Karlsruhe Institute of Technology

Automatische Problemdiagnose in Performance-Unit-Tests

Diplomarbeit
von

Christoph Heger

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Ralf H. Reussner
Zweitgutachter:	Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter:	Dr.-Ing. Jens Happe
Zweiter betreuender Mitarbeiter:	Dipl.-Inform. Dennis Westermann

Bearbeitungszeit: 01. September 2011 – 29. Februar 2012

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Karlsruhe, 28. Februar 2012

Danksagung

An erster Stelle bedanke ich mich bei meinem Betreuer Dr.-Ing. Jens Happe für alle wertvollen Diskussionen und die vorbildliche Betreuung über den gesamten Bearbeitungszeitraum. Des Weiteren bedanke ich mich bei meinem Zweitbetreuer Dipl.-Inform. Dennis Westermann für die stetige Unterstützung dieser Diplomarbeit.

Ich danke Prof. Dr. Ralf H. Reussner und Prof. Dr. Walter F. Tichy für die Bewertung dieser Diplomarbeit.

Weiter möchte ich mich bei Markus und Jan für das konstruktive Korrekturlesen und bei Alex für die großartige Zusammenarbeit bedanken.

Ganz besonders möchte ich mich bei meiner Familie, insbesondere meinen Eltern Roselinde und Wolfgang und meiner Freundin Isabel bedanken: Diese Diplomarbeit wäre ohne eure Unterstützung nicht möglich gewesen. Vielen Dank!

Zusammenfassung

Zu den Zielen der Software-Performance-Analyse zählt die Bewertung von Entwurfsentscheidungen im Hinblick auf die Performance, sowie die Erkennung von Performance-Problemen bereits während frühen Entwicklungsphasen. Eine proaktive Software-Performance-Analyse verhindert, dass Performance-Probleme erst spät im Entwicklungsprozess entdeckt werden. Die Aufgabe der Problemdiagnose eines Performance-Problems ist noch immer wenig automatisiert, obwohl sie erheblich Zeit und Aufwand von hochqualifizierten Analysten bedarf. Existierende Ansätze erlauben entweder nur die Erkennung von Performance-Problemen oder konzentrieren sich auf die Problemdiagnose im laufenden Betrieb eines Software-Systems. Ansätze die sowohl eine Regressionserkennung wie auch Unterstüzung bei der Problemdiagnose bieten erfordern dagegen spezielle Tests.

In dieser Diplomarbeit wird ein Ansatz vorgestellt, wie Performance-Probleme zwischen Software-Versionen erkannt und die Ursache identifiziert werden kann, indem bereits vorhandene Ressourcen wiederverwendet werden. Dazu wird die Problemdiagnose zu einem Großteil automatisiert. Wird bei einer Testmethode eines Unit-Tests ein Performance-Problem erkannt, wird durch die Analyse der Antwortzeit der getesteten Methoden das Problem auf eine oder mehrere Methoden eingeschränkt. Die Unit-Tests werden automatisch als Performance-Unit-Tests verwendet, ohne dass dazu ein Benutzereingriff notwendig ist. Die Korrelation der im Versionskontrollsystem gespeicherten Änderungen zur entsprechenden Software-Version mit den identifizierten Methoden gibt Entwicklern einen Hinweis, mit welcher Änderung das Performance-Problem eingeführt wurde. Entwickler werden dadurch ohne Mehraufwand bei der Erkennung von Performance-Problemen entlastet und bei der Problemdiagnose unterstützt. Eingebunden als Zusatzprogramm in die kontinuierliche Integration oder automatisierte Übersetzung wird Entwicklern ein Werkzeug für die proaktive Software-Performance-Analyse gegeben, das eine umgehende Reaktion auf Performance-Probleme bereits während frühen Entwicklungsphasen ermöglicht.

Anhand eines künstlichen Software-Systems konnte gezeigt werden, dass verschiedene Regressionsarten erkannt und die Ursache identifiziert werden kann. Weiterhin hat eine industrielle Fallstudie gezeigt, dass der Ansatz im praktischen Einsatz hilft, Performance-Probleme frühzeitig zu erkennen.

Abstract

An important objective of software performance analysis is to judge different design alternatives in terms of performance, as well as to detect performance problems during early stages of development. Proactive software performance analysis prevents performance problems from surfacing late in the development process. The task of root cause analysis is still poorly automated, though it demands plenty of time and effort of highly skilled analysts. Existing approaches either allow purely the detection of performance problems or focus on root cause analysis in running software systems. Other approaches provide the ability to detect performance problems as well as to perform root cause analysis, but demand special tests.

This thesis presents an approach to detect performance problems between versions of a software system and to identify the root cause by reusing existing resources. For this purpose, the root cause analysis is automated as far as possible. If a performance problem is detected for a test method of an unit test, the root cause analysis restricts the root cause to one or more methods by analyzing the response times of the tested methods. The unit tests are automatically used as performance unit tests without requiring any user intervention. The changes stored in the version control system for the corresponding software version are correlated with the identified methods. Thus, developers receive an advice, which change introduced the performance problem. Developers are supported in performance problem detection and root cause analysis without extra effort. Added as plugin to continuous integration or automated build, developers have a tool for proactive software performance analysis which enables quick response to performance problems during early stages of development.

With the help of an artificial software system we show that various regression types are detected and the root cause is identified. Furthermore, an industrial case study demonstrates that the approach helps to identify performance problems early in practice.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation . . . 2
1.2. Zielsetzung . . . 2
1.3. Anforderungen . . . 3
1.4. Struktur . . . 3

2. Grundlagen **5**
2.1. Performance Cockpit . . . 5
2.2. Instrumentierung . . . 7
2.2.1. Aspektorientierte Programmierung . . . 7
2.2.2. Kieker Rahmenwerk . . . 9
2.2.2.1. Zusammenfassung . . . 12
2.2.3. Strukturelle Reflektion Javassist . . . 12
2.2.4. Bytecode-Instruktionszähler ByCounter . . . 14
2.3. Versionskontrollsysteme . . . 17
2.3.1. Git . . . 17
2.4. Kontinuierliche Integration . . . 20
2.4.1. Automatisierte Unit-Tests . . . 22
2.4.2. Jenkins . . . 22
2.4.3. Apache Maven . . . 23
2.5. Statistik . . . 24
2.5.1. Grundlagen . . . 24
2.5.2. Varianzanalyse (ANOVA) . . . 26
2.5.3. Tuckey-HSD-Test . . . 30

3. Verwandte Arbeiten **31**
3.1. Regressionserkennung . . . 31
3.2. Problemdiagnose . . . 32
3.3. Abgrenzung . . . 34

4. Ansatz **35**
4.1. Allgemeiner Überblick . . . 35
4.2. Herausforderungen . . . 36
4.3. Regressionserkennung . . . 37
4.4. Problemdiagnose . . . 39
4.4.1. Aufbau des Aufrufbaumes . . . 40
4.4.2. Problemdiagnose bei gleichem Aufrufbaum . . . 41
4.4.3. Problemdiagnose bei ungleichem Aufrufbaum . . . 42
4.4.4. Korrelation mit Änderungen . . . 43
4.5. Adaptive Bytecode-Instrumentierung . . . 44
4.6. Automatisierung . . . 45
4.6.1. Lokale Erstellung . . . 45

4.6.2. Kontinuierliche Integration 47

5. Implementierung **49**
5.1. Architektur 49
5.2. Regressionserkennung und Problemdiagnose 50
5.3. Datenbereitstellung 50
5.4. Systemvermessung 54
5.5. Limitierungen 55

6. Validierung **57**
6.1. Testsystem 57
6.2. Experimententwurf 58
6.3. Experimentergebnis 60
6.4. Zusammenfassung 63

7. Evaluation **67**
7.1. Project Voldemort 67
7.2. Einschränkungen 67
7.3. Experimententwurf 68
7.4. Fallstudie 69
7.4.1. Regressionserkennung 69
7.4.2. Problemdiagnose 71
7.5. Zusammenfassung 71

8. Abschluss **75**
8.1. Nutzen 75
8.2. Weiterführende Arbeiten 76

Literaturverzeichnis **77**

Anhang **85**
A. Evaluation 85

1. Einleitung

Innovative Produkte und stetige Automatisierung sorgen dafür, dass komplexe Software-Systeme bereits heute in Unternehmen beinahe in jeden Geschäftsprozess involviert sind. Nicht nur bei strategischen Entscheidungen können Software-Systeme daher maßgeblich am Erfolg oder Misserfolg eines Unternehmens beteiligt sein. Kann ein solches Software-System seine nicht-funktionalen Anforderungen aufgrund unzureichender Leistungsfähigkeit nicht erfüllen, kann dies eine Vielzahl an negativen Konsequenzen für das Unternehmen nach sich ziehen. Beim elektronischen Handel mit Wertpapieren wurde es Anlegern von Online-Marklerfirmen im Oktober '97 nach einem Kurzsturz an den Aktienmärkten zum Verhängnis, als die Webseiten einiger Online-Marklerfirmen unter der hohen Last zusammenbrachen und deren Anleger hunderttausende Dollar an Kapitel verloren. Die Performance moderner Software-Systeme ist noch immer ein Problem in der Softwaretechnik. Unter Performance versteht man dabei den Grad, zu dem eine Software Anforderungen an Antwortzeiten und Durchsatz einhalten kann. Die Diagnose von Performance-Problemen in großen und verteilten Software-Systemen ist trotz der verfügbaren Werkzeuge immernoch eine komplexe, zeitaufwendige und vorallem kostspielige Aufgabe [SW01].

Der Forschungsbereich der sich diesen Problemen annimmt wird Software Performance Engineering (SPE) bezeichnet und gewann in den letzten Jahren an immer größerer Bedeutung. SPE ist ein systematischer, quantitativer Ansatz Software-Systeme zu konstruieren, die ihre nicht-funktionalen Anforderungen erfüllen. Proaktives Performance-Management identifiziert und löst Performance-Probleme frühzeitig. SPE basiert auf zwei verschiedenen Ansätzen. Zum einen gibt die Vorhersage basierend auf Performance-Modellen Aufschluss darüber ob die gewählte Architektur die Performance-Ziele erreichen kann, bevor ausführbare Software-Artefakte verfügbar sind, zum anderen liefert die Performance-Evaluation basierend auf Messungen präzisere Ergebnisse und kann auch auf Black-Box-Systeme angewendet werden.

Ideales proaktives Performance-Management in einem Entwicklungs-Team würde sich dadurch charakterisieren, dass es einen ausgewiesenen Performance-Ingenieur gibt, der für die Nachverfolgung und Kommunikation von Performance-Problemen verantwortlich ist. Die Entwickler sind mit SPE vertraut und in der Anwendung dieser Technologien trainiert. Es existiert ein Prozess für die Identifizierung von Performance-Gefahren und wie darauf reagiert wird. Das Projekt verfügt über einen angemessenen Plan zum Performance-Risiko-Management, der auf den Defizitkosten und den Kosten für SPE-Aktivitäten basiert [SW01]. Für die Entwickler bedeutet dies eine zusätzliche Belastung, da zur Vielzahl

an Anforderungen mit denen sie fertig werden müssen, nun eine weitere nicht unerhebliche hinzukommt. Mit der Bereitstellung von Software-Werkzeugen für die kontinuierliche Performance-Analyse, die SPE-Techniken und Aktivitäten kapseln, infolgedessen nicht das Wissen eines Performance-Analysten vorraussetzen und zusätzlich eine abstrahierte Repräsentation als Benutzeroberfläche bieten, erlauben es einem Entwickler die Performance eines Software-Artefaktes oder Software-Systems ohne großen Mehraufwand zu evaluieren.

1.1. Motivation

Desto größer die Zeitspanne zwischen der Einführung und der Erkennung einer Leistungsregression ist, umso schwieriger gestaltet sich die Suche nach der Problemursache und umso kostspieliger ist die Problemlösung. Damit Performance-Probleme frühzeitig erkannt werden bedarf es Werkzeuge die mindestens einmal täglich ausgeführt werden können und so präzise sind, dass selbst winzige Performance-Änderungen erkannt werden [BKT05]. Die Aufgabe der Problemdiagnose eines Performance-Problems ist noch immer wenig automatisiert, obwohl sie erheblich Zeit und Aufwand von hochqualifizierten Analysten bedarf [MM08]. Kontinuierliche Performance-Analysen stellen sicher, dass durch Änderungen hervorgerufene Performance-Probleme frühzeitig erkannt werden. Aktuelle Werkzeuge wie ContiPerf [Con] und JUnitPerf [JUn] konzentrieren sich dabei darauf eine obere Schranke der Antwortzeit für eine Testmethode zu überwachen. Das Konzept der Annotationen erlaubt die Verwendung von JUnit-Tests als Performance-Unit-Tests. Die manuelle Erstellung der Performance-Unit-Tests erfordert aber Zeit und verlangt die Definition einer sinnvollen oberen Schranke der Antwortzeit für eine Testmethode. Der Entwickler wird bei der Problemdiagnose bisher nicht unterstützt. Die Veränderung der Antwortzeit zwischen Software-Versionen bleibt in aktuellen Werkzeugen ebenso unberücksichtig. Die Werkzeuge sind deshalb erst dann hilfreich, wenn die obere Schranke überschritten wird. In diesem Fall ist jedoch unklar, ob das Performance-Problem in der zuletzt übergebenen Änderung oder das Ergebnis mehrer Änderungen ist, die auf mehrere Software-Versionen verteilt sind. Die Ursache des Performance-Problems muss manuell diagnostiziert und auf eine oder mehrere Änderungen zurückgeführt werden.

Die Problemdiagnose kann dabei zu einem Großteil automatisiert werden. Wird bei einer Testmethode ein Performance-Problem erkannt, kann durch die Analyse der Antwortzeit der getesteten Methoden das Problem auf eine oder mehrere Methoden eingeschränkt werden. Werden die im Versionskontrollsystem gespeicherten Änderungen zur entsprechenden Software-System mit den identifizierten Methoden korreliert, kann ein Hinweis auf die Problemursache gegeben werden. Des Weiteren können die Unit-Tests automatisch als Performance-Unit-Tests verwendet werden, ohne dass dazu ein Benutzereingriff notwendig ist. Der Mehraufwand für Entwickler wird reduziert und gleichzeitig der Nutzwert gesteigert. Entwickler werden dadurch bei der Erkennung von Leistungsregressionen entlastet und bei der Problemdiagnose unterstützt.

1.2. Zielsetzung

Das Ziel dieser Arbeit ist die Bereitstellung einer Methode für die automatische Erkennung und Problemdiagnose von Performance-Problemen nach der Übergabe von Änderungen an das Versionskontrollsystem basierend auf Performance-Unit-Tests. Dazu werden Unit-Tests, Versionskontrollsysteme, Projektverwaltungswerkzeuge und die kontinuierliche Integration, allesamt bereits Bestandteil moderner Software-Projekte, für die proaktive Software-Performance-Analyse orchestriert. Eingebunden als Zusatzprogramm in die kontinuierliche Integration oder automatisierte Übersetzung werden den Entwicklern die SPE-Techniken und das Domänenwissen in gekapselter Form und ohne Mehraufwand zugänglich gemacht.

In dieser Diplomarbeit werden folgende Ziele adressiert:

1. Entwicklung eines Werkzeuges als Machbarkeitsstudie, das autonom Messdaten einer Software-Version erhebt, mit den Messdaten anderer Software-Versionen vergleicht, auf Performance-Probleme analysiert, die Probleme diagnostiziert und die Performance-Probleme mit den Änderungen im Versionskontrollsystem verknüpft. In diesem Kontext wird die Eignung des Kieker-Rahmenwerkes zur Messdatenerhebung evaluiert, indem das Kieker-Rahmenwerk zur Instrumentierung und Erhebung der Messdaten eingesetzt wird. Alternativ wird die Eignung von ByCounter zur Zählung der tatsächlich ausgeführten Bytecode-Instruktionen und Schätzung der Antwortzeit evaluiert.
2. Durchführung einer Fallstudie mit einem realen Software-Projekt zur Evaluation der Regressionserkennung und Problemdiagnose.
3. Identifizierung der Probleme die adressiert werden müssen, um beliebige Sofware-Projekte zu überwachen, deren Entwicklungsprozess auf Git, Apache Maven und Jenkins aufbaut.

Die Anforderungen an die Instrumentierung sind: i) Unabhängig und autonom vom Entwickler, ii) vernachlässigbare Verfälschung der Messdaten und iii) beliebige Methoden eines Software-Systems müssen adaptiv und reversibel instrumentiert werden. Die Anweisungen der Instrumentierung die in den Rumpf einer Methode eingefügt werden dürfen die eigentliche Antwortzeit der Methode nur um einen vernachlässigbaren Wert erhöhen.

1.3. Anforderungen

Sowohl an die Methodik als auch den Prototyp werden folgende Anforderungen gestellt:

- Einfache Nutzung (kein Mehraufwand für Entwickler)
- Kompatibilität mit bei SAP eingesetzten Technologien (Java, Apache Maven, Git und JUnit)
- Geringe Fehlerkennungsrate
- Reproduzierbarkeit der Ergebnisse
- Unterstützung bei der Ursachenfindung

Diese Anforderungen müssen bei der Entwicklung der Methodik zur automatischen Regressionserkennung und Problemdiagnose berücksichtigt werden. Außerdem werden dadurch Rahmenbedingungen für den zu implementierenden Prototyp geschaffen.

1.4. Struktur

Beginnend werden in Kapitel 2 die für das Verständnis der Diplomarbeit wichtigen Grundlagen beschrieben. In Kapitel 3 wird eine Übersicht über die verwandten Arbeiten gegeben. Der Ansatz für die automatische Problemdiagnose in Performance-Unit-Tests wird anschließend im Detail in Kapitel 4 vorgestellt. Der Prototyp wird in Kapitel 5 vorgestellt und beschrieben. Die Validierung des Prototyps mit einem künstlichen Software-System ist Gegenstand von Kapitel 6, gefolgt von der Fallstudie mit welcher die Anwendbarkeit der Methode evaluiert wird. Den Abschluss bildet Kapitel 8 indem die Diplomarbeit zusammengefasst wird und offene Punkte sowie weiterführende Arbeiten beschrieben werden.

2. Grundlagen

In diesem Kapitel werden die Grundlagen für das Verständnis dieser Diplomarbeit vorgestellt. Zunächst wird in Abschnitt 2.1 das Performance Cockpit Rahmenwerk für die systematische Durchführung von Experimenten vorgestellt, aus denen sich im ersten Schritt Leistungsmodelle ableiten und im zweiten Schritt mit diesen Leistungsmodellen Qualitätsmerkmale vorhersagen lassen. Im darauffolgenden Abschnit 2.2 wird die Instrumentierung für die Erhebung der Messdaten beschrieben. In diesem Kontext wird das Kieker Rahmenwerk für die kontinuierliche Beobachtung und Erhebung von Messdaten durch Instrumentierung der Software-Systeme in Abschnitt 2.2.2 vorgestellt. Ergänzend wird in Abschnitt 2.2.4 der Instruktionszähler ByCounter zur Zählung der ausgeführten Bytecode-Instruktionen erklärt. Anschließend wird die Versionskontrolle im Allgemeinen und Git als Versionskontrollsystem im Speziellen in Abschnitt 2.3 vorgestellt. In Abschnitt 2.4 wird die kontinuierliche Integration erläutert. Am Ende werden in Abschnitt 2.5 die statistischen Grundlagen arithmetisches Mittel, Varianz, Standardabweichung, Konfidenzintervall, α-Quantil, Varianzanalyse und der Tuckey-Test eingeführt.

2.1. Performance Cockpit

Mit zunehmender Komplexität der Software-Systeme wird das Verständnis der Leistungseigenschaften eines Software-Systems eine immer größere Herausforderung. Durch die Unterstützung der Leistungsanalysten bei der Durchführung verschiedener Aufgaben wird versucht mit dieser Entwicklung schrittzuhalten. Westermann et al. stellt in [WHHH10] den Ansatz des Performance Cockpits, als ein erweiterbares Rahmenwerk für die auf Messungen basierende Leistungsevaluation durch die systematische Durchführung von Experimenten vor. Das Rahmenwerk erlaubt die Ableitung von Leistungsmodellen durch automatisierte Messungen und die Ausführung von Regressions-Benchmarks. Dabei wird das Wissen über die Leistungsevaluation in einer einzigen Anwendung gekapselt, deren Kern bereits die grundlegende Funktionalität zur Kontrolle der Benchmark-Systeme bereitstellt. Die verschiedenen Domänenexperten müssen dadurch nur noch die Teile des erweiterbaren Systems ergänzen, die zu ihrer Domäne gehören. Die Ziele des Rahmenwerks sind ein geringerer Aufwand beim Aufsetzen von Leistungstests zu erreichen, einen besseren Wissenstransfer zu gewährleisten, eine flexible Messumgebung bereitzustellen und durch eine bessere Benutzerfreundlichkeit die Leistungsanalysen einem breiteren Publikum zugänglich zu machen. Der Ansatz besteht darin, die verschiedenen Aufgaben auf verschiedene Rollen aufzuteilen. Der Leistungsanalyst spezifiziert die Ziele der Studie, definiert die Systemgrenzen und listet die zu evaluierenden Dienste auf. Die notwendigen Informationen

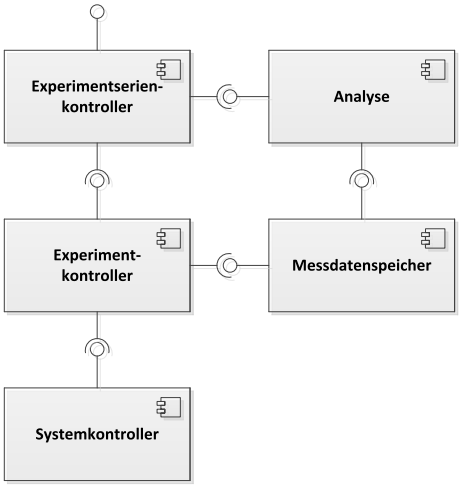


Abbildung 2.1.: Architektur des Performance-Cockpits nach [WHHH10]

über das Testsystem werden über ein Meta-Modell des Rahmenwerks zur Konfiguration modelliert. Zusätzlich wählt der Leistungsanalyst die Adapter aus, welche die verschiedenen Komponenten des Software-Systems mit dem Performance Cockpit verbinden. Die Adapter sind Zusatzprogramme für das Rahmenwerk, die von System-, Benchmark- und Werkzeugexperten entwickelt werden. Dadurch muss sich der Leistungsanalyst nicht mit technischen Details beschäftigen. Darüberhinausgehend kann der Leistungsanalyst von vorherigen ähnlichen Konfigurationen vergleichbarer Aufbauten profitieren. In den weiteren Schritten wählt der Leistungsanalyst die Metriken und relevanten Parameter aus, danach gibt er an, welche dieser Parameter während des Experimentes variiert werden sollen und wählt die gewünschte Arbeitslast aus. Nach dem Experimentlauf kann er die Daten auswerten und interpretieren. Die auf den Messdaten ausgeführten statistischen Analysen verwenden unterschiedliche Verfahren, die durch Analyseexperten implementiert und als Analyseadapter in das Rahmenwerk integriert wurden [WHHH10].

Die Architektur des Rahmenwerks ist in Abbildung 2.1 in Form einer Komponentenübersicht abgebildet. Der Experimentkontroller ist verantwortlich für die Kommunikation mit den Systemkontrollern auf den Testsystemen. Er führt einzelne Experimentläufe aus und sammelt die Messdaten nach einem Experimentlauf ein. Der Systemcontroller ist für die Leitung bestimmter Systeme im allgemeinen Testaufbau verantwortlich. Für jeden Knoten im Testaufbau existiert mindestens eine Instanz des Systemkontrollers. Er wickelt die Kommunikation mit dem Experimentkontroller ab und besteht aus mehreren systemspezifischen Zusatzprogrammen einschließlich der Adapter zu den verschiedenen Komponenten des zu leitenden Systems. Die Komponente zur Aufbewahrung der Messdaten ist der zentrale Speicherort des Performance Cockpits und verantwortlich für die Persistenz der Messdaten. Die Messdatenspeicherungskomponente stellt Modelle und Schnittstellen bereit, mit denen Messdaten und Analyseergebnisse gespeichert und abgefragt werden können. Die Analysekomponente ist verantwortlich für die Auswertung der Messdaten und dafür, die durch den Leistungsanalysten definierten Ergebnisse zu liefern. Sie bietet verschiedene statistische Analysemethoden zum Auswerten der Messdaten. Der Experimentserienkontroller lädt die vom Leistungsanalysten modellierte Konfiguration und startet eine Serie von Experimenten basierend auf dieser Konfiguration. Die Konfiguration beschreibt welche Parameter, innerhalb welcher Bereiche und welcher Schrittgrößen variiert werden sollen [WHHH10].

Die Analysen des Performance Cockpits werden verwendet, um die Messreihen verschiedener Software-Versionen auf signifikante Unterschiede zu untersuchen. Außerdem wird die

Persistenz dazu verwendet die Messreihen der Testmethoden der Unit-Tests zu speichern. Dadurch wird eine Referenz aufgebaut, mit der eine neue Software-Version verglichen und auf ein Performance-Problem untersucht werden kann.

2.2. Instrumentierung

Die Instrumentierung eines Software-Systems bezeichnet den Vorgang bei dem Anweisungen zum Messen bestimmter Merkmale an ausgewählten Punkten im Software-System integriert werden. Ein Merkmal kann dabei die Ausführungszeit einer Methode sein, aber auch wie oft eine Methode aufgerufen oder wie viele Bytecode-Instruktionen ausgeführt wurden. Der Einfluss der Instrumentierung auf die Antwortzeit einer Methode muss vernachlässigbar sein, damit präzise Messwerte resultieren und selbst winzige Änderungen der Performance erkannt werden. Zusätzlich darf der Entwickler mit der Instrumentierung keinerlei Aufwand haben. Dies bedeutet, dass ein Software-System nach Bedarf automatisch und autonom instrumentiert und vermessen wird. In Abschnitt 2.2.1 wird das Paradigma der aspektorientierten Programmierung eingeführt, das die Grundlage der aspektorientierten Instrumentierung des Kieker Rahmenwerkes in Abschnitt 2.2.2 bildet. In Abschnitt 2.2.2 wird ebenfalls aufgezeigt, dass die angebotene Instrumentierung des Kieker Rahmenwerkes für die in dieser Diplomarbeit vorgestellte Methode ungenügend ist. Deshalb wird in Abschnitt 2.2.3 die strukturelle Reflexion mit Javassist beschrieben, mit der die Anweisungen der Instrumentierung direkt in den Rumpf einer Methode eingesetzt werden können. Als Alternative zur Messung der Antwortzeit wird in Abschnitt 2.2.4 ByCounter vorgestellt. Mit Bytecounter können die tatsächlich ausgeführten Bytecode-Instruktionen gezählt werden, woraus widerrum die Antwortzeit geschätzt werden kann.

2.2.1. Aspektorientierte Programmierung

Ein wichtiger Punkt ist die Art und Weise wie die Anweisungen der Instrumentierung in das eigentlich zu instrumentierende Software-System gelangen. Die Anweisungen der Instrumentierung generieren Messdaten an bestimmten Stellen im Software-System. Die Wartbarkeit des Software-Systems wird reduziert, wenn diese Anweisungen direkt in das Software-System integriert werden, da dies die Lesbarkeit reduziert und die Komplexität erhöht. Nach [vHRH+09] wird das Paradigma der aspektorientierten Programmierung (AOP) [KLM+97] als ein adäquates Mittel zur Integration dieser Anweisungen in eine Anwendung beschrieben. Eine beliebte Java-basierte Implementierung von AOP ist AspectJ [vHRH+09]. Die Anweisungen der Instrumentierung werden nicht als Bestandteil des Software-Systems betrachtet. Sie dienen lediglich der Generierung von Messdaten für die Analyse, um die Leistungsfähigkeit des Software-Systems über verschiedene Software-Versionen hinweg zu verfolgen und zu vergleichen. Deshalb werden die Anweisungen der Instrumentierung nur bei Bedarf in das Software-System integriert.

Die AOP wird von Kiczales et al. in [KHH+01] als Technik für eine bessere Trennung der Belange in Software-Systemen vorgeschlagen. Dabei baut die AOP auf vorherigen Technologien wie prozedurale und objektorientierte Programmierung auf, die bereits signifikante Verbesserungen in der Modularität von Software herbeiführten. Kiczales et al. beschreiben die zentrale Idee von AOP so, dass die AOP für querschneidende Belange (engl. crosscutting concerns) dem entspricht, was die objektorientierte Programmierung für die Kapselung und Vererbung von Objekten bedeutet. Unter querschneidenden Belangen versteht man dabei Belange, die nicht klar vom restlichen System getrennt werden können. Die AOP erlaubt die Definition von Schnittpunkten im Kontrollfluss von Programmen, an denen die in AOP definierten Aspekte zur Ausführung kommen. In AspectJ werden diese grundlegenden Punkte in der Ausführung eines Programms als Vereinigungspunkte (engl. join

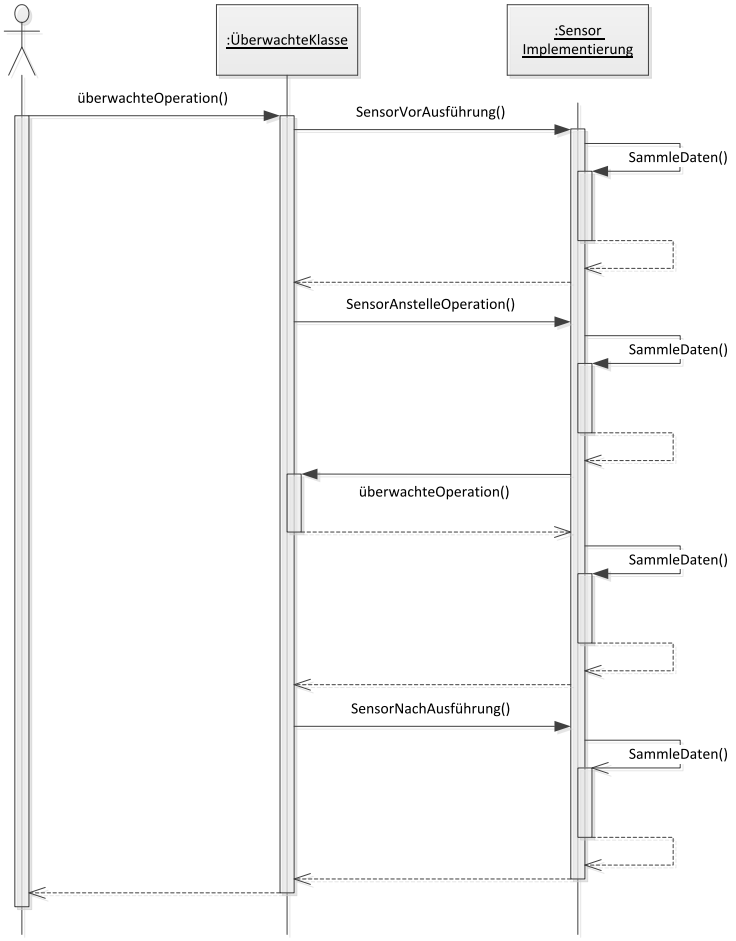


Abbildung 2.2.: Sequenzdiagramm mit den drei Ausführungszeitpunkten basierend auf [vHRH+09]

points) bezeichnet. Das Modell der Vereinigungspunkte bildet den Rahmen der Referenzierung, die die koordinierte Ausführung des Aspekts und des herkömmlichen Codes in einem Programm ermöglicht. Von den im dynamischen Kontrollfluss unterstützen Vereinigungspunkten von AspectJ ist der Methodenaufruf für unseren Ansatz der ausschlaggebende. Verschiedene Vereinigungspunkte lassen sich zu sogenannten Schnittpunkten (engl. pointcuts) zusammenfassen. Ein Schnittpunkt kann beispielsweise aus einem Vereinigungspunkt für einen Methodenaufruf und einem Vereinigungspunkt für das Setzen eines bestimmten Feldes bestehen. Die Schnittpunktbezeichner (engl. pointcut designators) sind dann für den Abgleich bestimmer Vereinigungspunkte während der Ausführung mit dem Kontrollfluss zuständig. Der Ausführungshinweis (engl. advice) ist ein Methoden ähnlicher Mechanismus um den Ausführungszeitpunkt von bestimmtem Code der Vereinigungspunkte eines Schnittpunktes anzugeben. In unserem Kontext bedeutet dies, die Ausführung vor der eigentlichen Methode, nach der eigentlichen Methode oder anstelle der Methode wie Abbildung 2.2 in Form eines Sequenzdiagramms darstellt. Ein Aspekt ist schließlich eine Implementierung ähnlich der einer Klasse in Java. Er kann die Deklarationen von Schnittpunkten, Ausführungshinweisen aber auch alle Arten von Deklarationen, die in Klassen erlaubt sind, beinhalten [KHH+01]. Das Einweben der Aspekte in ein Software-System kann auf drei verschiedene Arten geschehen, deren Resultat identisch ist. Erstens ist dies direkt beim Kompilierungsvorgang möglich, zweitens nach der Kompilierung oder drittens beim Laden der Klassen durch die Java Virtual Machine (JVM). Unser Ansatz wird sich dem Einweben beim Laden der Klassen bedienen, da dies uns erlaubt Java-Programme

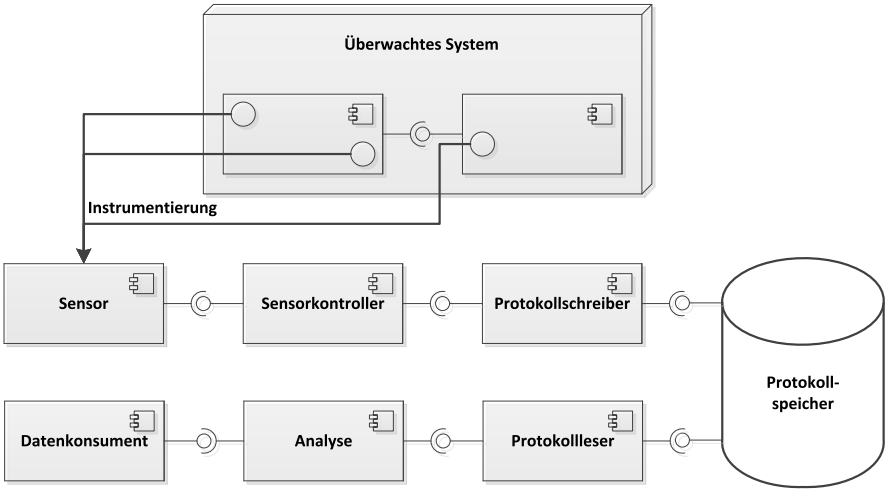


Abbildung 2.3.: Kieker Architektur basierend auf [vHRH+09] und [EvHWH11]

beliebig zu instrumentieren.

2.2.2. Kieker Rahmenwerk

Van Hoorn et al. stellen in [vHRH+09] das komponentenbasierte Rahmenwerk Kieker zur andauernden Beobachtung, Aufzeichnung und Analyse des Laufzeitverhaltens von Software-Systemen vor. Die zugrundeliegende Idee ist durch die Extraktion von Informationen aus einem laufenden System, die Basis für selbst-adaptive Software-Systeme zu liefern. Dies beinhaltet die Ableitung von Entwurfsmodellen aus der Programmausführung und die Erkennung von Laufzeitfehlern zu ermöglichen. Die Komponenten zur Instrumentierung der Software, Protokollierung und Analyse sind leicht erweiterbar und können einfach ersetzt werden, um die spezifischen Anforderungen in einem Anwendungskontext zu erfüllen. Die Abgrenzung der kontinuierlichen Beobachtung eines Software-Systems und die damit einhergehende Aufzeichnung von Messdaten gegenüber der Profilierung besteht darin, dass im ersten Fall der zusätzliche Aufwand durch Instrumentierung, Protokollierung etc., so gering wie möglich gehalten werden muss. Die wesentlichen Eigenschaften, die Kieker auszeichnen ist die allgemeine, erweiterbare Datenstruktur der Messdaten z.B. der Antwortzeit. Diese Datenstruktur wird sowohl von der Protokollierung, als auch der Analyse geteilt. Die erweiterbaren Modelle zum Lesen und Schreiben der Messdaten ermöglichen es, die Messdaten ins Dateisystem, zu Nachrichtensystemen oder relationalen Datenbanken zu schreiben, aber auch davon zu lesen [vHRH+09].

Aspektorientierte Instrumentierung

AspectJ erlaubt das Einweben von Instruktionen der Instrumentierung in den Java Bytecode von Software-Systemen und Bibliotheken. Das Kieker Rahmenwerk beinhaltet zwei, auf AspectJ basierende Instrumentierungsvarianten: *OperationExecutionAspectAnnotation* und *OperationExecutionAspectFull*. Beide Instrumentierungsvarianten können sowohl bei der Übersetzung als auch während des Ladevorgangs einer Klasse in ein Software-System eingewebt werden. Nach der Instrumentierung wird die Ausführung von Methoden und der entsprechenden Ablaufverfolgung als auch Zeitwerte aufgezeichnet. Wird die Variante mit dem Namen *OperationExecutionAspectAnnotation* verwendet, müssen die zu vermessenden Methoden des Software-Systems mit der von Kieker bereitgestellsten Annotation *@OperationExecutionMonitoringProbe* annotiert werden. Das Einwebeprogramm (engl. AspectJ weaver) ist ein sogenannter Java-Agent, der bei der Java Virtual Machine (JVM) registriert wird und die Instruktionen der Instrumentierung in die Java-Klassen einwebt. Mit

Hilfe der Konfigurationsdatei *aop.xml* kann das Einwebeprogramm konfiguriert werden. Die Methoden des Software-Systems können dann mit der Annotation *@OperationExecutionMonitoringProbe* instrumentiert werden [EvHJ11].

