



Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

**Ein Campus.
Der See.
Deine Vision.**

SEITENBAU

Regression Test Selection durch Codeanalyse in einem Javaprojekt

Bachelorarbeit

Jannis Liebscher

HTWG Konstanz

08.07.2024

Bachelorarbeit

Regression Test Selection durch Codeanalyse in einem Javaprojekt

von

Jannis Liebscher

Im Tiefen Brunnen 2, 78239 Rielasingen-Worblingen

Zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

an der

Hochschule Konstanz Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Angewandte Informatik

Matrikelnummer: 301645

1. Prüfer: Prof. Dr. Markus Joachim Eiglsperger.

2. Prüfer: Achim Bitzer

Ausgabedatum: 10.04.2024

Abgabedatum: 09.07.2024

Hiermit erkläre ich, Jannis Liebscher, geboren am 14.05.2001 in Singen,

- (1) dass ich meine Bachelorarbeit mit dem Titel:

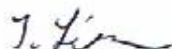
„Regression Test Selection durch Codeanalyse in einem Javaprojekt“

in der Fakultät Informatik unter Anleitung von Professor Dr. Markus Joachim Eiglsperger selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angeführten Hilfen benutzt habe;

- (2) dass ich die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.
- (3) dass die eingereichten Abgabe-Exemplare in Papierform und im PDF-Format vollständig übereinstimmen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Rielasingen, 28.06.2024



Abstract

Diese Bachelor Thesis widmet sich der Erstellung eines Maven Plugins, welches Regression Test Selection (RTS) in einem Java-Projekt durchführt. Dabei soll ein statischer Ansatz durch Codeanalyse verfolgt werden. Durch die Auswahl einiger Tests kann, im Gegensatz zur erneuten Ausführung aller Regressionstests, Zeit beim Build eingespart werden. Des Weiteren wird das Plugin in die bestehende Projekt-Infrastruktur eines Java-Projekts integriert, um es in der automatisierten Build-Pipeline verwenden zu können. Außerdem wird ein Praxistest durchgeführt, wobei das Plugin anhand einer User-Story getestet wird.

Inhaltsverzeichnis

Glossar	6
1 Einleitung	7
1.1 Motivation	7
1.2 Projektkontext	7
1.2.1 Die verschiedenen Testarten	8
1.3 Aufgabenstellung	9
1.3.1 Zielsetzung	9
1.3.2 Anforderungen	10
2 Grundlagen	11
2.1 Regression Test Selection	11
2.2 Ansätze	13
2.3 Metriken	13
2.3.1 Safety	13
2.3.2 Precision	14
2.3.3 Performance	14
2.3.4 Zielwerte	14
3 Verwandte Arbeiten	15
3.1 Ekstazi	15
3.2 STARTS	16
3.3 FaultTracer	17
4 Implementierung	18
4.1 Konzeption	18
4.2 Datenstruktur	19
4.2.1 Abbildung des Projekts	19
4.2.2 Abbildung der Abhängigkeiten	20
4.3 Finden der Abhängigkeiten	22
4.4 Dependency Injection	23
4.5 Testauswahl	23
4.6 Maven Plugin	24

4.7	Integration	25
4.8	Analyse der Maven pom.xml	25
5	Evaluierung	27
5.1	Praxistest des Plugins	27
5.2	Versuch 1 - Story nachstellen	27
5.2.1	Methodik	27
5.2.2	Fehleranalyse	28
5.2.3	Die verwendete Story	28
5.2.4	Zeitmessung	28
5.3	Ergebnisse Versuch 1	28
5.3.1	Plugin Performance	29
5.3.2	Unit Tests	29
5.3.3	Integrationstests	31
5.4	Versuch 2 - Anpassung einzelner Dateien	32
5.4.1	Methodik	33
5.4.2	Fehleranalyse	33
5.5	Ergebnisse Versuch 2	33
5.5.1	Bug im Plugin	33
5.5.2	Versuchsdaten	34
6	Fazit	36
6.1	Erfüllung der Ziele	36
6.2	Beantwortung der Forschungsfragen	37
6.3	Zusammenfassung	38
6.4	Ausblick	38
7	Literaturverzeichnis	40
	Abbildungsverzeichnis	41
	Verzeichnis der Codeausschnitte	42
	Anhang	43

Glossar

(User-)Story:	In Scrum beschreibt ein Nutzer bzw. Kunde in einer User-Story die Anforderungen zu seinem Produkt. Im Entwicklerteam von VPP steht der Begriff synonym für eine Softwareänderung, die umgesetzt werden soll.
Scrum:	Eine agile Arbeitsweise, welche von Teams verwendet werden kann.
Feature Branch:	Ein (Git-)Branch, welcher erstellt wird, um eine Softwareänderung durchzuführen. Nach Abschluss wird dieser in den Main/Master oder Entwicklungs-Branch gemerged.
Reflection:	Die Fähigkeit eines Programms seine eigene Struktur zu kennen und/oder diese zu modifizieren [Ref]. In Java ermöglicht es beispielsweise Klassen andere Klassen anhand eines Strings aufzurufen.

1 Einleitung

1.1 Motivation

In der heutigen Softwareentwicklung ist das automatisierte Testen des Codes von entscheidender Bedeutung. Die Wartezeit auf das Ergebnis der automatisierten Tests, sollte dabei so gering wie möglich ausfallen, damit zeitnah am Code weitergearbeitet werden kann. Regression Test Selection (RTS) stellt eine Möglichkeit dar, dieses Ziel zu erreichen.

Die vorliegende Thesis widmet sich der Herausforderung, auf Basis eines Commits bzw. eines Git Push automatisiert zu ermitteln, welche Tests in einem Java Softwareprojekt ausgeführt werden müssen. Tests, die von der Änderung unberührt bleiben, sollen ignoriert werden. Die Arbeit konzentriert sich auf die Entwicklung eines Algorithmus zur Codeanalyse, welcher anhand der Codeänderungen in einem Commit die zugehörigen Tests identifiziert. Das Ganze soll sich als Maven Plugin verpackt in ein bestehendes Projekt integrieren lassen. Durch unterschiedliche Optionen und Parameter soll es zudem möglich sein, dass Plugin in verschiedenen Java-Projekten zu verwenden. Um das Plugin effektiv nutzen zu können, ist es außerdem notwendig, dass es auch in der automatisierten Build Pipeline verwendet werden kann. Da moderne Pipelines immer öfter containerisiert sind, ist es zudem wichtig die Pipeline so anzupassen, dass das Plugin Informationen zwischen verschiedenen Pipeline Schritten austauschen kann.

1.2 Projektkontext

Bei dem Java-Projekt, für welches die Regression Test Selection (RTS) entwickelt wird, handelt es sich um das Vorgang-Verwaltungs-Programm VVP, welches von der Firma SEITENBAU entwickelt wird. Mit 250.000 LoC (Lines of Code) handelt es sich bei VVP um ein größeres Projekt. Es enthält verschiedene Module, wie Kalender und Terminpläne, ein Modul zur Vorgangerstellung und Einsicht, sowie ein Dashboard für wichtige und aktuelle Informationen. VVP ist eine Java Webanwendung, die auf dem Spring Framework aufbaut. Als Build Tool kommt Apache Maven zum Einsatz. Zur Projekt-Infrastruktur gehört ein Maven Mirror, eine firmeninterne Docker Registry, sowie eine Jenkins Pipeline. Wird ein Commit vom lokalen Entwickler PC in das Bitbucket Git-Repository gepusht, wird das der Pipeline mittels Git-Hooks mitgeteilt. Jenkins baut und testet anschließend das Projekt auf dem Stand des Commits. Dazu werden von Kubernetes verwaltete Con-

tainer genutzt, die jeweils einen der Pipeline Schritte ausführen. Einige Tests (z.B. Visual regression Tests) werden nur bei parametrisierten Builds ausgeführt, welche von Hand gestartet werden.

1.2.1 Die verschiedenen Testarten

In VVP gibt es eine Reihe automatisierter sowie einige manuelle Tests bzw. Testfälle. Für diese Arbeit sind nur die automatisierten Tests interessant. In Jenkins sind die unterschiedlichen Testarten jeweils einem eigenen Pipeline-Step zugeordnet.

Dazu gehören:

- Unit-Tests
- Integration Tests
- System Tests GL
- System Tests SGW
- System Tests GUI
- System Tests VRT
- System Tests BF
- Performance Tests

Unit und Integration Tests werden automatisch bei jedem Pipeline-Durchlauf durchgeführt. Die Integration Tests testen im Fall dieses Projekts nicht nur das Zusammenspiel einiger Klassen, sondern Teile des gesamten Systems. System Tests SGW und GL testen die beiden Module SGW und GL. Das Service Gateway (SGW) kümmert sich um den Datenaustausch mit anderen Systemen, während das Modul GL (Geschäftslogik) den Großteil der übrigen Funktionen bereitstellt. Bei Systemtest GUI handelt es sich um automatisierte Selenium Tests. Die Systemtests VRT (Visual Regression Test) testen die Grafische Benutzeroberfläche, indem sie Screenshots von bestimmten Seiten machen und diese mit einer Referenz vergleichen. Sie werden nur ausgeführt, falls es Änderungen an der GUI gab. Die Systemtests BF (Barrierefreiheit) testen die GUI auf Barrierefreiheit. Sie werden beim Abschluss einer Story ausgeführt, falls es Änderungen an

der GUI gab. Die Performance Tests überprüfen die Performance des gesamten Systems und werden nur vor einer neuen Lieferung ausgeführt. Eine Lieferung beschreibt in diesem Fall, das Ausliefern des aktualisierten Quellcodes, womit der Kunde die neue Softwareversion installieren kann.

Für die statische Analyse eignen sich vor allem die Unit- und Integrationstests. Die übrigen Testarten agieren teilweise auf einer grafischen Oberfläche oder beinhalten remote calls, was eine statische Analyse erschwert, oder sogar teilweise unmöglich macht.

1.3 Aufgabenstellung

Die Aufgabenstellung liegt in der Entwicklung eines Tools zur Regression Test Selection und seiner Integration in den automatisierten Build Prozess. Dieses Tool soll auf die Anwendung im Projekt VVP zugeschnitten sein, aber dabei so allgemein wie möglich gehalten werden. Da bei SEITENBAU viele weitere Projekte ebenfalls auf Java und Spring Boot basieren, kann es so auch in anderen Projekten Verwendung finden.

1.3.1 Zielsetzung

Das Ziel dieser Arbeit ist die Erstellung eines Maven Plugins, welches eine Regression Test Selection durchführen kann. Außerdem soll das Plugin in eine Jenkins Pipeline integriert werden. Insgesamt sollen folgende Ziele erfüllt werden:

Z.1. Anhand einer Liste von geänderten Dateien wählt das Plugin aus dem Set aller Tests diejenigen aus, die von den Änderungen verursachte Fehler aufdecken könnten.

Z.2. Die Zeitersparnis durch ausgeschlossene Tests übertrifft die Zeit, die zur Berechnung der betroffenen Tests benötigt wird

Z.3. Das Plugin bietet eine Möglichkeit, die Informationen über ausgewählte Tests an ein Test-Execution-Framework zu übergeben

Z.4. Das Plugin wird in eine Jenkins Pipeline integriert.

Folgende **Fragen** ergeben sich aus den Zielen:

F.1. Wie können durch statische Codeanalyse, Abhängigkeiten zwischen Testklassen und Produktivcode aufgedeckt werden?

F.2. Wie kann die Analyse in Bezug auf ihre Performance optimiert werden?

F.3. Welche Möglichkeiten gibt es für die Weitergabe der ausgewählten Testfälle?

F.4. Welche Schritte sind notwendig um das Plugin in eine Jenkins Pipeline zu integrieren?

1.3.2 Anforderungen

Aus dem Projektkontext ergeben sich einige Anforderungen, die nicht direkt zu den Zielen gehören.

