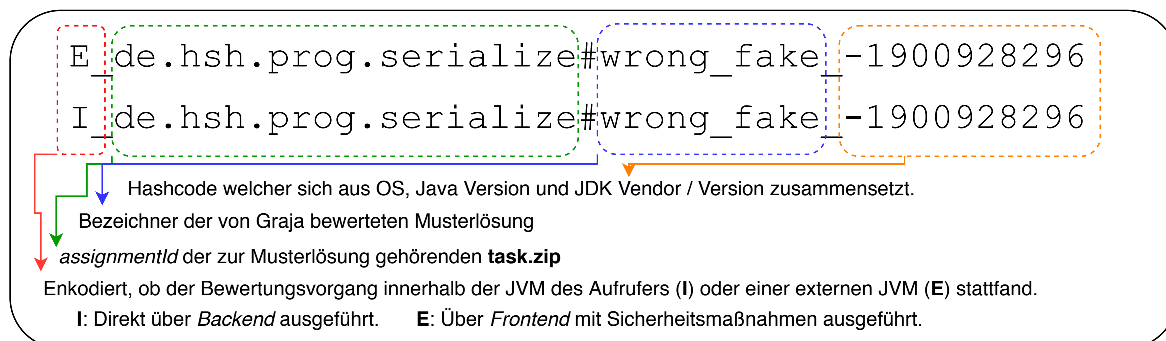


Abbildung 45: Enkodierung einer *CommentDumper*-Aufzeichnung

Die aufgezeichneten Kommentare werden in einem Verzeichnis mit dem in Abbildung 45 dargestellten Namenschema abgespeichert. Während der Aufzeichnung wird eine Linearisierung des Kommentarbaums durchgeführt. Kommentare ohne Inhalt werden in einer leeren Datei abgespeichert. Das Namenschema der extrahierten Kommentare ist in Listing 24 abgebildet. Alle Kommentare des *AssignmentResult*-Objekts verwenden das Präfix *ASSIGN_*. Positionsangaben der Kommentare werden durch die Attribute *DESCRIPTION, HEADER, LEADING* und *TRAILING* repräsentiert. Kommentare des Bewertungsschemas verwenden die Positionsangabe als Präfix. Darauffolgend ist dann der Pfad des Kommentars innerhalb des Bewertungsschemas vom Wurzelknoten ausgegeben. Knoten innerhalb des Pfads werden durch einen Punkt separiert, Aspektgruppen durch deren Namen dargestellt und für ProFormA-Tests wird das Testmodul und, falls vorhanden, die Test-Unterreferenz angegeben. Somit ist eine eindeutige

[237] Version: `graja_mit_werkzeugen.zip`

Zuordnung zwischen gleichen Kommentaren innerhalb verschiedener Aufzeichnungen gegeben.

```
1 ASSIGN_DESCRIPTION.txt
2 ASSIGN_HEADER.txt
3 ASSIGN_LEADING.txt
4 ASSIGN_TRAILING.txt
5 DESCRIPTION_root.(test=compile).txt
6 DESCRIPTION_root.functionality.(test=junit#ref=de.hsh.prog.
     serialize.grader.Grader#shouldOutputCorrectResults).txt
7 DESCRIPTION_root.functionality.txt
8 ...
```

Listing 24: Von *Comment-Dumper* extrahierte und linearisierte Kommentare

Außerdem wird pro Aufzeichnung eine `metadata.json` generiert, die Metainformationen bezüglich des verwendeten Betriebssystems, JDK und der Aufgaben-Musterlösung-Kombination enthält. Falls ein Bewertungsvorgang fehlerhaft verlaufen ist, durch z.B. eine Exception oder dem Auslaufen der Zeit wird das Attribut *error* auf *wahr* gesetzt.

```
1 {
2   "os":"Windows 10",
3   "jdkVendor":"Oracle Corporation",
4   "jdkName":"Java HotSpot(TM) 64-Bit Server VM",
5   "jdkVersion":"1.8.0_231",
6   "samplesolution":"wrong_fake",
7   "assignment":"de.hsh.prog.serialize",
8   "error": false
9 }
```

Listing 25: Beispiel einer von *Comment-Dumper* generierten *metadata.json*

Bei einem Bewertungsfehler, bei dem es zu keinem Ergebnis kommt, da kein *Result*-Objekt erstellt wurde, wird nur eine Datei mit dem Namen `ERROR.txt` und allen bis dahin erstellten Kommentaren generiert. Dafür kann die Funktion `dumpError` verwendet werden.