Die Annotation kann dazu verwendet werden um alle Methoden eines Software-Systems zu instrumentieren. Davon ausgenommen sind Konstruktoren [EvHJ11]. In der Konfigurationsdatei *aop.xml* werden die Klassen angegeben, die für die Instrumentierung durch das AspectJ-Einwebeprogramm beachtet werden müssen. Das AspectJ-Einwebeprogramm wird dadurch angewiesen alle Klassen in Paketen zu berücksichtigen, die beispielsweise mit dem Namen *org.sopeco.* oder *test.org.sopeco.* beginnen. Dahingegen kann das Einwebeprogramm aber auch ausdrücklich angewiesen werden, keine Instrumentierung in Klassen einzuweben, die sich in Paketen befinden, die mit dem Namen *org.sopeco.runner.* beginnen. Der Aspekt der eingewebt werden soll wird ebenfalls angegeben. Es kann sich dabei beispielhaft um den Aspekt *OperationExecutionAspectAnnotation* handeln, bei dem die zu vermessenden Methoden mit der Annotation *@OperationExecutionMonitoringProbe* versehen werden. Das Kieker Rahmenwerk bietet aber auch die Möglichkeit Software-Systeme basierend auf AspectJ zu instrumentieren, ohne dabei Annotationen zu verwenden. Hierzu stellt das Kieker Rahmenwerk den eingangs als zweitens erwähnten Aspekt respektive Instrumentierungsvariante *OperationExecutionAspectFull* zur Verfügung. Dieser Aspekt stellt sicher, dass alle Methoden instrumentiert und aufgezeichnet werden, die sich in den zu berücksichtigenden Klassen oder Paketen befinden. Hierfür müssen keine Methoden im Software-System annotiert werden, es genügt den Aspekt in der Konfigurationsdatei *aop.xml* als einzuwebenden Aspekt anzugeben. Aufrufe zu instrumentierten Methoden werden unterbrochen und die Ausführung fährt mit der Methode der Instrumentierung fort, welche die Logik zur Messung der Antwortzeit beinhaltet. Bevor die Ausführung an die eigentlich aufgerufene Methode gegeben wird, wird ein neuer Aufzeichnungeintrag erstellt und der Zeitstempel gelesen. Nach dem die Ausführung zur Instrumentierung zurückkehrt wird die Antwortzeit berechnet und im Aufzeichnungseintrag gespeichert, der dann an die Instanz des Überwachungskontrollers des Kieker Rahmenwerkes übergeben wird [vHRH+09].

Mehraufwand

Damit präzise Messwerte resultieren wird an die Instrumentierung die Anforderung gestellt, dass ihr Einfluss auf die Messwerte vernachlässigbar sein muss. Die Instrumentierung des Kieker Rahmenwerkes verursacht einen Mehraufwand, der Einfluss auf die Messwerte nimmt. Das Einweben der Instruktionen der Instrumentierung in den Bytecode einer Klasse beim Laden in die JVM ebenso wie die Ausführung dieser Instruktionen verursachen einen Mehraufwand, der auf die eigentliche Ausführungszeit einer Methode addiert wird. Abbildung 2.4 zeigt ein Sequenzdiagramm mit dem vereinfachten internen Kontrollfluss des Kieker Rahmenwerkes für die Überwachung einer Methode. Das Sequenzdiagramm ist am unteren Rand mit verschiedenen Ausführungszeiten annotiert: Die Ausführungszeit der Instruktionen der überwachten Methode (Δ_A), die Zeit zum Anstoßen der Ausführung der Instruktionen der Instrumentierung vor und nach Ausführung der Instruktionen der überwachten Methode (Δ_B), die Zeit für die Erhebung der Messdaten (Δ_C) und die Zeit zum Speichern der Messaten (Δ_D). Die drei Ursachen des Mehraufwandes der auf die Ausführunszeit der überwachten Methode addiert wird ist:

1. Anstoßen der Ausführung der Instruktionen der Instrumentierung (Δ_B),
2. Erhebung der Messdaten (Δ_C) und
3. Speichern der Messdaten (Δ_D) [vHRH+09].

Der Mehraufwand des Kieker Rahmenwerkes beträgt $< 2,5\mu s$ auf einem X6270 Blade Server [vHRH+09]. Unberücksichtigt ist die Aufwärmphase in der die Messungen von einer

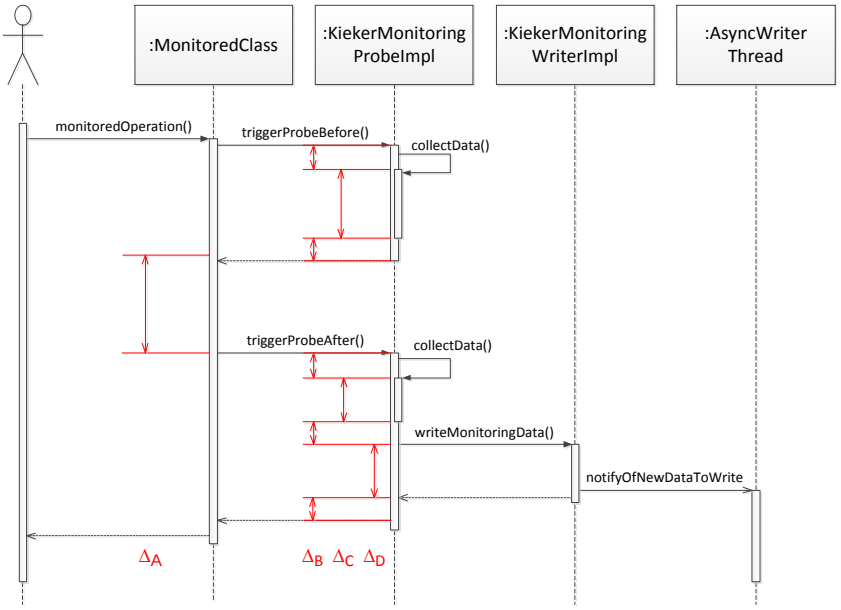


Abbildung 2.4.: Sequenzdiagramm für die Ausführung einer instrumentierten Methode mit Kieker nach [vHRH+09]

signifikanten Streuung betroffen sind, verursacht durch das Laden der Klassen und Einweben der Instruktionen der Instrumentierung. Von 100 000 000 Aufrufen einer Operation, werden deshalb nur die letzten 100 000 Aufrufe aufgezeichnet und analysiert [vHRH+09]. Eine signifikante Streuung macht die Analyse der Messungen und den Vergleich verschiedener Messreihen schwierig. Übertragen auf die Methode dieser Diplomarbeit müsste eine Testmethode eines Unit-Tests 100 000 000 mal ausgeführt werden. Dadurch ergibt sich ein unvertretbarer Aufwand, wird in Betracht gezogen, dass dies für mehrere Unit-Tests und Software-Versionen erfolgen muss. Deshalb muss eine Methode entwickelt werden, die es erlaubt, dass Kieker Rahmenwerk weiterhin zu verwenden, aber bei einer geringen Anzahl an Ausführungen einer Testmethode eine geringe Streuung erreicht.

Adaptive Instrumentierung

Andauernde Beobachtung kann die frühtzeitige Erkennung von Qualitätsproblemen wie den Rückgang der Leistung bedeuten. Ehlers et al. präsentieren in [EvHWH11] einen Ansatz, bei dem Kieker durch ein Zusatzprogramm um eine selbst-adaptive Instrumentierung erweitert wird. Die grundlegenden Schritte sind dabei: Injezierung der Sensoren, Aktivierung der Sensoren, Sammlung der Messdaten, Verarbeitung der Messdaten und Adaption der Instrumentierung basierend auf der Analyse der Messdaten. Die Sensoren sind dabei überall im System verteilt, so dass alle Ausführungen von Operationen beobachtet werden können. Da es nach [EvHWH11] nicht sinnvoll ist, alle Sensoren zur gleichen Zeit zu aktivieren, werden als aktive Sensoren zunächst meist diejenigen an den öffentlichen Schnittstellen gewählt. Zeigen die Messdaten eines Sensors auf dieser Ebene dann eine Anomalie, kann durch die adaptive Instrumentierung die Sensorik der darunterliegen Schicht aktiviert werden. Im Kern besteht die in [EvHWH11] vorgestellte adaptive Instrumentierung von Kieker aus einem regelbasierten System. Wird eine Messstelle im Software-System erreicht, prüft der Sensor, ob dessen Vorgänger aktiviert ist und welche Anomaliebewertung dieser derzeit besitzt. Überschreitet die Anomaliebwertung einen gewissen Schwellwert, oder befindet sich die aktuelle Messstelle auf höchster Ebene, auf der es keinen Vorgänger gibt, wird der Sensor aktiviert und liefert Messdaten. Die Anomaliebwertung basiert dabei

auf statistischen Verfahren zur Zeitreihenvorhersage, einem Hypothesentest und einer Berechnungsfunktion. Der Hypothesentest testet dabei, ob der vorhergesagte Wert als Mittel der Grundgesamtheit akzeptiert werden kann oder nicht. Die Berechnungsfunktion liefert einen Wert der Auskunft darüber gibt, ob der Messwert der Erwartung entspricht oder nicht [EvHWH11].

2.2.2.1. Zusammenfassung

Alexander Wert hat in [Wer12] den Einfluss der vollständigen Instrumentierung des Kieker Rahmenwerkes auf die Messwerte untersucht. Er hat gezeigt, dass die vollständige Instrumentierung nicht für die Erhebung der Messdaten verwendet werden kann. Die Instrumentierung von Methoden durch das manuelle Hinzufügen von Annotationen, erfüllt die Anforderung der Autonomie der Instrumentierung nicht. Die adaptive Instrumentierung ist zum Zeitpunkt dieser Diplomarbeit nicht in der veröffentlichten Version von Kieker enthalten. Um das Ziel dieser Diplomarbeit zu erreichen ist es deshalb notwendig, eine adaptive Instrumentierung zu entwickeln.

2.2.3. Strukturelle Reflektion Javassist

Als Reflektion (engl. reflection) wird die Möglichkeit bezeichnet, Datenstrukturen zu prüfen und das Verhalten eines Software-Systems zu ändern. In Java ist diese Möglichkeit sehr eingeschränkt und erlaubt einem Software-System nur eine Klasse zu erzeugen, den Wert eines Feldes zu setzen oder abzurufen und eine Methode mit Hilfe der Programmierschnittstelle zur Reflektion (engl. Java reflection API) aufzurufen. Die meisten Erweiterungen, die sich den Einschränkungen dieser Programmierschnittstelle annehmen, ermöglichen die Reflektion des Verhaltens. Unter der Verhaltensreflexion versteht man die Möglichkeit eine Operation wie einen Methodenaufruf zu unterbrechen und das Verhalten dieser Operation zu verändern. Durch die Verhaltensreflektion kann das Verhalten einer Operation verändert werden, es können jedoch nicht die in einem Software-System verwendeten Datenstrukturen geändert werden, die während der Kompilierung statisch festgelegt werden. Die strukturelle Reflektion dagegen erlaubt einem Software-System beispielsweise die Definition einer Klasse, die Definition einer Methode und die Veränderung eines Datums nach Bedarf [Chi00].

Chiba präsentiert in [Chi00] Javassist, eine Klassenbibliothek für die strukturelle Reflektion in Java. Javassist kann ohne Modifikationen eines bestehenden Laufzeitsystems oder Übersetzers verwendet werden. Chiba beschreibt die grundlegende Idee hinter Javassist so, dass die strukturelle Reflektion als Bytecode-Transformation beim Übersetzen oder beim Laden ausgeführt wird. Javassist erlaubt keine strukturelle Reflektion, nachdem ein übersetztes Software-System in die JVM geladen wurde. Die strukturelle Reflektion ist deshalb nur beim Laden möglich. Eine weitere Eigenschaft von Javassist ist die Abstraktion von der Bytecode-Ebene. Dadurch ist kein tiefgehendes Verständnis des Java Bytecodes erforderlich. In Java wird der Bytecode durch die Übersetzung eines Software-Systems gewonnen und in Klassendateien abgespeichert. Dabei entspricht jede dieser Klassendateien einer bestimmten Klasse. Die strukturelle Reflektion in Javassist wird durch Übersetzen der Änderung der strukturellen Reflektion in eine gleichwertige Bytecode-Transformation der Klassendateien durchgeführt. Im Anschluss an die Transformation werden die modifizierten Klassendateien in die JVM gelanden, wodurch dann keine weiteren Änderungen mehr erlaubt sind. Dafür kann Javassist mit einer standardmäßigen JVM verwendet werden. Javassist wird mit einem eigenen Klassenladeprogramm (engl. class loader) verwendet. Java erlaubt Software-Systemen ihre eigenen Klassenladeprogramme zu definieren, die eine Klassendatei aus dem Dateisystem oder dem Netzwerk laden. Diese Klassenladeprogramme fodern die JVM zum Laden einer Klasse auf, die aus dem Bytecode konstruiert wird.

Der Rückgabewert ist ein Klassenobjekt (java.lang.class), das die geladene Klasse repräsentiert. Nachdem eine Klasse manuell mit einer Instanz eines eigenen Klassenladeprogramms geladen wurde, werden alle von dieser geladenen Klasse referenzierten Klassen ebenfalls durch dieses Klassenladeprogramm geladen. Die JVM lädt eine referenzierte Klasse bei Bedarf automatisch mit diesem Klassenladeprogramm [Chi00].

In Javassist repräsentiert ein spezielles Übersetzungsklassenobjekt (`CtClass`) den Bytecode einer in die JVM geladenen Klasse. Dieses Übersetzungsklassenobjekt enthält ausreichende symbolische Information um eine Klasse zu vergegenständlichen. Ausgehend von diesem Übersetzungsklassenobjekt, kann die Klassendefinition geprüft und geändert werden. Änderungen der Klassendefinition spiegeln sich im Bytecode wider, den das Übersetzungsklassenobjekt repräsentiert. Javassist erlaubt es einem eigenen Klasseladeprogramm auch eine neue Klasse von Grund auf zu definieren ohne eine Klassendatei lesen zu müssen. Dadurch kann ein Software-System nach Bedarf dynamisch neue Klassen definieren. Zu einer solchen leeren Klasse können dann Methoden und Felder hinzugefügt werden. Der Teil von Javassist, der dafür gedacht ist eine Klasse zu prüfen, die durch ein Übersetzungsklassenobjekt dargestellt wird, ist kompatibel zur Reflektion von Java mit der Ausnahme, dass Javassist keine Methoden anbietet, um eine Instanz einer Methode zu erzeugen oder aufzurufen. Der Unterschied zwischen Javassist und der Reflektion von Java sind die Methoden von Javassist, um Klassendefinitionen zu ändern. Die Methoden lassen sich in drei Kategorien unterteilen: Erstens in Methoden zum Verändern von Modifikatoren, zweitens in Methoden für das Ändern der Klassenhierarchie und drittens in Methoden zum Hinzufügen neuer Elemente. Die drei Entwurfsziele beschreibt Chiba wie folgt: Das erste Ziel ist eine Abstraktion der Bytecode-Ebene bereitzustellen, sodass kein Wissen über den Bytecode von Java erfordlich ist. Das zweite Ziel ist die strukturelle Reflektion so effizient wie möglich auszuführen und als drittes Ziel Software-Systemen zu helfen, die strukturelle Reflektion auf sichere Art und Weise auszuführen, was die Typsicherheit angeht. Das dritte Entwurfsziel soll Software-Systeme daran hindern, fälschlicherweise eine Klasse zu produzieren, die unrichtige Typen beinhaltet. Um dieses Ziel zu erreichen, erlaubt Javassist nur einen begrenzten Umfang an Änderungen einer Klassendefinition. Im Allgemeinen soll dadurch verhindert werden, dass Software-Systeme sich nicht selbst durch Relfektion zum Absturz bringen. Deshalb stellt Javassist keine Methoden bereit, um eine Methode oder ein Feld einer Klasse zu entfernen, da dies eine Unrichtikeit auf Typebene herbeiführt. Javassist stellt hingegen Methoden zum Hinzufügen einer neuen Methode zu einer Klasse bereit. Um Abstraktions- und Leistungsprobleme zu umgehen, verwendet Javassist ein spezielles Übersetzungsmethodenobjekt (`CtMethod`), das einen Methodenrumpf spezifiziert. Dieser Methodenrumpf wurde von einer gegebenen Methode kopiert. Da der Methodenrumpf von einer übersetzten Methode kopiert wurde, ist beim Laden keine Übersetzung von Quellcode erforderlich. Entwickler können einen Methodenrumpf in Java beschreiben und im Voraus übersetzen. Javassist liest den Bytecode dieser übersetzten Methode ein und fügt ihn einer anderen Klasse hinzu. Dadurch wird die Leistung von Javassist gesteigert, da keine Übersetzung beim Laden durchgeführt werden muss. Obwohl Javassist das Entfernen einer Methode von einer Klasse nicht unterstützt, werden Methoden bereitgestellt um Methodenrümpfe zu verändern. Ein Methodenrumpf kann mit einem gegebenen Methodenrumpf substituiert werden. Außerdem bietet Javassist die Möglichkeit Ausdrücke innerhalb eines Methodenrumpfes zu modifizieren [Chi00].

Das Klassenladeprogramm von Javassist erlaubt einem geladenen Software-System mit diesem Klassenladeprogramm das Laden von Klassen zu kontrollieren. Wurde ein Software-System mit dem Klassenladeprogramm von Javassist geladen und es beinhaltet eine Klasse C, dann kann es das Laden von C durch das Klassenladeprogramm unterbrechen, um den Bytecode von C selbstreflektierend zu modifizieren wie in Abbildung 2.5 veranschaulicht. Zur Vermeidung einer unendlichen Rekursion durch die Unterbrechung beim Laden einer

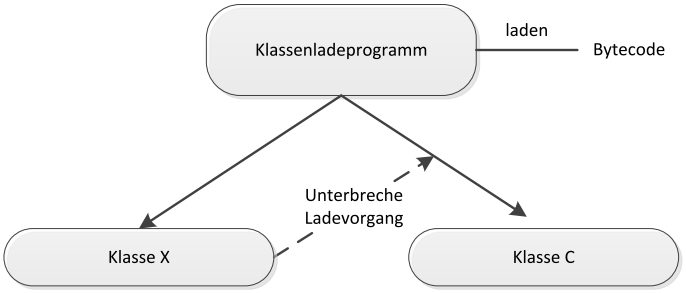


Abbildung 2.5.: Selbstreflektierendes Laden einer Klasse durch das Klassenladeprogramm von Javassist [Chi00]

Klasse, sind weitere Unterbrechungen nicht gestattet. Das standardmäßige Klassenladeprogramm von Java verbietet aus Sicherheitsgründen das selbstreflektierende Laden von Klassen. In Java wird eine Klasse C, die durch ein eigenes Klassenladeprogramm geladen wurde als eine andere Klasse angesehen, als eine im Software-System mit dem Namen C angegebene Klasse, die mit dem Klassenladeprogramm geladen wurde, mit dem das Software-System selbst geladen wurde [Chi00].

Um dieses Problem zu umgehen nutzen Chiba et al. in [CST03] einen HotSwap genannten Mechanismus aus, um dynamisch bereits geladene Klassendateien neu zu laden, damit deren Klassendefinitionen aktualisiert werden. Der HotSwap-Mechanismus erlaubt jedoch nicht das Neuladen von Klassendateien zu denen eine neue Methode oder ein neues Feld angefügt wurde. Der HotSwap-Mechanismus ist eine Erweiterung der Java Platform Debugger Architektur (JPDA) und wurde vom Java2 SDK 1.4 eingeführt [CST03]. Die neue Erweiterung kapselt die Fähigkeit, modifizierten Code in einer laufenden Anwendung mittels der Programmierschnittstelle des Debuggers zu substituieren. Beispielsweise kann man dadurch eine einzige Klasse neu übersetzen und die alte Instanz mit der neuen Instanz ersetzen [JPD02]. Falls eine neudefinierte Methode aktive Rahmen im Kellerspeicher (engl. stack) hat, führen diese aktiven Rahmen weiterhin den Bytecode der ursprünglichen Methode aus. Die neudefinierten Methoden werden bei neuen Aufrufen verwendet. Das Neudefinieren einer Klasse führt nicht dazu, dass ihre Initialisierung ausgeführt wird. Die Werte von bereits zuvor existierten statischen Variablen bleiben erhalten, wohingegen uninitialisierte (neue) statische Variablen ihren Standardwert zugewiesen bekommen. Gab es von einer neudefinierten Klasse bereits Instanzen, dann werden alle diese Instanzen die Felder besitzen, die durch die neudefinierte Klasse definiert werden. Zuvor existierende Felder werden ihre Werte beibehalten. Allen neuen Feldern wird ebenfalls ihr Standardwert zugewiesen. Initialisierungen der Instanz oder Konstruktoren werden nicht ausgeführt [JVM03].

2.2.4. Bytecode-Instruktionszähler ByCounter

Java Bytecode wird von der JVM ausgeführt, welche von der unterliegenden Hardware- und Software-Plattform abstrahiert. Dadurch ensteht eine Portabilität kompilierter Klassen über die verschiedenen Plattformen hinweg, für die eine JVM angeboten wird. Kuperberg et al. stellt in [KKR08] ByCounter vor, einen portablen Ansatz für das Zählen von Bytecode-Instruktionen und Methodenaufrufen zur Laufzeit. Dabei wird der Bytecode der Anwendung anstelle der JVM instrumentiert. ByCounter kann mit jeder JVM genutzt werden und das instrumentierte Software-System kann ebenso auf jeder JVM zur Ausführung kommen. Bestehende Methodensignaturen in instrumentierten Klassen müssen dabei nicht geändert werden. Dadurch entstehen keine strukturellen Änderungen im vorhandenen Software-System [KKR08].

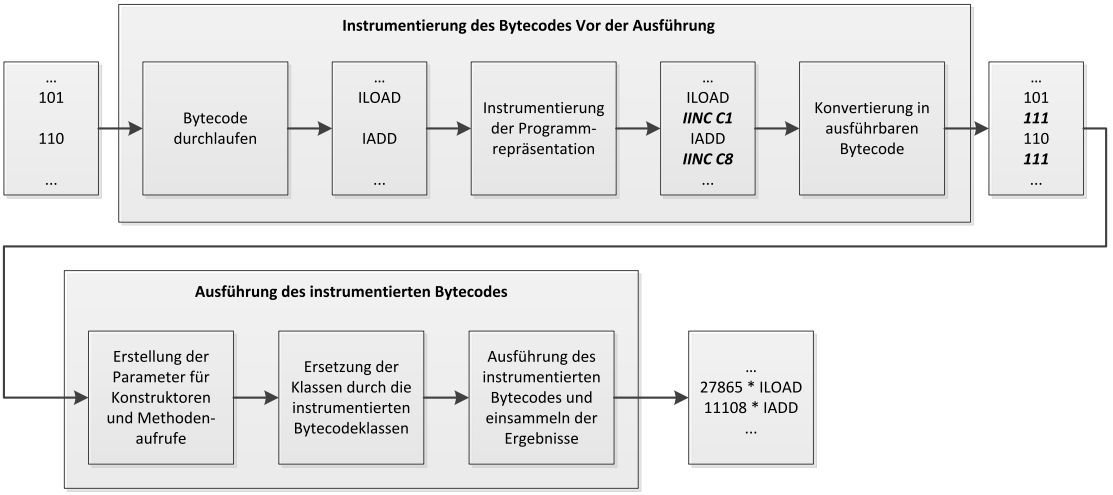


Abbildung 2.6.: Überblick über den ByCounter-Prozess basierend auf [KKR08]

In Abbildung 2.6 sind alle sechs Schritte die von ByCounter durchgeführt werden illustriert. In Schritt eins wird der existierende Bytecode einer Klassendatei in eine navigierbare, strukturierte Repräsentation gebracht. Dabei behilft sich ByCounter mit dem ASM Bytecode Engineering Rahmenwerk [BLC02a], das eine Repräsentation des Bytecodes einer Klasse mit semantischen Details bietet. Durch die ASM Programmierschnittstelle können individuelle Bytecode-Instruktionen in den Bytecode hinzugefügt, geändert oder entfernt werden. In Schritt zwei wird die Instrumentierung zur Zählung der Bytecode-Instruktionen in die Repräsentation eingefügt. Während der Ausführung der instrumentierten Methode werden diese Zähler initialisiert, inkrementiert, evaluiert und schließlich gemeldet. In Schritt drei wird die instrumentierte Repräsentation der Klasse zurück in ausführbaren Bytecode konvertiert. Mit diesem Schritt ist die Instrumentierung abgeschlossen, auf die dann die Ausführung folgt. In Schritt vier wird die Ausführung der instrumentierten Methode vorbereitet. Wird die instrumentierte Klasse im gleichen Kontext wie die uninstrumentierte Klasse ausgeführt, sind die Vorbereitungen absolut identisch mit denen der Ausführung der uninstrumentierten Klasse und sollten demnach bereits durch den vorhanden Kontext erfüllt werden. Die uninstrumentierte Klasse wird in Schritt fünf durch die instrumentierte Klasse im ausgebrachten Software-System ersetzt. Dies kann im einfachsten Fall durch das Neustarten des Software-Systems passieren nach dem der originale Bytecode durch den instrumentierten Bytecode ersetzt wurde. Abschließend wird die instrumentierte Methode in Schritt sechs ausgeführt, um die ausgeführten Bytecode-Instruktionen und Methodenaufrufe zu zählen und zu melden. Danach kann bei Bedarf der ursprüngliche uninstrumentierte Zustand der Klasse wiederhergestellt werden [KKR08]. ByCounter bietet die Möglichkeit die Ausführungszeit der Bytecode-Instruktionen über einen ByBench genannten Benchmark für ein System zu ermitteln. Dadurch kann die Ausführungszeit einer Methode auf Grundlage der ausgeführten Bytecode-Instruktionen geschätzt werden. Bytecode-Instruktionen haben unterschiedliche Ausführungszeiten wie die nachfolgende Studie zeigt.

Lambert und Power haben in [LP08] das Plattform unabhängige Antwortzeitverhalten von 137 Bytecode-Instruktionen untersucht. Die Autoren beschränken sich mit den 137 ausgewählten Bytecode-Instruktionen auf den imperativen Kern der JVM und konnten 13 Instruktionen identifizieren deren Ausführungszeit sich signifikant von den Ausführungszeiten der verbleibenen Instruktionen unterscheidet. Diese Ausreißer sind zusammen mit der entsprechenden Ausführungszeit in Tabelle 2.2 aufgelistet. Die charakteristischen Größen der Verteilung sind in Tabelle 2.1 aufgezeigt. Etwa 91% der untersuchten imperativen

Instruktionen werden durch das Minimum und das Maximum eingeschlossen. 50% davon fallen zwischen das $0,25$-Quantil und das $0,75$-Quantil, 24% fallen zwischen das Minimum und das $0,25$-Quantil während die restlichen 16% zwischen das $0,75$-Quantil und das Maximum fallen. Die Instruktionen die zwischen das $0,25$-Quantil und das $0,75$-Quantil fallen haben eine Ausführungszeit im Bereich von $4,488 * 10^{-9}$ Sekunden [LP08].

Minimum	$0,25$-Quantil	Mittelwert	$0,75$-Quantil	Maximum
$1,093 * 10^{-9}$	$3,583 * 10^{-9}$	$4,527 * 10^{-9}$	$8,072 * 10^{-9}$	$1,22 * 10^{-8}$

Tabelle 2.1.: Fünf Punkte Zusammenfassung des Antwortzeitverhaltens der 137 vermessenen JVM Instruktionen [LP08]. Alle Messwerte der Tabelle sind in Sekunden.

Die Ausreißer können dabei so charakterisiert werden, dass die Instruktionen in drei breite Kategorien fallen. Die erste Kategorie repräsentiert Instruktionen, die eine primitive Typkonvertierung durchführen. Die Konvertierung einer Fließkommadarstellung in eine Ganzzahldarstellung des Typs *long* oder *int* und umgekehrt, beansprucht eine beachtlichte Menge an Zeit. Die Konvertierung von verschiedenen Zahlendarstellungen benötigt umgefähr zehnmal solange als Konvertierungen von Fließkomma zu Fließkomma und Ganzzahl zu Ganzzahl. Die zweite Kategorie stellen Instruktionen dar, die Restoperationen ausführen wie beispielsweise *irem*, *lrem*, *frem* und *drem*. Diese Instruktionen sind in Klassendateien zu finden, deren Java-Quelltext Modulooperationen beinhalten und benötigen im Allgemeinen zehnmal länger verglichen mit anderen imperativen Instruktionen. Die dritte Kategorie innerhalb der Menge der Ausreißer bilden die Instruktionen *ldiv* und *idiv*. Die Menge der Ausreißer deutet den Autoren zufolge scheinbar darauf hin, dass berechnungsintensive Programme signifikant längere Ausführunszeiten haben im Vergleich zu anderen imperativen Programmen [LP08].

Instruktion	Ausführungszeit in Sekunden
l2f	$2.205176 * 10^{-8}$
i2d	$2.635641 * 10^{-8}$
irem	$3.179264 * 10^{-8}$
l2d	$3.321960 * 10^{-8}$
idiv	$3.449739 * 10^{-8}$
frem	$4.191904 * 10^{-8}$
drem	$5.049472 * 10^{-8}$
f2i	$8.028337 * 10^{-8}$
f2l	$8.267345 * 10^{-8}$
d2l	$8.795759 * 10^{-8}$
d2i	$8.976430 * 10^{-8}$
ldiv	$1.132759 * 10^{-7}$
lrem	$1.180681 * 10^{-7}$

Tabelle 2.2.: 13 als Ausreißer identifizierte Instruktionen [LP08]

Die Autoren haben mit dem Mittelwert als Distanzmetrik eine Cluster-Analyse durchgeführt um die Bytecode-Instruktionen weiter zu klassifizieren. Die Analyse teilte die untersuchten Bytecode-Instruktionen in 11 Gruppen ein. Die ersten drei Gruppen umfassen dabei bereits 124 der 137 untersuchten Instruktionen. Das entspricht ungefähr 91% aller Instruktionen und ist gleichzusetzen mit den Instruktionen die durch das Minimum und Maximum eingeschlossen werden. Die restlichen acht Gruppen werden von den Ausreißern gebildet [LP08]. Lambert und Power geben in [LP08] nur die genauen Ausführungszeiten

zu den 13 Ausreißern an. Die genauen Ausführungszeiten der restlichen 124 Bytecode-Instruktionen bleiben unerwähnt. Des Weiteren haben sich die Autoren auf 137 Bytecode-Instruktionen beschränkt und die verbleibenden 119 Bytecode-Instruktionen blieben unberücksichtigt.

2.3. Versionskontrollsysteme

Versionskontrollsysteme (VKS) erlauben Software-Entwicklern gleichzeitiges Modifizieren von Quellcode, die Behandlung von in Konflikt stehenden Änderungen und die Übergabe von Aktualisierungen als neue Version. Solche Systeme fördern die Zusammenarbeit und den gemeinsamen Beitrag einer großen Entwicklerbasis zu geteiltem Quellcode. Dies führte dazu, dass Software-Entwickler weltweit VKS zu Verwaltung einer großen Anzahl diverser Software-Systeme einsetzen. Die VKS unterstützen neben der verteilten inkrementellen Mitwirkung an gemeinsamen Quellcode über speicherbasierte Ablagen (engl. Repositories) auch über eine Reihe von Werkzeugen, die Zugriff auf Dateien, die Verwaltung, Nachverfolgung und Verzweigung von Versionen bieten. Außerdem bieten sie dem Entwickler Unterstützung durch automatisches Auflösen von Konflikten und geben ihm Rückmeldung, wenn diese fehlschlägt oder andere Ereignisse durch andere Entwickler einreten [MK08].

Ein VKS ist ein System, das die Änderungen an einer Datei oder einer Menge von Dateien über die Zeit hinweg aufzeichnet, so dass eine bestimmte Version später wieder aufgerufen werden kann. Da dies mit nahezu jeder Dateiart gemacht werden kann, sprechen wir in diesem Kontext hier von Dateien die Teil eines Software-Systems sind, wie beispielsweise Klassendateien mit Java-Code. Ein VKS erlaubt die Wiederherstellung eines frühreren Zustandes einer Datei, oder sogar die Wiederherstellung eines früheren Zustandes des gesamten Projektes, den Vergleich von Änderungen über die Zeit hinweg, das Anzeigen der Person die zuletzt Änderungen vorgenommen hat, die Probleme verursachen könnten, wer ein Problem eingeführt hat und wann und einiges mehr. Eine simple und äußerst fehleranfällige lokale Variante eines VKS ist Dateien in, mit einem Zeitstempel versehene Verzeichnisse zu kopieren. Um den Problemen die mit diesem Ansatz behaftet sind aus dem Wege zu gehen, haben Entwickler vor langer Zeit ein lokales VKS entwicklet, das eine einfache Datenbank benutzt, um alle Änderungen von Dateien zubehalten, die der Versionskontrolle unterliegen. Eines der bekannteren VKS ist das von Tichy in [Tic82] vorgestellte RCS. Dieses Werkzeug behält die Differenz zwischen zwei Dateien von einer Änderung zur nächsten in einem speziellen Format auf der Festplatte. Dadurch kann RCS die Version die eine Datei zu einem Zeitpunkt hatte wiederherstellen, indem alle Änderungen aufsummiert werden. Die nächste Entwicklung waren zentralisierte VKS wie Subversion [CSFP04], die einen einzelnen zentralen Server besitzen. Dieser zentrale Server beinhaltet alle versionierten Dateien und eine Reihe von Abnehmern, die sich Dateien aus diesem zentralen Ort abholen. Einer der Nachteile dieses Ansatzes ist der einzelne Bruchpunkt dieses Aufbaus in Form des zentralen Servers. Steht der Server nicht zur Verfügung kann niemand versionierte Änderungen speichern an denen gearbeitet wird. Falls die zentrale Datenbank in denen die Änderungen verwaltet werden korrumpiert wird und keine ordnungsgemäße Sicherungen gemacht wurden, geht die gesamte Änderungshistorie des Projektes verloren. An diesem Punkt kommen verteilte VKS wie Git ins Spiel. In einem verteilten VKS holen sich Entwickler nicht einfach nur das letzte Speicherabbild einer Datei, sondern sie spiegeln sich die gesamte Speicherablage (engl. Repository). Jedes Ausgeben von Dateien ist dadurch eine vollständige Sicherung des gesamten Datenbestandes [Cha09].

2.3.1. Git

Der Hauptunterscheid zwischen Git und anderen VKS ist die Art und Weise wie Git die Änderungen verwaltet. Die meisten anderen Systeme speichern die Dateien und die

gemachten Änderungen zu jeder Datei über die Zeit hinweg, wie in Abbildung 2.7 illustriert [Cha09].

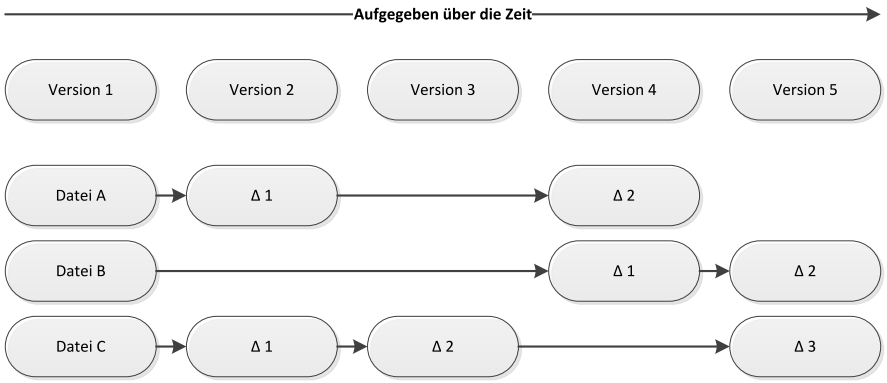


Abbildung 2.7.: Informationsverwaltung anderer VKS wie z.B. in Subversion [Cha09]

Git verwaltet seine Daten dagegen eher wie eine Menge von Schnappschüssen eines minimalen Dateisystems. Jedesmal wenn eine Änderung aufgegeben wird oder der Zustand eines Projektes in Git gespeichert wird, macht Git im Prinzip einen Abzug des aktuellen Zustandes der Dateien und speichert eine Referenz zu diesem Schnappschuss. Haben sich Dateien nicht verändert, speichert Git die Dateien nicht erneut sondern fügt einfach eine Referenz auf die vorherige, identische Datei hinzu, die bereits gespeichert ist. Die Verwaltungsstruktur der Änderungen von Git ist daher eher wie in Abbildung 2.8.