Das Plugin soll unter Maven Version 3.9.5 und Java 17 lauffähig sein. Außerdem muss es Multi-Module Maven Projekte unterstützen. Projekte, in denen das Plugin verwendet wird, sollten ebenfalls Java Version 17 nutzen. Die Anpassungen an der Jenkins Pipeline beziehen sich auf Jenkins Version 2.440.2. Um Informationen über die geänderten Dateien zu erhalten, ist zudem ein Versionsverwaltungssystem notwendig. In Verbindung mit Git soll dabei eine reibungslose Integration ermöglicht werden.

2 Grundlagen

2.1 Regression Test Selection

Regressionstests sind Testfälle, die in der modernen Softwareentwicklung wiederholt ausgeführt werden, um sicherzustellen, dass Software Änderungen keine unerwarteten Fehler erzeugen. Je mehr unterschiedliche Testfälle zur Build Pipeline hinzugefügt werden, desto wahrscheinlicher ist es, dass durch Modifikationen herbeigeführte Fehler aufgedeckt werden. Allerdings steigt damit auch die Zeit, die eine Pipeline für einen Durchlauf benötigt.

Regression Test Selection, kurz RTS, stellt eine Möglichkeit dar, die Laufzeit der Pipeline zu verringern. Normalerweise wird regelmäßig, d.h. bei jedem Merge oder Commit, die komplette Test Suit ausgeführt. Auch dieses Verfahren lässt sich als RTS Strategie verstehen auf das sich die in Kapitel 2.3 aufgeführten Metriken anwenden lassen. Diese Strategie wird nachfolgend, wie auch in anderen Arbeiten zum Thema RTS[RH96, RH97, RH98, EWS⁺22], als retest-all bezeichnet. Speziell bei Unittests, aber auch bei anderen Testarten, ist ein einzelner Testcase oder auch eine Testklasse selten für das gesamte System zuständig. Im Falle von Unittests wird im Idealfall nur eine einzige Klasse getestet, eventuelle Abhängigkeiten werden gemocked, d.h. mittels eines Test Frameworks simuliert, so dass auf die eigentliche Klasse nicht zugegriffen wird. Daraus folgt, dass im Falle einer Codeänderung nur ein Teil der ausgeführten Regressionstests tatsächlich in der Lage ist, eventuell produzierte Fehler aufzudecken. Diese Grundüberlegung soll anhand von Abbildung 1 genauer erläutert werden. Die Abbildung zeigt ein vereinfachtes, von IntelliJ generiertes UML-Diagramm eines Beispielprojekts, dessen Code im Anhang (2) gefunden werden kann.

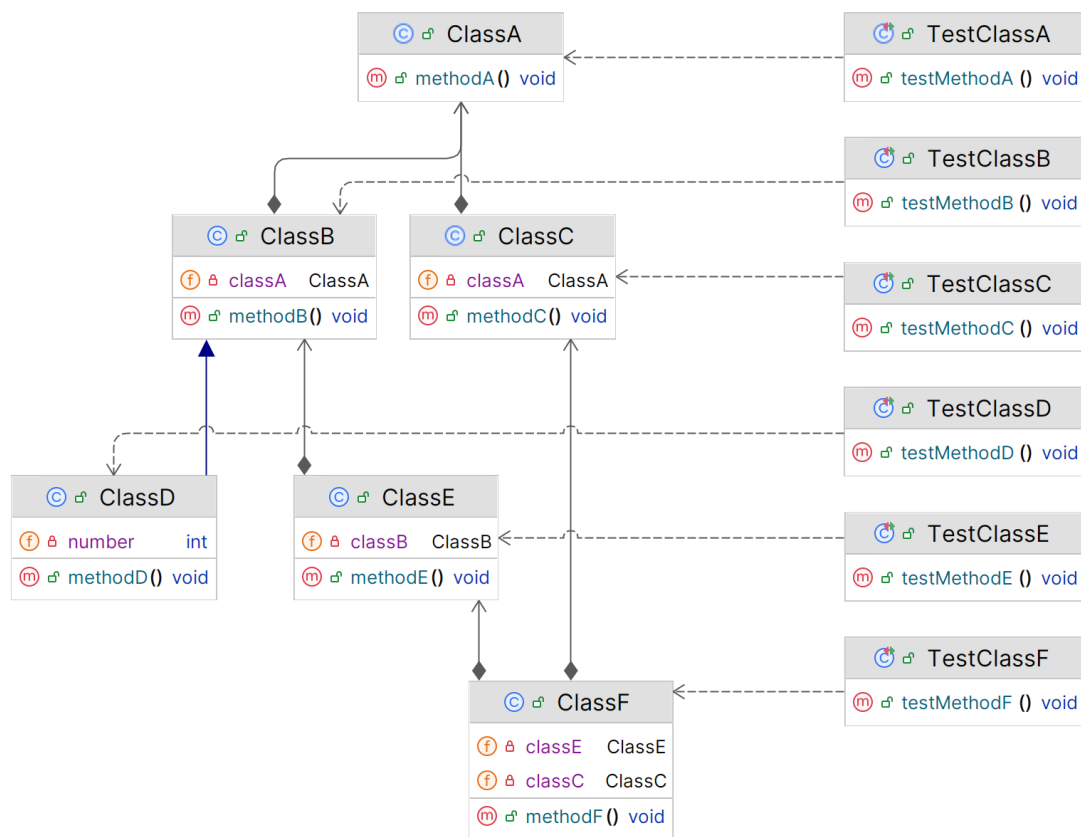


Abbildung 1: Dependency Beispiel

Das Projekt besteht aus der Komposition einiger Klassen, die sich gegenseitig als Klassenfeld enthalten und so Methoden der enthaltenen Klasse aufrufen können. ClassD stellt einen Spezialfall dar, da sie eine Unterklasse von ClassB ist. Die Testklassen stellen jeweils den Unit-Test einer bestimmten Klasse dar.

Im einfachsten Fall wird ClassF modifiziert. Da außer TestClassF keine andere Klasse eine Abhängigkeit auf ClassF hat, reicht es, diesen Test auszuführen um eventuell produzierte Fehler aufzudecken. Alle weiteren Test würden kein verändertes Verhalten zeigen. Wird ClassB modifiziert, so sind die Tests für ClassB, ClassD, ClassE und ClassF betroffen. ClassE beinhaltet ClassB als Feld und kann so auf Methoden von ClassB zugreifen. Dasselbe gilt für ClassF, da diese Klasse indirekt über ihr Feld ClassE von ClassB abhängig ist. Schließlich ist auch ClassD betroffen, da sie von ClassB erbt und somit auf nicht überschriebene Methoden oder Instanzvariablen von ClassB zugreifen kann.

RTS zielt nun darauf ab, eben jene Testcases/Klassen auszuwählen, die Fehler aufdecken könnten. Tests, die lediglich unveränderten Code ausführen, werden übersprungen, um Zeit zu sparen. Alternativ ist es außerdem möglich, Tests nicht auszuwählen, sondern lediglich zu priorisieren, also zuerst auszuführen. Dies hätte im Fehlerfall ein frühes Scheitern der Build Pipeline zur Folge.

In der Bewertung des dynamischen RTS-Tools Ekstazi [GEM15b] wird die Regression Test Selection in 3 Phasen unterteilt: Analysis(*A*), Execution(*E*) und Collection(*C*). Beschrieben werden die einzelnen Phasen wie folgt:

- Die Analysis (*A*) Phase wählt die auszuführenden Tests aus.
- Die Execution(*E*) Phase führt die ausgewählten Tests aus.
- Die Collection(*C*) Phase sammelt Informationen über die aktuelle Revision, um die Analyse der nächsten Revision zu ermöglichen.

Diese Unterteilung lässt sich auf andere RTS Strategien übertragen, die Reihenfolge kann allerdings variieren.

2.2 Ansätze

RTS Ansätze lassen sich grob in statisch und dynamisch unterteilen. Statische Ansätze analysieren den Quell- oder Bytecode, um Abhängigkeiten zwischen Produktionscode und Testklassen zu finden. Dagegen überwachen dynamische Ansätze auf welche Dateien oder Klassen während der Testausführung zugegriffen wird.

2.3 Metriken

RTS Techniken lassen sich durch drei Parameter bewerten: Safety, Precision und Performance [RH96, LHS⁺16].

2.3.1 Safety

Safety bezieht sich darauf, dass die RTS Technik wirklich alle Tests auswählt, die bei einer bestimmten Codeänderung einen Fehler aufdecken könnten. Anders ausgedrückt:

Eine RTS Technik ist sicher (safe), wenn sie niemals einen Test „vergisst“, also nicht auswählt, der nach einer Codeänderung fehlschlägt. Im Idealfall sollte eine RTS Technik komplett sicher sein, damit keine Fehler übersehen werden, da die entsprechenden Tests nicht ausgeführt wurden. Gerade für statische RTS Techniken ist es allerdings schwer komplette Sicherheit zu garantieren.

2.3.2 Precision

Precision bezieht sich auf die Anzahl bzw. die Genauigkeit der ausgewählten Testfälle. Eine RTS Technik, welche nur Tests auswählt, die aufgrund einer gegebenen Änderung fehlschlagen könnten, ist zu 100% precise.

2.3.3 Performance

Performance beschreibt schlicht, wie schnell die RTS ausgeführt werden kann. Sollte die Zeit für die Auswahl der Tests größer sein, als die Zeit, die durch das Auslassen einiger Testfälle gespart wird, ist die RTS Technik nicht Performant genug und hat keinen praktikablen Nutzen. In diesem Fall ist retest-all die bessere Alternative.

2.3.4 Zielwerte

Das Ziel von RTS Techniken ist es die Precision zu steigern. Der Overhead muss dabei so gering bleiben, dass die Performance die von retest-all übertrifft. Die Sicherheit sollte dabei nicht zu tief fallen und wenn möglich sogar bei 100% bleiben. Wie tief die Sicherheit fallen darf, hängt jeweils vom individuellen Projekt und dem Aufwand für die Korrektur von übersehenen Fehlern ab.

Retest-all lässt sich ebenfalls als RTS Technik mit folgenden Eigenschaften betrachtet:

- Da immer alle Tests ausgewählt werden, ist sie zu 100% safe.
- Da immer alle Tests ausgewählt werden, werden auch immer alle „überflüssigen“ Tests ausgewählt. Damit ist retest-all zu 0% precise.
- Da retest-all den Standard darstellt, ist auch die Performance als neutral zu bewerten.

3 Verwandte Arbeiten

Es existieren eine Reihe von Ansätzen zur RTS, sowie ein paar fertige Tools. Nachfolgend werden einige der bekannteren Arbeiten aufgelistet. Zudem wird darauf eingegangen, warum sie für das in Kapitel 1.2 genannte Projekt nicht anwendbar sind.

3.1 Ekstazi

Bei Ekstazi handelt es sich um ein dynamisches RTS Tool in Form eines Maven Plugins. Es überwacht während der Testausführung, welche Dateien von einem Test aufgerufen werden. Auf diese Weise wird ein dependency file erstellt, welcher zu jeder Test Klasse alle ihre Abhängigkeiten enthält. Des Weiteren speichert Ekstazi die Checksummen zu jeder .class Datei um so festzustellen, ob eine Datei geändert wurde, ohne auf ein Versionsverwaltungssystem zurückgreifen zu müssen [GEM15a].

Ekstazi stellt derzeit eines der wenigen, nutzbaren Tools zur Regression Test Selection dar, wurde aber schon länger nicht mehr aktualisiert. Die aktuelle Version von Ekstazi im Maven Central Repository stammt aus dem Jahr 2018 [mvn]. Die größte Hürde für den Einsatz in VPP stellt die nicht vorhandene Unterstützung von JUnit 5 dar. Die Tatsache, dass es sich bei Ekstazi um ein dynamisches RTS Tool handelt, bringt außerdem eigene Probleme mit sich. So kann die dynamische Überwachung der Test Dependencies zur Laufzeit sehr zeitintensiv sein. Des Weiteren können in Echtzeitsystemen Timeouts auftreten[LHS⁺16].