Um automatisch eine Liste an Aufgaben und Musterlösungen abzuarbeiten kann die Graja-GUI Version der beigelegten Graja-Installation[238] mit bestimmten Umgebungsvariablen gestartet werden. Die Umgebungsvariable `BA_RUN_BATCH_DUMP` schaltet die automatische Aufzeichnung an. Anhand von `BA_BATCH_RECORDING_DUMP_TASKS` wird das Verzeichnis, in dem die task.zip-Dateien der Aufgaben liegen spezifiziert. Über die Umgebungsvariable `BA_BATCH_RECORDING_DUMP_CONFIG` wird eine Konfigurationsdatei angegeben, die die automatische Aufzeichnung steuert. Die für diese Arbeit verwendete Konfigurationsdatei ist im elektronischen Anhang unter `Daten/batchDump.txt` auffindbar.

Während der automatisierten Aufzeichnung wird die Benutzeroberfläche von Graja-GUI einfrieren und signalisieren, dass eine Aufzeichnung stattfindet. Der Fortschritt kann über die Konsolenausgabe beobachtet werden. Nach Abschluss der Aufzeichnung beendet sich Graja-GUI selbstständig.

[238] Version: `graja_mit_werkzeugen.zip`

```
## Zufaellig Gewaehlte ##

dom.grid.zip
   - correct
   - bad.firstcol
   - bad.secondandfourthcol

de.hsh.prog.kunzestatistik.zip
   - correct
   - correct_withEnum
   - wrong_staticField

de.hsh.prog.serialize.zip
   - correct
   - wrong_fake
   - wrong_output

dom.charstat.dis.zip
   - correct

de.hsh.prog.innenwinkel.zip
   - correct
   - wrong_prompt

de.hsh.prog.doublespace.zip
   - correct
   - correct_fencepostStyle
   - wrong_lastLineBreakMissing

dom.fraction.zip
   - correct
   - wrong_validationandstyle
   - wrong_getter

de.hsh.prog.factorsengine.zip
   - correct
   - wrong_concurrentModifikationException
   - wrong_threadname
   - wrong_progress
   - wrong_jobsnotinrunningjobs

de.hsh.prog.verdoppelnelementeinliste.zip
   - wrong
   - correct1

dom.student1.zip
   - bad.syntax
   - correct
   - bad.noargcheck

## Restliche ##

```

```
53 de.hsh.prog.detectzip.zip
54     - correct
55     - wrong_byteOrder
56     - wrong_detectFilename
57
58 de.hsh.prog.ballspiele.zip
59     - correct
60     - wrong_noinheritance
61     - wrong_tooManySupermethods
62
63 de.hsh.prog.simplegraphic.dis.zip
64     - correct
65     - wrong_output1
66     - wrong_windows1252Spanish
67
68 de.hsh.prog.klassedashedline.zip
69     - correct
70     - wrong_drawMissesCutAtEndPoint
71     - wrong_roundingError
72
73 de.hsh.prog.maxfromcmdline.zip
74     - correct
75     - wrong_endlessLoop
```

Listing 26: Für die Rauschidentifikation verwendeten Aufgaben und Musterlösungen

Werkzeug: Stacktrace-Dumper

Mithilfe des Werkzeugs *Stacktrace-Dumper* werden die *Stacktrace*-Elemente eines durch Graja oder Grader-Code erstellten Kommentars ausgegeben. *Stacktrace-Dumper* kann mit der im elektronischen Anhang vorhandenen Graja-Installation[239] verwendet werden. Um alle *Stacktraces* auszugeben muss Graja-GUI mit den Umgebungsvariablen `BA_USE_CONTENT_STACKTRACE_DUMP` und `BA_CONTENT_STACKTRACE_DUMP_LOCATION` gestartet werden.