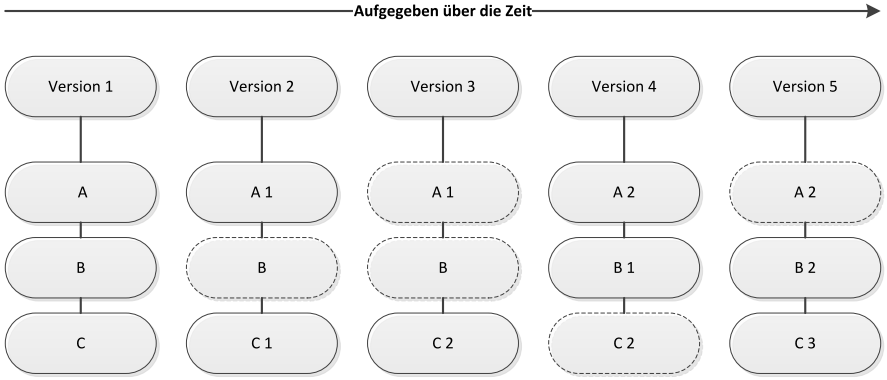


Abbildung 2.8.: Informationsverwaltung von Git [Cha09]

Bevor in Git etwas gespeichert wird, wird es mit einer Prüfsumme versehen. Dadurch ist es unmöglich den Inhalt einer Datei oder eines Verzeichnisses zu ändern, ohne das Git dies nicht bemerkt. Der verwendete Mechanismus zur Erstellung der Prüfsumme wird als SHA-1 Prüfsumme bezeichnet. Dies ist eine Zeichenkette mit einer Länge von 40 Zeichen, bestehend aus hexadezimalen Zeichen (0-9 und a-f), die auf Basis des Inhalts einer Datei oder einer Verzeichnisstruktur berechnet wird. Eine SHA-1 Prüfsumme kann beispielsweise wie folgt aussehen: *24b9da6552252987aa493b52f8696cd6d3b00373*. In der Datenbank von Git kann man alles über diese Prüfsumme adressieren. Werden Änderungen in Git aufgegeben, wird die Prüfsumme für jedes Verzeichnis berechnet und der Inhalt jeder Datei als Blob-Objekt in der Speicherablage gespeichert. Anschließend wird ein Commit-Objekt mit allen Metadaten und einem Zeiger auf den Baum des Wurzelprojektes erstellt. Damit kann ein Schnappschuss wiederhergestellt werden, sobald er benötigt wird. Die Speicherablage verfügt deshalb über drei Arten von Objekten: Ein Blob-Objekt für den Inhalt jeder Datei, einem Baum mit der Auflistung des Inhalts jedes Verzeichnisses und den Verweisen von

den Dateinamen zu den Blob-Objekten und einem Commit-Objekt mit dem Zeiger auf den Wurzelbaum und den Metadaten. Dieses Konzept ist in Abbildung 2.9 dargestellt [Cha09].

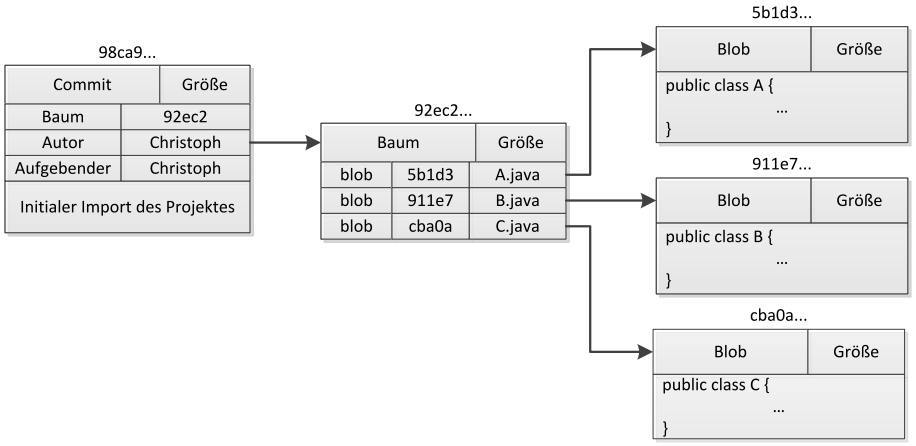


Abbildung 2.9.: Daten in der Speicherablage von Git [Cha09]

Werden erneut Änderungen vorgenommen und aufgegeben, speichert das neue Commit-Objekt eine Referenz auf das alte, unmittelbar zuvor aufgegebene Commit-Objekt. Nachdem also zweimal weitere Änderungen aufgegeben wurden, sieht die Historie in Git etwa wie in Abbildung 2.10 aus.

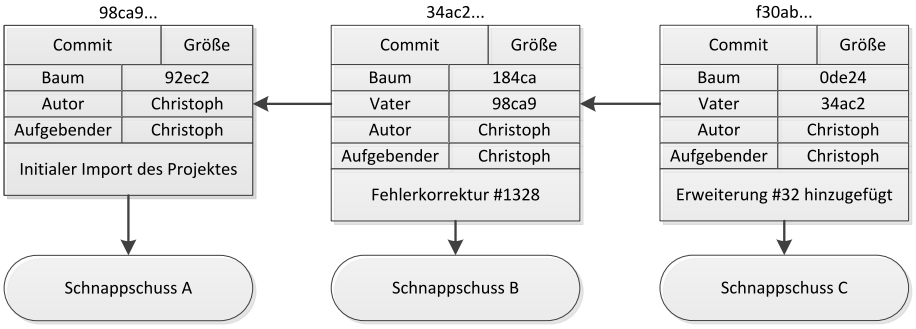


Abbildung 2.10.: Historie in Git nach dreimaligem Aufgeben von Änderungen [Cha09]

Git bietet die Möglichkeit im Versionsgraphen Abzweigungen zu erstellen. Eine Abzweigung (engl. branch) ist im Grunde ein Zeiger auf eines dieser Commit-Objekte. Der Standardzweig wird in Git als Master-Zweig bezeichnet. Werden nur Änderungen aufgegeben und dadurch neue Commit-Objekte erzeugt, erhält man einen Master-Zweig der immer auf das zuletzt erstellte Commit-Objekt zeigt und sich mit jeder Ankunft von Änderungen automatisch vorwärts bewegt [Cha09].

Wird nun beispielsweise ein neuer lokaler Zweig mit dem Namen Test angelegt, wird einfach ein neuer Zeiger erstellt, der bewegt werden kann. Damit Git den Überblick behält in welchem Zweig gerade gearbeitet wird, gibt es einen speziellen Zeiger mit der Bezeichnung Kopf (engl. head). Der Kopfzeiger referenziert nach dem Anlegen eines neuen Zweiges zunächst weiter den Master-Zweig. Lässt man sich jedoch den Schnappschuss des neu angelegten Testzweiges ausgeben (engl. checkout), wird der Kopfzeiger vom Master-Zweig auf den Testzweig umgehängt [Cha09]. Git bietet aber auch die Möglichkeit die gewünschte Abzweigstelle anzugeben. So könnte man ebenso bei der Erstellung des Testzweiges angeben, dass man auf Basis eines anderen Schnappschusses starten will. Wechselt man

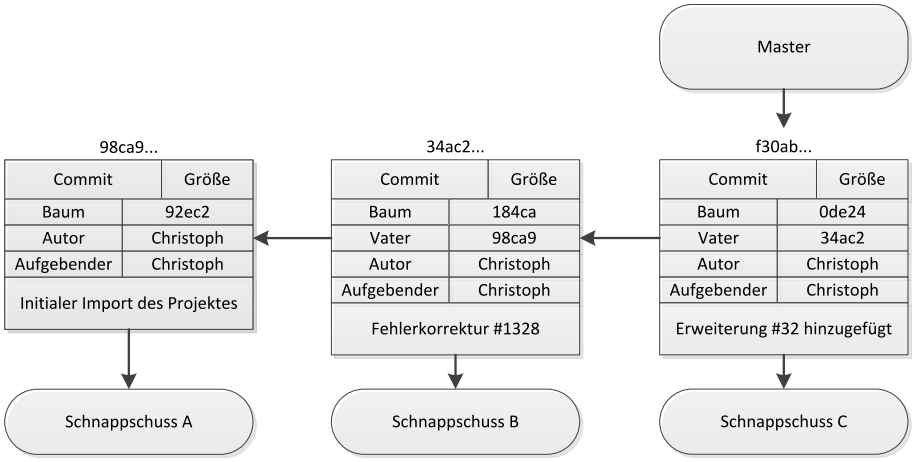


Abbildung 2.11.: Illustration des Zweigkonzeptes [Cha09]

dann im Anschluss vom Master-Zweig in den Testzweig indem man sich den Testzweig in den Arbeitsbereich ausgeben lässt, ist dort nicht wie zuvor der *Schnappschuss C* sondern *Schnappschuss B* vorzufinden [Cha09].

2.4. Kontinuierliche Integration

Die kontinuierliche Integration (engl. continuous integration) nach [DMG07] bezeichnet einen Prozess der Qualitätssicherung in Software-Projekten. Integration bezeichnet dabei den Vorgang, indem separierte Software-Artefakte zusammengefügt werden, um zu bestimmen wie diese als Ganzes funktionieren. Damit soll sichergestellt werden, dass Komponenten problemlos zusammenarbeiten. Nach Fowler ist die kontinuierliche Integration "eine Software-Entwicklungsmethode bei der die Mitglieder einer Gruppe ihre Arbeit häufig integrieren, normalerweise integriert eine Person mindestens täglich – was zu mehreren Integrationen pro Tag führt. Jede Integration ist durch automatische Erstellungen (inklusive Tests) verifiziert, um Integrationsfehler so früh wie möglich zu erkennen" [Fow06]. Die Erstellung einer Software bezeichnet dabei eine Reihe ausgeführter Aktivitäten wie die Übersetzung, das Testen, die Inspektion und die Ausbringung der Software. Wird mit der Integration bis Projektende gewartet, führt dies meist zu aller Art von Software-Qualitätsproblemen. Diese sind teuer und haben oftmals Projektverzögerungen zur Folge. Die kontinuierliche Integration adressiert diese Risiken. Die Schritte in einem typischen Szenario der kontinuierlichen Integration sehen oftmals folgendermaßen aus [DMG07]:

1. Ein Entwickler ändert Code und gibt diese Änderungen ins Versionskontrollsystem (VKS) auf. Das für die Integration zuständige System prüft die Ablage des VKS in regelmäßigen Abständen auf Änderungen (z.B. alle paar Minuten).

2. Das Integrationssystem erkennt bald darauf, dass Änderungem im VKS vorhanden sind und holt sich eine aktuelle Kopie des Codes von der Ablage des VKS. Anschließend wird ein Skript zur Integration der Software ausgeführt, das die auszuführenden Aktionen definiert.

3. Das Integrationssystem generiert einen Bericht als Rückmeldung und sendet diesen beispielsweise an bestimmte Projektmitglieder.

4. Anschließend wird die Prüfung des VKS auf Änderungen durch das Integrationssystem fortgesetzt [DMG07].

Abbildung 2.12 veranschaulicht die Bestandteile der kontinuierlichen Integration. Damit der größte Nutzen aus der kontinuierlichen Integration gezogen und diese so effektiv ein-

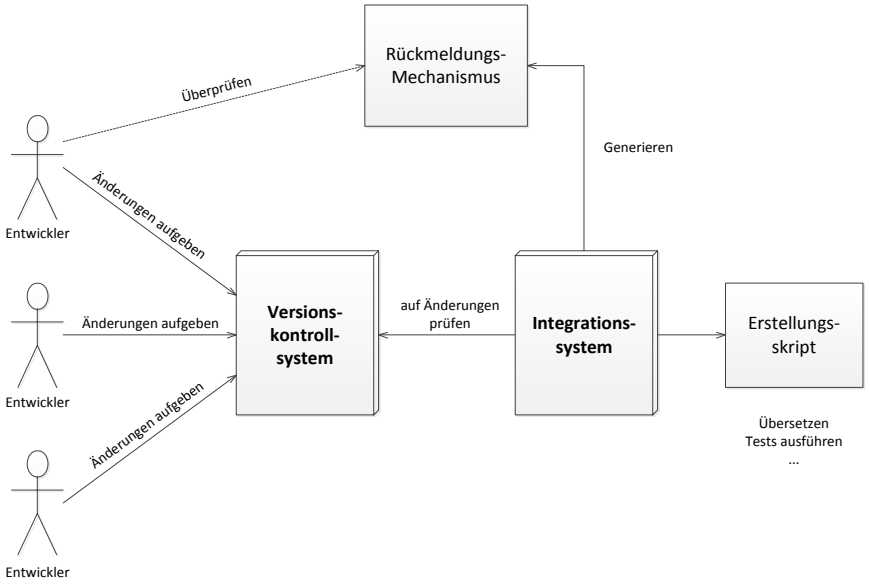


Abbildung 2.12.: Bestandteile eines Systems zur kontinuierlichen Integration [DMG07]

gesetzt werden kann wie möglich, beschreibt Duvall et al. in [DMG07] sieben zu praktizierende Grundsätze:

1. Änderungen am Code sollen häufig und mindestens täglich ins VKS aufgegeben werden.

2. Fehlerbehafteter Code, der nicht übersetzt werden kann oder einen Test nicht besteht, darf nicht ins VKS aufgegeben werden.

3. Die Fehler aufgrund derer eine Software nicht erstellt werden kann müssen umgehend bereinigt werden.

4. Die ordnungsgemäße Funktionalität der Software muss mit automatisierten Tests sichergestellt werden. Diese Tests sollen im Kontext der automatisierten Erstellung ausgeführt werden.

5. Bevor Änderungen ins VKS aufgegeben werden, müssen alle Tests bestanden werden.

6. Entwickler führen eigene (lokale) Integrationen der Software durch, um einen Fehlschlag der kontinuierlichen Integration zu vermeiden. Hierzu rufen sie die aktuellen Änderungen anderer Entwickler aus dem VKS ab.

7. Fehlerbehafteter Code soll nicht aus dem VKS abgerufen werden. Die Entwickler sollen entweder abwarten, bis eine Korrektur verfügbar ist oder dem betroffenen Entwickler bei der Korrektur unterstützen.

Die kontinuierliche Integration bietet dadurch ein Sicherheitsnetz, welches das Risiko reduziert, dass Fehler in die Codebasis eingeführt werden. Ein wichtiger Bestandteil der kontinuierlichen Integration ist die Automatisierung wiederkehrender Aktivitäten, wie beispielsweise Übersetzen von Code, Testen, Inspektion und Rückmeldung der Resultate. Durch die Automatisierung wird sichergestellt, dass der Prozess jedesmal gleich abläuft und einer geordneten Struktur folgt. Außerdem wird der Prozess immer dann ausgeführt, sobald eine Änderung in das VKS aufgegeben wird [DMG07].

2.4.1. Automatisierte Unit-Tests

Ohne automatisierte Tests ist es für Entwickler schwierig neuen Änderungen zu vertrauen. Sie müssten aus dem VKS abgerufene Änderungen selbst auf Fehler überprüfen. Deshalb setzen viele Entwickler auf Testwerkzeuge wie JUnit [Bec] zur Ausführung von Tests. Unit-Tests testen das Verhalten kleiner Elemente eines Software-Systems wie beispielsweise das Verhalten einer einzigen Klasse. Die eins zu eins Beziehung zwischen einem Unit-Test und einer Klasse wird manchmal, aber selten, aufgrund der starken Bindung der Klassen untereinander, um weitere Klassen erweitert. Unit-Tests sollten höchstens Abhängigkeiten zu anderen Klassen haben. Diese Abhängigkeiten sind selbst nur einfache Klassen und besitzen keine große Objekthierarchie. Dennoch verwenden sogar Unit-Tests einfache Modellobjekte (engl. mocks) mit denen reale, komplexe Objekte substituiert werden. Ist ein solches Abhängigkeitsobjekt selbst von einer externen Entität abhängig, wie beispielsweise dem Dateisystem, so handelt es sich bei diesem Test nicht länger um einen Unit-Test sondern vielmehr um einen Komponententest. Das Schlüsselelement bei Unit-Tests ist, die Unabhängigkeit von externen Entitäten wie Datenbanken, die tendenziell den Zeitaufwand zur Vorbereitung und Ausführung eines Tests erhöhen. Unit-Tests können früh in der Entwicklung erstellt und ausgeführt werden [DMG07].

JUnit

JUnit [Bec] ist die im Java-Umfeld am weitesten verbreitete und am häufigsten eingesetzte Bibliothek zum Testen von Software-Systemen [Sma11]. Annotationen bilden in JUnit eine wichtige Grundlage. Ein Unit-Test wird in JUnit durch eine gewöhnliche Java-Klasse repräsentiert, deren Methoden die Testfälle darstellen. Durch die von JUnit bereitgestellte Annotation `Test` werden die Methoden der Testklasse als Testfall gekennzeichnet. Die Testklasse selbst kann mit der Annotation `RunWith` annotiert werden. Der von dieser Annotation bereitgestellte Parameter ermöglicht die Angabe einer Dienstklasse. Diese Dienstklasse ermittelt dann welche Testklassen tatsächlich ausgeführt werden. Die in der Annotation `SuiteClasses` als Parameter angebenen Testklassen, geben die zu berücksichtigen Testklassen an. Es kann hier eine beliebige Anzahl an Testklassen definiert werden. Testmethoden sind bei JUnit parameterlos [Mef06].

2.4.2. Jenkins

Jenkins [Jen] ist ein in Java geschriebenes, quelloffenes Werkzeug für die kontinuierliche Integration. Die zahlreichen verfügbaren Zusatzprogramme umfassen u.a. Versionskontrollsysteme, Erstellungswerkzeuge, Qualitätsmetriken, Erstellungsbenachrichtigungen, die Integration von externen Systemen. Eine bestimmte Aufgabe wird im Erstellungsprozess von Jenkins als Erstellungsaufgabe (engl. build job) bezeichnet. Eine solche Erstellungsaufgabe kann beispielsweise die Übersetzung des Quellcodes und die Ausführung der Unit-Tests umfassen. Eine Erstellungsaufgabe kann aber auch andere verwandte Aufgaben ausführen, wie beispielsweise die Ausführung von Integrationstests, das Messen der Codeüberdeckung (engl. code coverage) oder anderer Qualitätsmetriken, die Generierung technischer Dokumentation oder das verteilen (engl. deploy) der Anwendung. Jenkins bietet Unterstützung für das Projektverwaltungswerkzeug Maven, sowie das Versionskontrollsystem Git. Der Quellcode eines Projektes wird von Jenkins aus dem Versionskontrollsystem abgerufen. Der Erstellungsprozess arbeitet anschließend in einem eigenen Arbeitsbereich. Nachdem ein Erstellungsprozess fertig ist, können weitere Aufgaben anstehen. Hierzu bietet Jenkins die Möglichkeit sogenannte Post-Erstellungsaktionen (engl. post-build actions) zu definieren. Eine solche Post-Erstellungsaktion kann beispielweise die Anzeige der Testergebnisse der Unit-Tests nach der Ausführung einer Erstellungsaufgabe sein. Für diesen und andere Zwecke können Zusatzprogramme definiert werden, die als solche Post-Erstellungsaktionen von Jenkins ausgeführt werden [Sma11].

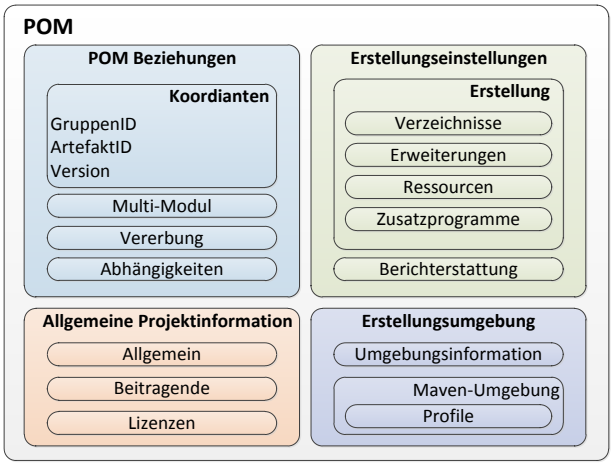


Abbildung 2.13.: Das Projektmodell [Son11]

2.4.3. Apache Maven

Die formale Definition von Apache Maven [Mav] in [Son11] lautet in ihrer Übersetzung: "Maven ist ein Projektverwaltungswerkzeug, das ein Projektmodell (engl. project object model), eine Menge an Standards, einen Projektlebenszyklus, ein System zur Verwaltung von Abhängigkeiten und eine Logik zur Ausführung der Ziele von Zusatzprogrammen zu bestimmten Phasen im Lebenszyklus umfasst. Wenn man Maven verwendet, beschreibt man das Projekt unter Verwendung eines wohldefinierten Projektmodells. Maven kann daraufhin querschneidende Logik von einer Menge an geteilten (oder benutzerdefinierten) Zusatzprogrammen anwenden". Maven ist demnach ein mächtiges Werkzeug der Projektverwaltung. In Maven wird der Erstellungsprozess deklarativ mit Hilfe des Projektmodells (POM) beschrieben. Die Metadaten des Projektes wie beispielsweise die Pfade zu den Klassendateien und die Abhängigkeiten wie externe Bibliotheken sind im POM enthalten. Aus diesen Informationen erstellt Maven einen auf dieses Projekt angepassten Erstellungsprozess und führt diesen aus [Pop08]. Das Projektmodell ist in Abbildung 2.13 dargestellt. In den allgemeinen Projektinformationen steht der Name des Projektes, die URL des Projektes, die geldgebende Organisation sowie eine Liste der Entwickler und Beitragenden zusammen mit der Projektlizenz. Die Erstellungseinstellungen passen den standardmäßigen Erstellungsprozess auf die Bedürfnisse des Projektes an. Hier können dem Erstellungsprozess beispielsweise neue Zusatzprogramme hinzugefügt werden. Die Erstellungsumgebung kann aus verschiedenen Profilen bestehen. Diese Profile können sodann in unterschiedlichen Umgebungen (z.B. Entwicklung oder Produktion) angewendet werden. Die POM-Beziehungen definieren die Abhängigkeiten von anderen Projekten, erben POM-Einstellungen übergeordneter Projekte, definiert die eigenen Koordinaten und kann zusätzlich weitere Module beinhalten [Son11]. Das Projektmodell beschreibt dabei das zu erstellende Artefakt. Popp definiert ein Artefakt in [Pop08], als "[...]eine in sich abgeschlossene (Teil-)Funktionalität eines Projektes, die eigenständig ausgeliefert werden kann.". Besteht ein Projekt aus mehreren Komponenten, wird für jede Komponenten ein eigenes POM erstellt und sodann auch jeweils ein eigenes Artefakt. Ein übergeordnetes POM fügt dann alle Artefakte des Gesamtprojektes zusammen und generiert die Auslieferungsdatei [Pop08]. Der am häufigsten vorkommende Anwendungsfall von Maven ist die Erstellung und Verteilung eines Software-Projektes [Son11]. Maven legt großen Wert auf die allgemeine Wiederverwendung. Der Entwurf von Maven sieht vor, dass ein Großteil der Verantwortung an Zusatzprogramme delegiert wird, die widerrum den Lebenszyklus von Maven betreffen und den Zugriff auf sogenannte Ziele anbieten. Diese Ziele der Zusatzprogramme zeigen sich verantwortlich für Dinge wie die Übersetzung des Quellcodes, die

Erstellung von Jar-Dateien und jede andere Aufgabe die bei der Erstellung vorkommen muss. Der Kern von Maven ist sehr einfach gehalten und beinhaltet keine Funktionalität zum Übersetzen oder zum Ausführen von Unit-Tests. Die dazu notwendige Funktionalität ist in Zusatzprogrammen implementiert. Diese Zusatzprogramme werden aus der Ablage von Maven (engl. Maven Repository) abgerufen. Wird Maven erstmalig ausgeführt, werden die meisten grundlegenden Zusatzprogramme von Maven aus der zentralen Ablage von Maven aus dem Internet abgerufen. Dadurch kann der Umfang des Erstellungsprozesses leicht durch das Hinzufügen weiterer Zusatzprogramme erweitert werden. Zusätzlich zu den Zusatzprogrammen ruft Maven aber auch Abhängigkeiten wie beispielsweise Bibliotheken oder Komponenten aus der Ablage ab, die vom zu erstellenden Projekt referenziert werden. Diese beiden Tatsachen erlauben die allgemeine Wiederverwendung der Erstellungslogik [Son11].

2.5. Statistik

Das Unterkapitel beginnt mit Abschnitt 2.5.1 und den grundlegenden Definitionen arithmetisches Mittel, Varianz, Standardabweichung und Konfidenzinterval. Im Anschluss werden in Abschnitt 2.5.2 die Varianzanalyse und der Tukey-Test vorgestellt. Um den Einfluss verschiedener Alternativen zu beurteilen, beschreiben Georges et al. in [GBE07] die Varianzanalyse mit angehängtem Tuckey-Test als eine statistisch aussagekräftige Methode.

2.5.1. Grundlagen

In diesem Abschnitt werden grundlegende Begriffe der Statistik eingeführt, die für das Verständnis dieser Diplomarbeit von Bedeutung sind. Die nachfolgenden Definitionen und Erklärungen stammen aus [HEK05] und [GBE07].

Oft lässt sich eine Messreihe durch wenige charakteristische Größen kennzeichnen. Ein solches Charakteristikum ist z.B. eine Maßzahl wie das arithmetische Mittel. Dieses Lagemaß beschreibt in geeigneter Weise ein Zentrum der Messwerte. Sind $x_1, \ldots, x_n$ die gemessenen Werte eines Merkmals X, so ist das **arithmetische Mittel (Mittelwert, durchschnittlicher Wert)**

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.1}$$

ein sinnvolles Lagemaß [HEK05].

Bezeichnen wiederum $x_1, \ldots, x_n$ die Messwerte des betrachteten Merkmals X, so ist die **Varianz** der Messreihe, die wohl das am häufigsten verwandte Streuungsmaß ist, gerade

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{2.2}$$

Bei Verwendung dieses quadratischen Abstandsmaßes wird speziell das arithmetische Mittel $\bar{x}$ als Bezugsgröße gewählt, da es die **Minimumeigenschaft des arithmetischen Mittels**

$$\sum_{i=1}^{n} (x_i - \bar{x})^2 \leq \sum_{i=1}^{n} (x_i - c)^2 \tag{2.3}$$

für jede beliebige Zahl c besitzt [HEK05].

Die **Standardabweichung**

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.4}$$

ist gerade die Wurzel aus der Varianz s^2 einer Messreihe. Sie hat gegenüber der Varianz den Vorteil, dass sie die gleiche Dimension wie die Messwerte selbst und damit wie $\bar{x}$ besitzt. Varianz und Standardabweichung benutzen zwar als Bezugspunkt das arithmetische Mittel $\bar{x}$, d.h. sie messen einen mittleren Abstand von $\bar{x}$, werden jedoch nicht ins Verhältnis zu $\bar{x}$ gesetzt. Ein von $\bar{x}$ bereinigtes Streuungsmaß, das ohne direkte Nennung von $\bar{x}$ interpretiert werden kann, ist der **Variationskoeffizient**

$$v = \frac{s}{\bar{x}} \tag{2.5}$$

Der Variationskoeffizient misst das Verhältnis von Standardabweichung und arithmetischem Mittel. Er ist nur für solche Messreihen ein sinnvolles Maß, die ausschließlich positive Messwerte beinhalten und eignet sich natürlich insbesondere zum Vergleich der Streuung verschiedener Messreihen [HEK05].

Ein **Konfidenzintervall** für das Mittel, abgeleitet aus den entnommenen Messwerten einer zugrundeliegenden Grundgesamtheit, quantifiziert den Bereich an Messwerten, die mit einer gegebenen Wahrscheinlichkeit, das tatsächliche Mittel der Grundgesamtheit beinhalten. Für die Berechnung des Konfidenzintervalls gilt es zwei Fälle, abhängig von der Anzahl entnommener Messwerte der Grundgesamtheit zu unterscheiden:

1. Die Zahl der Messwerte n ist groß (typischerweise $n \geq 30$), und
2. die Zahl der Messwerte n ist klein (typischerweise $n < 30$).

An dieser Stelle wird nur die Berechnung des Konfidenzintervalls für $n \geq 30$ beschreiben, da die in dieser Diplomarbeit vorgestellte Methode immer mehr als 30 Messwerte sammelt. Für $n < 30$ wird [GBE07] zum Weiterlesen empfohlen. Die Berechnung eines Konfidenzintervalls erfordert, dass eine Reihe von Messungen $x_i, 1 \leq i \leq n$ einer Grundgesamtheit mit Mittel μ und Varianz σ^2 vorliegt. Das Mittel dieser Messungen $\bar{x}$ berechnet sich als

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \quad . \tag{2.6}$$

Im folgenden wird das tatsächliche Mittel μ durch das Mittel $\bar{x}$ der Messungen abgeschätzt und ein Intervall $[c_1, c_2]$ mit Werten um $\bar{x}$ berechnet, die das Konfidenzintervall zu einer gegebenen Wahrscheinlichkeit (Konfidenzniveau) definieren. Das Konfidenzintervall $[c_1, c_2]$ ist so definiert, dass die Wahrscheinlichkeit, das sich μ zwischen c_1 und c_2 befindet gerade $1 - \alpha$ entspricht, wobei α das Signifikanzniveau und $(1 - \alpha)$ das Konfidenzniveau bezeichnet. Die Berechnung des Konfidenzintervalls baut auf dem zentralen Grenzwertsatz auf. Der zentrale Grenzwertsatz besagt, dass für große Werte von n (typischerweise $n \geq 30$), $\bar{x}$ näherungsweise normalverteilt mit Mittel μ und Standardabweichung $\sigma/\sqrt{n}$ ist. Voraussetzung dafür ist, dass die Messwerte $x_i, 1 \leq i \leq n$ unabhängig sind und der gleichen Grundgesamtheit mit Mittel μ und endlicher Standardabweichung σ entstammen. Da das Signifikanzniveau α a priori gewählt wird, muss c_1 und c_2 so bestimmt werden, sodass $Pr[c_1 \leq \mu \leq c_2] = 1 - \alpha$ gilt. Gewöhnlich werden c_1 und c_2 so gewählt, dass sie ein symmetrisches Intervall um $\bar{x}$ bilden, wie beispielsweise $Pr[\mu < c_1] = Pr[\mu > c_2] = \alpha/2$. Durch Anwendung des zentralen Grenzwertsatzes ergibt sich

$$c_1 = \bar{x} - z_{1-\alpha/2}\frac{s}{\sqrt{n}} \tag{2.7}$$

$$c_2 = \bar{x} + z_{1-\alpha/2}\frac{s}{\sqrt{n}} \quad , \tag{2.8}$$

mit Mittel $\bar{x}$, der Anzahl an Messwerten n und der Standardabweichung s, die sich berechnet aus:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \quad . \tag{2.9}$$

Der Wert von $z_{1-\alpha/2}$ ist so definiert, dass eine normalverteilte Zufallsvariable Z, mit Mittel $\mu = 0$ und Standardabweichung $\sigma^2 = 1$ folgende Eigenschaft erfüllt:

$$Pr[Z \leq z_{1-\alpha/2}] = 1 - \alpha/2 \quad . \tag{2.10}$$

Der Wert von $z_{1-\alpha/2}$ kann Tabellen entnommen werden. Ein Konfidenzintervall mit einem Konfidenzniveau von 90% bedeutet, es gibt eine Wahrscheinlickeit von 90%, dass sich das tatsächliche Mittel μ der zugrundeliegenden Grundgesamtheit innerhalb dieses Konfidenzintervalls befindet. Eine Erhöhung des Konfidenzniveaus auf 95% bedeutet, dass die Wahrscheinlichkeit erhöht wird, dass das tatsächliche Mittel innerhalb des Konfidenzintervalls liegt. Da die Messwerte nicht geändert werden, besteht die einzige Möglichkeit die Wahrscheinlichkeit zu erhöhen, dass sich das Mittel innerhalb des neuen Konfidenzintervalls befindet darin, das Intervall zuverbreitern. In Konsequenz bedeutet dies, dass ein Konfidenzintervall mit 95% breiter sein wird, als ein Konfidenzintervall mit 90% genauso wie ein Konfidenzintervall mit 99% breiter sein wird, als ein Konfidenzintervall mit 95% [GBE07].

Ist $x_{(1)} \leq \ldots \leq x_{(n)}$ die geordnete Messreihe zu den Messwerten $x_1, \ldots, x_n$ so bezeichnet man als **α-Quantil** $\tilde{x}_\alpha (0 < \alpha < 1)$ dieser Reihe gerade den Wert

$$\tilde{x}_\alpha = \begin{cases} x_{(k)} & \text{falls } n * \alpha \text{ keine ganze Zahl ist} \\ \frac{1}{2} * (x_{(k)} + x_{(k+1)}) & \text{falls } n * \alpha \text{ eine ganze Zahl ist} \end{cases} \tag{2.11}$$

Das α-Quantil $\tilde{x}_\alpha$ wird für ganze Zahlen $n*\alpha$ so gewählt, dass $n*\alpha$ Messwerte kleiner oder gleich und $n(1-\alpha)$ Werte größer oder gleich $\tilde{x}_\alpha$ sind. Ist $n*a$ keine ganze Zahl, so soll für das α-Quantil gelten, dass $k-1$ Werte kleiner und $n-k$ Werte größer als $\tilde{x}_\alpha$ sind, wobei k die auf $n*\alpha$ folgende ganze Zahl ist. Der Median $\tilde{x}_{0,5}$ ist ein spezielles α-Quantil, nämlich gerade das 0,5-Quantil der Messreihe. Weitere Quantile, die einen speziellen Namen tragen sind das 0,25-Quantil (unteres Quartil) und das 0,75-Quantil (oberes Quartil) [HEK05].

Eine Messreihe kann durch eine Kastenzeichnung (engl. box plot) in eine übersichtliche graphische Darstellung gebracht werden. In dieser Zeichnung sind Minimum, 0,25-Quantil, Median, 0,75-Quantil und Maximum als 5-Punkt-Zusammenfassung für ein Merkmal angegeben. Abbildung 2.14 veranschaulicht diesen Sachverhalt grafisch. Darin enthalten sind mit dem Median ein Lagemaß und implizit auch zwei Streuungsmaße nämlich Spannweite und Quartilsabstand [HHN11]. Der Quartilsabstand ist dabei definiert als Differenz $\tilde{x}_{0,75} - \tilde{x}_{0,25}$.

2.5.2. Varianzanalyse (ANOVA)

Um den Einfluss verschiedener Alternativen beurteilen zu können, bedarf es strenger statistischer Verfahren. In diesem Abschnitt wird nach [GBE07] die Varianzanalyse mit angehängtem Tuckey-Test als solches Verfahren vorgestellt, um signifikante Unterschiede zwischen den Messreihen zu erkennen.

Ein einfacher Ansatz um zwei Alternativen miteinander zu vergleichen ist zu bestimmen, ob sich die Konfidenzintervalle der beiden Messreihen überlappen. Wenn sie sich überlappen, kann nicht gefolgert werden, dass die beobachteten Unterschiede zwischen den Mitteln nicht

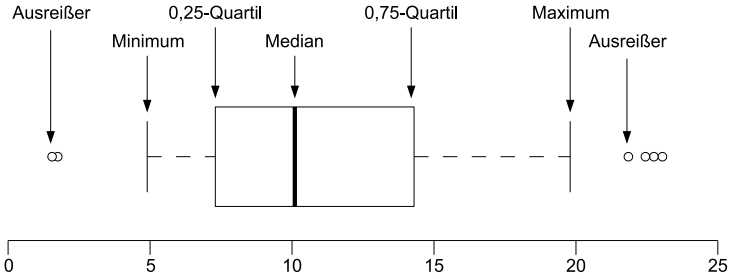


Abbildung 2.14.: Beispiel einer Kasten-Zeichnung

durch zufällige Schwankungen in den Messwerten begründet sind. Es ist möglich, dass die beobachteten Unterschiede zwischen den Mitteln von zufälligen Einflüssen stammen. Wenn die Konfidenzintervalle sich nicht überlappen, wird gefolgert, dass es keinen Beweis gibt, um zu behaupten, dass es keinen statistisch signifikanten Unterschied gibt. Es gibt immer noch eine Wahrscheinlichkeit α, dass die beobachtete Differenz in den Messungen einfach durch zufällige Fehler in den Messungen verursacht werden. Deshalb kann also nicht mit hunderprozentiger Sicherheit gesagt werden, das es eine tatsächliche Differenz zwischen den verglichenen Alternativen gibt. Werden beispielsweise zwei Alternativen mit n_1 Messungen der ersten Alternative und n_2 Messungen der zweiten Alternative verglichen, dann werden als erstes die Mittel $\bar{x}_1$ und $\bar{x}_2$ bestimmt, sowie die Standardabweichungen s_1 und s_2. Als nächstes wird die Differenz der Mittel als $\bar{x} = \bar{x}_1 - \bar{x}_2$ berechnet. Die Standardabweichung s_x der Differenz der Mittel berechnet sich aus

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \quad . \tag{2.12}$$

Das Konfidenzintervall für die Differenz der Mittel ist gegeben durch

$$c_1 = \bar{x} - z_{1-\alpha/2} s_x \tag{2.13}$$

$$c_2 = \bar{x} + z_{1-\alpha/2} s_x \quad . \tag{2.14}$$

Wenn das Konfidenzintervall die Null enthält, wird gefolgert, dass zum gewählten Konfidenzniveau kein statistisch signifikanter Unterschied zwischen den beiden Alternativen besteht. Diese Behauptung gilt, wenn die Anzahl der Messwerte aus beiden Messreihen groß ist, beispielsweise $n_1 \geq 30$ und $n_2 \geq 30$ [GBE07]. Für den Fall, dass die Zahl der Messwerte zumindest in einer der beiden Alternativen kleiner als 30 ist, kann nicht länger angenommen werden, dass die Mittel normalverteilt sind [GBE07]. Auf diesen Fall wird an dieser Stelle nicht weiter eingegangen und auf [GBE07] zum Weiterlesen verwiesen, da die Methode dieser Diplomarbeit immer mehr als 30 Messungen tätigt.