Aus der Funktionsweise von Ekstazi geht zudem hervor, dass Probleme mit neu hinzugefügten Dateien auftreten können. Während neue Tests einfach präventiv ausgeführt werden, kann Ekstazi, bei z.B. der Ersetzung einer Klasse durch eine Andere, diese neue Dependency erst nach erneuter Testausführung erkennen. Gerade im Falle von Dependency Injection kann es hier zu Problemen kommen. Neue Klassen, die ein bestehendes Interface implementieren oder eine bestehende Klasse erweitern, können in andere Klassen injiziert werden, ohne angepasst werden zu müssen. Da das Spring Framework Dependency Injection vereinfacht und somit fördert, kann dieser Edge Case in Spring Projekten nicht einfach ignoriert werden.

3.2 STARTS

STARTS (STAtic Regression Test Selection) ist ebenfalls ein Maven plugin, welches aber im Gegensatz zu Ekstazi statische Regression Test Selection ausführt [Sta]. Dafür analysiert es den Bytecode und erzeugt mithilfe des Kommandozeilenwerkzeugs jdeps einen Dependency Graph. Dieser enthält zu jedem Typen seine jeweiligen Abhängigkeiten. Über diesen Graphen können dann, für einen gegebenen, geänderten Typ, die von der Änderung betroffenen Typen und somit die betroffenen Testklassen gefunden werden. Auch STARTS nutzt Checksummen, um die geänderten Klassen zu bestimmen. Eine bekannte Ursache für das nicht Auswählen betroffener Testfälle, stellt Reflection dar[LSM17]. Ohne erheblichen Mehraufwand ist es für statische RTS Ansätze nicht möglich, dieses Konzept zu berücksichtigen, obwohl es auch für dieses Problem schon Ansätze gibt[SHTZ⁺19].

Ein bisher nicht genanntes Problem stellt aber auch dynamischer Polymorphismus in Bezug auf Dependency Injection dar. Codebeispiel 1, welches einen Teilcode des Projekts aus Abbildung 1 beinhaltet, verdeutlicht das Problem. Die Klasse ClassE enthält ClassB als Member, welche auch als Abhängigkeit im Constant Pool auftaucht. Die Klasse ClassD erweitert nun ClassB und kann daher mittels Dependency Injection zur Laufzeit von ClassE verwendet werden. Für die Test Selection folgt dadurch: Wird ClassD geändert, beeinflusst dies möglicherweise das Testergebnis der Klasse ClassETest (welche ClassE testet). ClassETest sollten also ausgeführt werden. Da aber weder ClassE noch ClassB eine Abhängigkeit auf ClassD haben, ist dies nicht der Fall.

Codebeispiel 1: Polymorphismus und DI

```
public class ClassB{}  
public class ClassD extends ClassB{}  
  
public class ClassE  
{  
    private ClassB member;  
}
```

3.3 FaultTracer

FaultTracer ist ein Forschungs-Prototyp für statische Regression Test Selection, welcher nicht als Plugin oder Ähnliches zur Verfügung steht. Im Gegensatz zu den anderen aufgelisteten Arbeiten, arbeitet FaultTracer ausschließlich auf Methoden Ebene. Das heißt, dass eine Klasse nicht als ganzes eine Dependency darstellt, sondern jede ihrer individuellen Methoden für sich. Auch bei der Auswahl der Tests werden nicht ganze Testklassen, sondern einzelne Methoden ausgewählt [ZKK11]. Diese feinere Granularität ermöglicht es FaultTracer, im Allgemeinen weniger Tests auszuwählen, als vergleichbare Werkzeuge, die auf Klassenebene operieren. Studien haben jedoch gezeigt, dass die Zeitersparnis durch weniger ausgewählte Tests durch die längere Analysedauer wieder aufgehoben wird. Das geht so weit, dass FaultTracer sogar teilweise langsamer ist, als retest-all [LHS⁺16], [GEM15b].

4 Implementierung

Die Regression Test Selection wird über ein Maven Plugin ausgeführt, wobei für einige Schritte ein Zugriff auf Git benötigt wird. Da es sich um eine statische Analyse handelt, die den Quellcode des Projekts einliest und auswertet, muss dieser verfügbar sein. Das Einlesen und Parsen des Quellcodes geschieht mithilfe der Bibliothek JavaParser [JPa]. Zudem gelten weiterhin die in Kapitel 1.3.2 gelisteten Anforderungen. Da das Beispielprojekt VPP keine Reflection nutzt, wird dieses Konzept in der Implementierung nicht beachtet.

4.1 Konzeption

Da im Normalfall eine Java Datei genau eine Klasse enthält (mit Ausnahme von inneren Klassen), werden diese Begriffe in diesem Kapitel synonym verwendet. Interfaces und Enums können für den Zweck von RTS genauso behandelt werden wie Klassen. Daher wird auf diese Spezialfälle nicht gesondert eingegangen. Wie unter anderem aus dem Paper über STARTS[LHS⁺16] hervorgeht, birgt die präzisere Class-Level Regression Test Selection einen recht hohen Overhead. Daher fiel die Entscheidung beim Entwickeln dieses Plugins auf Class bzw. File-Level RTS.

Die zentrale Frage, die für die (statische) Regression Test Selection beantwortet werden muss, ist folgende: Wenn sich eine Datei ändert, welche Testklassen können daraufhin potenziell ein anderes Verhalten zeigen? Der Ansatz, welcher in dieser Arbeit verfolgt wird, lässt sich grob in 3 Schritte unterteilen:

- Einlesen des Quellcodes und Aufnahme der Klassen in eine geeignete Datenstruktur.
- Verbindung der Klassen, welche Abhängigkeiten zueinander haben.
- Auswahl der betroffenen Testklassen, auf Grundlage einer Menge von geänderten Klassen.

Die in Kapitel 2.1 genannten Phasen lassen sich auch hier wieder finden. Die Analysis(*A*) Phase wird von Schritt 1 und 2 abgedeckt, während die Auswahl der betroffenen Testklassen der Collection(*C*) Phase gleich kommt. Execution(*E*) bezeichnet schließlich das Ausführen der ausgewählten Tests.

Es stellt sich die Frage, ob sich Punkt 1 und 2, die beide zur Analysis(A) Phase beitragen, nicht zu einem Schritt vereinen lassen. Warum dies nicht möglich ist, wird in Kapitel 4.2 genauer erläutert.

4.2 Datenstruktur

Die Struktur zur Abbildung der Klassen und ihrer Abhängigkeiten zueinander bildet einen gerichteten, azyklischen Graphen. Dieser besteht aus den drei Klassen `ClassNode`, `PackageNode` und `PackageTree`. Auf diese drei Klassen, zu sehen in Abbildung 2, wird in Kapitel 4.2.2 genauer eingegangen. Sobald der oben genannte Schritt 2, die Verbindung der abhängigen Klassen, ausgeführt wurde, handelt es sich streng genommen um zwei verschachtelte, gerichtete Graphen, welche allerdings getrennt voneinander betrachtet werden können.

4.2.1 Abbildung des Projekts

Der erste Graph dient der Darstellung der Projektstruktur im Programmcode. Alle Klassen des Projekts werden eingelesen und jeweils in einer `ClassNode` gespeichert. Um bei der Suche nach bestimmten `ClassNodes` Zeit zu sparen, wird die Paketstruktur von Java nachgebildet. Dafür gibt es neben der Klasse `ClassNode` die beiden Klassen `PackageNode` und `PackageTree`. Eine `PackageNode` repräsentiert ein Paket des Projekts und zugleich einen Knoten im Graph. Wie ein Java Paket, kann eine `PackageNode` sowohl Klassen (`ClassNodes`), als auch weitere Unterpakete beinhalten. Die Referenzen auf Unterpakete sind in diesem Fall die Kanten des Graphen.

`PackageTree` stellt die eigentliche Datenstruktur dar und verwaltet das Einfügen, sowie Suchen von `ClassNodes` oder `PackageNodes`. Dabei wird sichergestellt, dass die Struktur der verschachtelten Pakete korrekt aufrechterhalten wird. Eine neue `PackageNode` „com.example“ wird beispielsweise in die `PackageNode` „com“ eingefügt. Sollte „com“ noch nicht existieren wird das Paket in diesem Schritt erstellt. Auch `ClassNodes` werden beim Einfügen in die entsprechenden `PackageNodes` abgelegt. So wird die gesamte Paketstruktur des eingelesenen Projekts nachgestellt. Wird im `PackageTree` eine Klasse gesucht, kann der `PackageTree` anhand des fully qualified name (Paketname + Klassen-

name) durch die Knoten(Pakete) navigieren. So muss nur ein Bruchteil der ClassNodes betrachtet werden, um die gesuchte Klasse zu finden. Auch die Suche nach allen Klassen eines bestimmten Pakets wird so auf dieselbe Weise beschleunigt, ein Fakt welcher beim Sammeln der Abhängigkeiten hilfreich ist. Dieser erste Graph ist lediglich eine Representation des Java-Projekts und gib noch keinen Aufschluss auf die Abhängigkeiten unter den Klassen.

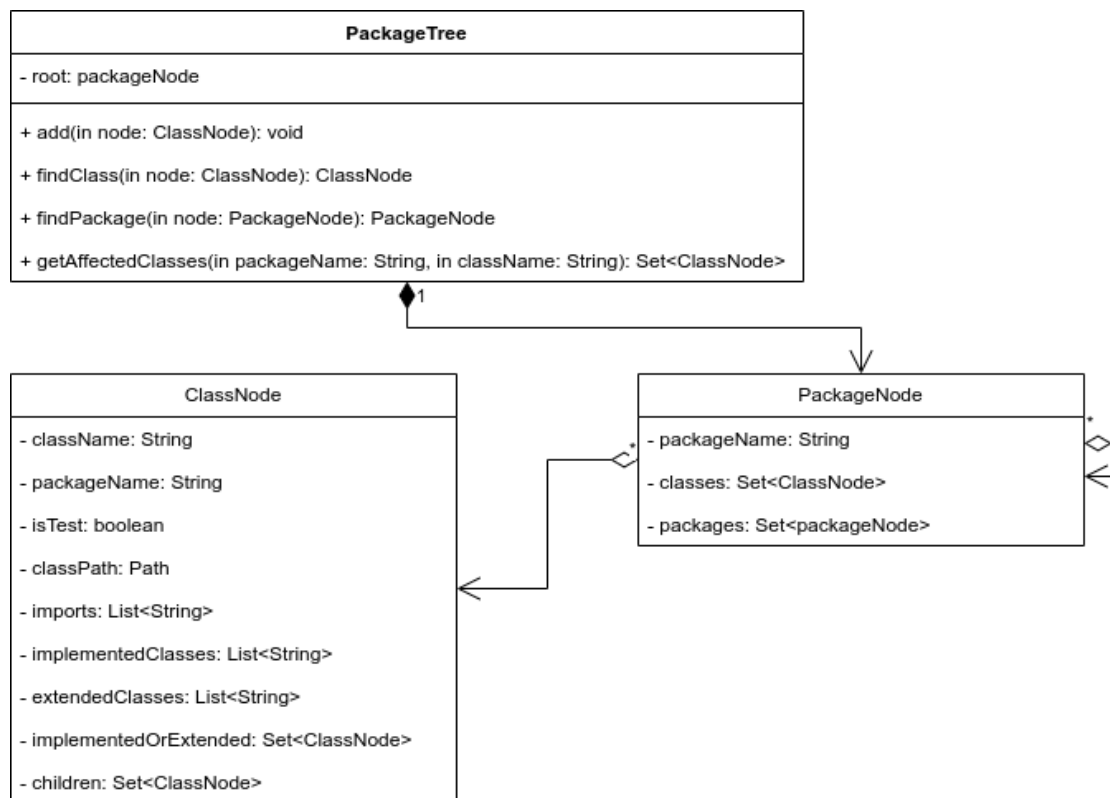


Abbildung 2: UML-Diagramm der Datenstruktur

4.2.2 Abbildung der Abhängigkeiten

Zum Speichern der Java Quelldateien dient die Klasse `ClassNode`, welche die für die Regression Test Selection relevanten Attribute beinhaltet. Dazu gehören vor allem Klassen- und Packagename, Importstatements, sowie implementierte (Interface) und erweiterte (Superclass) Klassen. Obwohl Java keine Mehrfachvererbung für Klassen unterstützt, muss

für das Attribut `extendedClasses` eine Liste gewählt werden, da eine `ClassNode` auch ein Interface enthalten kann, welches wiederum von mehreren Interfaces erben kann.