Nach einem Bewertungsvorgang werden nun die jeweiligen Klassen der aufgezeichneten Kommentarbausteine mit einer Sequenznummer und dem dazugehörigen *Stacktrace* in die vorher spezifizierte Datei geschrieben. In Listing 27 ist ein Teilausschnitt eines solchen *Stackdump* abgebildet. Die `Paragraph`-Instanz stellt die 1135te `Content`-Instanz des Bewertungsdurchlaufs dar. Darunter befindet sich jeweils ein *Stacktrace*, der die Herkunft des generierenden Codes verrät. Eine Grundfunktion des im Kapitel 5.2.3.2 beschriebenen *Stacktrace*-Machanismus ist somit innerhalb von Graja realisierbar.

```
1  de.hsh.graja.util.comment.Paragraph @ 1135
2  de.hsh.graja.modules.checkstyle.GrajaListener.addError(
      GrajaListener.java:208)
3  com.puppycrawl.tools.checkstyle.Checker.fireErrors(Checker.java
      :391)
4  com.puppycrawl.tools.checkstyle.Checker.processFiles(Checker.
      java:290)
5  com.puppycrawl.tools.checkstyle.Checker.process(Checker.java
      :217)
6  de.hsh.graja.modules.checkstyle.CheckstyleModuleRunner.run(
      CheckstyleModuleRunner.java:80)
7  de.hsh.graja.core.Core.runAllModules(Core.java:419)
8  de.hsh.graja.core.Core.gradeAssignment(Core.java:230)
9  de.hsh.graja.core.Core.gradeRequest(Core.java:107)
10 de.hsh.graja.core.apcli.BackendCommandProvider.
      gradeBackendRequest(BackendCommandProvider.java:85)
11 de.hsh.graja.core.apcli.BackendCommandProvider.executeCommand(
      BackendCommandProvider.java:51)
12 de.hsh.graja.util.cli.Command.execute(Command.java:342)
13 de.hsh.graja.util.cli.Main.main(Main.java:61)
14 de.hsh.graja.Cli.main(Cli.java:62)
```

Listing 27: Beispiel eines abgefangenen *Stacktrace* einer *Content*-Instanz

[239] Version: `graja_mit_werkzeugen.zip`

Werkzeug: Comment-Dump-Analyzer

Um die durch das Werkzeug *Comment-Dumper* generierten Rohdaten auszuwerten, wurde das Werkzeug *Comment-Dump-Analyzer* entwickelt. Es befindet sich im elektronischen Anhang unter `Hilfswerkzeuge/CommentDumpAnalyser`. Das Werkzeug wird mithilfe von Gradle gebaut und verwendet für die Berechnung der Levenshtein-Distanz die Bibliothek *Apache Commons Text*[240]. Um Zeichenketten abzugleichen wird auf die bereits bestehende Bibliothek *java-diff-utils*[241] zurückgegriffen.

Generierte Ausgaben des Werkzeugs werden im Verzeichnis `analyseDump` abgespeichert, das jeweils im derzeitigen Arbeitsverzeichnis des Programms erstellt wird. Um das Werkzeug aufzurufen, ist eine Rohdatenmenge notwendig, die mit dem ersten Kommandozeilenargument übergeben wird. Hierbei bieten sich z.B. die Verzeichnisse innerhalb der ZIP-Dateien `CommentDump.zip` oder `CommentDumpAfterRngChanges.zip` unter `Daten` im elektronischen Anhang an.

Das Werkzeug enthält eine Rauschunterdrückung, die vor einem Abgleich mithilfe regulärer Ausdrücke verschiedene Textmuster mit neutralen Zeichenketten ersetzt. Die regulären Ausdrücke, die in den Unterkapiteln des Kapitels 5.3.2 erwähnt werden, finden sich in der Klasse `ReplacePatterns` und werden automatisch zu einem gemeinsamen regulären Ausdruck zusammengefasst. Dieser finale Ausdruck ist *greedy matching* und löst bei dem ersten gefundenen Muster einen Treffer aus. Die Reihenfolge innerhalb des zusammengefassten Ausdrucks ergibt sich durch die Reihenfolge der Konstanten des Aufzählungstypen `ReplacePatterns`. Sollte bei den abzugleichenden Zeichenketten trotzdem ein Unterschied auffindbar sein, kann die Levenshtein-Distanz dazu genutzt werden, diesen Treffer zu unterdrücken, sofern der berechnete Wert unter dem definierten Schwellwert liegt.

Nach der Ausführung werden nur Berichte, in denen Differenzen sichtbar sind, in das Verzeichnis `analyseDump` geschrieben. Diese Berichte enthalten jeweils Informationen über die verwendeten Datensätze und gefundenen Differenzen. Außerdem werden nach einem Durchlauf Statistiken generiert, diese sind z.B. in Listing 28 abgebildet. Alle Steuerungsparameter des Werkzeugs sind in Tabelle 7 beschrieben. Diese können in der Klasse `Main` geändert werden.