Eine allgemeinere und robustere Methode um Alternativen miteinander zu vergleichen ist die Varianzanalyse. Die Varianzanalyse ist ein statistisches Verfahren um mehrere Alternativen miteinander zu vergleichen. Sie beantwortet die Frage, ob die verschiedenen Stufen eines Einflussfaktors statistisch signifikant unterschiedliche Wirkungen auf das interessierende Merkmal zeigen und quantifiziert ihre Effekte [HEK05]. Konkret werden in diesem Kontext als Alternativen die Messreihen der unterschiedlichen Versionen aus dem Versionskontrollsystem 2.3 und als Einflussfaktor die Version selbst betrachtet. Die Idee der Varianzanalyse besteht darin die Abweichung in den Messwerten in die Abweichung zwischen den Messreihen aufgrund realer Unterschiede und zufällige Fehler und in die Abweichung innerhalb einer Messreihe aufgrund von Messfehlern zu zerlegen. Der Unterschied zwischen den Messreihen ist auf reale Unterschiede zurückzuführen, wenn die Abweichung

	Alternativen						
Messung	1	2	$\cdots$	i	$\cdots$	p	Gesamtmittel
1	y_{11}	y_{21}	$\cdots$	y_{i1}	$\cdots$	y_{p1}	
2	y_{12}	y_{22}	$\cdots$	y_{i2}	$\cdots$	y_{p2}	
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$	
j	y_{1j}	y_{2j}	$\cdots$	y_{ij}	$\cdots$	y_{pj}	
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$	
n	y_{1n}	y_{2n}	$\cdots$	y_{in}	$\cdots$	y_{pn}	
Spaltenmittel	$\bar{y}_{1.}$	$\bar{y}_{2.}$	$\cdots$	$\bar{y}_{i.}$	$\cdots$	$\bar{y}_{p.}$	$\bar{y}_{..}$
Effekte	α_1	α_2	$\cdots$	α_i	$\cdots$	α_p	

Tabelle 2.3.: Aufbereitete Darstellung der Messwerte für die Varianzanalyse (basierend auf [HEK05] und [Kou10])

zwischen den Messreihen statistisch größer ist, als die Abweichung aufgrund von Messfehlern [Kou10]. Die nachfolgende Erklärung stammt aus [HEK05], [GBE07] und [Kou10].

Die Details der Varianzanalyse lassen sich am einfachsten erklären, wenn dazu eine aufbereitete Darstellung der Messwerte zur Hilfe genommen wird, wie in Tabelle 2.3. In der Tabelle sind p unabhängige Messreihen der Länge n und die Messwerte y_{ij} für $i = 1, \ldots, p$ und $j = 1, \ldots, n$ übersichtlich dargestellt. Für die Varianzanalyse wird die Annahme getroffen, dass die Fehler der Messreihen unabhängig und normalverteilt sind. Die Messwerte $y_{i1}, \ldots, y_{in}$ der i-ten Messreihe entstammen einer Normalverteilung $N(\mu_i, \sigma^2)$ mit unbekanntem Erwartungswert μ_i und unbekannter, aber für alle Messreihen gleicher Varianz $\sigma^2 > 0$.

Man berechnet für jede der p Messreihen den Mittelwert, das sogenannte **Spaltenmittel**

$$\bar{y}_{i.} = \frac{1}{n} \sum_{j=1}^{n} y_{ij} \tag{2.15}$$

der natürlich auch ein Schätze für den Erwartungswert μ_i ist, sowie das **Gesamtmittel**

$$\bar{y}_{..} = \frac{1}{(p * n)} \sum_{i=1}^{p} \sum_{j=1}^{n} y_{ij} \quad . \tag{2.16}$$

Anschließend lässt sich jeder Messwert y_{ij} durch das Spaltenmittel $\bar{y}_{i.}$ und dem Fehler e_{ij} als

$$y_{ij} = \bar{y}_{i.} + e_{ij} \tag{2.17}$$

ausdrücken. Der Fehler e_{ij} ist dabei die Abweichung des gemessenen Wertes y_{ij} gegenüber des zur Messreihe gehörenden Spaltenmittels $\bar{y}_{i.}$. Wobei sich das Spaltenmittel wiederum durch das Gesamtmittel $\bar{y}_{..}$ und seiner Abweichung vom Gesamtmittel α_i als

$$\bar{y}_{i.} = \bar{y}_{..} + \alpha_i \tag{2.18}$$

ausrücken lässt. Die Abweichung vom Gesamtmittel α_i wird auch als Effekt der Messreihe i bezeichnet. Durch Gleichung 2.17 und 2.18 lässt sich ein Messwert y_{ij} schließlich durch das Gesamtmittel $\bar{y}_{..}$, den Effekt α_i der Messreihe i und den Fehler e_{ij} für die j-te Messung dieser Messreihe ausdrücken als

$$y_{ij} = \bar{y}_{..} + \alpha_i + e_{ij} \quad . \tag{2.19}$$

Gleichung 2.19 spiegelt dabei die zu Beginn genannte grundlegende Idee der Varianzanalyse wieder. Im nächsten Schritt werden die Quadratsummen berechnet. Die Quadratsumme

$$SSE = \sum_{i=1}^{p}\sum_{j=1}^{n}(e_{ij})^2 = \sum_{i=1}^{p}\sum_{j=1}^{n}(y_{ij} - \bar{y}_{i.})^2 \tag{2.20}$$

bezieht sich auf die Variation innerhalb der Messreihen und dient der Charakterisierung der Abweichung aufgrund von zufälligen Fehlern. Die Quadratsumme

$$SSA = n\sum_{i=1}^{p}(\alpha_i)^2 = n\sum_{i=1}^{p}(\bar{y}_{i.} - \bar{y}_{..})^2 \tag{2.21}$$

charakterisiert die Variation aufgrund der Effekte. Die letzte Quadratsumme bezieht sich auf die Variation zwischen den Messwerten und dem Gesamtmittel:

$$SST = \sum_{i=1}^{p}\sum_{j=1}^{n}(y_{ij} - \bar{y}_{..})^2 \tag{2.22}$$

Es kann dann gezeigt werden, das

$$SST = SSA + SSE \quad \text{ist.} \tag{2.23}$$

Im Anschluss werden die Reihenmittel mit Hilfe des F-Tests auf signifikante Unterschiede geprüft. Der F-Test analysiert die verschiedenen Varianzen und schließt auf einen signifikanten Unterschied zwischen den Messreihen, wenn die Variation zwischen den Messreihen 'wesentlich größer' ist als die innerhalb der Reihen. Zum vorgegebenen Konfidenzniveau γ wird die Nullhypothese

$$H_0 : \mu_1 = \mu_2 = \ldots = \mu_p \tag{2.24}$$

verworfen und genau dann auf einen signifikanten Unterschied zwischen den Reihenmitteln geschlossen, wenn für den Quotient der beiden mittleren Quadratsummen MSA und MSE

$$F = \frac{MSA}{MSE} > F_{p-1,p(n-1);1-\gamma} \tag{2.25}$$

gilt. Der F-Test nimmt an, dass die Nullhypothese H_0 einer F-Verteilung mit $p-1$ und $p(n-1)$ Freiheitsgraden genügt, wodurch deren $(1-\gamma)$-Quantil zur Festlegung des 'wesentlich größer' herangezogen werden kann. Die Freiheitsgrade und die Quantile $F_{p-1,p(n-1);1-\gamma}$ können Tabellen entnommen werden wie sie beispielsweise im Anhang von [HEK05] zu finden sind. Nun werden nur noch die mittleren Quadratsummen von SSE und SSA benötigt, die durch folgende Gleichungen gegeben sind:

$$MSA = \frac{SSA}{p-1} \tag{2.26}$$

$$MSE = \frac{SSE}{p(n-1)} \quad . \tag{2.27}$$

Wird die Nullhypothese H_0 durch den F-Test verworfen, ist ein weiterer Test notwendig, um herauszufinden, welche Reihenmittel sich unterscheiden [HEK05]. Zu diesem Zweck wird normalerweise der Tuckey-Test angewendet, welcher im nachfolgenden Abschnitt näher beschrieben wird. Der Vorteil des Tukey-Tests gegenüber einfacheren Ansätzen wie paarweisen t-Tests für den Vergleich der Reihenmittel ist, dass es die Wahrscheinlichkeit limitiert, eine falsche Schlussfolgerung zu machen, in dem Fall, dass es keine statistisch signifikante Unterschiede zwischen allen Reihenmitteln gibt und im Falle, dass die meisten Reihenmittel gleich sind und sich aber nur ein oder zwei unterscheiden [GBE07].

2.5.3. Tuckey-HSD-Test

Der Tuckey-Test prüft, ob sich die Mittelwerte von p Messreihen statistisch signifikant unterscheiden. Er berechnet für jedes Paar $(i, i'), i < i'$, von Messreihen die **Differenz der Reihenmittel**

$$\bar{y}_{i.} - \bar{y}_{i'.} \tag{2.28}$$

und erkennt auf zum Konfidenzniveau γ signifikante Unterschiede der Mittelwerte der Reihen i und i', falls die Differenz betraglich größer ist als die Grenzdifferenz. Bei gleichen Reihenlängen $n = n_i$ berechnet sich die **Grenzdifferenz** als

$$HSD_\gamma = \sqrt{\frac{MSE}{n} * q_{p,p(n-1);1-\gamma}} \tag{2.29}$$

zum Niveau γ [HEK05]. Die studentisierte Spannweite $q_{p,p(n-1)}$ kann dabei wieder Tabellen entnommen werden, wie sie im Anhang von [HEK05] zu finden sind. Ist die Differenz der Reihenmittel $\bar{y}_{i.} - \bar{y}_{i'.}$ also größer als die Grenzdifferenz HSD_γ, gelten die Messreihen i und i' als signifikant unterschiedlich zum Konfidenzniveau γ.

Die Varianzanalyse erlaubt zusammenfassend die Variation der einzelnen Versionsstände der Versionskontrollsysteme als Eingabeparameter. Soll beispielsweise die Leistung für fünf Versionsstände für einen gegebenen Performance-Unit-Test auf einer gegebenen Hardware-Plattform verglichen werden, so ist der zu variierende Faktor die Version und die fünf Alternativen die fünf Versionsstände [GBE07].

3. Verwandte Arbeiten

Der Ansatz die Leistung zweier verschiedener Versionen desselben Software-Systems, mit denselben Eingaben auf der gleichen Plattform zu vergleichen ist relativ neu. Bisher stand im Fokus früherer Arbeiten die Identifizierung von Unterschieden in der Leistung zwischen zwei Ausführungen desselben Programms mittels unterschiedlicher Eingaben oder unterliegenden Plattformen [MK09]. In diesem Kapitel werden verwandte Arbeiten vorstellt.

3.1. Regressionserkennung

Regressionsleistungstests erkennen Einbrüche in der Leistung eines Systems unter Last. Solche Regressionen beziehen sich auf Situationen, in der die Leistung der Software im Vergleich zu früheren Versionen nachlässt, obwohl die neue Version sich genauso verhält wie die alte Version. Diese Tests sind oft Bestandteil traditioneller Regressionstests, um Leistungsengpässe und Entwurfsprobleme frühzeitig aufzudecken. Foo et al. stellen in [FJA+10] einen automatischen Ansatz vor, der Leistungssignaturen durch Aufzeichnung der Korrelationen zwischen Metriken ableitet. Verletzungen dieser Leistungssignaturen werden als potentielle Leistungsprobleme gebrandmarkt. Der Ansatz vergleicht automatisch einen neuen Test mit einer Menge erwarteter und korrelierter Metriken, die aus früheren Tests extrahiert wurden. Ein Bericht identifiziert automatisch die Metriken des neuen Tests, die eine signifikante Abweichung von früheren Tests zeigen. Die Autoren überlassen es dann dem Analysten diese Metriken darauf zu prüfen, ob sie tatsächlich Leistungsprobleme darstellen. Die vier wesentlichen Phasen des Ansatzes sind die Normalisierung der Metriken, Diskretisierung der Metriken, Erkennung von Abweichungen der Leistungssignaturen und Erstellung des Berichts. Die Normalisierung der Metriken dient dem Entfernen von Unregelmäßigkeiten. Dabei werden die Daten der Metrik extrahiert, die mit der erwarteten Dauer des Testlaufes übereinstimmen. Die eingesetzten Verfahren des maschinellen Lernens erfordern die Diskretisierung der historischen und neuen Testdaten. Dazu wird die Zeit in gleichgroße Intervalle aufgeteilt. Jedes Interval wird durch einen Vektor der Mediane repräsentiert. Jedem Element im Vektor wird dabei eine von drei Kategorien zugeordnet. In der dritten Phase werden die Leistungssignaturen extrahiert, indem häufig beobachtete Korrelationen zwischen den Metriken der historischen Daten aufgezeichnet werden. Davon leiten die Autoren Regeln für die Daten des neuen Tests ab und brandmarken Metriken in diesen Regeln, die signifikante Abweichungen im Vertrauensbereich zeigen. In der letzten Phase wird der Bericht mit allen gebrandmarkten Metriken erstellt. Der Bericht hebt die Regeln hervor, welche durch die Metriken verletzt werden [FJA+10].

Die Idee der automatischen, wiederholbaren, umfangreichen und in den Entwicklungsprozess integrierten Leistungstests wird bereits in [BKT05] geschildert. Bulej et al. schlagen für Regressionstests von Software-Systemen einen sogennaten Regressions-Benchmark zur Erkennung von Performance-Problemen vor. Dazu muss der Regressions-Benchmark in den Entwicklungsprozess integriert werden und sowohl vollständig automatisiert, umfassend als auch wiederholbar sein. Dabei wird zwischen einfachen und komplexen Benchmarks unterschieden. Bei einem einfachen Benchmark wird eine isolierte Funktionaltät unter einer künstlich erzeugten Last getestet. Durch die geringe Laufzeit gibt es wenig Raum für eine Beeinflussung, wodurch präzise Messungen erzielt werden. Ein komplexer Benchmark testet eine Reihe der bereitgestellten Funktionalität unter realer Last. Da einige Funktionalität dabei nebenläufig ausgeführt wird, können Seiteneffekte komplexer Interaktionen die Messungen beeinflussen. Die Messungen einfacher Benchmarks werden mit einem t-Test verglichen wohingegen bei komplexen Benchmarks eine Cluster-Analyse angewendet wird [BKT05].

3.2. Problemdiagnose

Mostafa und Krintz präsentieren in [MK09] ein Performance-Aware Revision Control Support (PARCS) genanntes Rahmenwerk, das Entwicklern Rückmeldung darüber geben soll, wie sich eine aufgegebene Änderung auf das Verhalten und die Leistungsfähigkeit einer Anwendung auswirkt. Entwicklern soll geholfen werden zu verstehen, wie lokale inkrementelle Modifikationen die Leistungsfähigkeit beeinflussen. Wird eine neue Version des Quellcodes an das VKS übergeben, führt PARCS das Programm mittels Testeingaben aus. Der Ansatz von Mostafa und Krintz baut auf der Profilierung des Aufrufkontextbaums (CCT) und der Differenzierung der Leistung auf. Der Aufrufkontext einer Methode ist dabei eine Liste aller Methoden die zu irgendeinem bestimmten Zeitpunkt im Kellerspeicher (engl. stack) ansässig sind. Der Aufrufkontext jedes dynamischen Methodenaufrufs, der während der Ausführung eines Programms auftritt, wird durch den CCT aufgezeichnet. Alle Ausführen derselben Methode, die aus demselben Aufrufkontext resultieren werden dabei in einen einzigen Knoten aggregiert. PARCS erweitert das Konzept des CCT dahingehend, dass wenn innerhalb einer Methode Y die Methode X mehr als einmal aufgerufen wird und somit denselben Aufrufkontext besitzt, zwischen den Aufrufstellen in Form der Codezeile unterschieden wird. Er vergleicht dabei zwei verschiedene Versionen desselben Software-Systems, mit denselben Eingaben auf der gleichen Plattform. PARCS instrumentiert jeden Methodenein- und austritt des Software-Systems und eine modifizierte JVM generiert einen CCT zur Laufzeit, während das Software-System im Hintergrund ausgeführt wird. Der CCT wird von PARCS mit Metriken wie der gesamten und mittleren Ausführungezeit und der Aufrufzahl einer Methode annotiert. Außerdem führt PARCS einen statischen Bytecode-Vergleich auf den Versionen aus, um Änderungen auf Methodenebene zu extrahieren. Durch den Bytecode-Vergleich wird eine Liste mit allen gelöschten, modifizierten und umbenannten Methoden erstellt. Die Arbeit auf Bytecode-Ebene wird als vorteilhaft beschrieben, da die meisten Änderungen wie die Verschiebung der Deklaration von Variablen innerhalb einer Methode, Umbenennung von Variablen oder die Ersetzung einer For-Schleife durch eine While-Schleife auf Bytecode-Ebene nicht reflektiert werden und somit automatisch ignoriert werden. PARCS vergleicht zwei CCT miteinander um topologische Unterschiede zu identifizieren. Diese Unterschiede werden anschließend in Kategorien klassifiziert, die eine mögliche Ursache hinter der Differenz in der Leistung kennzeichen. Alle Teilbäume die an Knoten beginnen, von denen Differenzen ausgehen, werden herausgeschnitten. Die resultierenden CCTs sind dann identisch. Diese Bäume werden dann auf Differenzen in den Metriken untersucht, mit denen die Knoten annotiert sind. Dadurch werden Paare von Knoten identifiziert, die eine signifikante Differenz in den Metriken aufzeigen. Das sind Differenzen, die größer sind als diejenigen eines nicht deterministischen

Effektes. PARCS fügt jede topologische und metrische Differenz zu einer Menge von wahrscheinlichen Code-Änderungen hinzu und meldet dies an den Entwickler zurück [MK09].

Ehlers et al. schlagen in [EvHWH11] einen Ansatz zur Lokalisierung von Leistungsanomalien in Software-Systemen durch den Gebrauch von selbst-adaptiver Überwachung vor. Die selbst-adaptive, regelbasierte Überwachung ändert autonom die Reichweite der Überwachung dynamisch zur Laufzeit. Die Sensoren sind dabei überall im System verteilt, so dass alle Ausführungen von Operationen beobachtet werden können. Da es nach [EvHWH11] nicht sinnvoll ist, alle Sensoren zur gleichen Zeit zu aktivieren, werden als aktive Sensoren zunächst meist diejenigen an den öffentlichen Schnittstellen gewählt. Zeigen die Messdaten eines Sensors auf dieser Ebene dann eine Anomalie, kann durch die adaptive Instrumentierung die Sensorik der darunterliegen Schicht aktiviert werden. In ihrem Ansatz konzentrieren sich die Autoren auf die Lokalisierung der Ursachen von Leistungsanomalien, deren Auswirkungen das normale Verhalten eines Systems verändern, das der Anwender wahrnimmt. Die vorgeschlagene adaptive Überwachung ist als Erweiterung des Kieker Rahmenwerks implementiert, das in Abschnitt 2.2.2 erklärt wird und hier der Überwachung und Analyse des Laufzeitverhaltens von Software-Systemen dient. Ehlers et al. verwenden die Object Constraint Language (OCL), um die Regeln der Übwachung zu spezifizieren. Die Regeln referenzieren Leistungsattribute, insbesondere eine Anomaliebewertung als Ansprechschwelle, die ihre Werte während der Laufzeit ändert. Die Adaption basiert auf der laufenden Evaluation der Überwachungsregeln. Die Anomaliebwertung basiert dabei auf statistischen Verfahren zur Zeitreihenvorhersage, einem Hypothesentest und einer Berechnungsfunktion. Der Hypothesentest testet dabei, ob der vorhergesagte Wert als Mittel der Grundgesamtheit akzeptiert werden kann oder nicht. Die Berechnungsfunktion liefert einen Wert, der Auskunft darüber gibt, ob der Messwert der Erwartung entspricht oder nicht [EvHWH11].

Die Bedeutung und die Komplexität beim Testen von Software hat zu einer großen Anzahl verschiedener Ansätze geführt, wie dem Ableiten von Testfällen aus Modellen und Quellcode, verwenden unterschiedlicher Testziele wie dem Überdeckungskriterium und vielen darunterbefindlichen Techniken und Algorithmen [FZ11]. Fraser und Zeller fassen in [FZ11] klassische White-Box-Tests ins Auge, bei denen Testfälle aus dem Software-System abgeleitet werden. Dabei wird keine Annahme über die Existenz einer formalen Spezifikation gemacht. Der vorgestellte Ansatz nimmt eine maschinell erstellte Methodensequenz und erforscht sowohl mögliche Vorbedingungen, als auch implizierte Nachbedingungen. Der resultierende parametrisierbare Unit-Test beinhaltet nur eine kleine Menge relevanter Vor- und Nachbedingungen und nur Anweisungen, die mit dem aktuellen Test in Verbindung stehen. Ihr Ansatz baut auf zwei Ideen auf: Zum einen wird zwischen Test-Quellcode und Testeingaben unterschieden, was ihnen erlaubt große Teile generierten Quellcodes zur Vorbereitung von Eingaben zu verwerfen und durch symbolische Parameter zu ersetzen. Die andere Idee ist, dass durch Variation wichtiges Verhalten von unrelevantem Verhalten unterschieden werden kann. Dabei verändern sie einige Nachbedingungen durch Mutation, in dem Defekte gesät werden. Ihre Annahme dabei ist, dass nur diejenigen Teile der Nachbedingung relevant sind, die von den gesäten Defekten betroffen sind. Also umso mehr Fehler eine Nachbedingung auffängt, desto relevanter ist sie. Die gleiche Idee wenden Fraser und Zeller auch auf die Eingaben an. Wenn eine Änderung der Eingabe keine Auswirkungen hat, dann scheint dieser bestimmte Aspekt der Eingabe, der sich geändert hat, irrelevant für den Test zu sein. Ähnliche Techniken werden verwendet zur Verallgemeinerung und Auswahl von Vorbedingungen. Ihr Ansatz basiert auf existierendem Bytecode und beginnt mit einer automatisch generierten Methodensequenz. Am Ende des Prozesses erhalten sie einen parametrisierten Unit-Test, der nur aus Testanweisungen, den relevanten Vorbedingungen der Eingaben und den effektiven Nachbedingungen besteht [FZ11].

Das schnelle Auffinden der Ursache eines Software-Fehlers oder Leistungsproblems in pro-

duktiven Umgebungen ist eine wichtige Fähigkeit. Ungeachtet ihrer Bedeutung, ist die Aufgabe der Problemdiagnose noch immer wenig automatisiert, während sie erheblich Zeit und Aufwand von hochqualifizierten Analysten bedarf. Mirgorodskiy und Miller adressieren dieses Problem in [MM08]. Der Ansatz macht sich zunutze, dass Probleme oft mit seltenen Ausführungspfaden oder mit Pfaden, deren Eigenschaften von der Norm abweichen, korrespondieren. Durch das Auffinden der Stelle an der die Ausführung von der Norm abweicht, können sie womöglich das Problem lokalisieren. Ihr Ansatz überwacht das System dabei auf feingranularer Ebene, erkennt und instrumentiert kommunizierende Prozesse während der Ausführung und zeichnet dabei den Ablauf in verteilten Umgebungen auf Methodenebene auf. Die Ablaufanalyse vergleicht automatisch zwei Abläufe miteinander um ungewöhnliche Aktivitäten zu identifizieren und zeigt dem Analysten die wahrscheinliche Problemursache auf. Mit Hilfe selbst angetriebener Instrumentierung über Prozesse und Rechner hinweg, wird der Kontrollflussablauf von unmodifizierten und unbekannten Systemen gesammelt. Der Grundstein der selbst angetriebenen Instrumentierung ist ein autonomes Code-Fragment, das als Agent bezeichnet wird und das über ein externes Ereignis in das System injiziert wird. Nach der Injektion startet der Agent mit der Überwachung des Systems, indem er Anweisungen zur Ablaufverfolgung, dem Kontrollfluss vorausgehend, in den Code des Systems einfügt. Dieser Aspekt wird von den Autoren noch dahingehend erweitert, dass sich die Instrumentierung aus einem Prozess heraus in einen anderen Prozess propagieren kann. Dies ermöglicht ihnen kommunizierende Komponenten zu entdecken und verteilte Ablaufverfolgungsprotokolle des Kontrollflusses zu erstellen. Um Problemen mit nebenläufigen Prozessen aus dem Weg zu gehen, wird die globale Ablaufverfolgung in eine Sammlung von Ablaufverfolgungen pro Anfrage zerlegt. Für die Analyse der Problemursache mit begrenztem Aufwand konzentrieren Mirgorodskiy und Miller die Analyse auf Unterschiede in der Überdeckung zwischen erfolgreichen und fehlgeschlagenen Abläufen. Sie suchen nach Methoden, die entweder nur im erfolgreichen oder im fehlgeschlagenen Ablauf ausgeführt wurden. Solche Methoden sind mit dem Auftreten des Problems korreliert und deuten oft auf die Problemursache hin [MM08].

3.3. Abgrenzung

Der in dieser Diplomarbeit vorgestellte Ansatz, erfordert im Gegensatz zu [MK09] keine modifizierte Java Virtual Machine (JVM). Dadurch wird eine Unabhänigkeit von der unterliegenden JVM erreicht woraus eine größere Flexibilität resultiert. Außerdem wird keine statische Bytecode-Analyse verwendet. Eine statische Quellcode-Analyse soll in einer späteren Arbeit ermöglichen, die im Versionskontrollsystem gespeicherten Änderungen an einer Klassendatei mit der betreffenden Methode in Verbindung zu bringen. Der für die Problemdiagnose benutzte Aufrufbaum wird erst bei Bedarf und nicht durch die JVM aufgebaut. Durch den Vergleich der Performance unterschiedlicher Software-Versionen wird ein vom Konzept her anderer Ansatz verfolgt als in [EvHWH11] und [MM08]. Die Messungen erfolgen nicht im Produktivbetrieb des Software-Systems. Die Methode zur Erkennung von Performance-Problemen und deren Diagnose verwendet bereits im System enthaltene Unit-Tests. Die Unit-Tests werden nicht wie in [FZ11] automatisch generiert. Durch die Verwendung von Unit-Tests kann die Methode bereits in der frühen Entwicklungsphase eingesetzt werden und benötigt kein lauffähiges Software-System wie in [FJA+10]. Der Ansatz in [FJA+10] ist nicht darauf ausgerichtet die Problemursache zu diagnostizieren.

4. Ansatz

In diesem Kapitel wird der Ansatz zur Regressionserkennung und Problemdiagnose vorgestellt. Ausgehend von den gespeicherten Änderungen im Versionskontrollsystem (VKS) wird aufgezeigt, wie Leistungsregressionen durch systematische Messungen erkannt und die Problemursache diagnostiziert wird. Das Kapitel beginnt mit einem allgemeinen Überblick in Abschnitt 4.1. Die Regressionserkennung wird in Abschnitt 4.3 beschrieben gefolgt von der Beschreibung der Problemdiagnose in Abschnitt 4.4. Die Automatisierung erfolgt über die Integration in den automatischen Erstellungsprozess und ist Gegenstand der Betrachtung in Abschnitt 4.6.

4.1. Allgemeiner Überblick

Das Ziel ist Software-Entwickler bei der Erkennung von Leistungsregressionen zu entlasten und bei der Problemdiagnose zu unterstützen. Die Grundlage dazu bilden die Unit-Tests, das Versionskontrollsystem (VKS), die automatisierte Übersetzung und die kontinuierliche Integration. Über Zusatzprogramme wird das Performance-Unit-Test-Rahmenwerk (PUT-Rahmenwerk) je nach Anwendung entweder in die kontinuierliche Integration oder in die automatisierte Übersetzung integriert. Abbildung 4.1 veranschaulicht die Integration des Rahmenwerkes in den bestehenden Entwicklungsprozess. Wird das PUT-Rahmenwerk ausgeführt, werden alle Unit-Tests instrumentiert und ausgeführt. Dadurch wird für jede Testmethode eines Unit-Tests eine Messreihe generiert. Jede Messreihe wird mit einer Referenz verglichen und auf signifikante Unterschiede geprüft. Bei einem Unit-Test liegt genau dann eine Leistungsregression vor, wenn eine signifikante Verschlechterung der Antwortzeit erkannt wird. Für die Entwickler läuft die Regressionserkennung und Problemdiagnose vollständig automatisiert im Hintergrund ab. Die Automatisierung ist vergleichbar mit der Erstellungsaufgabe bei der kontinuierlichen Integration. Einmal eingerichtet wird kontinuierlich eine Erstellung des Software-Systems durchgeführt. Wird die Erstellung als instabil oder fehlgeschlagen gekennzeichnet werden die Entwickler umgehend darüber informiert. Die Automatisierung sorgt dafür, dass sich Entwickler höherwertigen Aufgaben zuwenden können. Das PUT-Rahmenwerk läuft bei der automatisierten Erstellung, bei der ein Entwickler eine lokale Integration durchführt, bevor er Änderungen ins VKS aufgibt, ebenso wie bei der kontinuierlichen Integration einfach mit. Sobald eine Leistungsregression erkannt und das Problem diagnostiziert ist, wird der Entwickler umgehend darüber informiert. Die Instrumentierung muss deshalb unabhängig vom Entwickler erfolgen. Durch das Hinzufügen der Anweisungen der Instrumentierung zum Software-System wird die

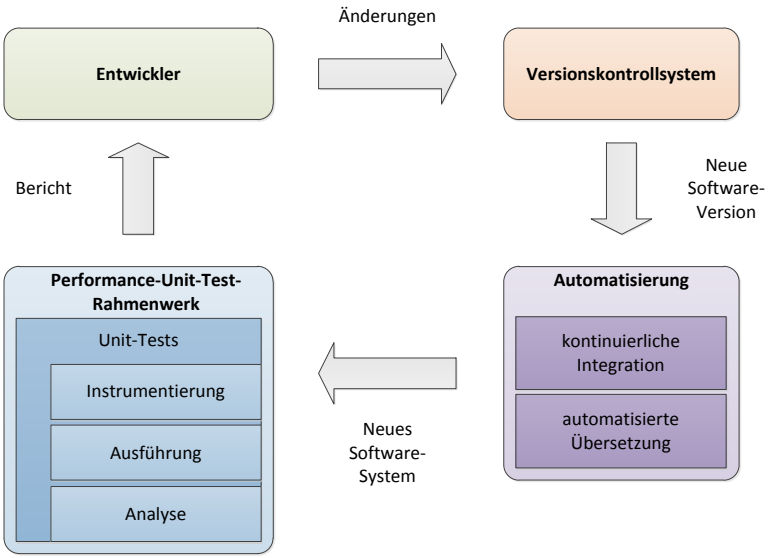


Abbildung 4.1.: Integration des Ansatzes in eine moderne Entwicklungsumgebung

eigentliche Antwortzeit des Software-Systems beeinflusst. Die Instrumentierung darf die Antwortzeit des Software-Systems nur um einen zu vernachlässigbaren Wert erhöhen. Nur so wird sichergestellt, dass authentische Antwortzeiten verglichen werden. Um diesen Anforderungen gerecht zu werden, erfolgt die Instrumentierung des Software-Systems über die adaptive Bytecode-Instrumentierung. Alternativ kann die Antwortzeit auf Grundlage der ausgeführten Bytecode-Instruktionen geschätzt werden. Die Instrumentierung hierfür wird dann durch ByCounter aus Abschnitt 2.2.4 vorgenommen. Mit beiden Instrumentierungsvarianten können zur Laufzeit Methoden und Klassen sowohl instrumentiert als auch deinstrumentiert werden. Bei der Zählung der Bytecode-Instruktionen bleiben die Anweisungen der Instrumentierung ungezählt. Dadurch haben diese Anweisungen keinerlei Einfluss auf die Messwerte. Die Problemdiagnose baut durch systematische Messungen den Aufrufbaum für jede Testmethode auf, bei der eine Leistungsregression erkannt wird. Der Aufrufbaum wird anschließend mit einer Referenz verglichen. Die Analyse identifiziert Methoden die sich zwischen den beiden Aufrufbäumen unterscheiden. Identifizierte Methoden und deren Aufrufpfade im Aufrufbaum werden mit den Änderungen im VKS zu dieser Software-Version korreliert. Um Methoden früherer Software-Versionen vermessen zu können, wird nach Bedarf ein ausführbares Software-System der Software-Version erstellt. Hierzu wird die Ablage des VKS geklont und die benötigte Software-Version in den Arbeitsbereich ausgegeben.

4.2. Herausforderungen

In der Regel konkurrieren auf jedem System nebenläufige Prozesse, um die Ausführung auf der CPU. Daraus resultieren Schwankungen in der Antwortzeit einer Testmethode bei der Ausführung eines Unit-Tests. Die Schwankungen sind für eine Streuung der Messwerte in der Messreihe verantwortlich. Die Schwankungen müssen deshalb auf ein Minimum reduziert werden. Eine Möglichkeit hierfür ist die Verwendung eines dedizierten Systems, auf dem das PUT-Rahmenwerk die Messungen durchführt. Alternativ wird die Antwortzeit aus den ausgeführten Bytecode-Instruktionen geschätzt. Bei großer Streuung der Messwerte aufgrund der Schwankungen, erkennt die Regressionserkennung fälschlicherweise eine Leistungsregression. Wird das PUT-Rahmenwerk in größeren Abständen angewendet,

muss die verantwortliche Software-Version identifiziert werden, in der die Leistungsregression eingeführt wird. Hierzu muss ein Intervall an Software-Versionen systematisch durchsucht werden. Das Intervall wird dabei durch die aktuelle Software-Version und die zuletzt überprüfte Software-Version definiert. Die binäre Suche im Intervall reduziert den Mehraufwand indem Software-Versionen, wie bei der bisect-Funktion von Git [Cha09], von der weiteren Betrachtung ausgeschlossen werden. Die Erkennungsstrategie sucht nach der frühesten Leistungsregression im Intervall. Weitere Erkennungsstrategien in der beispielsweise die größte Regression im Intervall gesucht wird sind Gegenstand zukünftiger Arbeiten. Existieren im betrachteten Intervall mehrere Leistungsregressionen, wird dabei nicht nach der Größe unterschieden.

Die Problemdiagnose identifiziert im Aufrufbaum Methoden, die sich zwischen den beiden Software-Versionen in ihrer Antwortzeit unterscheiden. Umbenannte Klassen oder Methoden stellen eine Herausforderung dar und müssen mit einer gewissen Wahrscheinlichkeit erkannt werden.

Eine weitere Herausforderung ist die Instrumentierung. Es muss eine Methode der Instrumentierung entwickelt werden, die eine signifikante Streuung der Messungen in der Aufwärmphase vermeidet und dadurch die Aufwärmphase vernachlässigbar wird. Wird die Aufwärmphase vernachlässigbar, reicht eine Anzahl > 30 an Ausführungen für die Erhebung der Messdaten einer Testmethode aus.

4.3. Regressionserkennung

Im folgenden wird aufgezeigt, wie aus einer gegebenen Menge von Software-Versionen desselben Software-Systems diejenige Software-Version identifiziert wird, in der als erstes eine Leistungsregression eingeführt wurde. Hierzu wird als erstes der Ansatz und die notwendigen Schritte allgemein beschreiben und anschließend an einem Beispiel veranschaulicht.