Um im späteren Verlauf der Analyse Testklassen leichter identifizieren zu können, wird die in JUnit genutzte „`@Test`“ Annotation gematcht, und das Ergebnis als Attribut „`is-Test`“ gespeichert. Da es Probleme mit vererbten Testklassen geben kann, bei denen die Testmethoden nur in der Oberklasse zu finden sind, wird zudem geschaut, ob der Klassenname den Substring „`Test`“ enthält. Außerdem enthält eine `ClassNode` ein Set von `ClassNode` Objekten (`children`), welches genutzt wird, um die Abhängigkeiten darzustellen.

Mit den `ClassNodes` als Knoten und den Referenzen auf abhängige Klassen als Kanten ergibt sich ein zweiter gerichteter Graph. Dieser stellt die Projektstruktur nur noch indirekt (durch die gespeicherten Importstatements) dar, erlaubt dafür aber die Abbildung der Klassen Abhängigkeiten. Zur Erleichterung der anschließenden Suche der betroffenen Testfälle wird die Richtung der Abhängigkeiten jedoch invertiert. Das heißt, dass eine `ClassNode` nicht speichert, auf welche Klassen sie eine Abhängigkeit hat. Stattdessen enthält eine `ClassNode X` eine Liste von `ClassNodes`, welche eine Abhängigkeit auf Klasse `X` haben. Da diese `ClassNodes` von `X` abhängig sind, werden sie im Verlauf dieser Arbeit auch als `Children` von `X` bezeichnet. Die Testklassen bilden daher den Großteil der Blätter des Graphen, da sie von den Klassen abhängig sind, die sie testen, selbst aber nicht aufgerufen werden (außer durch das Test-Framework). Weitere Blätter wären Controller, die über REST Schnittstellen aufgerufen werden, oder die Main Klasse. Klassen, die sich im selben Paket befinden, können ohne ein entsprechendes Importstatement aufgerufen werden. Diese müssen daher gesondert behandelt werden, mehr dazu in Kapitel 4.3.

Zurückblickend auf Codebeispiel 1 ergibt sich folgende Beziehung: Sowohl `ClassE` als auch `ClassD` sind „`Children`“ von `ClassB`. `ClassE`, da sie `ClassB` als member verwendet, `ClassD`, weil sie `ClassB` erweitert. Abbildung 3 veranschaulicht die entstehende Beziehung.

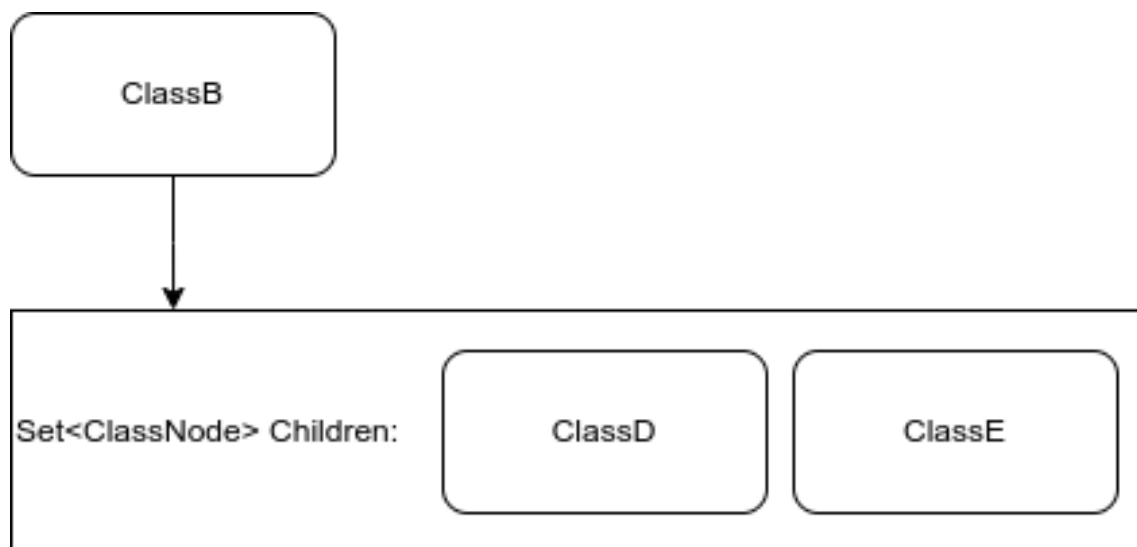


Abbildung 3: Vereinfachter Dependency Graph

4.3 Finden der Abhängigkeiten

Sobald das eingelesene Projekt in den PackageTree aufgenommen wurde, können abhängige Klassen miteinander verbunden werden. Dafür wird der Iterator des PackageTree aufgerufen, um jede enthaltene ClassNode zu erhalten. Anschließend wird für jedes Importstatement die entsprechende Klasse im PackageTree gesucht. Sollte es sich dabei um eine 3rd Party Bibliothek handeln wird diese im PackageTree nicht gefunden. In diesem Fall passiert nichts, da davon ausgegangen wird, dass sich 3rd Party Bibliotheken nicht ändern. In Kapitel 4.8 wird ein Ansatz vorgestellt, mit dem aber auch geänderte 3rd Party Bibliotheken berücksichtigt werden können.

Wird die Klasse gefunden, so wird die Klasse mit dem Importstatement bei der gefundenen als „Child“ eingetragen, um die Abhängigkeit anzuzeigen. Klassen aus demselben Paket müssen nicht explizit importiert werden. Um auch für diese alle Abhängigkeiten abzubilden, müssen die Java-Dateien noch einmal eingelesen werden. Der Inhalt kann dann auf das Vorkommen von Klassennamen desselben Pakets geprüft werden. Bei diesem Schritt hilft die Struktur des PackageTree wieder dabei, schnell die Klassen eines Pakets zu sammeln, die geprüft werden müssen.

Während dem Verbinden abhängiger Klassen wird zudem ein weiterer Schritt als Vorbereitung auf die Testauswahl ausgeführt. Von implementierten Interfaces und erweiterten

Klassen ist bisher nur der fully qualified name als String in der entsprechenden ClassNode gespeichert. Da zum Verbinden der abhängigen Klassen über alle ClassNodes iteriert wird, bietet es sich hier an, anhand dieses Strings die entsprechenden Klassen im PackageTree zu suchen und die entsprechende Referenz in der ClassNode abzuspeichern. Wird später beispielsweise das ClassNode Objekt der Oberklasse einer ClassNode gebraucht, kann direkt darauf zugegriffen werden, statt diese erneut im PackageTree suchen zu müssen, genaueres dazu in Kapitel 4.4.

4.4 Dependency Injection

Dependency Injection inklusive dynamischer Bindung ist ein grundlegendes Konzept des Spring Frameworks. Der bisher beschriebene Ansatz beachtet dieses Konzept jedoch nicht. Anhand von Codebeispiel 1 lässt sich das notwendige Verhalten für die Testauswahl herleiten. Aus dem Quellcode kann nicht abgeleitet werden, welche Implementierung zur (Test-)Laufzeit für den Typ ClassB verwendet wird (sowohl ClassB als auch ClassD kommen infrage). Nun könnten die verschiedenen Konfigurationsmöglichkeiten von Spring betrachtet werden, um die richtige Klasse zu finden. Dies würde zu einem genaueren Ergebnis führen, da nun klar wäre, welche Klasse zur Laufzeit aufgerufen wird. Einfacher ist es aber, für geänderte bzw. von Änderungen betroffene Klassen die Interface- und Superclass Children als betroffene Klassen aufzunehmen. Im Codebeispiel 1 sähe das wie folgt aus: Wenn sich ClassD geändert hat, wird in der entsprechenden ClassNode geprüft, ob diese Klasse weitere Klassen implementiert oder erweitert. Ist dies der Fall, so sind nicht nur Klassen betroffen, die ClassE direkt referenzieren, sondern auch jene, die ClassB referenzieren. Daher sollten alle „echten“ Children von ClassB zum Set der betroffenen Klassen hinzugefügt werden. In diesem Kontext sind „echte“ Children jene Klassen, die eine andere referenzieren, ohne diese zu implementieren/erweitern. Im Beispiel ist ClassE ein „echtes“ child von ClassB, eine neue Klasse ClassX welche ClassB erweitert aber nicht. Zugehörige Tests sollten demnach nicht ausgewählt werden.

4.5 Testauswahl

Nach der vorangegangenen Vorbereitung ist die eigentliche Testauswahl nicht mehr komplex. Die geänderten Klassen werden im PackageTree gesucht. Welche Dateien als „geändert“ betrachtet werden sollen, wird dem Algorithmus über das Maven Plugin mitgeteilt,

mehr dazu in Kapitel 4.6. Für jede geänderte Datei wird anschließend rekursiv über die Children, also die abhängigen Klassen iteriert. So werden alle direkt und alle transitiv abhängigen Klassen gefunden. Die abhängigen Klassen aller geänderten Dateien werden einem Set hinzugefügt, um Dopplungen zu eliminieren. Zuletzt werden alle Klassen, die nicht als Test markiert wurden, wieder entfernt.

4.6 Maven Plugin

Das Maven Plugin agiert als Wrapper für das Datenmodell und die Funktionen zum Finden der betroffenen Tests und erleichtert so die Benutzung. Es gibt zwei Möglichkeiten das Plugin aufzurufen. Einerseits kann direkt eine Liste der geänderten Dateien übergeben werden. Diese könnten z.B. aus dem Jenkins Job ausgelesen werden.

Alternativ kann dem Plugin eine Revision ID übergeben werden. Hierfür muss ein Umfeld geschaffen werden, (Ausführung in der Repository Root, sowie entsprechende Berechtigungen) in dem das Plugin Git Befehle ausführen kann. Das Plugin ermittelt dann mittels „git diff“ die geänderten Dateien zwischen Revision ID und dem HEAD des aktuellen Branches. Letzteres ist die bevorzugte Variante, da es die Integration in Jenkins erleichtert, mehr dazu in Kapitel 4.7.

In beiden Fällen braucht das Plugin Zugriff auf den Quellcode der aktuellen Revision um so, wie oben beschrieben, die betroffene Testfälle ermitteln zu können. Diese werden als kommaseparierte Liste in eine Datei geschrieben. Diese Datei kann nun als Parameter für das Maven Surefire Plugin verwendet werden, um die enthaltenen Testfälle auszuführen. Da in der Jenkins Pipeline sowohl die Regression Test Selection, als auch die Unit- und Integrationstests in einer eigenen Stage ausgeführt werden, ist ein direkter Aufruf des Surefire Plugins nicht sinnvoll.

Das Plugin bietet außerdem weitere Parameter an, um sein Verhalten zu steuern. Beispielsweise was im Fall einer geänderten Nicht-Java Datei geschehen soll. Diese werden nicht analysiert, daher können vom Plugin keine betroffenen Testfälle ermittelt werden. Der Nutzer kann hier entscheiden, ob in diesem Fall die komplette Analyse übersprungen, und alle Testfälle ausgeführt werden sollen.