Parameter	Beschreibung
DEBUG_PRINT	Treffer bestimmter Muster ausgeben
USE_LEVENSHTEIN	Mittels Levenshtein-Distanz Fehlalarme unterdrücken
LEVENSHTEIN_THRES	Schwellwert der Levenshtein-Distanz
IGNORE_BACKEND	Keine durch interne JVM generierte Daten verwenden
USE_ALIAS	Muster-Aliase als ersetzenden Text verwenden
PRINT_NOISE_HIT	Mustertyp bei Suchtreffer ausgeben
SUPPRESS_ERROR	Fehlgeschlagene Bewertungsdurchläufe ignorieren
SUPPRESS_NOISE	Rauschunterdrückung verwenden

Tabelle 7: Liste aller verfügbaren *Comment-Dump-Analyzer*-Steuerungsparameter

[240] `https://commons.apache.org/proper/commons-text/`
[241] `https://java-diff-utils.github.io/java-diff-utils/`

Statistiken des Werkzeugs: *Comment-Dump-Analyzer*

Die in Listing 28 dargestellten *Comment-Dump-Analyzer*-Statistiken wurden anhand des zweiten Datensatz, der im Elektronischen Anhang unter `Daten/CommentDump AfterRngChanges.zip` auffindbar ist, generiert. Dieser Datensatz unterscheidet sich von `Daten/CommentDump.zip` dadurch, dass die in Kapitel 5.3.4 angesprochenen Probleme mit den Graja-Testmodulen Checkstyle und PMD behoben wurden.

Es wurden die folgenden Einstellungen verwendet:

- Behandlung aller erkannten Differenzen als Fehlalarme, solange die Levenshtein-Distanz <= 5 ist
- Exklusion aller Datensätze, die durch die Aufrufer JVM generiert wurden
- Exklusion aller fehlerhaften Datensätze, bei denen Graja einen Bewertungsabbruch erzwungen hat und somit kein *Result*-Objekt erzeugt wurde
- Anwendung aller in `ReplacePatterns` definierten regulären Ausdrücke um durch Textersetzung eine Rauschunterdrückung vor Abgleich durchzuführen

```
{
  "cases": 40,
  "casesPassed": 35,
  "casesFailed": 5,
  "linesPassed": 76869,
  "violations": 13,
  "replacementsCount": 3198,
  "replacements": {
    "FILE_PREFIX": 92,
    "PATH_LINUX_TMP_CUTOFF": 46,
    "PATH_LINUX_FILE": 430,
    "PATH_WINDOWS": 430,
    "PATH_WINDOWS_SLASH_REVERSED": 46,
    "DATE_SIMPLETIMEFORMAT_DEFAULT": 780,
    "TIME_VALUE_IN_SECONDS": 790,
    "PROFORMA_GRADING_HINT_CNT": 584
  },
  "replacementsAliases": {
    "PATH": 1044,
    "TIMESTAMP": 1570,
    "OTHER": 584
  }
}
```

Listing 28: *Comment-Dump-Analyzer*-Statistiken nach Sortierung der Checkstyle-Dateien und PMD-Regelsätze und Exklusion der durch die Aufrufer JVM generierten Kommentare

Mit `cases` wird die Anzahl der verglichenen Paare angegeben. Die Anzahl der Abgleiche mit und ohne gefundenen Differenzen wird durch `casesFailed` und `casesPassed` angegeben. `linesPassed` enthält die Anzahl der ohne Differenzen abgearbeiteten Zeilen. Mit `violations` ist die Anzahl der gefundenen Differenzen innerhalb aller Paare gemeint. `replacementsCount` enthält die Anzahl an Ersetzungen, die zur Rauschunterdrückung vor einem Vergleich angewandt werden. Mithilfe von `replacements` wird

jeweils die Anzahl der gefundenen Muster mit einem regulären Ausdruck assoziiert. `replacementsAliases` ist das gleiche wie `replacements`, nur mit den jeweiligen Gruppen der regulären Ausdrücke (*TIME_VALUE_IN_SECONDS* und *DATE_SIMPLE TIMEFORMAT_DEFAULT* bilden z.B. die Gruppe *TIMESTAMP*).