Die Herausforderung bei der Regressionserkennung über die Antwortzeit liegt in der Differenzierung tatsächlicher Leistungsregressionen und Schwankungen in den Messwerten. Es wird angenommen, dass für ein gegebenes Intervall $I = [r_{k-n}, r_k], n \geq 1$ an Software-Versionen $r_i \in I$ festgestellt werden soll, ob eine Leistungsregression vorliegt. Hierzu sei eine zeitliche Ordnung der Form $r_k < r_{k-1} < \ldots < r_{k-n}$ so definiert, dass r_k die jüngste und r_{k-n} die älteste Software-Version im Intervall I ist. Um beurteilen zu können, ob eine Leistungsregression im Intervall I vorhanden ist oder nicht, wird die Nullhypothese $H_0 = \mu_{r_{k-n}} = \mu_{r_k}$ geprüft, dass sich die Reihenmittel $\mu_{r_{k-n}}$ und μ_{r_k} jeder Testmethode jedes Unit-Tests nicht signifikant unterscheiden. In dem zur Software-Version r_{k-n} gehörenden Software-System S_{k-n} werden dazu alle Unit-Tests $U \in S_{k-n}$ instrumentiert. Anschließend wird jeder Unit-Test U ausgeführt und die Antwortzeit als Messreihe T_U für jeden ausgeführten Unit-Test U erhoben. Damit der Hypothesentest die Nullhypothese testen kann, werden ebenfalls alle Unit-Tests $U' \in S_k$ instrumentiert und durch die Ausführung aller U' die Messreihen $T_{U'}$ für jeden ausgeführten Unit-Test U' generiert. Wird die Nullhypothese verworfen, wird auf einen signifikanten Unterschied zwischen den Reihenmitteln geschlossen. Sollten sich T_U und $T_{U'}$ signifikant unterscheiden, so wird die Existenz einer Leistungsregression im betrachteten Intervall I angenommen. Gilt für die Intervallgröße $m = n+1 \leq 2$ so gibt es neben r_k und r_{k-1} keine weiteren Software-Versionen im Intervall I. In diesem Fall wurde die Leistungsregression in Version r_k eingeführt. Gilt für die Intervallgröße jedoch $m > 2$, wird systematisch die früheste Software-Version $r_{k-i}, 0 \leq i < n$ bestimmt, in der eine Leistungsregression eingeführt wurde. Dazu wird eine klassische Intervallschachtelung mit einer binären Suche im Intervall I kombiniert und hierfür folgende Schritte ausgeführt:

1. Auswahl der nächsten Software-Version r_i, $\lfloor i \rfloor = m/2$, die sich in der Mitte zwischen der oberen Intervallgrenze r_k und der unteren Intervallgrenze r_{k-n} befindet. Gilt für die Intervallgröße m mod $2 \neq 0$ wird stets die nächst kleinere Ganzzahl gewählt.
2. Instrumentieren aller Unit-Tests $\tilde{U} \in S_i$ und Generierung der Messreihen $T_{\tilde{U}}$ nach deren Ausführung.
3. Mit dem Hypothesentest wird bestimmt, ob bereits im Intervall $I' = [r_{k-n}, r_i]$ eine Leistungsregression vorliegt. Wird die Nullhypothese $H_0 = \mu_{r_{k-n}} = \mu_{r_i}$ verworfen, werden die Software-Versionen zwischen r_{i+1} und r_k nicht weiter berücksichtigt. Deshalb wird als neues Intervall $I = [r_{k-n}, r_i]$ gesetzt, indem r_i als neue obere Grenze angenommen wird. Wird die Nullhypthese nicht verworfen, wird angenommen, dass im Intervall $I' = [r_{k-n}, r_i]$ noch keine Leistungsregression vorliegt und r_i als neue Referenz gesetzt. Dazu wird r_i zur neuen unteren Grenze des Intervalls I. Abhängig vom Hypothesentest resultiert somit entweder das neudefinierte Intervall $I = [r_i, r_k]$ oder $I = [r_{k-n}, r_i]$ als Intervall für die weitere Betrachtung.
4. Gilt für die neue Intervallgröße wieder $m = (k-i)+1 > 2$ bzw. $m = (n-i)+1 > 2$ wird für das neue Intervall I mit Schritt 1 begonnen.
5. Gilt hingegen für die neue Intervallgröße $m = (k-i)+1 \leq 2$ bzw. $m = (n-i)+1 \leq 2$, so ist die obere Grenze dieses Intervalls sodann die erste Software-Version, in der eine Leistungsregression eingeführt wurde.

Nachdem das Intervall $I = [r_{k-n}, r_k]$ überprüft wurde, wird r_k als Referenz für das kommende Intervall $[r_k, r_{k+n}]$ festgelegt. Dieses Intervall wird dann bei der nächsten Ausführung geprüft.

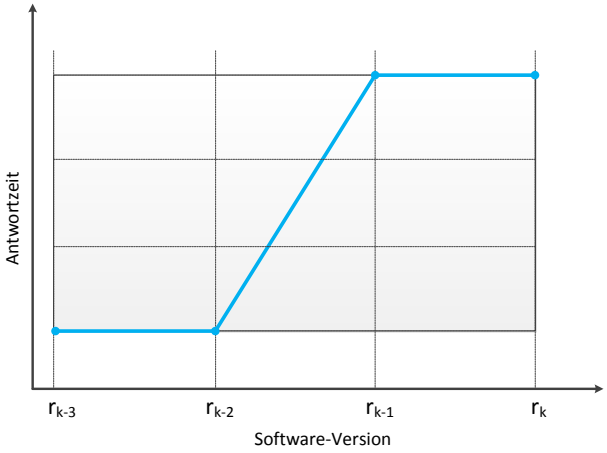


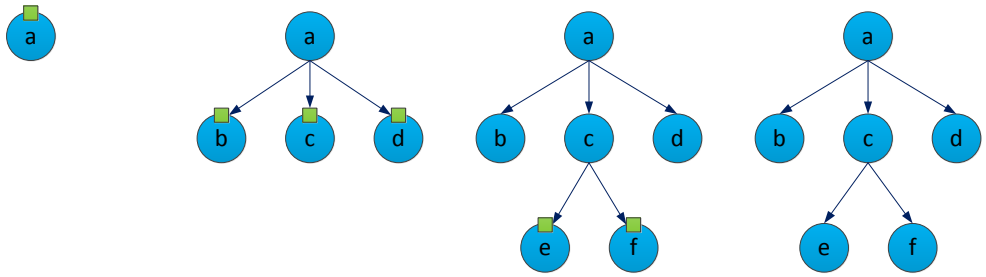
Abbildung 4.2.: Beispiel

Die beschriebene Vorgehensweise zur Regressionserkennung wird an dem Beispiel in Abbildung 4.2 verdeutlicht. Für dieses Beispiel sei ein Intervall I mit vier Software-Versionen $r_k, \ldots, r_{k-3}$ gegeben auf denen eine zeitliche Ordnung $r_k < r_{k-1} < \ldots < r_{k-3}$ so definiert sei, das r_k die jüngste und r_{k-3} die älteste Software-Version im betrachteten Intervall I darstellen. Die Software-Versionen $r_i \in I$ sind zusammen mit der entsprechenden Antwortzeit des zugehörigen Software-Systems S_i in Abbildung 4.2 veranschaulicht. Auf der Abszissenachse sind in diesem Diagramm die Software-Version $r_k, \ldots, r_{k-3}$ und auf der Ordinatenachse die Antwortzeit der jeweiligen Version aufgetragen. Es wird weiter angenommen, dass nur ein Unit-Test $U_i \in S_i$ existiert, dessen Antwortzeit gerade der in Abbildung 4.2 dargestellten Antwortzeit der entsprechenden Software-Version r_i entspricht. Das Ziel ist es nun, r_{k-1} als Software-Version zu identifizieren, da in ihr gemäß Abbildung 4.2 die erste

Leistungsregression im betrachteten Intervall I eingeführt wurde. Hierzu wird die untere Intervallgrenze r_{k-3} als Referenz ausgewählt und der Unit-Test U_{k-3} instrumentiert. Durch die Ausführung von U_{k-3} wird die Messreihe T_{k-3} generiert. Zunächst wird die Hypothese gestellt, dass r_k keine Leistungsregression einführt. Deshalb wird ebenfalls der Unit-Test U_k des entsprechenden Software-Systems S_k instrumentiert. Nach Ausführung von U_k nimmt der Hypothesentest die gestellte Hypothese an, woraus geschlossen wird, dass sich die Reihenmittel der Messreihe T_k und T_{k-3} nicht signifikant unterscheiden. Da die Intervallgröße $4 > 2$ ist, wird im folgenden nun festgestellt, ob es im Intervall eine ältere Software-Version $r_{k-y}, 3 > y > 0$ gibt, in der bereits eine Leistungsregression eingeführt wurde. Hierzu werden nun im folgenden die beschriebenen Schritte 1 bis 4 durchgeführt. Als Software-Version in der Mitte wird r_{k-2} ausgewählt. Gemäß Schritt 2 wird der Unit-Test $U_{k-2} \in S_{k-2}$ instrumentiert und durch dessen Ausführung T_{k-3} generiert. Mit der Analyse wird nun bestimmt, dass im Intervall $I' = [r_{k-3}, r_{k-2}]$ keine Leistungsregression vorliegt, indem T_{k-2} mit der Referenz T_{k-3} verglichen wird. Der Hypothesentest nimmt auch hier die gestellte Hypothese an, dass die Reihenmittel gleich sind. Deshalb wird als neues Intervall $I = [r_{k-2}, r_k]$ gesetzt. Für die neue Intervallgröße gilt aber immernoch $3 > 2$, deshalb wird für das neue Intervall I mit Schritt 1 begonnen. In der Mitte des neuen Intervalls $I = [r_{k-2}, r_k]$ befindet sich die Software-Version r_{k-1}. Die Nullhypothese $H_0 = \mu_{T_{k-1}} = \mu_{T_{k-2}}$ wird verworfen. Damit liegt bereits im Intervall $I' = [r_{k-2}, r_{k-1}]$ eine Leistungsregression vor, weshalb als neues Intervall $I = I'$ gesetzt wird. Für die neue Intervallgröße gilt $2 \leq 2$, damit ist die obere Grenze r_{k-1} dieses Intervalls sodann die erste Software-Version, in der eine Leistungsregression eingeführt wurde.

4.4. Problemdiagnose

Wird eine Leistungsregression durch die Regressionserkennung in Abschnitt 4.3 festgetellt, gilt es herauszufinden, wodurch diese ausgelöst wurde. Im folgenden wird aufgezeigt, wie die Ursache der Leistungsregression eingegrenzt wird und mit den Änderungen im VKS korreliert werden. Die Regressionserkennung liefert im Falle einer erkannten Leistungsregression mindestens einen Unit-Test und zwei aufeinanderfolgende Software-Versionen, zwischen denen sich die Antwortzeit des Unit-Tests signifikant unterscheidet. Für die weitere Erklärung wird angenommen, dass es sich dabei um die beiden Software-Versionen $r_i, r_j : i - 1 = j$ handelt. Die Software-Version r_i bezeichnet dabei diejenige Software-Version, die verdächtigt wird eine Leistungsregression eingeführt zu haben, wohingegen die Software-Version r_j die Software-Version bezeichnet, die bei der Regressionserkennung als Referenz diente. Diese beiden Software-Versionen bilden gerade das Intervall I der Regressionserkennung, das am Ende übrig bleibt. Neben den Software-Versionen stehen aber auch die im VKS gespeicherten Änderungen zwischen der Software-Version r_i und r_j zur Verfügung. Durch systematische Messungen werden zunächst diejenigen Methoden m_k identifiziert, deren Messreihen sich zwischen der Software-Version r_i und r_j signifikant unterscheiden und zudem Bestandteil des Aufrufbaumes CT des Unit-Tests sind. Der Aufrufbaum wird durch die Menge der aufgerufenen Methoden M_i für die Software-Version r_i und die Menge M_j für die Software-Version r_j definiert. Dabei kann sowohl $M_i \subseteq M_j$ als auch $M_i \supseteq M_j$ gelten. Im folgenden wird außerdem angenommen, dass der Unit-Test in den zugehörigen Software-Systemen zu r_i und r_j existiert. Die Knoten des Aufrufbaumes repräsentieren die aufgerufenen Methoden. Die Kanten des Aufrufbaumes repräsentieren die Aufrufe zwischen einzelnen Methoden. Die beiden Aufrufbäume werden sodann systematisch durchlaufen und die Methoden auf signifikante Unterschiede zwischen den beiden Aufrufbäumen untersucht. Eine Methode die einen signifikanten Unterschied in der Antwortzeit zwischen den Aufrufbäumen aufweist, wird als solche identifiziert. Die identifizierten Methoden können dann mit Änderungen im VKS in Verbindung gebracht werden. Die grundlegende Vorgehensweise der Problemdiagnose für jeden Unit-Test lässt sich in



(a) Instrumentierung der ersten Ebene im Aufrufbaum (b) Instrumentierung der zweiten Ebene im Aufrufbaum (c) Instrumentierung der dritten Ebene im Aufrufbaum (d) Resultierender Aufrufbaum

Abbildung 4.3.: Beispiel – Aufbau des Aufrufbaumes

drei Schritte kategorisieren:

1. Aufbau der Aufrufbäume CT_i und CT_j des Unit-Tests für die Software-Version r_i und r_j, die die Methodenaufrufe und die Aufrufhierarchie in der jeweiligen Software-Version wiederspiegeln.
2. Methoden m_k mit signifikanten Unterschieden in ihrer Antwortzeit zwischen CT_i und CT_j identifizieren.
3. Identifizierte Methoden m_k mit den gespeicherten Änderungen im VKS zwischen den beiden Software-Versionen r_i und r_j in Verbindung bringen.

4.4.1. Aufbau des Aufrufbaumes

Der Aufbau eines Aufrufbaumes beginnt mit den Testmethoden eines Unit-Tests. Ein Unit-Test besteht in der Regel aus einer oder mehreren Testmethoden. Die Wurzel des jeweiligen Aufrufbaumes bildet die Testmethode selbst, für die der Aufrufbaum erstellt wird. Die aufgerufenen Methoden bilden die Ebenen des Aufrufbaumes unterhalb der Wurzel. Für jede dieser Methoden werden die ausgeführten Methodenaufrufe ausgelesen und dem Aufrufbaum ein Knoten mit entsprechender Kante zwischen aufrufender Methode und aufgerufener Methode hinzugefügt. Für jede aufgerufene Methode wiederholt sich dieser Vorgang. Über die Instrumentierung wird jeder Methodenaufruf innerhalb des Methodenrumpfes einer Methode instrumentiert. Hierzu werden im Methodenrumpf Anweisungen eingefügt, die die Signatur der aufrufenden Methode und der aufgerufenen Methode an das Performance-Unit-Test-Rahmenwerk übergeben. Nachdem der Unit-Test ausgeführt wurde, wird für jeden ausgelesenen Methodenaufruf ein Tupel (m_u, m_v) bereitgestellt. Das Tupel (m_u, m_v) beinhaltet alle Informationen die benötigt werden, um die aufgerufene Methode m_v dem Aufrufbaum hinzufügen zu können. So beinhaltet das Tupel sowohl die aufrufende Methode m_u als auch die aufgerufene Methode m_v und gibt dadurch an, zwischen welchen beiden Methoden eine Kante hinzugefügt wird. Schrittweise wird so nach und nach der Aufrufbaum aufgebaut. Die beschriebene Vorgehensweise zum Aufbau des Aufrufbaumes am wird am Beispiel von Abbildung 4.3 veranschaulicht. Hiefür wird angenommen, dass der Unit-Test aus genau einer Methode a besteht, die zur Erbringung ihrer Funktionalität weitere Methoden aufruft. Im ersten Schritt wird die Methode a, wie in Abbildung 4.3(a) veranschaulicht, instrumentiert. Anschließend wird der Unit-Test ausgeführt, wodurch die Methoden b, c und d als aufgerufene Methoden ausgelesen werden. Danach wird die Instrumentierung von a wieder entfernt. Im darauf folgenden Schritt werden die Methoden b, c und d instrumentiert und der Unit-Test erneut ausgeführt. Dieser

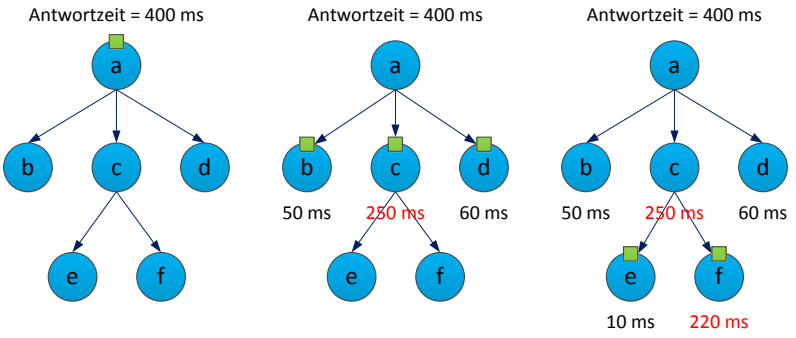
Schritt wird in Abbildung 4.3(b) veranschaulicht. Wie in Abbildung 4.3(c) zu erkennen ist, werden von den beiden Methoden b und d keine anderen Methoden aufgerufen. Die Methode c hingegen ruft sowohl die Methode e als auch die Methode f auf. Nachdem ebenfalls die Methoden e und f instrumentiert sind und der Unit-Test sodann ausgeführt wird, resultiert der in Abbildung 4.3(d) dargestellte Aufrufbaum, da weder die Methode e noch die Methode f weitere Methoden aufrufen.

4.4.2. Problemdiagnose bei gleichem Aufrufbaum

Um die Methoden $m_k \in (M_i \cap M_j)$ in CT_i und CT_j zu identifizieren, die sich in ihrer Antwortzeit zwischen r_i und r_j unterscheiden, werden systematische Messungen durchgeführt. Hierzu werden schrittweise die Methoden m_k in CT_i und CT_j vermessen. In jedem Schritt wird dadurch eine Messreihe $T_{m_k,i}$ für die Methode m_k in CT_i als auch eine Messreihe $T_{m_k,j}$ für die Methode m_k in CT_j generiert. Mit dem Hypothesentest kann die gestellte Hypothese, dass die Reihenmittel gleich sind, getestet werden. Wird die Hypothese angenommen, so wird die Methode m_k und der Teilbaum unterhalb von m_k als Ursache ausgeschlossen. Wird die Hypothese jedoch verworfen, wird auf einen signifikanten Unterschied zwischen den Reihenmitteln geschlossen. Handelt es sich bei m_k um einen Knoten des Baumes der kein Blatt ist, so müssen auch die Methoden der darunterliegenden Ebene verglichen werden. Handelt es sich dagegen um ein Blatt, wird die Methode m_k in die Liste der möglichen Ursachen aufgenommen. Diese Vorgehensweise lässt sich in die folgenden Schritte aufteilen:

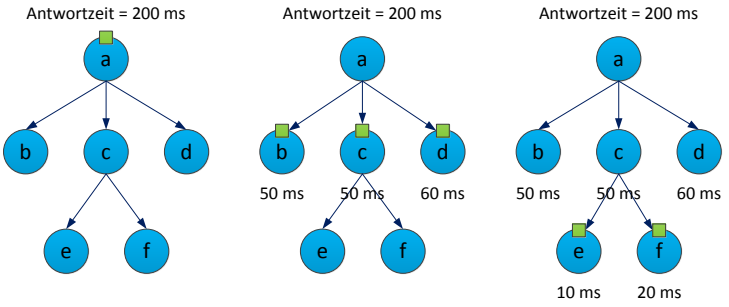
1. Generiere die Messreihe $T_{m_k,i}$ durch instrumentieren und vermessen der Methode m_k im Software-System der Software-Version r_i.
2. Generiere die Messreihe $T_{m_k,j}$ durch instrumentieren und vermessen der Methode m_k im Software-System der Software-Version r_j.
3. Testen der Nullhypothese $H_0 = \mu_{T_{m_k,i}} = \mu_{T_{m_k,j}}$.
4. Wird die Nullhypothese verworfen, wird für die von m_k aufgerufenen Methoden der darunterliegenden Ebene im Aufrufbaum mit Schritt 1 begonnen. Die Methode m_k wird als mögliche Ursache bewertet.
5. Wird die Nullhypothese angenommen, werden die von m_k aufgerufenen Methoden nicht weiter betrachtet.

Diese Vorgehensweise wird am weitergeführten Beispiel aus Abbildung 4.3 verdeutlicht, indem nun die Methoden identifiziert werden, die sich signifikant unterscheiden. Der Aufrufbaum besteht aus den Methoden $M = \{a, b, c, d, e, f\}$ und ist in den Software-Versionen r_i und r_j identisch. Durch die Regressionserkennung ist bekannt, dass sich die Antwortzeit der Methode a zwischen den Software-Versionen r_i und r_j statistisch signifikant unterscheidet. Die Antwortzeit der Methode a beträgt für die Software-Version r_i 200 ms und für die Software-Version r_j 400 ms wie die Abbildungen 4.4(a) und 4.5(a) für dieses Beispiel zeigen. Für die von Methode a aufgerufenen Methoden b, c und d wird sodann die Messreihen $T_{b,i}, T_{c,i}$ und $T_{d,i}$ generiert, indem die Methoden b, c und d im Software-System der Software-Version r_i instrumentiert werden und anschließend der Unit-Test ausgeführt wird. Dadurch werden die in Abbildung 4.4(b) zugeordneten Antwortzeiten der instrumentierten Methoden erhalten. Im zweiten Schritt werden die Messreihen $T_{b,j}, T_{c,j}$ und $T_{d,j}$ durch instrumentieren der Methoden b, c und d und Ausführung des Unit-Tests im Software-System der Software-Version r_j generiert. Dadurch weren die in Abbildung 4.5(b) zugeordneten Antwortzeiten der Software-Version r_j erhalten. Der Vergleich von $T_{b,i}$ mit $T_{b,j}$ ebenso wie der Vergleich $T_{d,i}$ mit $T_{d,j}$ ergibt keinen statistisch signifikanten Unterschied. Würden die Methoden b und d andere Methoden aufrufen, so würden diese an dieser Stelle nicht weiter betrachtet werden. Der Vergleich von $T_{c,i}$ mit $T_{c,j}$ hingegen ergibt einen statistisch



(a) Instrumentierung der ersten Ebene im Aufrufbaum (b) Instrumentierung der zweiten Ebene im Aufrufbaum (c) Instrumentierung der dritten Ebene im Aufrufbaum

Abbildung 4.4.: Beispiel – Software-Version r_i



(a) Instrumentierung der ersten Ebene im Aufrufbaum (b) Instrumentierung der zweiten Ebene im Aufrufbaum (c) Instrumentierung der dritten Ebene im Aufrufbaum

Abbildung 4.5.: Beispiel – Software-Version r_j

signifikanten Unterschied. Die Antwortzeit der Methode c unterscheidet sich zwischen den Software-Versionen r_i und r_j um 200 ms. Für die von c aufgerufenen Methoden $M' = \{e, f\}$ wird sodann mit Schritt 1 begonnen. Durch Instrumentieren der Methoden e und f und anschließendem Ausführen des Unit-Tests, werden die beiden Messreihen $T_{e,i}$ und $T_{f,i}$ für die Software-Version r_i generiert. In Abbildung 4.4(c) sind die zugeordneten Antwortzeiten dargestellt. Für die Software-Version r_j werden die Messreihen $T_{e,j}$ und $T_{f,j}$ nach Schritt 2 generiert. Die zugeordneten Antwortzeiten sind in Abbildung 4.5(c) dargestellt. Der Vergleich im dritten Schritt ergibt für die Methode e keinen signifikanten Unterschied der Antwortzeit. Im Gegensatz dazu ergibt der Vergleich der Messreihe $T_{f,i}$ mit $T_{f,j}$ einen statistisch signifikanten Unterschied, da sich die Antwortzeit der Method f um 200 ms unterscheidet. Wie aus dem Aufrufbaum CT_i in Abbildung 4.4 und dem Aufrufbaum CT_j in Abbildung 4.5 ersichtlich ist, werden keine Methoden von f aufgerufen. Da demnach $M' = \emptyset$ gilt, endet damit die Identifizierung der Methoden, die sich in ihrer Ausführungszeit zwischen den Software-Versionen r_i und r_j signifikant unterscheiden. Als Ursache der Leistungsregression kann deshalb die Methode f mit der Aufruffolge $a \rightarrow c \rightarrow f$ gewertet werden.

4.4.3. Problemdiagnose bei ungleichem Aufrufbaum

Die Aufrufbäume CT_i und CT_j müssen nicht gleich sein. Methoden können in der Software-Version r_i umbenannt werden oder sogar entfernt werden. Zusätzlich können aber auch

neue Methoden hinzukommen. Aber auch Klassen können in r_i umbenannt werden. Werden Methoden gelöscht oder kommen zusätzlich neue Methoden hinzu, ändert sich damit der Aufrufbaum. Die Problemdiagnose bei ungleichen Aufrufbäumen $CT_i \neq CT_j$ unterscheidet sich in der Identifikation der Methoden von der in Abschnitt 4.4.3 beschriebenen Vorgehensweise. Umbenannte und tatsächlich neue Methoden in CT_i werden dabei gleichbehandelt. Eine Methode $m_k \in M_i : m_k \ni M_j$ wird von der Analyse genauer untersucht. Die Analyse wählt die Methode $m'_k \in M_j$ aus, die sich an gleicher Stelle im Aufrufbaum CT_j befindet. Es wird dann geprüft, ob die aufrufenden und aufgerufenen Methoden von m_k und m'_k identisch sind. Ist dies nicht der Fall, wird die Methode m_k als mögliche Ursache der Leistungsregression identifiziert. Im anderen Falle, bei dem die eingehenden und ausgehenden Kanten jeweils identisch sind, werden m_k und m'_k instrumentiert und durch Ausführung des Unit-Tests die Messreihen $T_{m_k,i}$ und $T_{m'_k,j}$ generiert. Wird die gestellte Hypothese angenommen, schließt die Analyse aufgrund des identischen Kontextes der Methoden m_k und m'_k auf eine Umbenennung der Methode und fährt mit der in Abschnitt 4.4.3 beschrieben Vorgehensweise fort. Bei ungleichem Kontext wird der eventuell vorhandene Aufrufbaum unterhalb von m_k nicht weiter ausgewertet. Aufrufhäufigkeiten der Methoden werden bisher nicht ausgewertet.

4.4.4. Korrelation mit Änderungen

Nachdem nun eine oder mehrere Methoden als mögliche Ursache identifiziert sind, werden die im VKS gespeicherten Änderungen in einen kausalen Zusammenhang damit gebracht. In Abschnitt 2.3 wurde beschrieben, wie das VKS Git die Änderungen und seine Daten verwaltet. Jedesmal wenn eine Änderung aufgegeben wird oder der Zustand eines Projektes in Git gespeichert wird, macht Git einen Abzug des aktuellen Zustandes der Dateien und speichert eine Referenz zu diesem Schnappschuss. Haben sich Dateien nicht verändert, speichert Git die Dateien nicht erneut sondern fügt einfach eine Referenz auf die vorherige, identische Datei hinzu, die bereits gespeichert ist. In Java wird gewöhnlich der Quellcode jeder Klasse in einer Klassendatei gespeichert. Die Änderungen an einer Klasse und somit die Änderungen an einer Klassendatei werden daher von Git erfasst. Für die Klasse C, zu der die identifizierte Methode m_k gehört, kann geprüft werden, ob im VKS Änderungen in der Software-Version r_i vorliegen. Änderungen für eine Klasse C liegen genau dann vor, wenn die Klassendatei der Klasse C Bestandteil des Baumes ist. Wenn dies der Fall ist, so kann man die Änderungen als Differenz der Klassendatei zwischen r_i und r_j als Ursache werten. Im anderen Falle, bei dem die Klassendatei nicht Bestandteil des Baumes ist, ist davon auszugehen, dass es sich möglicherweise um Seiteneffekte der Änderungen anderer Klassen handeln könnte. Hierzu kann der Pfad im Aufrufbaum CT verwendet werden, um eine identifizierte Methode m_k mit einer Methode aus einer geänderten Klasse in Verbindung zu bringen. Dazu werden alle ausgehenden Pfade der Wurzel, die die Methode m_k beinhalten, hergenommen. Für jede Methode $m_x \neq m_k$ eines Pfades wird geprüft, ob für die zugehörige Klasse Änderungen im VKS in der Software-Version r_i vorliegen. Die so ermittelten Änderungen werden als mögliche Ursachen identifiziert. Die Vorgehensweise wird anhand eines Beispiels verdeutlicht. Hierzu wird die in Abbildung 4.6 dargestellten Änderungshistorie angenommen und an die vorhergehenden Beispiele in den Abbildungen 4.3, 4.4 und 4.5 weitergeführt. In diesem Beispiel wird die Software-Version r_i mit der Prüfsumme beginnend mit *98ca9* und die Software-Version r_j mit der Prüfsumme beginnend mit *34ac2* referenziert. Im vorhergehenden Beispiel wird die Methode f als diejenige identifiziert, die sich in ihrer Antwortzeit zwischen den beiden Software-Versionen signifikant unterscheidet. Es wird angenommen, dass die Methode f zur Klasse F gehört. Sodann kann der Änderungshistorie in Abbildung 4.6 entnommen werden, dass es für die zur Klasse F zugehörige Klassendatei *F.java* Änderungen im VKS gibt. Diese Differenz der Klassendatei *F.java* zwischen den Software-Versionen r_i und r_j wird für die spätere Berichterstattung extrahiert.

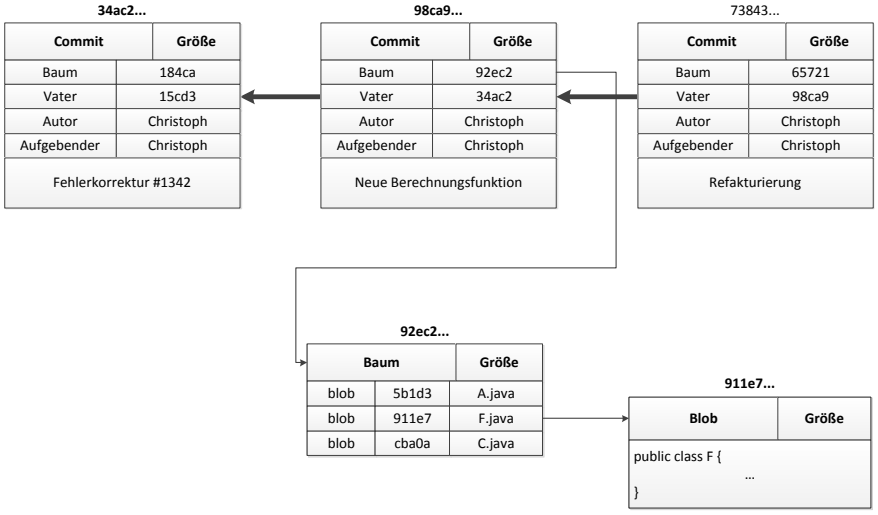


Abbildung 4.6.: Gespeicherte Änderungen im VKS

4.5. Adaptive Bytecode-Instrumentierung

Die in diesem Abschnitt beschriebene adaptive Bytecode-Instrumentierung ist in Zusammenarbeit mit Alexander Wert entstanden und wird auch in [Wer12] eingesetzt. Für die Streuung der Messungen in der Aufwärmphase mitverantwortlich ist der Ladevorgang und das damit verbundene Einweben der Instruktionen der Instrumentierung. Damit die Aufwärmphase bei der Erhebung der Messdaten vernachlässigt werden kann, erfolgt eine Trennung der Belange. Der Ladevorgang und das Einsetzen der Instruktionen der Instrumentierung wird strikt vom eigentlichen Messvorgang getrennt. Werden die Testmethoden eines Unit-Tests ausgeführt, sind bereits alle Testmethoden instrumentiert und die Unit-Tests bereits in die Java Virtual Machine (JVM) geladen. Die Aufwärmphase ist dadurch vernachlässigbar. Was verbleibt ist der in [vHRH+09] beschrieben Mehraufwand zur Ausführung der Instruktionen des Kieker Rahmenwerkes. Das Sequenzdiagramm in Abbildung 4.7 zeigt die Ausführung einer instrumentierten Methode. Das Sequenzdiagramm ist mit zwei Ausführungszeiten annotiert: Die Ausführungszeit ohne die Instrumentierung (Δ_A) und die Ausführungzeit die zur Erhebung und Schreiben der Messdaten benötigt wird (Δ_B). Die Ausführungszeit der instrumentierten Methode ist $\Delta_A + \Delta_B$.

Für die adaptive Bytecode-Instrumentierung wird Javassist zusammen mit dem HotSwap-Mechanismus aus Abschnitt 2.2.3 verwendet. Das Kieker Rahmenwerk aus Abschnitt 2.2.2 stellt die Instruktionen zur Instrumentierung der Methoden. Die Instrumentierung beliebiger Methoden ist ebenso möglich wie auch das Entfernen der Instrumentierung. Das Einsetzen der Instruktionen der Instrumentierung in die jeweilige Methode erfolgt mit Javassist. Es erlaubt den Rumpf einer Methode direkt zu verändern. Die in den Rumpf einer Methode eingesetzten Instruktionen gleichen den Instruktionen der aspektorientierten Instrumentierung *OperationExecutionAspectFull* aus Abschnitt 2.2.2 des Kieker Rahmenwerkes. Die Instruktionen der Instrumentierung werden in den Rumpf einer Methode so eingesetzt, dass der eigentliche Rumpf der Methode ummantelt wird. Dies entspricht der Instrumentierung durch die aspektorientierte Instrumentierung des Kieker Rahmenwerkes mit dem Unterschied, dass die Instruktionen der Instrumentierung direkt in den Bytecode einer Methode eingesetzt werden noch bevor die zugehörige Klasse in die JVM geladen wird und noch bevor irgendein Messvorgang gestartet wird. Damit die Instruktionen

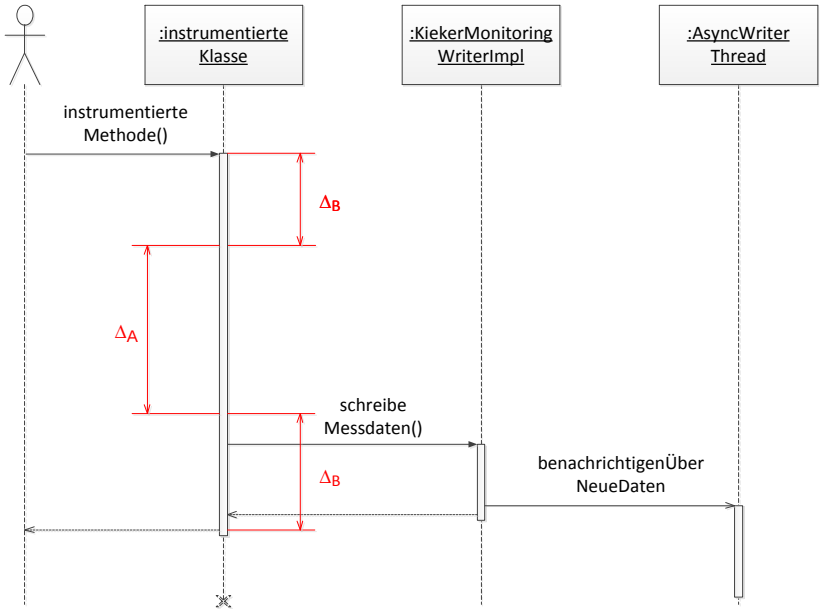


Abbildung 4.7.: Sequenzdiagramm für die Ausführung einer instrumentierten Methode

der Instrumentierung in den Rumpf einer Methode eingesetzt werden können, wird zunächst ein spezielles Übersetzungsmethodenobjekt (`CtMethod`) von Javassist erstellt, das den Methodenrumpf dieser Methode spezifiziert. Von Javassist bereitgestellte Methoden werden dazu verwendet, um zu Beginn des Methodenrumpfes aber auch am Ende des Methodenrumpfes die entsprechenden Instruktionen der Instrumentierung hinzuzufügen. Die Instrumentierung erfolgt bevor eine Klasse in die JVM geladen wird. Der HotSwap-Mechanismus ermöglicht es, dass der Bytecode einer bereits in die JVM geladenen Klasse durch den neuen aus der Instrumentierung resultierenden Bytecode zu ersetzen. Damit wird eine adaptive Instrumentierung der Methoden einer Klasse realisiert. Das direkte Einsetzen der Instruktionen der Instrumentierung in den Rumpf einer Methode kann mit dem Ausrollen einer Schleife verglichen werden. Die durch die aspektorientierte Progammierung bedingten Sprünge im Kontrollfluss zum Ausführen eines Aspektes vor und nach der Ausführung der eigentlichen Methode werden dadurch vermieden.

4.6. Automatisierung

In den folgenden beiden Abschnitten 4.6.1 und 4.6.2 wird die Integration des Ansatzes in die beiden Automatisierungswerkzeuge Apache Maven aus Abschnitt 2.4.3 und Jenkins aus Abschnitt 2.4.2 beschrieben. Über Zusatzprogramme kann das PUT-Rahmenwerk automatisiert ausgeführt und die Entwickler umgehend über Probleme informiert werden. Ziel der Automatisierung ist aber auch den Mehraufwand für die Entwickler zu minimieren und den Nutzen zu maximieren.

4.6.1. Lokale Erstellung

Im Rahmen der kontinuierlichen Integration führen Entwickler lokale Erstellungen des Software-Systems durch, bevor ihre Änderungen ins Versionskontrollsystem (VKS) aufgegeben werden. Dadurch soll das Risiko gemindert werden, dass fehlerbehaftete Änderungen in die Codebasis eingeführt werden, die dann wiederum zu einem Fehlschlag der Integration führen. Dazu werden die aktuellen Änderungen anderer Entwickler aus dem VKS abgerufen und zusammen mit den eigenen Änderungen eine vollständige Erstellung des

Software-Systems durchgeführt. Dies beinhaltet die Übersetzung des Quellcodes, die Ausführung von Tests etc. auf dem Rechner des Entwicklers. Wenn Entwickler nur ein paar Änderungen ausprobiert, die so nicht ins VKS aufgegeben werden sollen, werden meist leichtgewichtige Erstellungen des Software-Systems durchgeführt. Dabei wird oft nur der Quellcode übersetzt und einige Unit-Tests ausgeführt [DMG07]. Die Gemeinsamkeit in beiden Fällen liegt in der Verwendung eines Übersetzungswerkzeuges wie Apache Maven zur Automatisierung des Erstellungsprozesses. In den Grundlagen in Abschnitt 2.4.3 wird erläutert, dass ein Großteil der von Apache Maven verwendeten Funktionalität von Zusatzprogrammen bereitgestellt wird. Diese können im Projektmodell (POM), zusammen mit der Phase und dem auszuführenden Ziel angegeben werden. Die Integration des in dieser Arbeit vorgestellten Ansatzes erfolgt über ein solches Zusatzprogramm wie in Abbildung 4.8 dargestellt.