4.7 Integration

Um das Plugin automatisiert in einer Jenkins Pipeline auszuführen, sind einige Schritte notwendig. Zunächst wird das Plugin in das firmeninterne Maven Repository gepusht. So kann es beim Aufruf in der Pipeline direkt aus diesem heruntergeladen werden. Anschließend wird eine eigene RTS Stage in die Pipeline eingebaut, in welcher ein Maven Container zur Verfügung steht. Alternativ können bereits existierende Pipeline Steps, wie der Build, genutzt werden, da diese ebenfalls auf Maven Container zurückgreifen. Dies hat den weiteren Vorteil, dass das Git Repository nicht erneut geklont werden muss, um den Quellcode für die Analyse bereitzustellen.

Die Pipeline wird zudem um ein Skript erweitert, welches über „git rev-parse HEAD“ die aktuelle Revisions ID ausliest, speichert und über das copyArtifacts-Jenkins-Plugin für nachfolgende Pipeline Jobs verfügbar macht. Dies geschieht jedoch nur nach erfolgreichem Abschluss des Pipeline Jobs. So wird für den Fall, dass es fehlgeschlagene Tests gibt, oder eine neue Pipeline gestartet wird, bevor die vorherige beendet wurde, eine ältere Revision für die Regression Test Selection verwendet. Fehlgeschlagenen Tests werden so immer erneut ausgeführt.

Dies ist wichtig, da ansonsten folgender Fall eintreten könnte: ClassA wird verändert, TestClassA schlägt deshalb fehl. Im nächsten Commit wird beispielsweise nur eine Konfigurationsdatei angepasst, es werden also keine Tests ausgewählt. Die Pipeline ist nun „grün“, also erfolgreich, obwohl der Test für ClassA immer noch fehlschlagen würde, falls er ausgeführt wird.

Nachfolgende Pipeline Jobs laden also die Revision ID der letzten erfolgreichen Pipeline und rufen damit das RTS Plugin auf. Hat das Plugin die Liste der auszuführenden Tests in die Datei testsToExecute.txt geschrieben, wird die Datei über den Pipeline Step „stash“ gespeichert. Nachfolgende Steps wie die Unit-Test-Stage können die Datei dann über „unstash“ laden und als Aufrufparameter für das Surefire Plugin verwenden.

4.8 Analyse der Maven pom.xml

Im Zuge der Entwicklung des Maven Plugins wurde die Möglichkeit untersucht, auch geänderte 3rd Party Bibliotheken in der Regression Test Selection zu berücksichtigen.

Durch den Zugriff auf Git hat das Plugin die Möglichkeit, den alten Stand aller sich im Projekt befindenden pom.xml Dateien wiederherzustellen. Voraussetzung ist, dass die Revision, gegen die der aktuelle Stand verglichen werden soll, als Parameter übergeben wurde. Die Alten, sowie die neuen pom.xml Dateien können nun geparkt und analysiert werden. Dafür eignet sich die Bibliothek „org.apache.maven.maven-model“. Mit ihr können die pom.xml Dateien eingelesen und in Java Objekte umgewandelt werden. Aus diesen kann man die Dependencies extrahieren und jeweils (für Alt und Neu) in eine Liste aufnehmen. Jetzt wird die symmetrische Differenz gebildet, d.h. es wird geprüft, welche Dependencies sich nur in einer der beiden Listen befinden. Nur Dependencies, bei denen die Attribute groupId, artifactId und version übereinstimmen, werden als gleich betrachtet. So wird eine Liste aus Dependencies aufgebaut, die als „geändert“ betrachtet werden können.

An diesem Punkt können geänderte Dependencies, wie in Kapitel 4.2.1 beschrieben, ebenfalls in den PackageTree eingefügt werden. Geänderte 3rd Party Bibliotheken werden nun beim Finden der Abhängigkeiten (Kapitel 4.3) fast wie alle anderen Klassen behandelt. Nur der Schritt zu prüfen, ob die Bibliothek selbst Abhängigkeiten auf andere Klassen hat, wird übersprungen. Geänderter transitive Abhängigkeiten werden also nicht berücksichtigt.

Dieser Ansatz erlaubt es dem Plugin Tests auszuwählen, die z.B. Abhängigkeiten auf „org.projectlombok.lombok“ besitzen (Direkt, oder transitiv durch Klassen, von denen die Testklasse abhängig ist), sollte diese Bibliothek in einer anderen Version verwendet werden. Dies funktioniert auch für entfernte Bibliotheken, wobei in diesem Fall der Build fehlschlagen sollte, bevor es überhaupt zur Testauswahl bzw Ausführung kommt. Die Zeit für das Einlesen und analysieren der pom.xml Dateien beläuft sich bei dem Beispielprojekt VPP mit 100 direkten Abhängigkeiten auf wenige Sekunden.

Da dieser Ansatz zum Zeitpunkt des nachfolgenden Praxistests (5.1) jedoch nicht fertiggestellt war, lässt sich über die gesamte Performance keine Aussage treffen. Der Ansatz stellt jedoch eine Möglichkeit dar, das Plugin bzw RTS weiter zu verbessern.

5 Evaluierung

5.1 Praxistest des Plugins

Um sowohl die tatsächliche Zeitersparnis festzustellen, als auch um Fehlerquellen (fälschlicherweise ausgelassene Testfälle) zu erkennen, soll das Plugin in der Praxis eingesetzt werden. Dafür wird eine bereits abgeschlossene Story mit dem integrierten Plugin wiederholt. Da bei diesem Experiment zufälligerweise eine Story ausgewählt wurde, bei der es keine fehlgeschlagenen Tests gab, wird ein zweiter Versuch durchgeführt, um gezielt die Sicherheit (vgl. 2.3.1) des Plugins zu überprüfen. Daher werden in einem zweiten Versuchsaufbau einzelne Dateien modifiziert, um gezielt Test Fehlschläge zu provozieren. So können die vom Plugin ausgewählten Tests mit denen verglichen werden, die bei der Ausführung aller Tests fehlschlagen.

5.2 Versuch 1 - Story nachstellen

5.2.1 Methodik

Um den Produktionsbetrieb zu simulieren, wird in einem neuen Branch der Startpunkt einer bereits umgesetzten Story wiederhergestellt. Das heißt, dass eine Quellcode Version wiederhergestellt wird, wie sie unmittelbar vor Umsetzung der Story aktuell war. Anschließend wird das Plugin in die Build Pipeline integriert. Per Cherry-Pick werden nun die einzelnen Commits aus dem ursprünglichen Feature-Branch der Story auf den neuen Test-Branch angewandt. Die einzelnen Build Jobs werden jeweils zu Ende laufen gelassen, bevor der nächste Commit ausgewählt wird. Parallel wird in einem zweiten Branch derselbe Vorgang ohne das eingebaute Plugin gespiegelt. Dieser Branch dient als Kontrollgruppe und zugleich als Benchmark für die Testlaufzeit. Für diesen Test laufen die Unit- und Integrationstests. Das sind jene Tests die auch im normalen Betrieb automatisch bei jedem Commit ausgeführt werden. Da die Laufzeit für diese Tests teilweise um einige Minuten abweichen kann, werden alle Testlaufzeitwerte aus der Kontrollgruppe ohne Plugin gemittelt, um einen Durchschnitt zu erhalten. Gegen diesen Wert können anschließend die Build Jobs mit aktiviertem Plugin verglichen werden, um die Differenz zu erhalten.

5.2.2 Fehleranalyse

Das größte Fehlerpotential in diesem Versuch stellt die Auslastung der Build Server dar. Wie in Kapitel 5.3 zu sehen sein wird, variiert die Laufzeit zweier Jobs teils beachtlich, obwohl dieselben Schritte ausgeführt werden. Durch die Berechnung eines Mittelwerts sollte dieser Fehler jedoch minimiert werden.

5.2.3 Die verwendete Story

Bei der verwendeten Story ging es um die Erweiterung einer bestehenden Schnittstelle. In 19 Commits wurden in 5 Modulen des Projekts insgesamt 35 Dateien angepasst. Die Commits reichen von solchen, in denen nur eine README.md Datei angepasst wurde, über einzelne Java Dateien bis hin zu größeren Änderungen bis maximal 10 modifizierten Dateien.

5.2.4 Zeitmessung

Die Zeiten wurden jeweils der Jenkins Web-UI entnommen. Für Unit- und Integrations-tests beinhaltet das jeweils die komplette Stage. Dabei ist ein gewisser Overhead für Shell Skripte und Post-Stage Aktionen enthalten. Dieser beläuft sich jedoch auf lediglich 1%. Da zudem beide Branches (Plugin-Branch und Kontroll-Branch) diesen Overhead enthalten, wird dieser vernachlässigt.

Nicht zu vernachlässigen sind jedoch die Schwankungen in der Laufzeit, hervorgerufen durch die Auslastung der Server, auf denen Jenkins läuft. Da diese Server von einigen anderen Projekten bei SEITENBAU verwendet werden, gibt es hier teils große Laufzeitunterschiede, selbst wenn die Konfiguration dieselbe ist. Die Unittests des Kontroll-Branches laufen im Mittel 5:42 min mit einer Min/Max Differenz von 1:32 min und einer Standardabweichung von 0:30 min. Die Integrationstests laufen im Mittel 24:52 min mit einer Min/Max Differenz von 6:53 min und einer Standardabweichung von 2:08 min.

5.3 Ergebnisse Versuch 1

Sowohl der Branch mit aktiviertem RTS-Plugin, als auch der Kontroll Branch liefen ohne fehlgeschlagene Tests durch. Daher kann dieser Versuch nur Auskunft über die Performance des Plugins geben. Um Daten für die Werte Precision und Safety zu erhalten

wurde daher ein zweiter Versuch durchgeführt, mehr dazu in Kapitel 5.4.

5.3.1 Plugin Performance

Da das RTS-Plugin in die bereits vorhandene Build Stage der Pipeline integriert wurde, entsteht lediglich durch seine Ausführung ein gewisser Overhead. Dieser liegt bei durchschnittlich 23 Sekunden, mit einer Standardabweichung von 2 Sekunden. Die Laufzeit des Plugins variiert also nur minimal. Die Anzahl oder Art der modifizierten Dateien scheint keinen merklichen Einfluss auf die Laufzeit zu nehmen. Das wird durch weitere Zeitmessungen innerhalb des Plugins untermauert. Dieses misst die Zeit, die für einzelne Schritte benötigt wird, und gibt diese Werte über die Standardausgabe aus. Während das Einlesen der Klassen, sowie die Erstellung der Klassenabhängigkeiten jeweils fast 50% der Plugin Laufzeit ausmacht, ist das Finden der betroffenen Testklassen nur noch für einen Bruchteil der Gesamtlaufzeit verantwortlich. So lässt sich schlussfolgern, dass vor allem 2 Faktoren für die Plugin-Performance relevant sind:

Die Projektgröße, bzw die Anzahl der Java-Dateien, sowie die Anzahl der Klassenabhängigkeiten untereinander. Rückblickend auf Kapitel 4.3 findet sich noch ein weiterer Faktor, welcher eine Auswirkung auf die Plugin Laufzeit haben kann: Da für Klassen im selben Paket eine manuelle Überprüfung der Abhängigkeiten notwendig ist, lässt sich schlussfolgern, dass Große Pakete, also solche in denen sich viele Klassen befinden, die Laufzeit ebenfalls verlangsamen können.

5.3.2 Unit Tests

Abbildung 4 zeigt die Testlaufzeit der simulierten Story, vom ersten bis hin zum letzten Commit. Bis auf einen Ausreißer sind die Unit-Tests mit aktiviertem RTS-Plugin immer schneller als die ohne. Der Ausreißer ist dabei auf einen Pipeline-Job zurückzuführen, bei dem das RTS-Plugin (fast) alle Tests ausgewählt hat. Durch die Schwankungen im System war dann der Job mit dem Plugin langsamer als der ohne, obwohl beide Jobs dieselbe Arbeit verrichtet haben. Bei den übrigen Testläufen ist eine teils deutliche Reduktion in der benötigten Zeit zu erkennen. Der Durchschnitt beträgt dabei 2:31 Minuten, was gegenüber dem Durchschnitt von 5:42 Minuten des Kontroll-Branches eine Einsparung von 62,13% ergibt.