Pseudocode für einen Algorithmus, um Baumstrukturen auf strukturelle Äquivalenz zu überprüfen und die gefundenen Differenzen auszugeben

Der folgende Pseudocode findet sich auch im elektronischen Anhang unter `pseudocode.txt`.

```
interface node {
   fun childCount() : int
   fun positionInParent(): int
   fun containsPath(offsets: list<int>): bool
   fun query(offsets: list<int>): node
}

struct backtrack {
    previous: int
    node: node
}

struct nodepath {
    path: list<node>
    @Identity offsetPath: list<int>
}

fun recursiveTraversal(root: node, lookup: list<node>, backtrack: list<backtrack>,
                       seq: sequence<int>, previous: int) : void {

    next = seq.next()
    backtrack[next] = backtrack { previous, node }
    lookup[next] = root

    for(child : node)
        recursiveTraversal(child, lookup, backtrack, seq, next)
}
```

```
28 fun backtrackPath(start: backtrack, backtracks: list<backtrack>) : list<node> {
29     stack: stack<node>
30     stack.push(start.node)
31     while(start.previous != −1)
32         start = backtracks[start.previous]
33         stack.push(start.node)
34     return stack.asList()
35 }
36
37 fun calcOffsets(path: list<node>) : list<int> {
38     offsets: list<int>
39     for(node : path)
40         offsets.add(node.positionInParent())
41     return offsets
42 }
43
44 fun handleSubtreeEqualityCheck(handledPaths: set<nodepath>, violationPaths: list<nodepath>,
45                          cutOffA: nodepath, cutOffB: nodepath, childrenA: int, childrenB: int) : void {
46     if(childrenA == childrenB || handledPaths.contains(cutOffA))
47         return
48     if(childrenA > childrenB)
49         violationPaths.add(cutOffA)
50     else
51         violationPaths.add(cutOffB)
52     handledPaths.add(cutOffB)
53 }
54
55 fun findPositionByReference(in: list<node>, target: node) : int {
56     for(i = 0, i < in.size, i++)
57         if(in[i] == node)
58             return i
59 }
```

```
60 public fun isEqualTreeStructure(rootA: node, rootB: node) : list<nodepath> {
61     lookupA: list<node>
62     lookupB: list<node>
63     backtrackA: list<backtrack>
64     backtrackB: list<backtrack>
65     seqA: sequence<int>(0)
66     seqB: sequence<int>(0)
67     violationPaths: list<nodepath>
68     handledPaths: set<nodepath>
69 
70     if(rootA.childCount() == 0 && rootB.childCount() == 0)
71         return violationPaths
72 
73     recursiveTraversal(rootA, lookupA, backtrackA, seqA, -1)
74     recursiveTraversal(rootB, lookupB, backtrackB, seqB, -1)
75 
76     max = Math.min(lookupA.size, lookupB.size)
77 
78     for(i = 0, i < max, i++)
79         childrenA = lookupA[i].childCount()
80         childrenB = lookupB[i].childCount()
81 
82         backtrackedA = backtrackPath(backtrackA[i], backtrackA)
83         backtrackedB = backtrackPath(backtrackB[i], backtrackB)
84 
85         cutOffA = nodepath { backtrackedA, calcOffsets(backtrackedA) }
86         cutOffB = nodepath { backtrackedB, calcOffsets(backtrackedB) }
87 
88         if(cutOffA != cutOffB)
89             a1 = rootA.query(cutOffA.offsetPath)
90             b1 = rootB.query(cutOffA.offsetPath)
91 
```

```
92            if(a1 && b1 && a1.childCount() != b1.childCount())
93                actualBacktrackB = backtrackPath(backtrackB[findPositionByReference(backtrackB, b1)],
                      backtrackB)
94                actualCutOffB = nodepath { actualBacktrackB, calcOffsets(actualBacktrackB) }
95                handleSubtreeEqualityCheck(handledPaths, violationPaths, cutOffA, actualCutOffB, a1.
                      childCount(), b1.childCount())
96
97            a2 = rootA.query(cutOffB.offsetPath)
98            b2 = rootB.query(cutOffB.offsetPath)
99
100           if(a2 && b2 && a2.childCount() != b2.childCount())
101               actualBacktrackA = backtrackPath(backtrackA[findPositionByReference(backtrackA, a2)],
                      backtrackA)
102               actualCutOffA = nodepath { actualBacktrackA, calcOffsets(actualBacktrackA) }
103               handleSubtreeEqualityCheck(handledPaths, violationPaths, actualCutOffA, cutOffB, a1.
                      childCount(), b1.childCount())
104
105           continue
106
107       handleSubtreeEqualityCheck(handledPaths, violationPaths, cutOffA, cutOffB, childrenA, childrenB)
108
109   return violationPaths
110 }
```

Listing 29: Pseudocode für einen Algorithmus, um Baumstrukturen auf strukturelle Äquivalenz zu überprüfen und die gefundenen Differenzen auszugeben