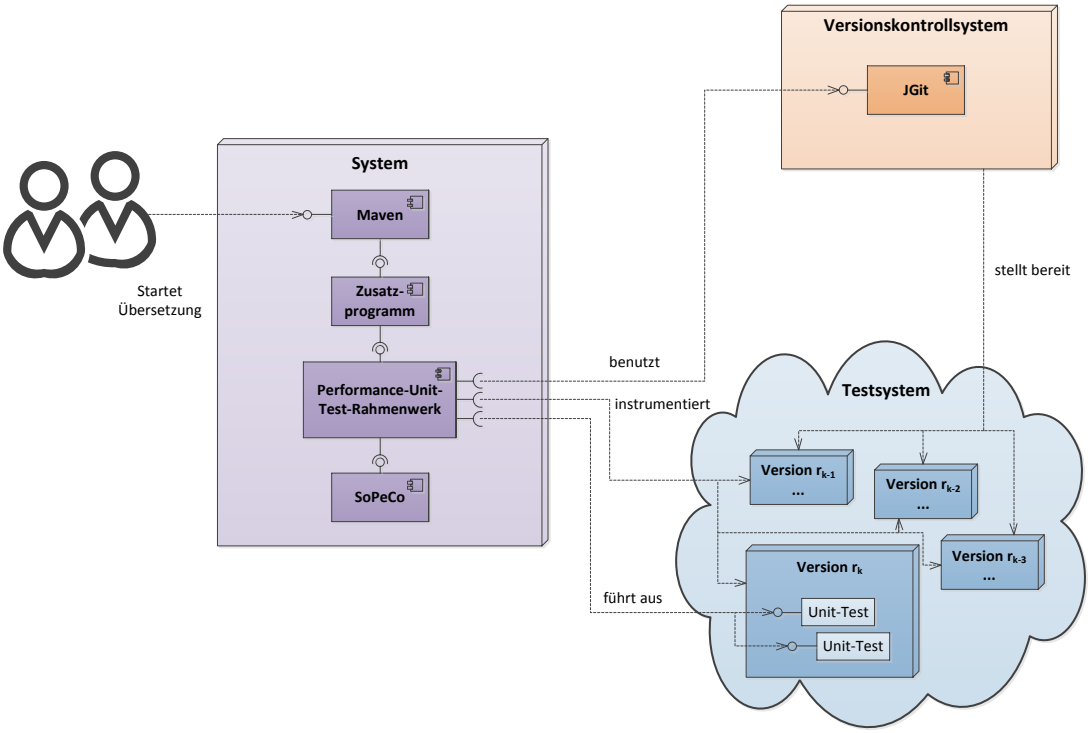


Abbildung 4.8.: Einbindung in die automatisierte Übersetzung

Ein Entwickler kann das PUT-Rahmenwerk als Zusatzprogramm während der Entwicklung zum POM hinzufügen. Dadurch wird bei jedem Übersetzungsvorgang des Software-Systems mit Maven automatisch die Leistung des Software-Systems überwacht. Wird eine Leistungsregression erkannt wird der Entwickler umgehend darüber informiert. Er kann daraufhin reagieren und Korrekturen vornehmen. Über definierte Eigenschaften des Zusatzprogramms können dem Entwickler bestimmte Freiheitsgrade eingeräumt werden. Eine anzubietende Eigenschaft kann beispielsweise die Erkennungsstategie sein. Der Aufwand für einen Entwickler ist bei dieser Variante der Automatisierung minimal und begrenzt sich auf drei Aktivitäten:

1. Manuelles Hinzufügen des Zusatzprogramms zum POM des Software-Projektes, an dem Änderungen vorgenommen werden.
2. Auswerten des generierten Berichts nach der Übersetzung.
3. Manuelles Entfernen des Zusatzprogramms aus dem POM, bevor die Änderungen

ins VKS aufgegeben werden.

Bei einer leichtgewichtigen Erstellung gilt es jedoch den Mehraufwand zu bedenken. Daher erscheint die Integration in die Eingangs beschriebene vollständige Erstellung des Software-Systems am sinnvollsten. Dadurch kann vermieden werden, dass Änderungen in die Codebasis eingeführt werden, die eine signifikante Leistungsregression auslösen.

4.6.2. Kontinuierliche Integration

In Jenkins können sogenannte Post-Erstellungsaktionen definiert werden. Eine häufige Anwendung ist die Darstellung von Testergebnissen. Dabei geht es nicht nur darum anzuzeigen, dass es fehlgeschlagene Tests gibt, sondern auch beispielsweise wie viele Tests ausgeführt wurden und wie lange die Ausführung gedauert hat. Jenkins bietet daher bereits die Unterstützung zur Darstellung von JUnit und JMeter [JMe]. Die Verarbeitung und Darstellung der Testergenisse erfolgt dabei über ein Zusatzprogramm. Ähnlich wie bei Maven kann die Funktionalität von Jenkins durch die Installation von Zusatzprogrammen erweitert werden. Diese Zusatzprogramme können als Post-Erstellungsaktion ausgeführt werden. Die Integration des Ansatzes in die kontinuierliche Integration erfolgt über ein ebensolches Zusatzprogramm für Jenkins. Das PUT-Rahmenwerk kann anschließend in einer Erstellungsaufgabe als Post-Erstellungsaktion ausgewählt werden. Hinzu kommt, dass das PUT-Rahmenwerk auch als eigenständige Erstellungsaufgabe separiert von der Erstellungsaufgabe des Software-Systems ausgeführt werden kann. In Abbildung 4.9 wird die Integration in Jenkins und somit in die kontinuierliche Integration veranschaulicht.

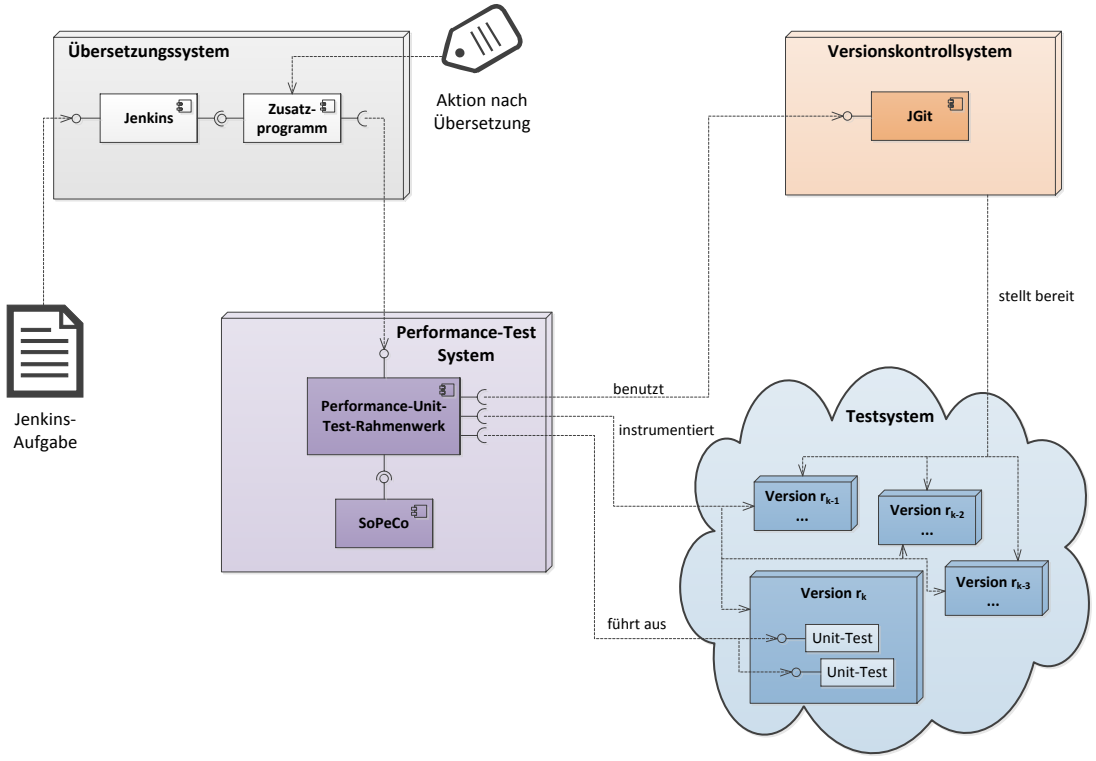


Abbildung 4.9.: Einbindung in die kontinuierliche Integration

Ein Entwickler kann beim Anlegen einer Erstellungsaufgabe das PUT-Rahmenwerk als Post-Erstellungsaktion auswählen. Das Zusatzprogramm kann das vom Rahmenwerk zurückgelieferte Ergebnis auswerten und bei erkannter Leistungsregression die Erstellung beispielsweise als instabil kennzeichnen. Die Entwickler werden dadurch umgehend darauf

hingewiesen, dass es Probleme gibt. Der Ansatz sieht dabei ein dediziertes System zur Durchführung der Regressionserkennung und Problemdiagnose vor. Dadurch werden die Einflüsse nebenläufig ausgeführter Erstellungen anderer Software-Systeme während der Messdatenerhebung umgangen. Das PUT-Rahmenwerk auf dem dedizierten System kann beispielsweise durch das Zusatzprogramm über einen Web-basierten Dienst oder einen entfernten Methodenaufruf (engl. remote method invocation) gesteuert werden. Die Ergebnisse der Regressionserkennung und Problemdiagnose werden nach abgeschlossener Analyse an das Zusatzprogramm zurückgegeben.

5. Implementierung

In diesem Kapitel wird der Prototyp beschrieben, der als Machbarkeitsstudie implementiert wurde. Die Architektur des Prototyps wird in Abschnitt 5.1 beschrieben. Die Implementierung der Regressionserkennung und Problemdiagnose wird anschließend in Abschnitt 5.2 erläutert. In Abschnitt 5.3 wird dargestellt, wie Messdaten für die Analyse bereitgestellt werden. Die Erhebung der Messdaten wird in Abschnitt 5.4 vorgestellt. Abschließend werden die aufgetretenen Probleme und die daraus resultierenden technischen Einschränkungen in Abschnitt 5.5 erläutert.

5.1. Architektur

In diesem Abschnitt wird ein Überblick über die Architektur der Prototypimplementierung gegeben. Für den Prototyp wird eine Schichtenarchitektur verwendet. Die bei SAP eingesetzten Technologien bilden die technischen Rahmenbedingungen. Der Prototyp wird in Java implementiert. Da für die Erstellung der SAP-Projekte Apache Maven eingesetzt wird und die Software-Projekte mit Git verwaltet werden, muss der Prototyp beide Systeme ansprechen können. Dies erfolgt über die in Apache Maven integrierte Programmierschnittstelle und bei Git über JGit [JGi]. Als Unit-Test Rahmenwerk wird JUnit eingesetzt über dessen Programmierschnittstelle Unit-Tests ausgeführt werden können. Der in Kapitel 4 beschriebene Ansatz wird auf drei Schichten aufgeteilt. Die drei Schichten lassen sich wie folgt kategorisieren:

1. **Generierung der Messreihen.** Damit Messdaten generiert werden, wird das Software-System zu einer Software-Version erstellt. Hierzu wird die Ablage des Versionskontrollsystems (VKS) einmalig geklont und die zu einer Software-Version gehörenden Klassendateien und Ressourcen in den Arbeitsbereich ausgegeben. Die automatisierte Erstellung liefert ein ausführbares Software-System mit allen Abhängigkeiten. Die adaptive Bytecode-Instrumentierung fügt die Anweisungen der Instrumentierung in den Bytecode der Testmethoden der Unit-Tests ein, ebenso wie bei anderen Methoden des Software-Systems. Über die Programmierschnittstelle von JUnit werden die Unit-Tests ausgeführt und dabei durch die Anweisungen der Instrumentierung Messdaten generiert. Das erstellte Software-System wird für eine spätere Wiederverwendung in einem Zwischenspeicher abgelegt.

2. **Verwaltung der Messreihen.** Die Generierung der Messreihen ist zeitaufwendig. Deshalb werden die Messreihen zu einer Software-Version in einer Datenbank per-

sistiert. Wird eine bereits generierte Messreihe angefordert kann diese aus der Datenbank abgerufen werden. Anderenfalls wird die Messreihe durch die erste Schicht generiert.

3. **Analyse der Messreihen.** Die Analyse vergleicht Messreihen und erkennt Leistungsregressionen, wenn signifikante Unterschiede vorliegen. Die Definition eines signifikanten Unterschiedes wird in der Analyse definiert. Welche Art von Leistungsregression gesucht wird, ist in der Regressionserkennung verankert.

Die Schichtenarchitektur erlaubt die Abstraktion der Details der jeweils darunterliegenden Schicht. Die erste Schicht ist verantwortlich für die technischen Details zur Generierung der Messreihen. Die zweite Schicht dagegen arbeitet nur noch auf Messreihen und einer Repräsentation der zugehörigen Software-Version. Außerdem hat sie den Überblick über die verfügbaren Software-Version, wie auch das betrachtete Intervall. Die dritte Schicht analysiert die bereitgestellten Messreihen. Dabei kann die Messreihe aus der Persistenz angefordert werden oder neu generiert werden. Abbildung 5.1 gibt einen Überblick über die implementierte Schichtenarchitektur. Die Funktionalität jeder Schicht wird in Verbundkomponenten (engl. composite components) gekapselt. Die Schichtenarchitektur und die Verbundkomponenten bieten den Vorteil einer losen Kopplung.

5.2. Regressionserkennung und Problemdiagnose

Die Regressionserkennung und Problemdiagnose analysiert Messreihen und Aufrufbäume, erkennt durch deren Vergleich Leistungsregressionen und diagnostiziert das Problem. Die Erkennungsstrategie definiert welche Art von Leistungsregression gesucht wird und die Analyse bei welchen Unterschieden es sich um eine Leistungsregression handelt. Die jeweilige Erkennungsstrategie ist ebenso wie die Problemdiagnose in der Komponente `Detection` enthalten. Die Messreihen werden von der Datenbereitstellung angefordert. Zunächst wird eine Referenz und anschließend eine zu prüfende Software-Version angefordert. Die Datenbereitstellung erhält Rückmeldung, ob eine Leistungsregression erkannt wurde oder nicht. Wird zwischen den Intervallgrenzen eine Leistungsregression erkannt, wird angefragt, ob weitere Software-Versionen zur Prüfung bereitgestellt werden können. Dieser Vorgang wiederholt sich solange, bis keine weiteren Software-Version zur Prüfung bereitgestellt werden können. Nach Abschluss des Vorgangs ist die früheste Leistungsregression im Intervall identifiziert. Dieser Vorgang ist in Abbildung 5.2 als Sequenzdiagramm für den Prototyp dargestellt. Das Ergebnis `regressionDetectionResult` der Regressionserkennung im Sequenzdiagramm beinhaltet eine Menge an Testmethoden der ausgeführten Unit-Tests, bei denen eine Leistungsregression erkannt wurde. Für jede dieser Testmethoden wird der Aufrufbaum in der als Referenz dienenden Software-Version und in der zu untersuchenden Software-Version erstellt. Die Aufrufbäume werden analysiert, wodurch die Methoden identifiziert werden, in denen sie sich unterscheiden. Die veränderten Java-Klassen werden ermittel, indem über JGit die Java-Dateien bestimmt werden, die zwischen den beiden Software-Versionen verändert wurden. Aus den Methoden des Aufrufbaumes werden die zugehörigen Klassen abgeleitet. Für jede der geänderten Klassen wird geprüft, ob sie beim Aufruf der identifizierten Methode im Aufrufbaum durchlaufen wird. Damit werden die Änderungen korreliert. Das Sequenzdiagramm in Abbildung 5.3 veranschaulicht die Implentierung dieses Vorgangs im Prototyp.

5.3. Datenbereitstellung

Die Datenbereitstellung stellt Messreihen für eine Software-Version zur Verfügung. Die Messdaten der Messreihen werden durch die Systemvermessung erhoben und von der Datenbereitstellung verwaltet. Die Messreihen der Testmethoden der Unit-Tests einer

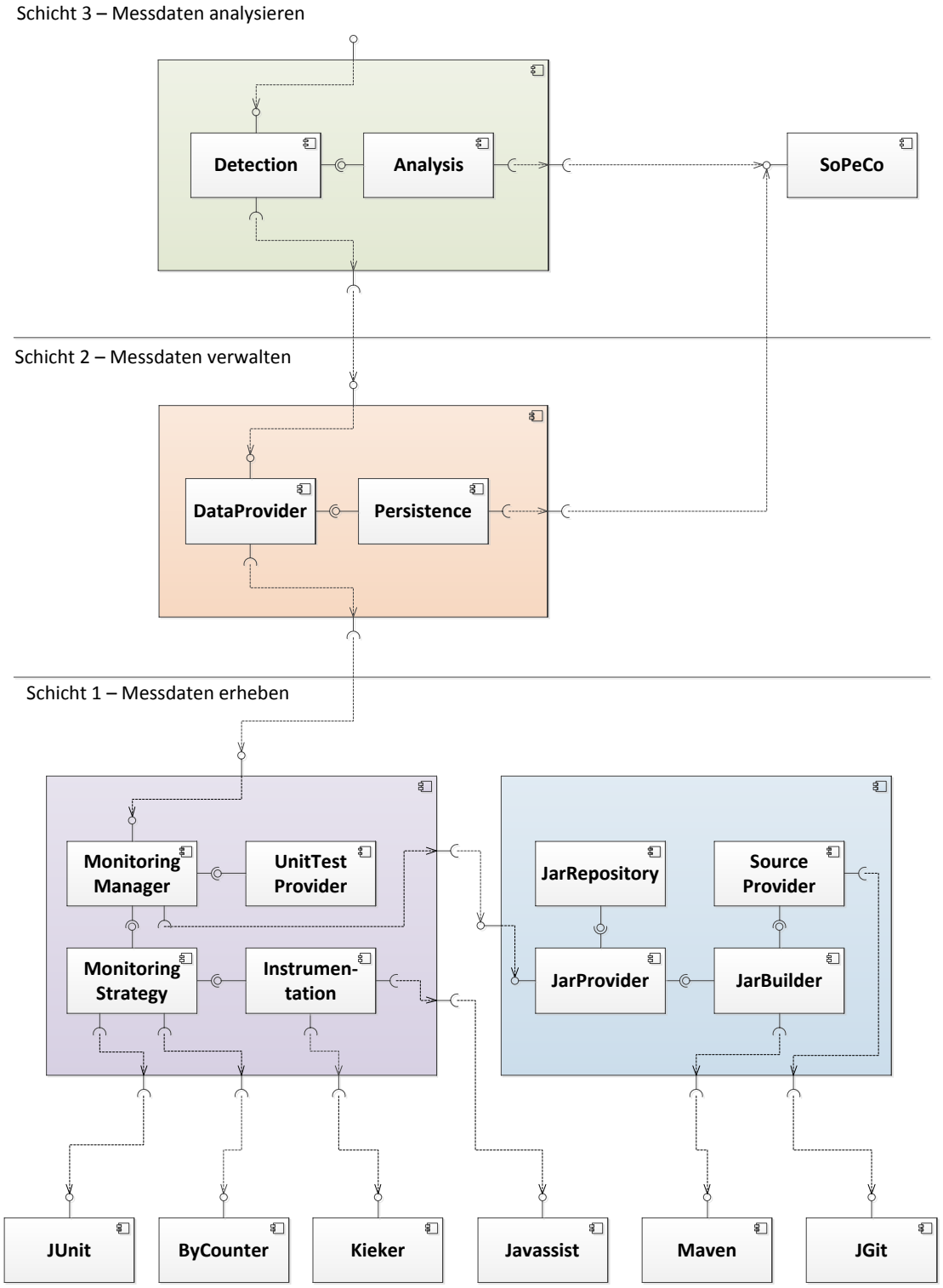


Abbildung 5.1.: Schichtenarchitektur mit Verbundkomponenten und Abhängigkeiten (ohne transitive Abhängigkeiten).

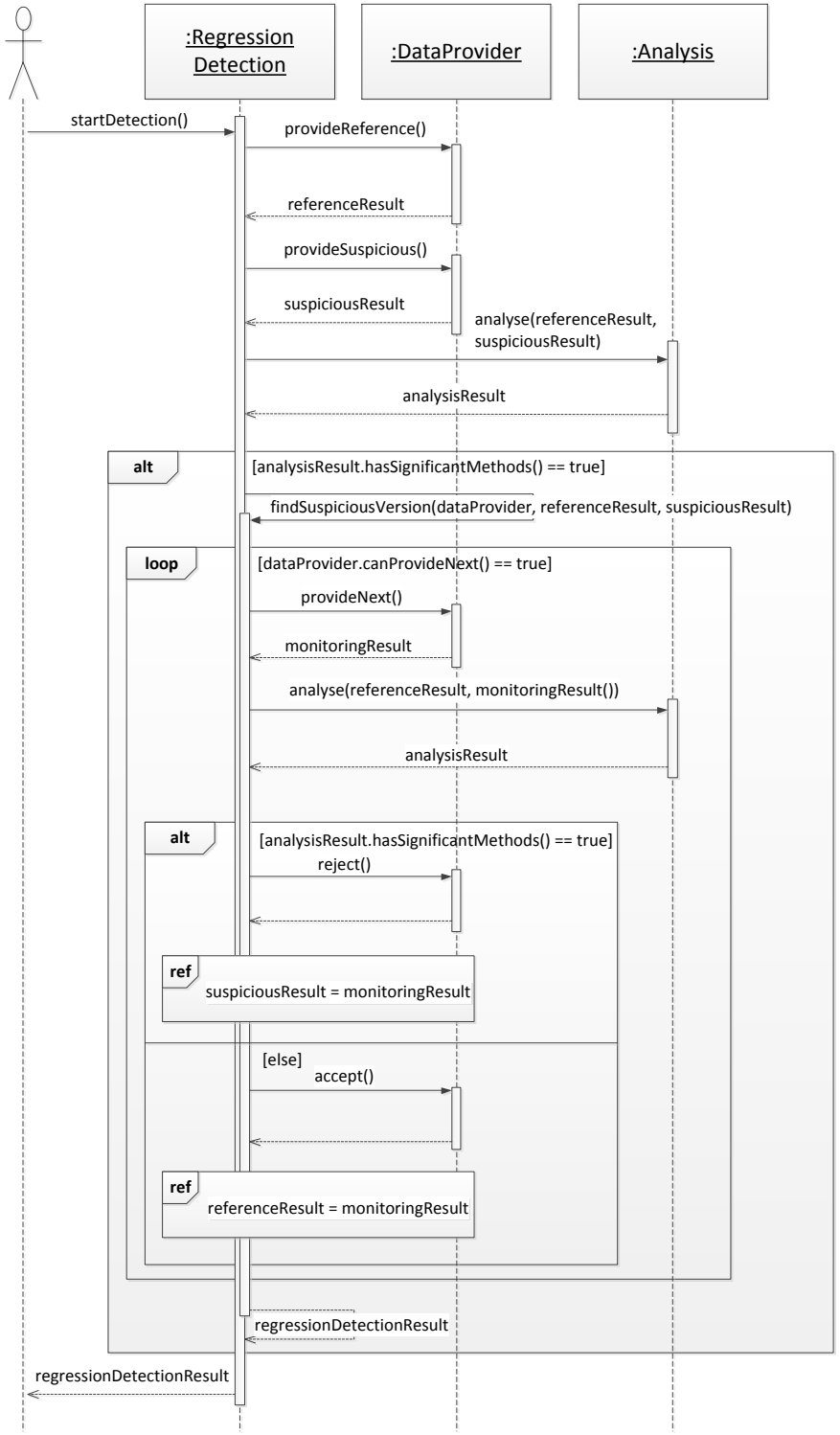


Abbildung 5.2.: Sequenzdiagramm der Regressionserkennung.

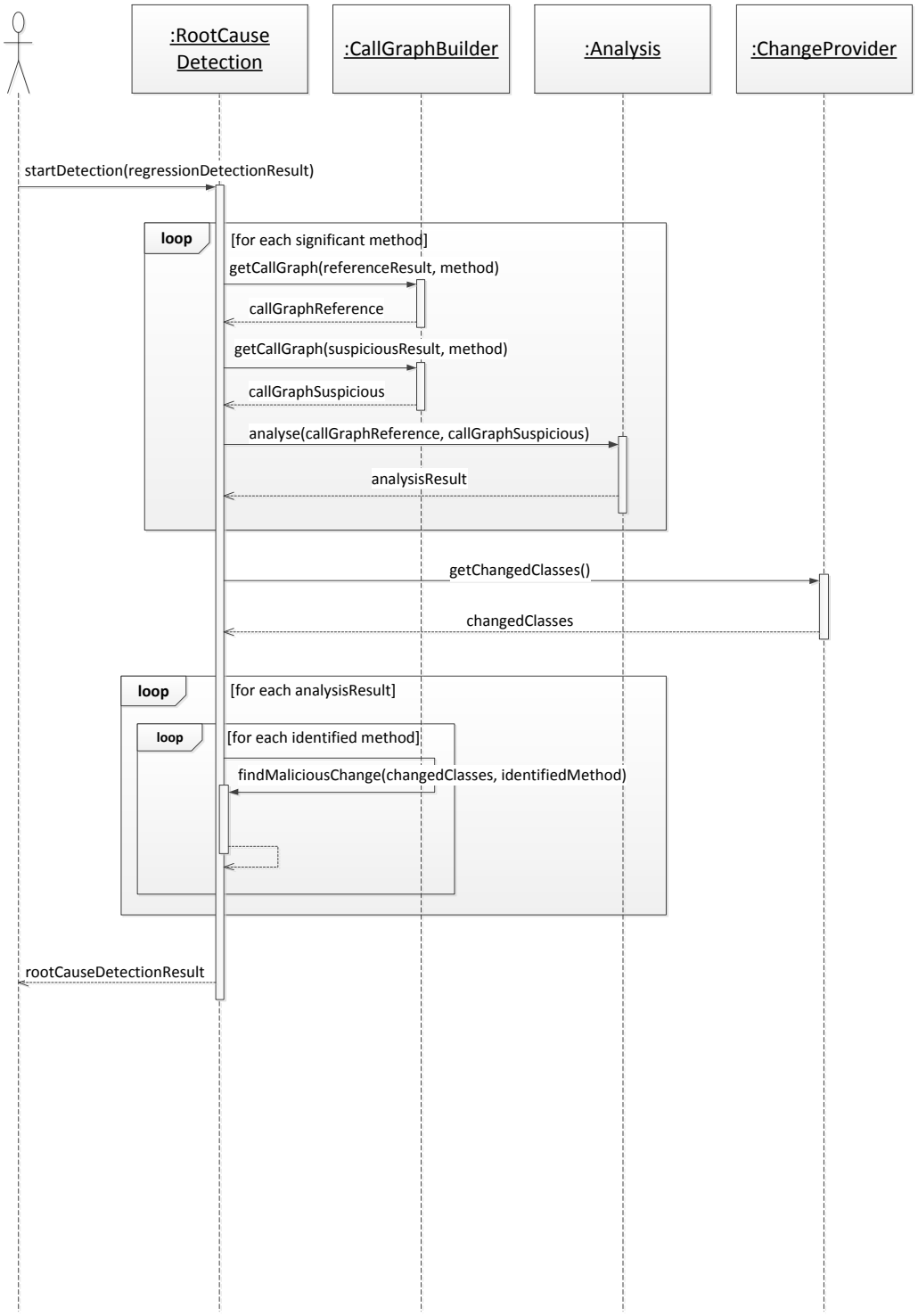


Abbildung 5.3.: Sequenzdiagramm der Problemdiagnose.

Software-Version, werden in einer Datenbank persistiert. Werden die Messreihen einer Software-Version angefordert, wird zunächst geprüft, ob diese bereits generiert sind und demnach in der Datenbank persistiert sind. Ist dies nicht der Fall werden die Messreihen generiert. Die Persistierung erfolgt über die Persistenz des Performance Cockpit (SoPeCo) aus Abschnitt 2.1. Als Datenbank wird für den Prototyp Apache Derby [Der] verwendet. Im Falle der in Abschnitt 4.3 beschriebenen Regressionserkennung, wird zunächst als Referenz die untere Intervallgrenze zurückgegeben. Die Messreihe wird zu dieser Software-Version aus der Datenbank abgerufen und als zu prüfende Software-Version die obere Intervallgrenze zurückgegeben. Die Messreihe hierzu wird durch die Systemvermessung generiert. Die binäre Suche zur Identifikaten der frühesten Leistungsregression im Intervall erhält von der Regressionserkennung als Eingabe, ob bei der zuletzt gelieferten Software-Version eine Leistungsregression erkannt wurde. Abhängig von der Rückmeldung schließt die binäre Suche das linke oder rechte Teilintervall von der weiteren Betrachtung aus.

5.4. Systemvermessung

Die Systemvermessung ist dafür verantwortlich Software-Systeme zu vermessen und dadurch die Antwortzeiten der Testmethoden bereitzustellen. Dazu werden die Klassendateien und Ressourcen, die für die Übersetzung des Software-Systems notwendig sind, aus der Ablage von Git ausgegeben. Die Funktionalität dazu wird von JGit bereitgestellt. Mit JGit wird eine entfernte Ablage auf das lokale System geklont und die angeforderte Software-Version in den Arbeitsbereich ausgegeben. Übersetzt und als Jar-Datei gepackt wird das Software-System von Apache Maven. Über das von Javassist angebotene `ClassPool`-Objekt wird die Jar-Datei und somit das Software-System in die JVM dynamisch eingebunden. Das Listing 5.1 zeigt die erforderlichen Anweisungen, um ein `ClassPool`-Objekt zu erstellen und eine Jar-Datei dem `ClassPool` hinzuzufügen.

```
ClassPool pool = new ClassPool();
pool.insertClassPath(jarfile.getName());
pool.appendSystemPath();
```

Listing 5.1: Jar-Datei nachladen

Die Testmethoden der Unit-Tests ebenso wie die getesteten Methoden werden entweder über Javassist im Kontext der adaptiven Bytecode-Instrumentierung oder über die Instrumentierung von ByCounter instrumentiert. Das Listing 5.2 zeigt die Anweisungen, die von der adaptiven Bytecode-Instrumentierung verwendet werden, um eine bestimmte Methode zu instrumentieren. Zunächst wird die zugehörige Klasse mit `pool.get(className)` aus dem `ClassPool`-Objekt `pool` abgerufen. Über `clazz.getDeclaredMethod(...)` wird das `CtMethod`-Objekt erstellt, das die zu instrumentierende Methode repräsentiert. Die Anweisungen zum Erheben der Messdaten werden mit `method.insertBefore(...)` vor die eigentlichen Anweisungen im Methodenrumpf und mit `method.insertAfter(...)` danach eingefügt.

```
CtClass clazz = pool.get("className");
clazz.defrost();

CtMethod method = clazz.getDeclaredMethod("methodName");

method.insertBefore("<Anweisungen>");
method.insertAfter("<Anweisungen>", false);
```

Listing 5.2: Instrumentierung einer Methode

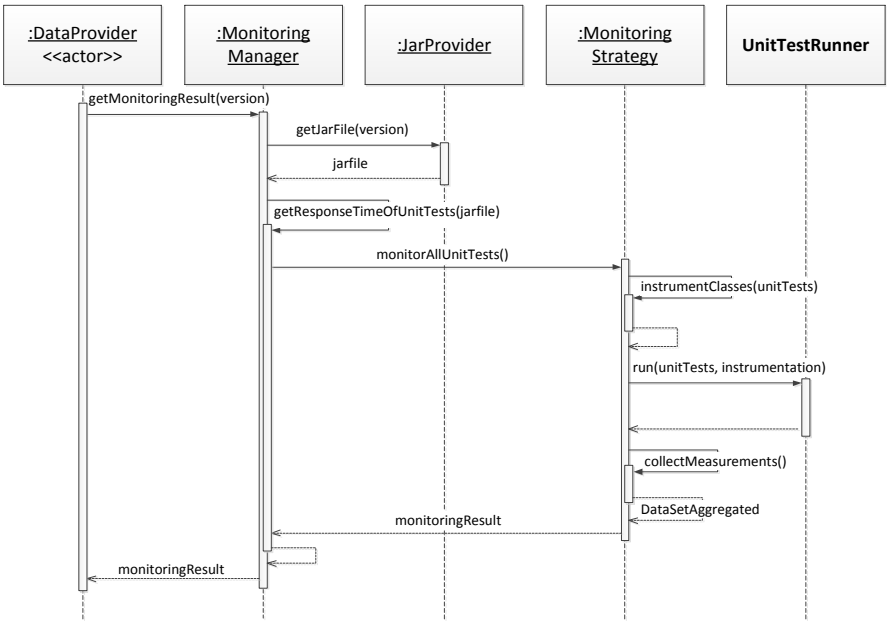


Abbildung 5.4.: Sequenzdiagramm der Systemvermessung

Anschließend wird der geladene Bytecode der Klasse mit Hilfe des `HotSwapper` durch den instrumentierten Bytecode ersetzt. Listing 5.3 zeigt die dazu notwendigen Anweisungen.

```
HotSwapper swapper = new HotSwapper(8000);
swapper.reload(clazz.getName(), clazz.toBytecode());
```

Listing 5.3: Bytecode ersetzen

Über die Programmierschnittstelle von JUnit werden die Unit-Tests ausgeführt und die Messdaten vom Kieker Rahmenwerk respektive ByCounter eingesammelt.

5.5. Limitierungen

Transitive Abhängigkeiten eines Software-Systems werden durch den Prototyp nicht aufgelöst. Deshalb werden alle Abhängigkeiten zusammen mit dem Software-System in eine Jar-Datei gepackt. Die Jar-Datei zu einer Software-Version wird jeweils über ein von Javasisst angebotenes `ClassPool`-Objekt in die JVM geladen. Der Vorteil dieser Vorgehensweise stellt sich aber auch als Problem heraus. Für jeden Klassenpool verwendet der Prototyp ein eigenes Klassenladeprogramm. Dadurch können gleichnamige Klassen verschiedener Software-Versionen gleichzeitig in die JVM geladen werden. Aufgrund der unterschiedlichen Klasseladeprogramme kann die JVM zwischen den gleichnamigen Klassen unterscheiden. Klassen die von den geladenen Klassen referenziert werden, werden dann ebenfalls mit dem gleichen Klasseladeprogramm geladen. Dies verursacht jedoch Probleme mit dem Kieker Rahmenwerk. Da die Anweisungen der Instrumentierung direkt in den Bytecode eingesetzt werden, wird die Instanz des Kieker Rahmenwerkes ebenfalls über das jeweilige Klassenladeprogramm geladen. Deshalb wird das Klassenladeprogramm des Klassenpools angewiesen, den Ladevorgang der Instanz des Kieker Rahmenwerkes an ein übergeordnetes Klassenladeprogramm zu delegieren. In der Theorie wird somit das Kieker-Rahmenwerk nur von einem Klassenladeprogramm geladen. In der Praxis gibt es dennoch Probleme. Indeterministisch gehen Messdaten eines Messvorgangs verloren, wenn mehrere Software-Versionen eines Software-Systems in der gleichen JVM nacheinander vermessen werden. ByCounter verwendet zur Instrumentierung die ASM Bibliothek [ASM][BLC02b]

zur Bytecode-Manipulation. Es kommt häufig vor, dass ASM Klassen nicht findet, obwohl diese der JVM definitiv bekannt sind. Die genaue Fehlerursache in beiden Fällen konnte bisher nicht ausfindig gemacht werden.

6. Validierung

Der Ansatz zur Regressionserkennung und Problemdiagnose wird in diesem Kapitel an einem künstlichen Software-System validiert. Die Validierung weiterer Szenarien ist Gegenstand weiterführender Arbeiten. In jedem Szenario der Regressionserkennung wird ein Intervall $I = [r_{k-3}, r_k]$ mit den Software-Versionen $r_{k-3}, r_{k-2}, \ldots, r_k$ betrachtet. Die Software-Version r_k stellt dabei die jüngste (obere Intervallgrenze) und r_{k-3} die älteste Software-Version (untere Intervallgrenze) im Intervall I dar. Auf den Software-Versionen im betrachteten Intervall ist somit die gleiche zeitliche Ordnung $r_k < r_{k-1} < \ldots < r_{k-3}$ definiert, wie auch in Abschnitt 4.3. Das verwendete Testsystem wird in Abschnitt 6.1 beschrieben. In Abschnitt 6.2 wird der Experimententwurf für die Validierung zusammen mit den nachgestellten Szenarien erläutert. Anschließend werden die Ergebnisse in Abschnitt 6.3 vorgestellt. Am Ende werden die Ergebnisse der Validierung in Abschnitt 6.4 nocheinmal zusammengefasst.

6.1. Testsystem

Das Testsystem für die Validierung ist ein künstliches Software-System, das aus den beiden Klassen `Fibonacci` und `FibonacciIterative` besteht, sowie dem Unit-Test `FibonacciTest`. Die Funktion der getesteten Klasse ist die Berechnung der Fibonacci-Zahl für eine als Parameter n übergebene Zahl. Die durch den Unit-Test `FibonacciTest` angesprochene Klasse `Fibonacci` abstrahiert dabei von der eigentlichen Implementierung der Berechnungsmethode `fibonacciIterative(n)`. Die getestete Klasse hat neben der Klasse mit der Implementierung der Berechnungsmethode keine weiteren Abhängigkeiten. Der Aufbau des Testsystems orientiert sich an der Definition eines Unit-Tests nach Duvall et al. aus [DMG07]. Nach dieser Definition testet ein Unit-Test das Verhalten einer Klasse. Die Ausführung des Unit-Tests ist in Abbildung 6.1 als Sequenzdiagramm dargestellt.

Die Erhebung der Messdaten erfolgt in einer virtuellen Maschine, die auf einer AMD Athlon II X4 CPU @ 2,60 GHz mit 8,00 GB Arbeitsspeicher (7,50 GB verwendbar) und 64-bit Betriebssystem Windows 7 Professional ausgeführt wird. Der virtuellen Maschine stehen zwei Prozessorkerne und 2,00 GB Arbeitsspeicher zur Verfügung. In der virtuellen Maschine wird die 32-bit Variante des Betriebssystems Windows 7 Professional ausgeführt. Die Unit-Tests werden in der Java Virtual Machine (JVM) 1.6.0_29 ausgeführt. Für die HotSwap-Funktionalität wird die JVM mit dem Paramter `-agentlib:jdwp=transport=dt_socket, server=y,suspend=n,address=8000` im Debug-Modus gestartet.