Auch unter Beachtung der durchschnittlichen Laufzeit des RTS-Plugins von 23 Sekunden

ergibt sich somit eine beachtliche Einsparung. Da das Plugin pro Pipeline-Job außerdem nur einmal laufen muss, kann die Plugin-Laufzeit auf die beiden Stages Unittests und Integrationtests aufgeteilt werden, wodurch sich nur jeweils ein Overhead von 11,5 Sekunden ergibt.

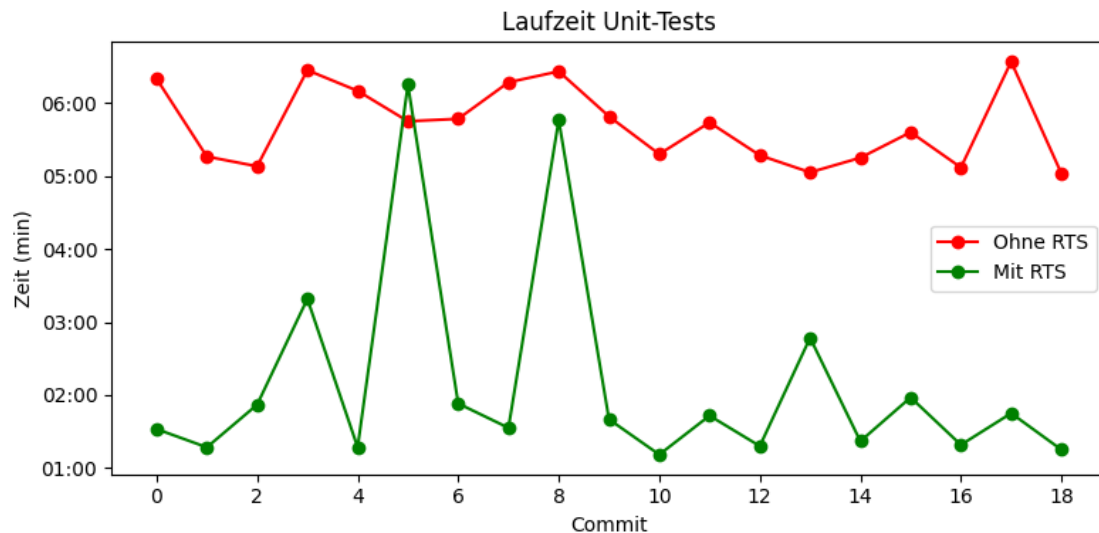


Abbildung 4: Laufzeit Unit-Tests mit und ohne RTS

Es stellt sich die Frage, ob die Größe von Commits (Anzahl der geänderten Dateien) eine Auswirkung auf die Anzahl der ausgewählten Tests und damit auf die Testlaufzeit hat. Abbildung 5 zeigt, dass das bei den Unittests nur bedingt der Fall ist. Größere Commits haben zwar tendenziell eine längere Laufzeit, jedoch gibt es sowohl nach oben, als auch nach unten Ausreißer.

Zudem zeigt sich, dass ein Großteil der Pipeline-Jobs in unter 2 Minuten abgeschlossen wird. Das 75%-Quantil für die Unittests liegt bei exakt 2 Minuten.

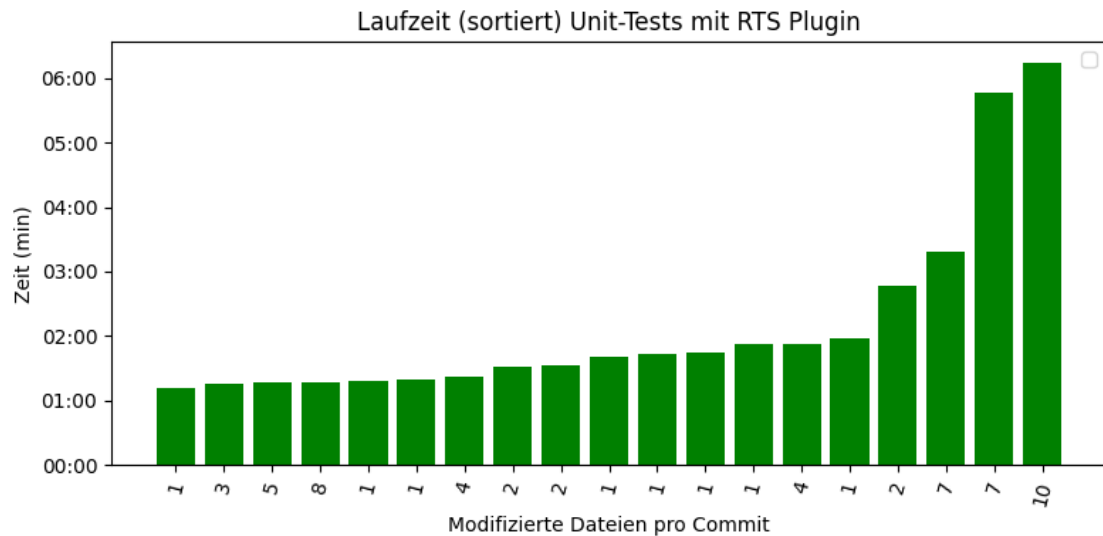


Abbildung 5: Sortierte Laufzeit der Unit-Tests

5.3.3 Integrationstests

Wird die durchschnittliche Pipeline-Laufzeit betrachtet, ergibt sich für die Integrationstests ein ähnliches Bild wie für die Unit-Tests. In Abbildung 6 sind die Testlaufzeiten der einzelnen Commits zu sehen. Die Hoch und Tiefpunkte stimmen mit denen der Unittests überein, allerdings gibt es einige Ausreißer (Vor allem Commit 2 und 14, zu einem gewissen Grad auch 6 und 7). Die durchschnittliche RTS-Laufzeit von 8:12 Minuten gegenüber 24:52 Minuten ergibt eine Einsparung von 67,02%. Wie in Abbildung 7 zu erkennen ist, ist diese Einsparung aber vor allem auf eine Reihe sehr schneller Jobs zurückzuführen, in denen nur wenige Tests ausgeführt wurden. Da die Integrationstest-Stage mit allen Tests recht lange benötigt, wirken sich Pipeline-Läufe mit wenigen oder gar keinen ausgeführten Tests stark auf den Durchschnitt aus. Dies spiegelt sich auch in der hohen Standardabweichung von 8:03 Minuten wieder.

Auch bei den Integrationstests spielt die Anzahl der modifizierten Dateien eine eher untergeordnete Rolle, wie aus Abbildung 7 hervorgeht.

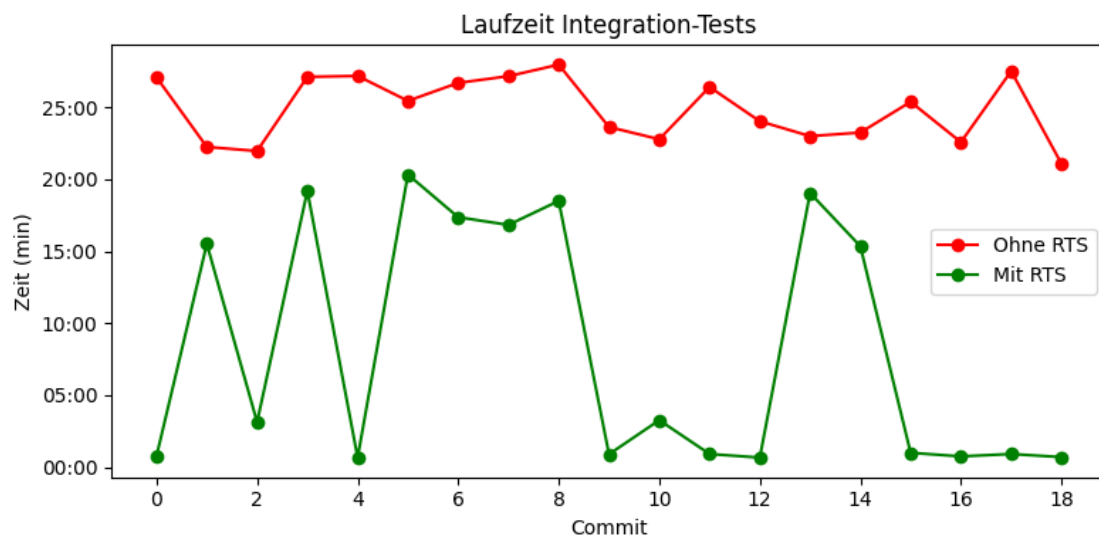


Abbildung 6: Laufzeit Integrationstests mit und ohne RTS

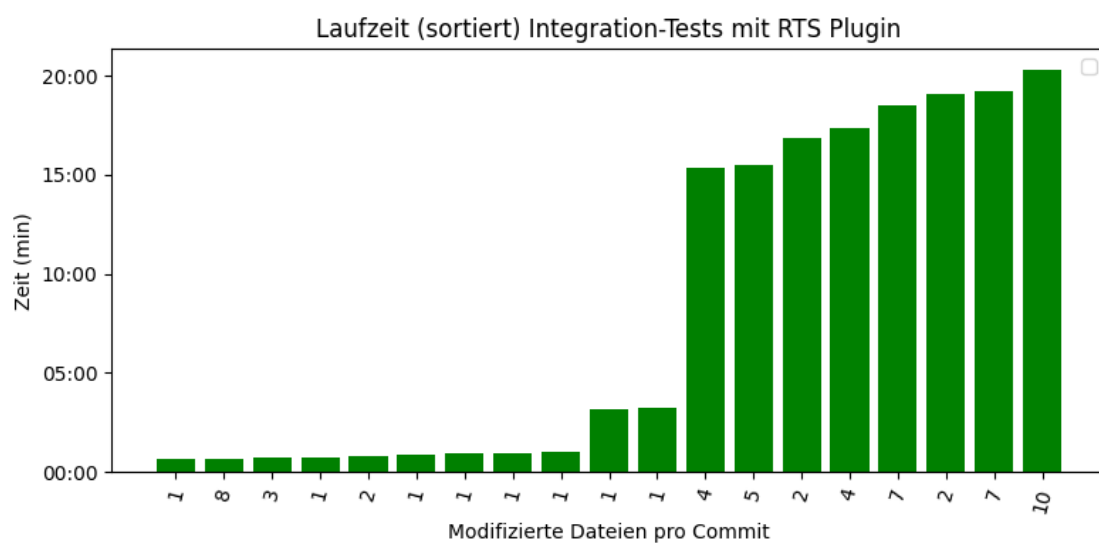


Abbildung 7: Sortierte Laufzeit der Integrationstests

5.4 Versuch 2 - Anpassung einzelner Dateien

Da es bei Versuch 1 keine Testfehlschläge gab, wurde dieser zweite Versuch ausgeführt.

5.4.1 Methodik

In diesem Versuch wurden einzelne Klassen wie Controller, Services, Mapper oder DAOs (Data Access Objects) modifiziert. Kurz gesagt wurden alle Methoden einer Klasse „zerstört“, indem sie manuell so angepasst wurden, dass sie immer null als Rückgabewert liefern. Im Falle von Methoden ohne Rückgabewert (void) wurde einfach jegliche Logik entfernt. Anschließend wurde die Kontroll-Branch Pipeline gestartet, welche wieder alle Tests ausführt. Je nachdem, welche Parameter bzw. Werte von den Tests überprüft werden, schlagen so die meisten abhängigen Tests fehl. Die fehlgeschlagenen Tests, sowie die Liste aller Tests, die das RTS-Plugin für die entsprechende Änderung ausgewählt hat, wurden gesammelt und anschließend verglichen. Mittels eines Pythonskripts wurde ausgewertet, welche Tests fehlgeschlagen sind, aber nicht ausgewählt wurden. Zudem wurden die fehlgeschlagenen sowie ausgewählten Tests gezählt, um Rückschlüsse auf die Precision des Plugins zu ziehen. In diesem Versuch wurde nicht mehr zwischen Unit- und Integrationstests differenziert, sondern beide Testarten gesammelt betrachtet.

5.4.2 Fehleranalyse

Dieser Versuchsaufbau birgt leider ein paar Ungenauigkeiten. Da sich diese jedoch nur durch eine zeitintensive, manuelle Analyse eines kleineren Projekts beseitigen lässt, wurde der Versuch trotzdem durchgeführt.

Zum einen lässt sich nicht sicherstellen, dass durch das „zerstören“ der Methoden auch wirklich alle abhängigen Tests fehlschlagen. Einzelne Tests könnten beispielsweise null als Rückgabewert erwarten.

Da der Zeitaufwand für das Modifizieren der Dateien, die Pipelinelaufzeit sowie das Sammeln der Daten mit 1 Stunde pro Datei nicht unerheblich ist, konnte außerdem nur eine kleine Anzahl getestet werden. Ein größeres Set an Versuchen könnte hier genauere Ergebnisse liefern.

5.5 Ergebnisse Versuch 2

5.5.1 Bug im Plugin

Bei einem ersten Versuchsdurchlauf wurde direkt ein Bug im Plugin gefunden. Nach dessen Behebung wurde der eigentliche Versuch, dessen Ergebnisse hier präsentiert wer-

den, durchgeführt. Allerdings hat dieser Bug selbst schon einige Erkenntnisse hervorgebracht.

Der „VppBaseIntegrationtest“ stellt die Grundlage (Oberklasse) für 150 weitere Integrationstests dar. Dieser Test enthält zahlreiche Objekte aus dem Produktivcode und ist daher von vielen Klassen bzw. Dateien abhängig. Dies hat zur Folge, dass „VppBaseIntegrationtest“, und somit die 150 Integrationstests, die ihn erweitern, sehr häufig ausgewählt werden, obwohl nur wenige der ererbenden Tests wirklich fehlschlagen können. Man könnte hier von „falschen“ Abhängigkeiten sprechen, da diese ererbenden Tests zwar theoretisch auf viele Klassen zugreifen, dies in der Praxis aber nicht tun. Es gibt jedoch keine einfache Möglichkeit festzustellen, welches diese „falschen“ Abhängigkeiten sind. Daher müssen vorsorglich alle ererbenden Tests ausgewählt werden, was zu einer (unnötig) langen Pipelinelaufzeit führt.

Die erste Erkenntnis aus diesem Versuch ist also, dass die Testarchitektur einen großen Einfluss auf die Precision von statischer Regression Test Selection hat. Würden alle Abhängigkeiten direkt in den eigentlichen Tests deklariert werden, könnte das Plugin genauer jene Tests identifizieren, die bei einer Änderung das Potenzial besitzen fehlschlagen.

5.5.2 Versuchsdaten

Das Verhältnis zwischen ausgewählten und fehlgeschlagenen Tests ist in Abbildung 8 zu sehen. Die grünen „Diff“ Balken geben hier Tests an, die fehlgeschlagen sind, vom RTS Plugin jedoch nicht ausgewählt wurden. Eine nachfolgende Untersuchung ergab, dass alle diese „vergessenen“ Tests auf denselben Fehler zurückzuführen sind. Der in Kapitel 4.4 beschriebene Mechanismus zur Beachtung von Dependency Injection bzw. dynamischer Bindung wurde nicht bei allen Schritten angewandt. Daher wurden Tests nicht ausgewählt, wenn folgende Bedingungen eintrafen:

- Eine Klasse K, welche nicht selbst verändert wurde, wurde als „betroffen“ markiert.
- Ein Test T hat eine Abhängigkeit auf ein Interface, bzw. die Oberklasse die von K implementiert bzw. erweitert wird.
- Mittels Dependency Injection verwendet T zur Laufzeit Klasse K.

Durch einige wenige Änderungen konnte dieser Bug probeweise behoben werden. Allerdings konnte der Praxistest aus Zeitgründen nicht wiederholt werden. Da das Beheben

dieses Bugs zu mehr ausgewählten Tests führt, wäre es notwendig, sowohl Versuch 1 als auch Versuch 2 zu wiederholen, um verlässliche Daten für die neue Version zu erhalten. Da alle „vergessenen“ Tests auf denselben Fehler zurückgeführt werden konnten, kann man nach dem Beheben des Bugs zuversichtlich sein, dass die Sicherheit für Projekte ohne Reflection bei 100% liegt.

Betrachtet man nur die 6 Versuche, bei denen alle fehlgeschlagenen Tests ausgewählt wurden, so kommt man bei durchschnittlich 7,17 fehlgeschlagenen Tests, pro modifizierter Datei, auf durchschnittlich 205,33 ausgewählte von insgesamt 1138 Tests. Da Tests in diesem Fall jedoch ganze Testklassen sind, sollten diese Werte mit Vorsicht betrachtet werden, wenn es darum geht die Effektivität des Plugins zu bewerten. Da Testklassen in ihrer Laufzeit stark variieren können, ist für diesen Zweck eine Analyse der Pipelinelaufzeit wie in Versuch 1 (Kapitel 5.2) besser geeignet.

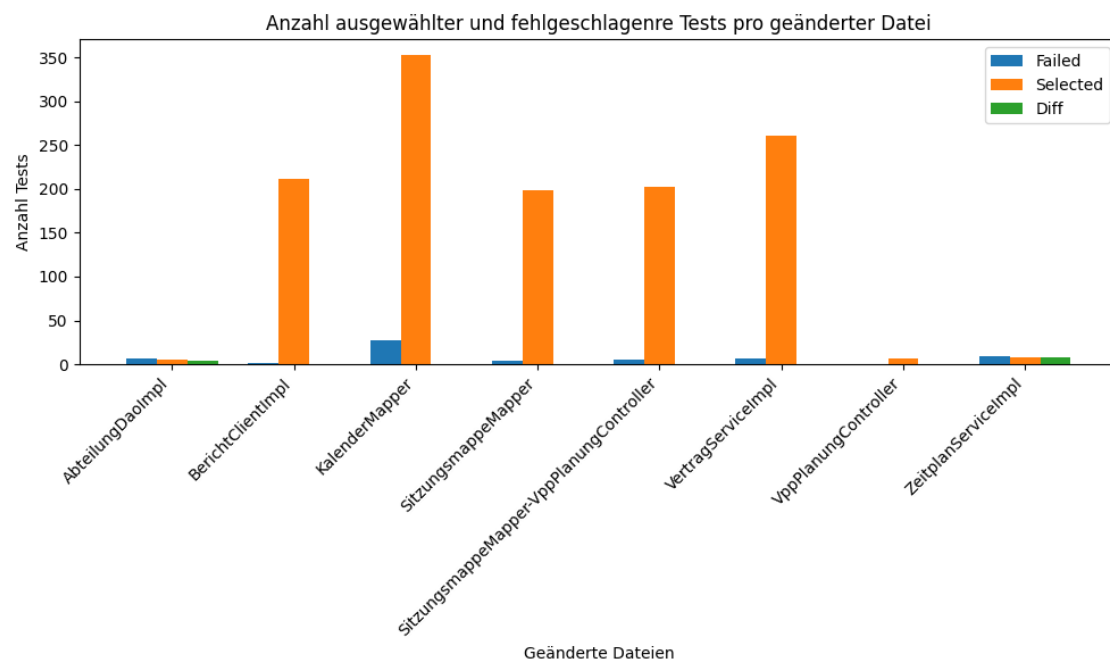


Abbildung 8: Ausgewählte und fehlgeschlagene Tests in Versuch 2

6 Fazit

6.1 Erfüllung der Ziele

Hier werden die in Kapitel 1.3.1 gesetzten Ziele erneut betrachtet, und auf ihre Erfüllung geprüft.

Z.1. Anhand einer Liste von geänderten Dateien wählt das Plugin aus dem Set aller Tests diejenigen aus, die von den Änderungen verursachte Fehler aufdecken könnten.

Das Plugin ist in der Lage eine Liste von geänderten Dateien als Parameter einzulesen. Alternativ errechnet es diese Liste mittels Git und einer Revisionsnummer welche als Parameter übergeben wird. In beiden Fällen wird auf Grundlage der geänderten Dateien ein Set von betroffenen Testfällen errechnet. Somit wurde dieses Ziel erfüllt.

Z.2. Die Zeitersparnis durch ausgeschlossene Tests übertrifft die Zeit, die zur Berechnung der betroffenen Tests benötigt wird

Die Laufzeit des Plugins liegt bei durchschnittlich 23 Sekunden, vgl. Kapitel 5.3.1. Zudem lag die Zeitersparnis sowohl bei den Unit- als auch den Integrationstests bei über 60% und somit mehreren Minuten. Somit wurde dieses Ziel erfüllt.

Z.3. Das Plugin bietet eine Möglichkeit, die Informationen über ausgewählte Tests an ein Test-Execution-Framework zu übergeben

Unter Berücksichtigung von Ziel **Z.4.** wurde hier eine einfache Lösung mittels einer output Datei gewählt, welche die auszuführenden Tests enthält. Dies stellt eine rudimentäre Lösung dar, die gerade für die Benutzung an einem lokalen Entwickler Rechner noch um z.B. den direkten Aufruf des Maven Surefire Plugins ergänzt werden kann. Der Inhalt dieser Datei kann jedoch durch einfache Shell Kommandos als Parameter an das Plugin übergeben werden. Somit wurde dieses Ziel erfüllt.

Z.4. Das Plugin wird in eine Jenkins Pipeline integriert.

In Kapitel 4.7 wurde beschrieben, durch welche Schritte sich das Plugin in eine Jenkins Pipeline integrieren lässt. Somit wurde dieses Ziel erfüllt.

6.2 Beantwortung der Forschungsfragen

Hier werden die in Kapitel 1.3.1 gestellten Forschungsfragen beantwortet, bzw. auf die Kapitel verwiesen, in denen sie beantwortet wurden.

F.1. Wie können durch statische Codeanalyse Abhängigkeiten zwischen Testklassen und Produktivcode aufgedeckt werden?

Wie in Kapitel 4 beschrieben, können durch Analyse der Import Deklarationen die Abhängigkeiten der Klassen untereinander aufgedeckt werden. Außerdem wird beschrieben, wie mit Spezialfällen, wie Klassen des selben Pakets oder Dependency Injection (Kapitel 4.4), umgegangen werden kann. Daraus ergibt sich ein Algorithmus, welcher zu einer geänderten Datei jene Testfälle findet, welche von dieser abhängig sind. Das Ergebnis ist eine Liste von Testklassen, welche möglicherweise ein geändertes Verhalten zeigen bzw. fehlschlagen.

F.2. Wie kann die Analyse in Bezug auf ihre Performance optimiert werden?

Durch die Verwendung einer geeigneten Datenstruktur kann bei der Analyse der Abhängigkeiten und der anschließenden Suche nach betroffenen Testklassen Zeit gespart werden (vgl. Kapitel 4.2).

F.3. Welche Möglichkeiten gibt es für die Weitergabe der ausgewählten Testfälle?

Eine einfache Möglichkeit stellt das Erstellen einer Datei dar, welche eine Liste an auszuführenden Tests enthält. Diese eignet sich auch für die Integration in eine Pipeline und wurde daher in dieser Arbeit ausgewählt.

F.4. Welche Schritte sind notwendig um das Plugin in eine Jenkins Pipeline zu integrieren?

Die wesentlichen Schritte für die Integration des Plugins in eine Jenkins Pipeline wird in Kapitel 4.7 ausführlich beschrieben.