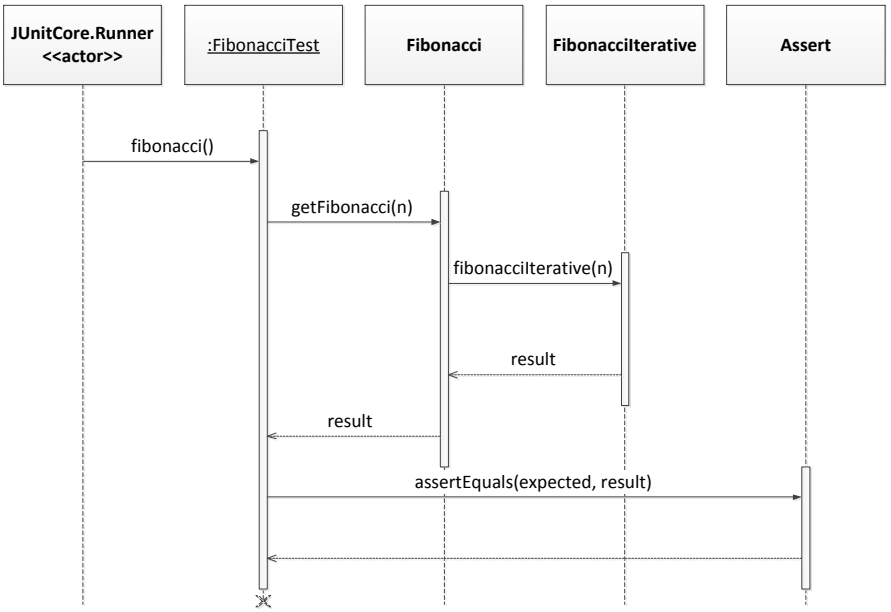


Abbildung 6.1.: Sequenzdiagramm Testsystem

6.2. Experimententwurf

Jedes Szenario besteht aus den vier Software-Versionen $r_{k-3}, \ldots, r_k$. In jeder Software-Version wird der Wert des Parameters n der Methode `getFibonacci(n)` erhöht oder verringert, damit die Antwortzeit der Methode `fibonacciIterative(n)` entsprechend zu oder abnimmt. Der Wert von n wird beibehalten, wenn sich die Antwortzeit zwischen zwei Software-Versionen nicht unterscheiden soll. Die Erhebung der Messdaten erfolgt durch fünfzigmalige Ausführung der Testmethode `fibonacci()`. Aufgrund der in Abschnitt 5.5 geschilderten technischen Einschränkungen erfolgt die Erhebung der Messdaten für eine Software-Version durch einen Benutzereingriff. Der Aufbau des Aufrufbaumes einer Testmethode für eine Software-Version erfolgt ebenfalls durch einen Benutzereingriff aus denselben Gründen.

Szenario	Software-Version			
	r_{k-3}	r_{k-2}	r_{k-1}	r_k
1	202000000	202000000	210000000	210000000
2	207000000	208000000	209000000	210000000
3	202000000	202000000	180000000	210000000
4	202000000	210000000	202000000	210000000
5	202000000	202000000	202000000	202000000
6	202000000	202000000	210000000	210000000

Tabelle 6.1.: Werte für n.

Für die Regressionserkennung werden die folgenden vier Szenarien erstellt:

1. **Signifikante Regression** Im ersten Szenario wird die Situation nachgestellt, in der die Antwortzeit einer Testmethode in mehreren Software-Versionen stabil ist, bevor eine Leistungsregression eingeführt wird. Die Leistungsregression wird in der Software-Version r_{k-1} eingeführt. Durch die Erhöhung von n in der Testmethode `fibonacci()` wird ein deutlicher Sprung in der Antwortzeit der Methode `fibonacciIterative(n)` erzielt. Die einzelnen Werte von n für die jeweilige Software-

Version kann Tabelle 6.1 entnommen werden. Es wird erwartet, dass der Prototyp die Software-Version r_{k-1} identifiziert.

2. **Graduelle Regression** Bei der graduellen Regression wird die Situation nachgestellt, in der durch aufeinanderfolgende Änderungen die Antwortzeit nur unwesentlich zunimmt. Dazu wird der Wert von n in jeder Software-Version stetig erhöht. Die Differenz der Antwortzeit zweier aufeinanderfolgender Software-Versionen ist dabei minimal. Tabelle 6.1 zeigt die Werte, die für n verwendwet werden. Die Werte sind so gewählt, dass die Nullhypothese $H_0 = \mu_i = \mu_j$ zum Konfidenzniveau von 99% für die Reihenmittel der Messreihen zweier benachbarter Software-Versionen nicht verworfen werden kann. Die Erwartung in diesem Szenario ist, dass der Prototyp r_k identifiziert.

3. **Vorausgehende Verbesserung** Im dritten Szenario wird eine Leistungssteigerung nachgestellt, bevor eine Leistungsregression eingeführt wird. Der Wert von n wird dazu in der Software-Version r_{k-1} deutlich verringert, wodurch die Antwortzeit der Methode `fibonacciIterative(n)` abnimmt. In der darauffolgenden Software-Version r_k wird dann die Leistungsregression eingeführt. Damit die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_k}$ verworfen wird, wird der Wert von n größer als in den anderen Software-Versionen gewählt. Tabelle 6.1 zeigt die Werte, die für n verwendet werden. Erwartet wird, dass r_k durch den Prototyp identifiziert wird.

4. **Oszillation** Die Oszillation stellt eine stark schwankende Antwortzeit der Testmethode nach. Für eine oszillierende Antwortzeit werden für n abwechselnd dieselben Werte verwendet. Damit wird in der Software-Version r_{k-2} und r_k eine Leistungsregression eingeführt. Die beiden Werte für n zeigt Tabelle 6.1. In diesem Szenario wird erwartet, dass die Software-Version r_{k-2} durch den Prototyp identifiziert wird.

Aufbauend auf dem ersten Szenario der Regressionserkennung werden für die Problemdiagnose zwei Szenarien erstellt:

5. **Direkte Änderung** Im ersten Szenario der Problemdiagnose wird die Änderung an einer Methode nachgestellt, die sich direkt auf die Antwortzeit dieser Methode auswirkt. Die Leistungsregression wird in der Software-Version r_k in der Methode `fibonacciIterative(n)` eingeführt. Hierfür wird als Änderung in der Methode `fibonacciIterative(n)` die Zahl 8000000 auf den Wert des übergebenen Paramters n addiert. Tabelle 6.1 zeigt den Wert, mit dem die Methode `getFibonacci(n)` aus der Testmethode `fibnoacci()` heraus aufgerufen wird. Es wird erwartet, dass der Prototyp die Methode `fibonacciIterative(n)` als Ursache identifiziert.

6. **Propagierte Änderung** Das Szenario der propagierten Änderung stellt die Situation nach, in der sich die Änderung auf die Antwortzeit einer anderen Methode auswirkt. In der Methode `fibonacciIterative(n)` wird die Leistungsregression ausgelöst, in dem der Wert von n in der Methode `fibonacci()` erhöht wird. Dies entspricht der Änderung in den Szenarien der Regressionserkennung. Die beiden Werte, die für n verwendet werden, zeigt Tabelle 6.1. In diesem Fall wird erwartet, dass der Prototyp die Methode `fibonacciIterative(n)` als Ursache identifiziert und die Änderung an der Klasse `FibonacciTest` über den Aufrufpfad `fibonacci()`→`getFibonacci(n)`→`fibonacciIterative(n)` korrelliert.

6.3. Experimentergebnis

In diesem Abschnitt werden die Ergebnisse des Experimentes für die in Abschnitt 6.2 entworfenen Szenarien vorgestellt. Sowohl die in Tabelle 6.2 aufgelisteten Kenngrößen, wie

auch die Diagramme in den Abbildungen 6.2, 6.3, 6.4 und 6.5 beziehen sich auf Messreihen der Testmethode `fibonacci()` des Unit-Tests `FibonacciTest` des Testsystems aus Abschnitt 6.1.

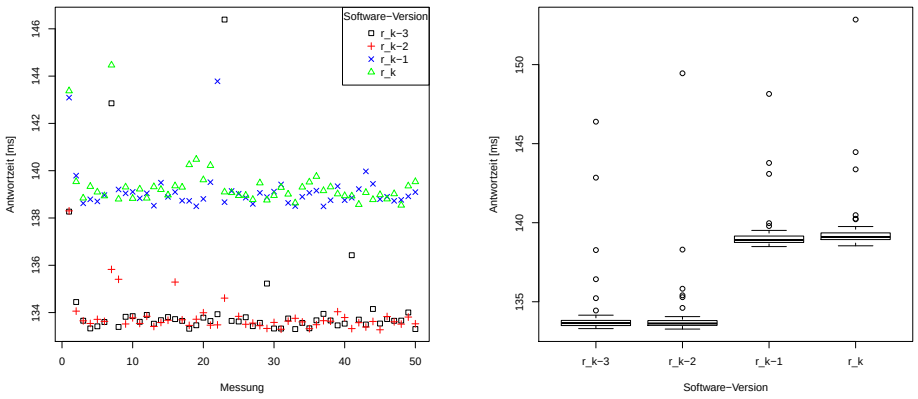
Szenario	Kenngröße	Software-Version			
		r_{k-3}	r_{k-2}	r_{k-1}	r_k
1	Minimum	133,3	133,3	138,5	138,5
	$0,25$-Quantile	133,5	133,5	138,8	138,9
	Median	133,7	133,6	138,9	139,1
	Mittelwert	134,3	134,2	139,3	139,6
	$0,75$-Quantile	133,8	133,8	139,2	139,4
	Maximum	146,4	149,5	148,1	152,8
2	Minimum	136,7	137,1	137,9	138,6
	$0,25$-Quantile	136,9	137,5	138,1	138,9
	Median	137,1	137,7	138,3	139,0
	Mittelwert	138,0	138,1	138,8	139,5
	$0,75$-Quantile	137,7	138,0	138,4	139,3
	Maximum	150,3	148,9	153,0	150,3
3	Minimum	133,1	133,1	118,7	138,4
	$0,25$-Quantile	133,3	133,4	118,9	138,7
	Median	133,5	133,5	119,1	138,9
	Mittelwert	134,0	134,1	119,6	139,5
	$0,75$-Quantile	133,6	133,6	119,3	139,2
	Maximum	148,8	149,4	125,9	154,1
4	Minimum	133,4	138,5	133,3	138,4
	$0,25$-Quantile	133,6	138,8	133,5	138,8
	Median	133,8	139,0	133,7	138,9
	Mittelwert	134,3	139,3	134,2	139,5
	$0,75$-Quantile	134,1	139,3	134,0	139,1
	Maximum	139,2	143,5	142,8	152,5

Tabelle 6.2.: Statistische Kenngrößen

1. **Signifikante Regression** In Tabelle 6.2 ist die mittlere Antwortzeit jeder Software-Version dargestellt. Die Leistungsregression in r_{k-1} ist deutlich erkennbar. Die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_k}$ wird verworfen und auf einen signifikanten Unterschied zwischen den Reihenmitteln geschlossen. Die Struktur der Messreihen wird in Abbildung 6.3(b) verglichen. Die Lage- und Streuungsunterschiede sind hier direkt ablesbar. Die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_{k-2}}$ wird angenommen. Dies wird durch den Vergleich der Messdaten in Abbildung 6.3(a) gestützt. Der Prototyp identifiziert r_{k-1}, da auch die Nullhypothese $H_0 = \mu_{r_{k-2}} = \mu_{r_{k-1}}$ verworfen wird.

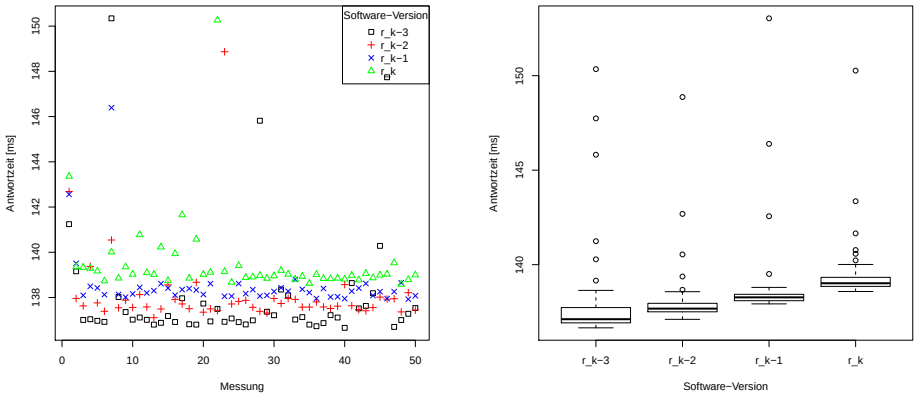
2. **Graduelle Regression** Der stetige Anstieg der Antwortzeit mit jeder Software-Version ist in beiden Diagrammen von Abbildung 6.3 erkennbar. Die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_k}$ wird durch den Hypothesentest verworfen. Die Kenngrößen in Tabelle 6.2 zeigen, dass zwischen r_{k-3} und r_k die größten Unterschiede existieren. Die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_{k-2}} = \mu_{r_{k-1}}$ wird angenommen, weshalb der Prototyp r_k identifiziert. Tabelle 6.2 zeigt, dass die Differenz der Reihenmittel $\mu_{r_{k-3}}$ und $\mu_{r_{k-1}}$ $0,8$ ms beträgt. Die Differenz der Reihenmittel, bei der die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_k}$ verworfen wird, beträgt $1,5$ ms.

3. **Vorausgehende Verbesserung** Der Vergleich der mittleren Antwortzeit in Tabelle 6.2 zeigt, dass die Antwortzeit in r_{k-3} und r_{k-2} stabil ist. Die Leistungsregressi-



(a) Messreihen (b) Kastenzeichnungen

Abbildung 6.2.: Szenario 1

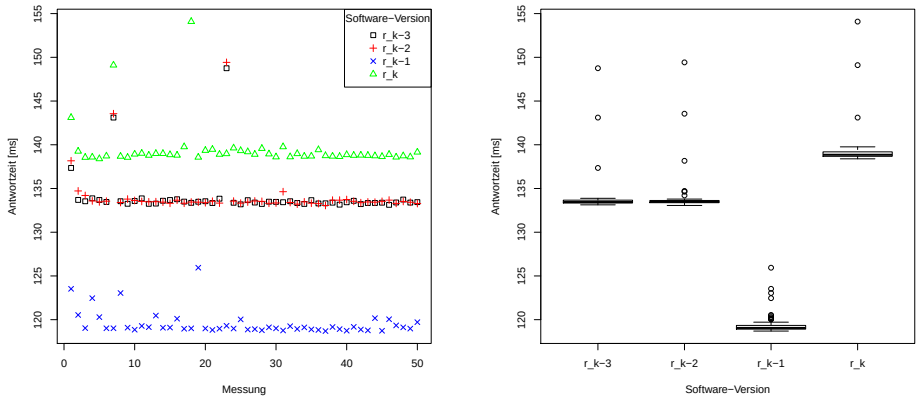


(a) Messreihen (b) Kastenzeichnungen

Abbildung 6.3.: Szenario 2

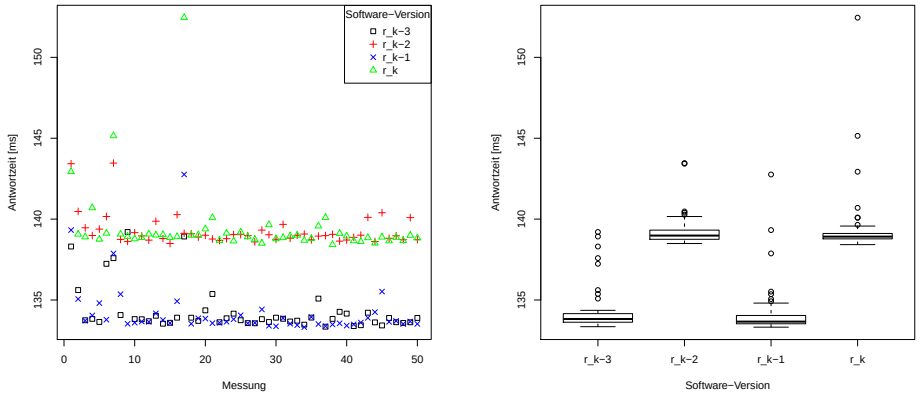
on und die vorausgehende Verbesserung ist deutlich erkennbar. Die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_k}$ wird verworfen und auf einen signifikanten Unterschied zwischen den Reihenmitteln geschlossen, die in Tabelle 6.2 aufgelistet sind. Durch den Hypothesentest kann die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_{k-2}}$ nicht verworfen werden. Der Vergleich der Struktur der Messreihen in Abbildung 6.5(b) zeigt die Ähnlichkeit beider Messreihen, die in Abbildung 6.5(a) noch deutlicher wird. Die Nullhypothese $H_0 = \mu_{r_{k-2}} = \mu_{r_{k-1}}$ wird verworfen. Der Prototyp erkennt, dass es sich in diesem Fall um keine Leistungsregression handelt. Die Software-Version r_k wird identifiziert, da $H_0 = \mu_{r_{k-1}} = \mu_{r_k}$ verworfen wird.

4. **Oszillation** Das Schwanken der mittleren Antwortzeit von Software-Version zu Software-Version ist in Tabelle 6.2 ablesbar. Die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_k}$ wird verworfen. Abbildung 6.5(b) zeigt die Strukturunterschiede beider Messreihen. Tabelle 6.2 zeigt, dass sich die Mittelwerte beider Messreihe um $5,2$ ms unterscheiden. Die anschließend gestellte Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_{k-2}}$ wird ebenfalls verworfen. Der Prototyp identifiziert daraufhin r_{k-2}.



(a) Messreihen (b) Kastenzeichnungen

Abbildung 6.4.: Szenario 3



(a) Messreihen (b) Kastenzeichnungen

Abbildung 6.5.: Szenario 4

Nachfolgend werden die Ergebnisse für Szenario 5 und 6 der Problemdiagnose vorgestellt. Es wird aufgezeigt, dass der Prototyp in Szenario 6 die Erwartung nur teilweise erfüllt.

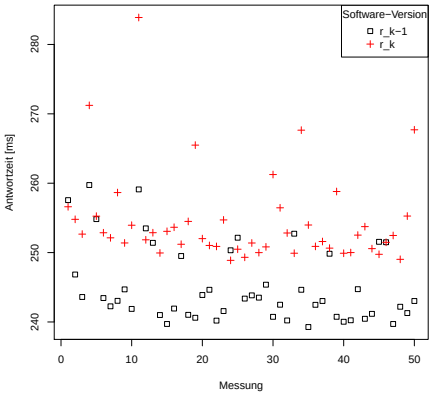
5. **Direkte Änderung** Der Hypothesentest testet die Nullhypothese $H_0 = \mu_{r_{k-1}} = \mu_{r_k}$ für die Messreihen der Methoden `fibonacci()`, `getFibonacci(n)` und `fibonacciIterative(n)`. Für jede Methode wird die Nullhypothese verworfen und auf einen signifikanten Unterschied zwischen den Reihenmitteln geschlossen. In Abbildung 6.6 werden die Messreihen jeder Methode miteinander verglichen. Der Prototyp identifiziert die Methode `fibonacciIterative(n)` mit dem Aufrufpfad `fibonacci()`→`getFibonacci(n)`→`fibonacciIterative(n)`.

6. **Propagierte Änderung** Entgegen der Erwartung wird die Nullhypothese $H_0 = \mu_{r_{k-1}} = \mu_{r_k}$ zum Konfidenzniveau von 99% für die Methode `fibonacciIterative(n)` angenommen. Abbildung 6.7(f) zeigt, dass sich die Messreihen in ihrer Struktur nur geringfügig unterscheiden, weshalb die Nullhypothese zum gegebenen Konfidenzniveau angenommen wird. Für die Methoden `fibonacci()` und `getFibonacci(n)` wird die Nullhypothese verworfen. Der Prototyp identifiziert entgegen der Erwartung die Methode `getFibonacci(n)` mit dem Aufrufpfad `fibonac-`

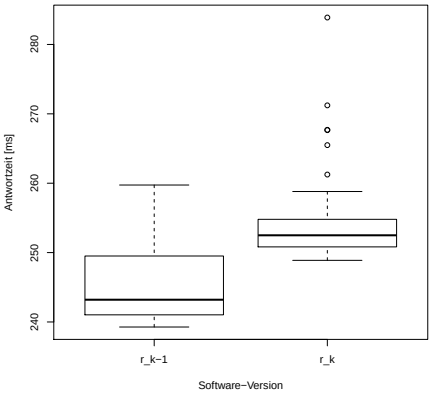
`ci()→getFibonacci(n).`

6.4. Zusammenfassung

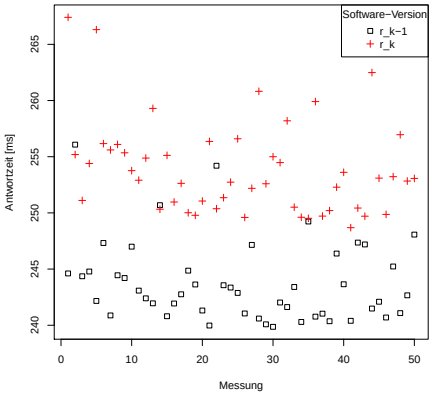
Die Ergebnisse der Validierung zeigen, dass der Prototyp in allen nachgestellten Szenarien die Erwartungen an die Regressionserkennung erfüllt. Voraussetzung ist jedoch, dass die Nullhypothese $H_0 = \mu_{r_{k-3}} = \mu_{r_k}$ verworfen wird. Wird die Nullhypothese angenommen, wird geschlossen, dass im betrachteten Intervall kein Performance-Problem vorliegt. Bei der Problemdiagnose zeigte sich, dass die propagierte Änderung eine geringere Auswirkung auf die Antwortzeit der Methode `fibonacciIterative(n)` hat, als angenommen. Trotz der Tatsache, dass der Prototyp die Erwartungen an die Problemdiagnose nur teilweise erfüllt, wurde gezeigt, dass er dennoch die notwendige Unterstützung bei der Ursachenfindung liefert.



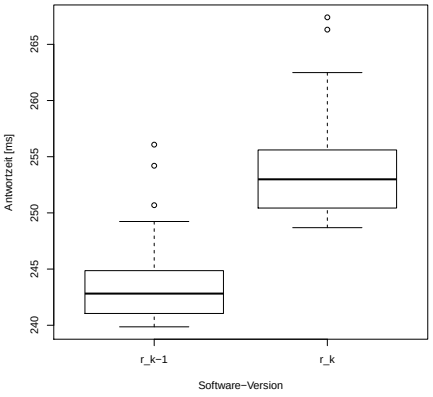
(a) `fibonacci()`



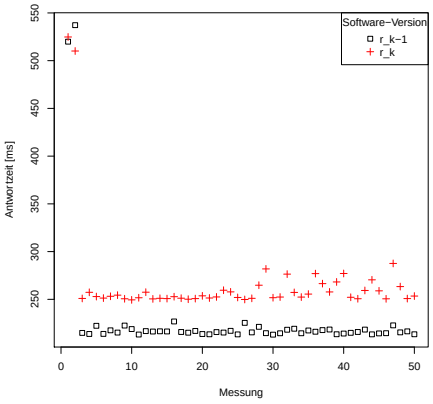
(b) `fibonacci()`



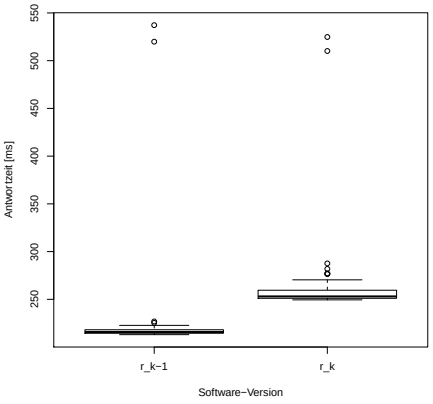
(c) `getFibonacci(n)`



(d) `getFibonacci(n)`

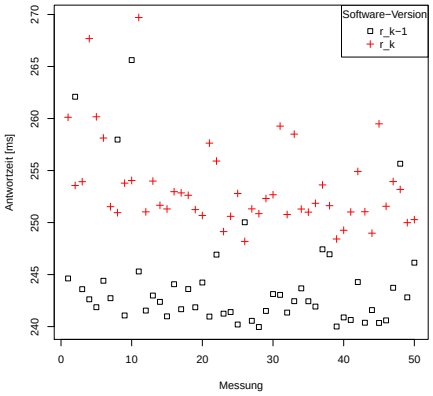


(e) `fibonacciIterative(n)`

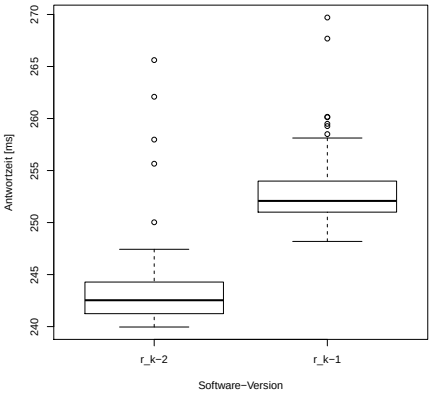


(f) `fibonacciIterative(n)`

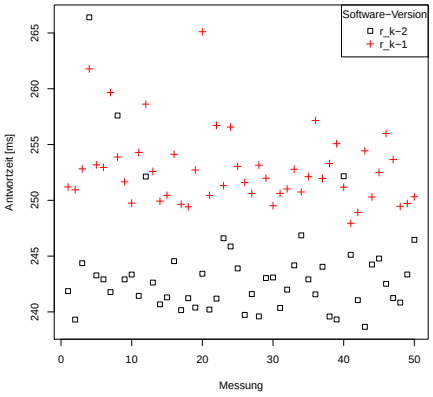
Abbildung 6.6.: Szenario 5



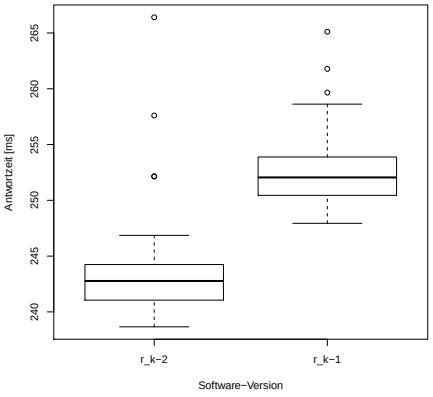
(a) `fibonacci()`



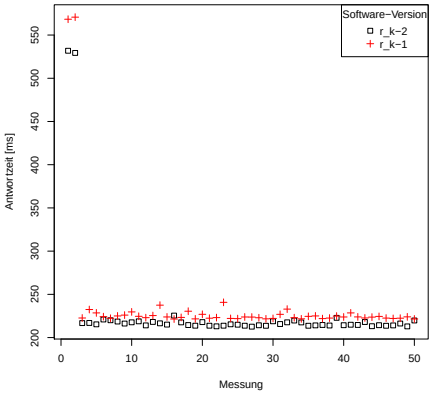
(b) `fibonacci()`



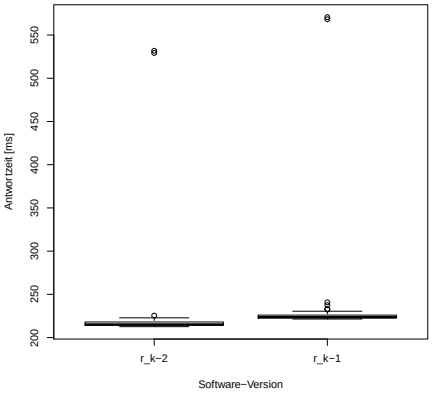
(c) `getFibonacci(n)`



(d) `getFibonacci(n)`



(e) `fibonacciIterative(n)`



(f) `fibonacciIterative(n)`

Abbildung 6.7.: Szenario 6

7. Evaluation

In diesem Kapitel wird die industrielle Fallstudie vorgestellt. Mit der Fallstudie wurde im praktischen Einsatz evaluiert, ob der Ansatz hilft Performance-Probleme zu erkennen. Das für die Fallstudie ausgewählte Software-Projekt wird in Abschnitt 7.1 vorgestellt. Die bei der Evaluation zu berücksichtigenden Einschränkungen werden in Abschnitt 7.2 erläutert. Anschließend wird der Experimententwurf zur Durchführung der Evaluation in Abschnitt 7.3 erklärt. In Abschnitt 7.4 werden die Ergebnisse der Fallstudie vorgestellt. Abschließend werden in Abschnitt 7.5 die Erkenntnisse aus der Evaluation nocheinmal zusammengefasst und diskutiert.

7.1. Project Voldemort

Project Voldemort [Vol] ist ein verteiltes, hochperformantes Speichersystem für Schlüssel-Wert-Paare, das sowohl von LinkedIn [Lin] entwickelt und eingesetzt wird aber auch für andere Anwender und Entwickler als quelloffenes System veröffentlicht wurde. Der Artikel über Amazons Dynamo [DHJ+07] inspirierte viele Aspekte von Project Voldemort [Kre]. Es wird bei LinkedIn für bestimmte Speicheraufgaben eingesetzt, bei denen eine einfache funktionale Partition ungenügend ist. Um einen hohen Grad an Performance und Verfügbarkeit zu gewährleisten werden nur einfache Schlüssel-Wert Datenzugriffe angeboten. Dabei können sowohl Schlüssel als auch die Werte komplexe zusammengesetzte Objekte sein. Die unterstützten Abfragen reduzieren sich auf das Lesen des Wertes über den Schlüssel `value = store.get(key)`, das Speichern eines Schlüssel-Wert-Paares `store.put(key, value)` und das Löschen eines Schlüssel-Wert-Paares über den Schlüssel `store.delete(key)` [Vol]. Die logische Systemarchitektur von Voldemort ist in Abbildung 7.1 dargestellt. Jede Schicht implementiert eine einfache Schnittstelle für das Lesen, Schreiben und Löschen von Schlüssel-Wert-Paaren. Eine Schicht ist verantwortlich für die Erfüllung einer bestimmten Funktionalität [Vol].

7.2. Einschränkungen

Es konnte kein Projekt ausfindig gemacht werden, das alle Anforderungen erfüllt. Das in Abschnitt 7.1 beschriebene Software-System Project Voldemort erfüllt die durch die technischen Einschränkungen aufgezwungenen Anforderungen weitestgehend. Es besteht aus einem Projekt, verwendet als Versionskontrollsystem Git und alle notwendigen Bibliotheken sind ebenfalls Bestandteil des Projektes. Für die Erstellung setzt das Projekt

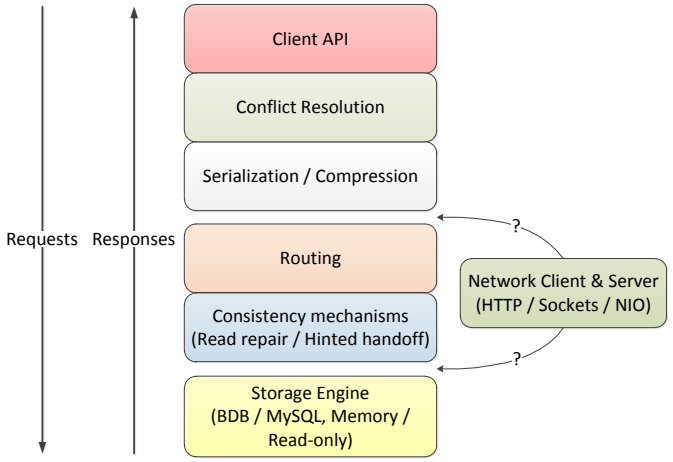


Abbildung 7.1.: Logische Architektur von Voldemort [Vol].

aber nicht auf Apache Maven sondern auf Apache Ant [Apa]. Das Ausführbare Software-System zu einer Software-Version muss deshalb durch einen Benutzereingriff erstellt werden. Zusätzlich muss vor jeder Erstellung Apache Ant angewiesen werden, alle Unit-Tests in das Software-System zu integrieren. Da der Prototyp aufgrund der Messdatenerhebung mit Kieker indeterministisch Messdaten verliert, sobald mehrere Software-Versionen nacheinander vermessen werden, müssen die Messdaten ohnehin durch einen Benutzereingriff erhoben werden. Die Tatsache, dass das ausführbare Software-System durch einen Benutzereingriff erstellt werden muss kann daher vernachlässigt werden. Die binäre Suche kann aufgrund der Anzahl aufgegebener Änderungen nicht angewendet werden, da aus zeitlichen Gründen nicht jedes dafür benötigte Software-System erstellt werden kann. Deshalb müssen gezielt Software-Versionen ausgewählt werden, die miteinander verglichen werden. Die Messdaten für die Problemdiagnose müssen aus demselben Grund ebenfalls durch einen Benutzereingriff erhoben werden. Durch die Benutzereingriffe müssen anschließend die Analysen manuell angestoßen werden. Die Ergebnisse der Analysen zeigen trotzdem, ob der Ansatz im praktischen Einsatz hilft Performance-Probleme zu erkennen, obwohl aktuell noch einige Benutzereingriffe notwendig sind.

7.3. Experimententwurf

Für die Evaluation wird die Git-Ablage [Gitb] von Voldemort, die auf GitHub [Gita] verwaltet wird, auf das lokale System kopiert. Da für die Erstellung von Voldemort nicht Apache Maven sondern Apache Ant eingesetzt wird und die Unit-Tests mit in die Jar-Dateien gepackt werden müssen sind bei der Erstellung Benutzereingriffe erforderlich. Die Erstellung von Voldemort erfolgt deshalb manuell durch Anpassung und Ausführung des Erstellungsvorgangs von Apache Ant. Als Einstiegspunkte für die Evaluation werden die freigebenenen Software-Versionen gewählt. Diese sind in der Änderungshistorie mit Markern gekennzeichnet und dadurch leicht auffindbar. Die Evaluation beschränkt sich dabei auf die Software-Versionen `0.90.1`, `0.90` und `0.81`. Verglichen werden jeweils benachbarte aufeinanderfolgende Software-Versionen wie beispielsweise `0.90.1` und `0.90`. Dadurch wird jeweils ein Intervall an Änderungen definiert. Aufgrund der in Abschnitt 7.2 aufgezeigten Einschränkungen können die einzelnen Software-Versionen nur getrennt vermessen werden. Die Erhebung der Messdaten erfolgt durch einen Benutzereingriff. Die erhobenen Messdaten werden in der Persistenz gespeichert. Die Analyse arbeitet anschließend ausschließlich auf den Messdaten aus der Persistenz. Für alle Unit-Tests, für die ein Performance-Problem erkannt wird, werden anschließend einzeln erneut Messdaten erhoben. Dadurch wird ge-

prüft, ob das Performance-Problem reproduziert werden kann und nicht das Resultat von Systemeinflüssen und Seiteneffekten ist.

Bei der Problemdiagnose müssen aufgrund der Einschränkungen ebenfalls Benutzereingriffe vorgenommen werden. Wird eine Leistungsregression einer Testmethode zwischen der Software-Version `0.81` und `0.90` oder `0.90` und `0.90.1` erkannt, so werden nicht alle dazwischen befindlichen Änderungen einer binären Suche nach der frühesten Leistungsregression unterzogen. Die Anzahl an Änderungen zwischen diesen Software-Versionen macht die Erstellung durch einen Benutzereingriff zeitlich unmöglich. Für die Evaluation muss deshalb die Suche nach der frühesten Software-Version im betrachteten Intervall, in der eine Leistungsregression eingeführt wird ausgeklammert werden. Die Problemdiagnose vergleicht die Aufrufbäume der identifizierten Testmethoden zwischen den Software-Versionen `0.81` und `0.90` oder `0.90` und `0.90.1` direkt.

Die Erhebung der Messdaten erfolgt in einer virtuellen Maschine, die auf einer AMD Athlon II X4 CPU @ 2,60 GHz mit 8,00 GB Arbeitsspeicher (7,50 GB verwendbar) und 64-bit Betriebssystem Windows 7 Professional ausgeführt wird. Der virtuellen Maschine stehen zwei Prozessorkerne und 2,00 GB Arbeitsspeicher zur Verfügung. In der virtuellen Maschine wird die 32-bit Variante des Betriebssystems Windows 7 Professional ausgeführt. Die Unit-Tests werden in der Java Virtual Machine (JVM) 1.6.0_29 ausgeführt. Für die HotSwap-Funktionalität wird die JVM mit dem Paramter `-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000` im Debug-Modus gestartet. Der Speicher für die Halde wird mit dem JVM-Parameter `-Xmx1024m` auf 1 GB erhöht.