6.3 Zusammenfassung

In dieser Arbeit wurde ein Plugin zur Regression Test Selection entwickelt, präsentiert und auf seine Effektivität geprüft. Dabei hat sich gezeigt, dass eine statische Analyse des Quellcodes sehr ressourcen-schonend durchgeführt werden kann, und dabei nutzbare Ergebnisse liefert, um die Pipelinelaufzeit zu verkürzen. Das Plugin liefert in seiner jetzigen Version noch kein perfektes Ergebnis in Bezug auf Sicherheit. Es wurde jedoch ein Lösungsansatz aufgezeigt durch welchen die Sicherheit auf 100% gesteigert werden kann, sollte das Projekt nicht auf Reflection zurückgreifen.

6.4 Ausblick

Diese Arbeit liefert einige Punkte, an denen zukünftige Arbeiten anknüpfen können. Der in Kapitel 4.8 präsentierte Ansatz zur Analyse der Maven pom.xml kann weiterverfolgt werden, um das Plugin auch für Änderungen in diesem Bereich sinnvoll nutzen zu können.

Weiterhin kann untersucht werden, wie die Genauigkeit dieser Form von statischer Regression Test Selection verbessert werden kann. Dadurch, dass direkt der Quellcode analysiert wird, wäre es theoretisch möglich, Abhängigkeiten zu identifizieren, die im Test gemockt werden. Beschränkt man diese Analyse auf die Testklassen, könnte bei überschaubarem Mehraufwand eine höhere Genauigkeit erzielt werden. Aber auch andere „falsche“ Abhängigkeiten, wie z.B. die in Kapitel 5.5.1 beschriebenen, könnten durch genauere Analyse identifiziert werden. Insgesamt sollte geprüft werden, welche Information dem Quellcode entnommen werden können, um die Genauigkeit der Abhängigkeiten zu erhöhen.

Auch an der Verbesserung der Plugin Performance könnte geforscht werden. Für das Beispielprojekt VPP ist die Plugin Laufzeit zwar gering, für größere Projekte kann diese aber signifikant sein. Hier könnte die Möglichkeit des Cachens, des in Kapitel 4.2 beschriebenen PackageTree untersucht werden. Sollte es möglich sein den PackageTree einmal zu bauen und anschließend anhand der geänderten Dateien zu aktualisieren, könnte auch an dieser Stelle Zeit eingespart werden.

Außerdem wären weitere Praxistests notwendig, um Schwächen und Fehler des Plugins aufzudecken, oder die genaue Auswirkung der vorgeschlagenen Verbesserungen zu analysieren. Auch die in Kapitel 5.5.1 angesprochene Architektur der Testklassen und die damit verbundene Genauigkeit des Plugins, bietet Möglichkeiten an diese Arbeit anzuknüpfen.

7 Literaturverzeichnis

Literatur

- [EWS⁺22] ELSNER, Daniel ; WUERSCHING, Roland ; SCHNAPPINGER, Markus ; PRETSCHNER, Alexander ; GRABER, Maria ; DAMMER, René ; REIMER, Silke: Build System Aware Multi-language Regression Test Selection in Continuous Integration. In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, S. 87–96
- [GEM15a] GLIGORIC, Milos ; ELOUSSI, Lamyaa ; MARINOV, Darko: Ekstazi: Lightweight Test Selection. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering Bd. 2*, 2015, S. 713–716
- [GEM15b] GLIGORIC, Milos ; ELOUSSI, Lamyaa ; MARINOV, Darko: Practical regression test selection with dynamic file dependencies. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. New York, NY, USA : Association for Computing Machinery, 2015 (ISSTA 2015). – ISBN 9781450336208, 211–222
- [JPa] *JavaParser*. <https://javaparser.org>, . – Accessed: 2024-05-30
- [LHS⁺16] LEGUNSEN, Owolabi ; HARIRI, Farah ; SHI, August ; LU, Yafeng ; ZHANG, Lingming ; MARINOV, Darko: An extensive study of static regression test selection in modern software evolution. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : Association for Computing Machinery, 2016 (FSE 2016). – ISBN 9781450342186, 583–594
- [LSM17] LEGUNSEN, Owolabi ; SHI, August ; MARINOV, Darko: STARTS: STAtic regression test selection. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, S. 949–954
- [mvn] *Maven Ekstazi*. <https://mvnrepository.com/artifact/org.ekstazi/org.ekstazi.core>, . – Accessed: 2024-05-07
- [Ref] *Reflection*. [https://de.wikipedia.org/wiki/Reflexion_\(Programmierung\)](https://de.wikipedia.org/wiki/Reflexion_(Programmierung)), . – Accessed: 2024-07-02

- [RH96] ROTHERMEL, G. ; HARROLD, M.J.: Analyzing regression test selection techniques. In: *IEEE Transactions on Software Engineering* 22 (1996), Nr. 8, S. 529–551. <http://dx.doi.org/10.1109/32.536955>. – DOI 10.1109/32.536955
- [RH97] ROTHERMEL, Gregg ; HARROLD, Mary J.: A safe, efficient regression test selection technique. In: *ACM Trans. Softw. Eng. Methodol.* 6 (1997), apr, Nr. 2, 173–210. <http://dx.doi.org/10.1145/248233.248262>. – DOI 10.1145/248233.248262. – ISSN 1049–331X
- [RH98] ROTHERMEL, G. ; HARROLD, M.J.: Empirical studies of a safe regression test selection technique. In: *IEEE Transactions on Software Engineering* 24 (1998), Nr. 6, S. 401–419. <http://dx.doi.org/10.1109/32.689399>. – DOI 10.1109/32.689399
- [SHTZ⁺19] SHI, August ; HADZI-TANOVIC, Milica ; ZHANG, Lingming ; MARINOV, Darko ; LEGUNSEN, Owolabi: Reflection-aware static regression test selection. In: *Proc. ACM Program. Lang.* 3 (2019), oct, Nr. OOPSLA. <http://dx.doi.org/10.1145/3360613>. – DOI 10.1145/3360613
- [Sta] *STARTS*. <https://github.com/TestingResearchIllinois/starts>, . – Accessed: 2024-05-08
- [ZKK11] ZHANG, Lingming ; KIM, Miryung ; KHURSHID, Sarfraz: Localizing failure-inducing program edits based on spectrum information. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, S. 23–32

Abbildungsverzeichnis

1	Dependency Beispiel	12
2	UML-Diagramm der Datenstruktur	20
3	Vereinfachter Dependency Graph	22
4	Laufzeit Unit-Tests mit und ohne RTS	30
5	Sortierte Laufzeit der Unit-Tests	31
6	Laufzeit Integrationstests mit und ohne RTS	32

7	Sortierte Laufzeit der Integrationstests	32
8	Ausgewählte und fehlgeschlagene Tests in Versuch 2	35

Code Ausschnitte

1	Polymorphismus und DI	16
2	Dependency Beispiel Code Teil 1	43
3	Dependency Beispiel Code Teil 2	43
4	Dependency Beispiel Test Code Teil 1	44
5	Dependency Beispiel Test Code Teil 2	44

Anhang

Codebeispiel 2: Dependency Beispiel Code Teil 1

```
public class ClassA
{
    public void methodA() {}
}

public class ClassB
{
    private ClassA classA;

    public void methodB() {
        classA.methodA();
    }
}

public class ClassC
{
    private ClassA classA;

    public void methodC() {
        classA.methodA();
    }
}
```

Codebeispiel 3: Dependency Beispiel Code Teil 2

```
public class ClassD extends ClassB
{
    private int number;

    public void methodD()
    {
        number++;
    }
}

public class ClassE
{
    private ClassB classB;

    public void methodE() {
        classB.methodB();
    }
}

public class ClassF
{
    private ClassC classC;
    private ClassE classE;

    public void methodF() {
        classC.methodC();
        classE.methodE();
    }
}
```

Codebeispiel 4: Dependency Beispiel

Test Code Teil 1

```
public class TestClassA {
    @Test
    public void testMethodA() {
        ClassA classA = new ClassA();
        classA.methodA();
    }
}

public class TestClassB {
    @Test
    public void testMethodB() {
        ClassB classB = new ClassB();
        classB.methodB();
    }
}

public class TestClassC {
    @Test
    public void testMethodC() {
        ClassC classC = new ClassC();
        classC.methodC();
    }
}
```

Codebeispiel 5: Dependency Beispiel

Test Code Teil 2

```
public class TestClassD {
    @Test
    public void testMethodD() {
        ClassD classD = new ClassD();
        classD.methodD();
    }
}

public class TestClassE {
    @Test
    public void testMethodE() {
        ClassE classE = new ClassE();
        classE.methodE();
    }
}

public class TestClassF {
    @Test
    public void testMethodF() {
        ClassF classF = new ClassF();
        classF.methodF();
    }
}
```

Commit	T-Unit	T-Integration	T-RTS-Plugin	T-Unit-RTS	T-Integration-RTS	Changed Files
7f177	06:20	27:05	00:22	01:32	00:47	2
3163	05:16	22:15	00:28	01:17	15:31	5
fdc03	05:08	21:58	00:21	01:52	03:08	1
4e76c	06:27	27:07	00:25	03:19	19:12	7
2b96e	06:10	27:11	00:22	01:17	00:40	8
3cc64	05:45	25:27	00:23	06:15	20:20	10
8f210	05:47	26:42	00:25	01:53	17:22	4
b7600	06:17	27:10	00:29	01:33	16:50	2
8cb99	06:26	27:58	00:24	05:46	18:30	7
a25c2	05:49	23:38	00:25	01:40	00:54	1
2c589	05:18	22:47	00:26	01:11	03:16	1
8a80e	05:44	26:25	00:24	01:43	00:55	1
702bf	05:17	24:02	00:22	01:18	00:40	1
1a87f	05:03	23:00	00:25	02:47	19:03	2
6fa25	05:15	23:15	00:25	01:22	15:21	4
593b7	05:36	25:23	00:25	01:58	01:00	1
f201a	05:07	22:35	00:20	01:19	00:45	1
53ece	06:34	27:31	00:20	01:45	00:55	1
c5e73	05:02	21:05	00:22	01:15	00:42	3

Tabelle 1: Rohdaten der Zeitmessung des RTS-Plugin

	T-max	T-min	Standardabweichung	Durchschnitt	Einsparung
T-Unit	06:34	05:02	00:31	05:42	0%
T-Integration	27:58	21:05	02:11	24:52	0%
T-RTS-Plugin	00:29	00:20	00:02	00:23	0%
T-Unit-RTS	06:15	01:11	01:25	02:09	62.13%
T-Integration-RTS	20:20	00:40	08:15	08:12	67.02%
Changed Files				3.26	

Tabelle 2: Rohdaten der Zeitmessung des RTS-Plugin

Failed-AbteilungDaoImpl	6
Selected-AbteilungDaoImpl	5
Diff-AbteilungDaoImpl	4
Failed-SitzungsmappeMapper	4
Selected-SitzungsmappeMapper	198
Diff-SitzungsmappeMapper	0
Failed-SitzungsmappeMapper-VppPlanungController	5
Selected-SitzungsmappeMapper-VppPlanungController	202
Diff-SitzungsmappeMapper-VppPlanungController	0
Failed-KalenderMapper	27
Selected-KalenderMapper	353
Diff-KalenderMapper	0
Failed-VppPlanungController	0
Selected-VppPlanungController	7
Diff-VppPlanungController	0
Failed-VertragServiceImpl	6
Selected-VertragServiceImpl	261
Diff-VertragServiceImpl	0
Failed-BerichtClientImpl	1
Selected-BerichtClientImpl	211
Diff-BerichtClientImpl	0
Failed-ZeitplanServiceImpl	9
Selected-ZeitplanServiceImpl	8
Diff-ZeitplanServiceImpl	8

Tabelle 3: Anzahl ausgewählter Tests Versuch 2