7.4. Fallstudie

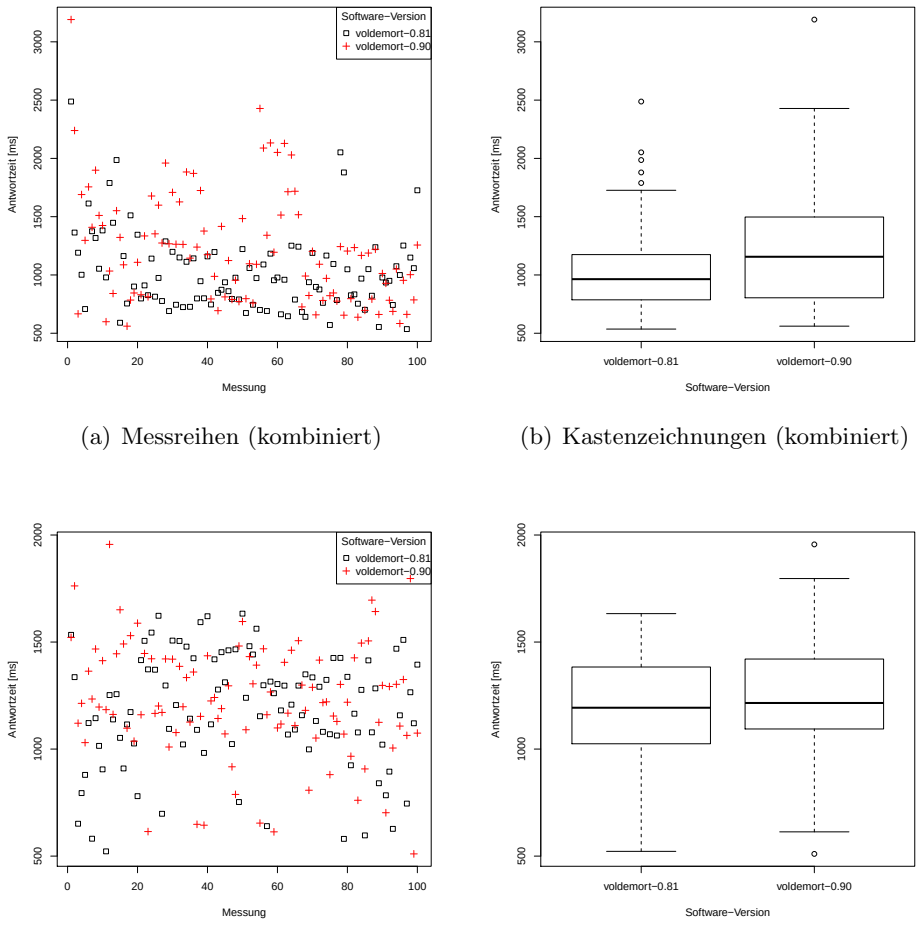
Mit Hilfe des Prototyps aus Kapitel 5 wird der Ansatz an dem in Abschnitt 7.1 beschriebenen Projekt evaluiert. Aufgrund der durch die Einschränkungen notwendigen Benutzereingriffe werden Regressionserkennung und Problemdiagnose nachfolgend getrennt betrachtet. Gleich zu Beginn der Fallstudie stellt sich heraus, dass es Unit-Tests gibt, die nicht instrumentiert werden können. Die genaue Problemursache kann bisher nicht festgestellt werden. Dadurch reduziert sich die Anzahl der Unit-Tests, die für die Evaluation von Belang sind und für die schließlich Messdaten erhoben werden. Eine Übersicht über die Zahl der verwendeten Unit-Tests jeder Software-Version gibt Tabelle 7.1. Die Unit-Tests sind namentlich in Tabelle A.1 in Anhang A aufgelistet. Die Fallstudie beginnt mit der Beschreibung der Ergebnisse aus der Regressionserkennung in Abschnitt 7.4.1. Die Ergebnisse der Problemdiagnose werden abschließend in Abschnitt 7.4.2 beschrieben.

Software-Version	Unit-Tests	Testmethoden
`0.90.1`	47	179
`0.90`	46	174
`0.81`	35	142

Tabelle 7.1.: Anzahl verwendeter Unit-Tests

7.4.1. Regressionserkennung

Bei der Regressionserkennung zeigt sich eine relative Häufigkeit an Fehlerkennungen. In Abbildung 7.2 ist dies für die Testmethode `testSorting(int)` des Unit-Tests `ExternalSorterTest` dargestellt. Die Abbildungen 7.2(a) und 7.2(b) zeigen die erhobenen Messdaten der Testmethode `testSorting(int)`, wenn alle Unit-Tests nacheinander in der gleichen JVM ausgeführt werden. Die Nullhypothese $H_0 = \mu_{testSorting-0.81} = \mu_{testSorting-0.90}$ für `testSorting(int)` wird verworfen und auf einen signifikanten Unterschied zwischen



(a) Messreihen (kombiniert)

(b) Kastenzeichnungen (kombiniert)

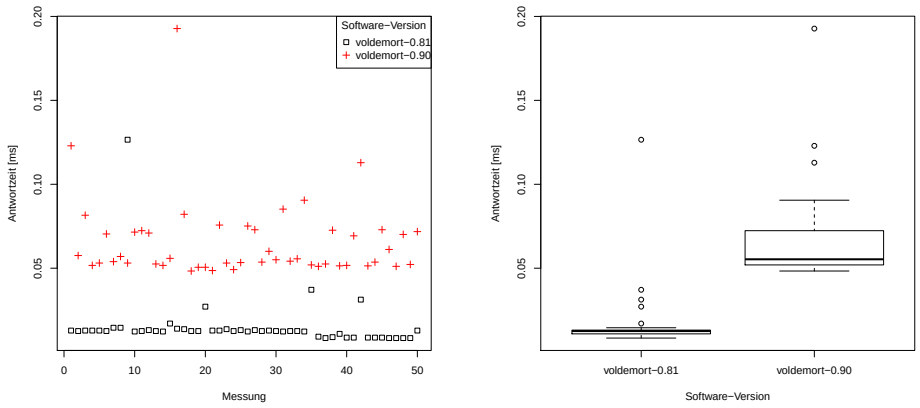
(c) Messreihen (einzeln)

(d) Kastenzeichnungen (einzeln)

Abbildung 7.2.: Fehlerkennung – `testSorting(int)`

den Reihenmitteln $\mu_{testSorting-0.81}$ und $\mu_{testSorting-0.90}$ der beiden Messreihen geschlossen. Die zur Kontrolle erhobenen Messdaten der Testmethode `testSorting(int)` sind in den Abbildungen 7.2(c) und 7.2(d) dargestellt. Die Nullhypothese wird für diese Messdaten angenommen. Die Analyse weiterer Fehlerkennungen und eine erneute Erhebung der Messdaten führte zum gleichen Ergebnis. Daraus wird geschlossen, dass sich die Unit-Tests gegenseitig beeinflussen, wenn diese nacheinander in der gleichen JVM ausgeführt werden. Eine Lösung dieses Problems ist die Ausführung jedes Unit-Tests in einer eigenen JVM. Dadurch wird der gegenseitige Einfluss der Unit-Tests eliminiert. Die Erhebung der Messdaten wird dadurch in ihrer Komplexität gesteigert.

Für die Testmethode `testByteComparison()` wird zwischen der Software-Version `0.90` und `0.81` eine reproduzierbare Leistungsregression erkannt. In Abbildung 7.3 sind die erhobenen Messdaten der Testmethode `testByteComparison()` dargestellt. Sowohl für diese Messdaten als auch für die zur Kontrolle erhobenen Messdaten wird die gestellte Nullhypothese verworfen, dass die Reihenmittel gleich sind. Die zur Kontrolle erhobenen Messdaten sind in Abbildung 7.4(a) und 7.4(b) dargestellt. Die gegenseitige Beeinflussung der Unit-Tests wird auch hier deutlich. Der absolute Unterschiede zwischen den Reihenmitteln beträgt lediglich 0.05141 ms. Der eher als klein zu bezeichnende Unterschied konnte trotzdem erkannt werden. Trotz der Änderungen zwischen den getesteten Software-Versionen wurden entgegen den Erwartungen keine weiteren Leistungsregressionen erkannt.



(a) Messreihen (b) Kastenzeichnungen

Abbildung 7.3.: Regressionserkennung – `testByteComparison()`

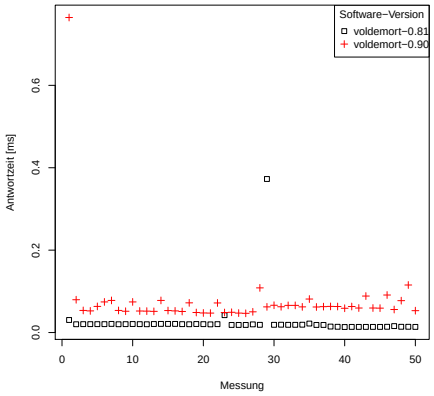
7.4.2. Problemdiagnose

Als Ursache der Leistungsregression für die Testmethode `testByteComparison()` wird die Testmethode selbst identifiziert. In Abbildung 7.4 sind die zur Kontrolle erhobenen Messdaten für `testByteComparison()` und die von ihr aufgerufene Methode `compare(byte[], byte[])` dargestellt. Die gestellte Nullhypothese, dass die Reihenmittel gleich sind, wird verworfen. Für die Methode `compare(byte[], byte[])` wird die gestellte Nullhypothese $H_0 = \mu_{compare-0.81} = \mu_{compare-0.90}$ angenommen. In der Software-Version `0.90` ruft die Methode `compare(byte[], byte[])` die Methode `compare(byte[],byte[],int,int)` auf. Die hinzugekommene Methode wird von der Problemdiagnose nicht erkannt. Die Ursache dafür ist die angenommene Nullhypothese für die Methode `compare(byte[], byte[])`. Die Problemdiagnose geht davon aus, dass in den von `compare(byte[], byte[])` aufgerufenen Methoden kein Performance-Problem vorliegt. Diese Schlussfolgerung ist auch richtig, da die Methode `compare(byte[],byte[],int,int)` nicht das Performance-Problem verursacht. In der Testmethode `testByteComparison()` sind zusätzliche Anweisungen hinzugekommen, mit denen die Methode `compare(byte[],byte[],int,int)` gestetet wird und wodurch sich die Antwortzeit der Testmethode erhöht. Wie auch die Messdaten in Abbildung 7.4 belegen, wurde das Performance-Problem tatsächlich auf die verursachende Methode eingeschränkt. Durch die Korrelation mit der Änderung der Testmethode werden dem Entwickler die hinzugekommenen Anweisungen angezeigt, die im Anhang in Abbildung A.1 dargestellt sind. Nach der Interpretation der Ergebnisse ist die Problemdiagnose für den Entwickler in diesem Fall abgeschlossen. Entwickler können aus den Ergebnissen ohne großen Aufwand schließen, dass das durch die Erweiterung des Testfalls ausgelöste Performance-Problem für das Software-System kein Risiko darstellt.

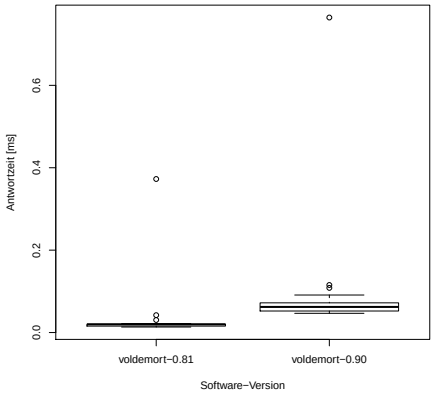
7.5. Zusammenfassung

Die Ergebnisse der Fallstudie zeigen, dass der Ansatz im praktischen Einsatz Performance-Probleme erkennt und hilft die Ursache zu identifizieren. Die erhobenen Messdaten zeigen, dass die absoluten Unterschiede der Reihenmittel eher klein sind. Ob sich solch ein Unterschied letztlich tatsächlich als Performance-Problem im Software-System bemerkbar macht, hängt von Faktoren wie der Aufrufhäufigkeit der getesteten Methode ab. Der testende Unit-Tests gibt darüber wenig Aufschluss. Daher kann die Adaption vorhandener Unit-Test ein zu fein granularar Ansatz sein. Abhilfe schaffen kann die Verwendung

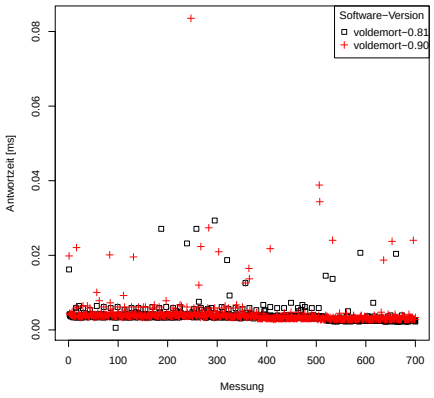
spezieller Unit-Tests, deren Fokus auf Performance-Tests liegt. Die aufgetretenen Fehlerkennungen stellen für Entwickler einen unnötigen Aufwand dar. Es ist erforderlich, das die Fehlerkennungen auf ein Minimum reduziert oder gänzlich eliminiert werden. Eine Lösung könnte die Ausführung jedes Unit-Tests in einer eigenen JVM sein. Die dadurch gesteigerte Komplexität bei der Messdatenerhebung muss gegen den Nutzen abgewogen werden. In der Fallstudie wurde ein Performance-Problem für eine Testmethode erkannt, das nicht von einer gestesteten Methode verursacht wird. Ein solches Performance-Problem, das durch Änderungen an der Testmethode selbst verursacht wird, ist nur bedingt interessant. Vielmehr interessieren Änderungen an den getesteten Methoden. Im Rahmen der Fallstudie konnte ein derartiges Performance-Problem nicht erkannt werden.



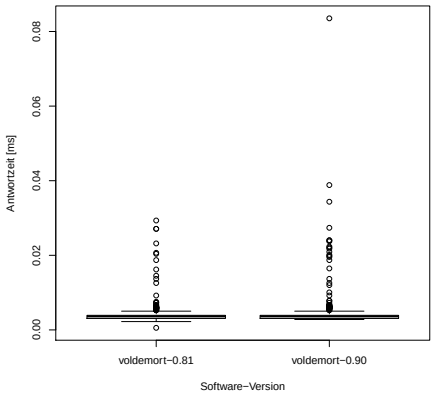
(a) `testByteComparison()`



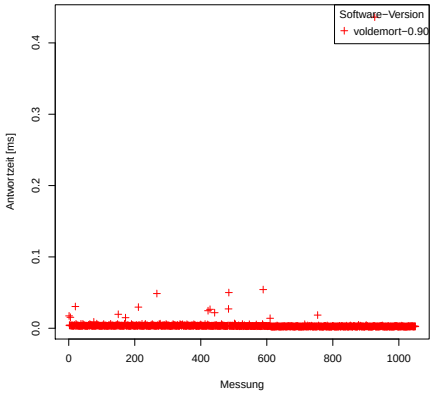
(b) `testByteComparison()`



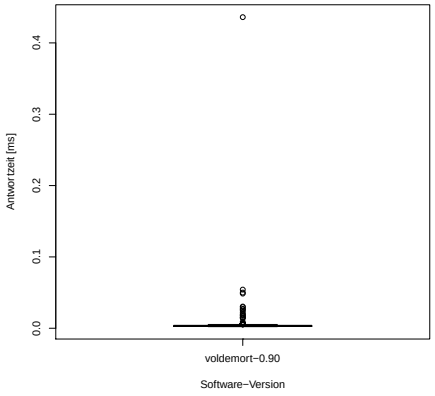
(c) `compare(byte[],byte[])`



(d) `compare(byte[],byte[])`



(e) `compare(byte[],byte[],int,int)`



(f) `compare(byte[],byte[],int,int)`

Abbildung 7.4.: Problemdiagnose

8. Abschluss

In dieser Diplomarbeit wurde ein Ansatz entwickelt, der Entwickler bei der Erkennung von Leistungsregressionen entlastet und bei der Problemdiagnose unterstützt ohne sie zusätzlich zu belasten.

Zu diesem Zweck wurde ein Verfahren entwickelt, das Leistungsregressionen zwischen Software-Versionen erkennt. Für das Ziel der Unterstützung des Entwicklers bei der Problemdiagnose wurde eine Methodik entwickelt, die durch den Vergleich von Aufrufbäumen die Ursache der Leistungsregression einschränkt und mit den im Versionskontrollsystem gespeicherten Änderungen verknüpft. Zur Automatisierung wurde der Ansatz in die kontinuierliche Integration eingebunden.

Die Eignung des Kieker Rahmenwerkes zur Messdatenerhebung wurde evaluiert. Dabei wurde festgestellt, dass die Instrumentierung des Kieker Rahmenwerkes die gestellten Anforderungen an Unabhängigkeit vom Entwickler, Adaptivität und vernachlässigbare Verfälschung der Messdaten nicht erfüllt. Für die Erhebung der Messdaten wurde deshalb eine Instrumentierung mitentwickelt, die bebliebige Methoden eines Software-Systems adaptiv und reversibel instrumentieren kann. Die Instrumentierung zeichnet sich durch die Unabhängigkeit vom Entwickler und einen vernachlässigbaren Einfluss auf die Messdaten aus. Als Alternative wurde die Eignung von ByCounter zur Zählung der ausgeführten Bytecode-Instruktionen evaluiert.

Als Machbarkeitsstudie wurde ein Werkzeug für die bei SAP eingesetzten Technologien entwickelt, das autonom Messdaten einer Software-Version erhebt, mit den Messdaten anderer Software-Versionen vergleicht, auf Performance-Probleme analysiert und die diagnostizierten Performance-Probleme mit den Änderungen im Versionskontrollsystem verknüpft.

Mit der durchgeführten Validierung wurde gezeigt, dass der aus der Machbarkeitsstudie resultierende Prototyp, erfolgreich verschiedene Leistungsregressionen erkennt und die Ursache identifiziert.

Weiterhin hat eine industrielle Fallstudie gezeigt, dass der Ansatz im praktischen Einsatz dabei hilft, Performance-Probleme zu erkennen und bei der Ursachenfindung zu unterstützen.

8.1. Nutzen

Der entwickelte Ansatz kann von Software-Entwicklern zur frühzeitigen Erkennung von Performance-Problemen eingesetzt werden, die durch Änderungen am Quellcode eines

Soft-ware-Systems hervorgerufen werden. Durch die Automatisierung der Problemdiagnose werden Entwickler bei der Ursachenfindung unterstützt. Die Verknüpfung mit dem Versionskontrollsystem hilft den Entwicklern die verantwortliche Änderung zu identifizieren. Eingebunden als Zusatzprogramm in die kontinuierliche Integration oder automatisierte Erstellung werden den Entwicklern die Software-Performance-Engineering-Techniken und das Domänenwissen in gekapselter Form und ohne Mehraufwand zugänglich gemacht. Entwicklern wird dadurch eine proaktive Software-Performance-Analyse ermöglicht. Außerdem kann das Vertrauen der Entwickler in die durchgeführten Änderungen gestärkt werden, gibt man ihnen die Gewissheit, dass aus einer Änderung resultierende Performance-Probleme umgehend angezeigt werden.

8.2. Weiterführende Arbeiten

Im Verlauf dieser Diplomarbeit mussten einige Punkte zum Gegenstand weiterführender Arbeiten erklärt werden: Die Strategie der Regressionserkennung kann verbessert werden, indem ein Schwellwert für jede Testmethode eines Unit-Tests definiert wird. Die Regressionserkennung überwacht, dass der Schwellwert nicht überschritten wird. Die empirischen Daten der Evaluation können keineswegs als vollständig betrachtet werden, liefern aber einen ersten Anhaltspunkt für einen geeigneten Schwellwert: Das $0,75$-Quantil liegt bei 0,196414 ms. Weitere Verbesserungen der Regressionserkennung kann die Erkennung der größten Leistungsregression im betrachteten Intervall sein und die Erkennung des langsamsten Unit-Tests.

Die Problemdiagnose kann mit einer statischen Quellcode-Analyse verbessert werden: Änderungen aus dem VKS werden den betroffenen Methoden zugeordnet. Dadurch kann zwischen relevanten und unrelevaten Änderungen an einer Klassendatei unterschieden werden. Es wird eine präzise Problemdiagnose ermöglicht. Unberücksichtigt sind Seiteneffekte hervorgerufen durch Änderungen an Konfigurationsdateien. Es muss eine Methode gefunden werden, wie Änderungen an Konfigurationsdateien mit Performance-Problemen korreliert werden können.

Die in dieser Diplomarbeit gesammelte Erfahrung deutet daraufhin, dass die Adaption bereits vorhandener Unit-Tests eine zu fein granulare Ebene darstellen könnte. Abhilfe können gezielte Performance-Tests sein, die sich die Technologie der Unit-Tests zunutze machen. Das Konzept der Adaption von Unit-Tests wird dabei beibehalten, eröffnet Entwicklern aber die Möglichkeit, bestimmte Unit-Tests für die Überwachung der Performance zu kennzeichnen. Bei diesen Unit-Tests könnten dann komplexere Testfälle implementiert sein, die sich an Komponententests anlehnen.

Der Aufwand bei der Erhebung der Messdaten kann durch eine intelligentere Herangehensweise reduziert werden: Kann eine Änderung an einer Klassendatei der oder den betroffenen Methoden zugeordnert werden, kann der zugehörige Unit-Test ermittelt werden. Infolgedessen wird die Regressionserkennung nur auf die ermittelten Unit-Tests angewendet. Die Anzahl der auszuführenden Unit-Tests wird reduziert und folglich der erforderliche Aufwand.

Bei der Validierung und Evaluation sind technische Probleme aufgetreten die abschließend untersucht und gelöst werden müssen. Nur dadurch wird der Einsatz ohne Benutzereingriffe möglich.

Die Implementierung des Prototyps muss um die Kompatibilität mit auf OSGi aufbauenden Software-Systemen erweitert werden. Damit wird der Prototyp aktuellen Entwicklungen und modernen Software-Systemen gerecht. Außerdem wird das Problem der Abhängigkeiten obsolet. Diese werden automatisch bereitgestellt. Die grafische Darstellung der Problemdiagnose sowie das Zusatzprogramm für die Integration in Jenkins müssen entwickelt werden.

Literaturverzeichnis

[Apa] *Apache Ant*. http://ant.apache.org/, besucht: 19.02.2012.

[ASM] *ASM Java bytecode engineering library*. http://asm.ow2.org/, besucht: 08.02.2012.

[Bec] Kent Beck: *JUnit testing framework*. http://junit.sourceforge.net/, besucht: 01.02.2012.

[BKT05] Lubomír Bulej, Tomáš Kalibera und Petr Tůma: *Repeated results analysis for middleware regression benchmarking*. Perform. Eval., 60:345–358, May 2005, ISSN 0166-5316. http://dx.doi.org/10.1016/j.peva.2004.10.013.

[BLC02a] Eric Bruneton, Romain Lenglet und Thierry Coupaye: *ASM: a code manipulation tool to implement adaptable systems*. Adaptable and extensible component systems, 30, 2002. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.5769&rep=rep1&type=pdf.

[BLC02b] Eric Bruneton, Romain Lenglet und Thierry Coupaye: *ASM: A code manipulation tool to implement adaptable systems*. In: *In Adaptable and extensible component systems*, 2002.

[Cha09] Scott Chacon: *Pro Git*. Apress, 2009, ISBN 978-1-430-21833-3. http://progit.org/ebook/progit.pdf, besucht: 11.11.2011.

[Chi00] Shigeru Chiba: *Load-Time Structural Reflection in Java*. In: *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, Seiten 313–336, London, UK, 2000. Springer-Verlag, ISBN 3-540-67660-0. http://dl.acm.org/citation.cfm?id=646157.679856.

[Con] *ContiPerf 2*. http://databene.org/contiperf, besucht: 09.02.2012.

[CSFP04] Ben Collins-Sussman, Brian Fitzpatrick und Michael Pilato: *Version Control with Subversion*. O'Reilly Media, 2004, ISBN 978-0-596-55619-8. http://svnbook.red-bean.com/en/1.6/svn-book.pdf, besucht: 11.11.2011.

[CST03] Shigeru Chiba, Yoshiki Sato und Michiaki Tatsubori: *Using HotSwap for Implementing Dynamic AOP Systems*. In: *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

[Der] *Apache Derby*. http://db.apache.org/derby/, besucht: 08.02.2012.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall und Werner Vogels: *Dynamo: amazon's highly available key-value store*. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, Seiten 205–220, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-591-5. http://doi.acm.org/10.1145/1294261.1294281.

[DMG07] Paul Duvall, Stephen M. Matyas und Andrew Glover: *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007, ISBN 0321336380.

[EvHJ11] Nils Ehmke, André van Hoorn und Reiner Jung: *Kieker 1.4 User Guide*, 2011. `http://ignum.dl.sourceforge.net/project/kieker/kieker/kieker-1.4/kieker-1.4_userguide.pdf`, besucht: 18.12.2011.

[EvHWH11] Jens Ehlers, André van Hoorn, Jan Waller und Wilhelm Hasselbring: *Self-adaptive software system monitoring for performance anomaly localization*. In: *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, Seiten 197–200, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0607-2. `http://doi.acm.org/10.1145/1998582.1998628`.

[FJA+10] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou und Parminder Flora: *Mining Performance Regression Testing Repositories for Automated Performance Analysis*. In: *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, Seiten 32–41, Washington, DC, USA, 2010. IEEE Computer Society, ISBN 978-0-7695-4131-0. `http://dx.doi.org/10.1109/QSIC.2010.35`.

[Fow06] Martin Fowler: *Continuous Integration*, 2006. `http://www.martinfowler.com/articles/continuousIntegration.html`, besucht: 31.01.2012.

[FZ11] Gordon Fraser und Andreas Zeller: *Generating parameterized unit tests*. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, Seiten 364–374, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0562-4. `http://doi.acm.org/10.1145/2001420.2001464`.

[GBE07] Andy Georges, Dries Buytaert und Lieven Eeckhout: *Statistically rigorous java performance evaluation*. SIGPLAN Not., 42:57–76, October 2007, ISSN 0362-1340. `http://doi.acm.org/10.1145/1297105.1297033`.

[Gita] *GitHub*. `https://github.com/`, besucht: 19.02.2012.

[Gitb] *GitHub.com - Project Voldemort*. `https://github.com/voldemort/voldemort`, besucht: 19.02.2012.

[HEK05] Joachim Hartung, Bärbel Elpelt und Karl Heinz Klösener: *Statistik: Lehr- und Arbeitsbuch der angewandten Statistik*. Oldenbourg Wissenschaftsverlag, München, Deutschland, 2005, ISBN 978-3-486-57890-4.

[HHN11] Reinhold Hatzinger, Kurt Hornik und Herbert Nagel: *R - Einführung durch angewandte Statistik*. Pearson Studium, München, Deutschland, 2011, ISBN 978-3-86894-060-2.

[Jen] *Jenkins*. `http://jenkins-ci.org/`, besucht: 01.02.2012.

[JGi] *JGit*. `http://eclipse.org/jgit/`, besucht: 28.02.2012.

[JMe] *Apache JMeter*. `http://jmeter.apache.org/`, besucht: 03.02.2012.

[JPD02] *Java Platform Debugger Architecture Enhancements*, 2002. `http://docs.oracle.com/javase/1.4.2/docs/guide/jpda/enhancements.html`, besucht: 14.12.2011.

[JUn] *JUnitPerf*. `http://clarkware.com/software/JUnitPerf.html`, besucht: 09.02.2012.

[JVM03] *JavaTM Virtual Machine Debug Interface Reference*, 2003. http://docs.oracle.com/javase/1.4.2/docs/guide/jpda/jvmdi-spec.html#RedefineClasses, besucht: 14.12.2011.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William Griswold: *An Overview of AspectJ*. In: Jørgen Knudsen (Herausgeber): *ECOOP 2001 - Object-Oriented Programming*, Band 2072 der Reihe *Lecture Notes in Computer Science*, Seiten 327–354. Springer Berlin / Heidelberg, 2001, ISBN 978-3-540-42206-8. http://dx.doi.org/10.1007/3-540-45337-7_18.

[KKR08] Michael Kuperberg, Martin Krogmann und Ralf Reussner: *ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations*. In: *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation , Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)*, 2008.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean Marc Loingtier und John Irwin: *Aspect-oriented programming*. In: Mehmet Aksit und Satoshi Matsuoka (Herausgeber): *ECOOP'97 - Object-Oriented Programming*, Band 1241 der Reihe *Lecture Notes in Computer Science*, Seiten 220–242. Springer Berlin / Heidelberg, 1997, ISBN 978-3-540-63089-0. http://dx.doi.org/10.1007/BFb0053381, 10.1007/BFb0053381.

[Kou10] Samuel Kounev: *Performance Engineering of Enterprise Software Systems*. Vorlesung, 2010. http://sdqweb.ipd.kit.edu/wiki/Vorlesung_Performance_Engineering_of_Enterprise_Software_Systems_SS2010, besucht: 06.11.2011.

[Kre] Jay Kreps: *Project Voldemort: Scaling Simple Storage at LinkedIn*. http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin/, besucht: 16.02.2012.

[Lin] *LinkedIn*. http://www.linkedin.com/, besucht: 18.02.2012.

[LP08] Jonathan M. Lambert und James F. Power: *Platform Independent Timing of Java Virtual Machine Bytecode Instructions*. Electron. Notes Theor. Comput. Sci., 220:97–113, December 2008, ISSN 1571-0661. http://dl.acm.org/citation.cfm?id=1480238.1480259.

[Mav] *Apache Maven*. http://maven.apache.org/, besucht: 01.02.2012.

[Mef06] Klaus Meffert: *JUnit Profi-Tipps*. entwickler.press, 2006, ISBN 3-935042-76-0.

[MK08] Nagy Mostafa und Chandra Krintz: *Offline Framework for Performance Comparison of Software Revisions*. http://www.cs.ucsb.edu/research/tech_reports/reports/2008-19.pdf, besucht: 11.11.2011, Ongoing work that has not yet been published., November 2008.

[MK09] Nagy Mostafa und Chandra Krintz: *Tracking performance across software revisions*. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, Seiten 162–171, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-598-7. http://doi.acm.org/10.1145/1596655.1596682.

[MM08] Alexander V. Mirgorodskiy und Barton P. Miller: *Diagnosing distributed systems with self-propelled instrumentation.* In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, Seiten 82–103, New York, NY, USA, 2008. Springer-Verlag New York, Inc., ISBN 3-540-89855-7. http://dl.acm.org/citation.cfm?id=1496950.1496957.

[Pop08] Gunther Popp: *Konfigurationsmanagement mit Subversion, Ant und Maven.* dpunkt.verlag, 2008, ISBN 978-3-89864-487-7.

[Sma11] John Ferguson Smart: *Jenkins: The Definitive Guide.* Wakaleo Consulting, 2011. http://www.wakaleo.com/books/jenkins-the-definitive-guide, besucht: 01.02.2012.

[Son11] Sonatype: *Maven: The Complete Reference.* Sonatype, Inc., 2011. http://www.sonatype.com/books/mvnref-book/reference/index.html, besucht: 01.02.2012.

[SW01] Connie Smith und Lloyd Williams: *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2001, ISBN 0-201-72229-1.

[Tic82] Walter F. Tichy: *Design, implementation, and evaluation of a Revision Control System.* In: *Proceedings of the 6th international conference on Software engineering*, ICSE '82, Seiten 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press. http://dl.acm.org/citation.cfm?id=800254.807748.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey und Dennis Kieselhorst: *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework.* Technischer Bericht TR-0921, Department of Computer Science, University of Kiel, Germany, November 2009. http://www.informatik.uni-kiel.de/ifi/forschung/technische-berichte/bericht/?tx_publication_pi1%5Bsingle%5D=168.

[Vol] *Project Voldemort.* http://project-voldemort.com/, besucht: 16.02.2012.

[Wer12] Alexander Wert: *Uncovering Performance Anti-Patterns by Systematic Experiments*, 2012. Diplomarbeit.

[WHHH10] Dennis Westermann, Jens Happe, Michael Hauck und Christian Heupel: *The Performance Cockpit Approach: A Framework for Systematic Performance Evaluations.* In: *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, Seiten 31–38. IEEE Computer Society, 2010.

Abbildungsverzeichnis

2.1. Architektur des Performance-Cockpits nach [WHHH10] 6
2.2. Beispiel: Ausführungszeitpunkte 8
2.3. Kieker Architektur basierend auf [vHRH+09] und [EvHWH11] 9
2.4. Sequenzdiagramm Kieker 11
2.5. Beispiel: Selbstreflektierendes Laden 14
2.6. Überblick über den ByCounter-Prozess basierend auf [KKR08] 15
2.7. Informationsverwaltung anderer VKS wie z.B. in Subversion [Cha09] 18
2.8. Informationsverwaltung von Git [Cha09] 18
2.9. Daten in der Speicherablage von Git [Cha09] 19
2.10. Historie in Git nach dreimaligem Aufgeben von Änderungen [Cha09] 19
2.11. Illustration des Zweigkonzeptes [Cha09] 20
2.12. Bestandteile eines Systems zur kontinuierlichen Integration [DMG07] 21
2.13. Das Projektmodell [Son11] 23
2.14. Beispiel einer Kasten-Zeichnung 27

4.1. Integration des Ansatzes in eine moderne Entwicklungsumgebung 36
4.2. Beispiel 38
4.3. Beispiel – Aufbau des Aufrufbaumes 40
4.4. Beispiel – Software-Version r_i 42
4.5. Beispiel – Software-Version r_j 42
4.6. Gespeicherte Änderungen im VKS 44
4.7. Sequenzdiagramm Instrumentierung 45
4.8. Einbindung in die automatisierte Übersetzung 46
4.9. Einbindung in die kontinuierliche Integration 47

5.1. Schichtenarchitektur des Prototyps 51
5.2. Sequenzdiagramm Regressionserkennung 52
5.3. Sequenzdiagramm Problemdiagnose 53
5.4. Sequenzdiagramm Systemvermessung 55

6.1. Sequenzdiagramm Testsystem 58
6.2. Szenario 1 61
6.3. Szenario 2 61
6.4. Szenario 3 62
6.5. Szenario 4 62
6.6. Szenario 5 64
6.7. Szenario 6 65

7.1. Voldemort Architektur 68
7.2. Fehlerkennung – `testSorting(int)` 70
7.3. Regressionserkennung – `testByteComparison()` 71
7.4. Problemdiagnose 73

A.1. Differenz `ByteUtilsTest.java` 85

Tabellenverzeichnis

2.1. Antwortzeitverhalten von 137 JVM Instruktionen [LP08] 16
2.2. 13 als Ausreißer identifizierte Instruktionen [LP08] 16
2.3. Aufbereitete Darstellung der Messwerte 28

6.1. Werte für n. 58
6.2. Statistische Kenngrößen 60

7.1. Anzahl verwendeter Unit-Tests 69

A.1. Verwendete Unit-Tests 86

Anhang

A. Evaluation

```
---------------- test/unit/voldemort/utils/ByteUtilsTest.java ----------------
diff --git a/test/unit/voldemort/utils/ByteUtilsTest.java b/test/unit/voldemort/utils/
ByteUtilsTest.java
index 71d69a0..7fee09a 100644
--- a/test/unit/voldemort/utils/ByteUtilsTest.java
+++ b/test/unit/voldemort/utils/ByteUtilsTest.java
@@ -165,6 +165,23 @@
         assertEquals(0, ByteUtils.compare(new byte[] { -1 }, new byte[] { -1 }));
         assertEquals(1, ByteUtils.compare(new byte[] { -1 }, new byte[] { -4 }));
         assertEquals(-1, ByteUtils.compare(new byte[] { -4 }, new byte[] { -1 }));
-    }

+        // Tests byte comparison with sub-sequence
+        assertEquals(1, ByteUtils.compare(new byte[] { 0 }, new byte[] { 0 }, 0, 0));
+        assertEquals(0, ByteUtils.compare(new byte[] { 0 }, new byte[] { 0 }, 0, 1));
+        try {
+            assertEquals(0, ByteUtils.compare(new byte[] { 0 }, new byte[] { 0 }, 0, 2));
+            fail("Should have thrown an exception");
+        } catch(IllegalArgumentException e) {}
+        assertEquals(-1, ByteUtils.compare(new byte[] { 2 }, new byte[] { 0, 1, 2, 3 }, 3, 4));
+        assertEquals(0, ByteUtils.compare(new byte[] { 3 }, new byte[] { 0, 1, 2, 3 }, 3, 4));
+        assertEquals(1, ByteUtils.compare(new byte[] { 5 }, new byte[] { 0, 1, 2, 3 }, 3, 4));
+
+        assertEquals(0, ByteUtils.compare(new byte[] { 2, 3 }, new byte[] { 0, 1, 2, 3 }, 2, 4));
+        try {
+            assertEquals(0, ByteUtils.compare(new byte[] { 2, 3 }, new byte[] { 0, 1, 2, 3 }, 2, 5));
+            fail("Should have thrown an exception");
+        } catch(IllegalArgumentException e) {}
+        assertEquals(1, ByteUtils.compare(new byte[] { 5 }, new byte[] { 0, 1, 2, 3 }, 2, 4));
+    }
 }
```

Abbildung A.1.: Differenz `ByteUtilsTest.java`

Unit-Test $=voldemort	Software-Version 0.90.1	0.90	0.81
$.protocol.pb.ProtocolBuffersRequestFormatTest	x	x	x
$.routing.ZoneRoutingStrategyTest	x	x	-
$.serialization.VersionedSerializerTest	x	x	x
$.utils.pool.KeyedResourcePoolTest	x	x	x
$.xml.ClusterMapperTest	x	x	x
$.cluster.failuredetector.ThresholdFailureDetectorTest	x	x	x
$.store.readonly.ReadOnlyUtilsTest	x	x	x
$.store.memory.InMemoryStorageEngineTest	x	x	x
$.scheduled.DataCleanupJobTest	x	x	x
$.versioning.VectorClockTest	x	x	x
$.store.stats.StoreStatsJmxTest	x	x	-
$.store.stats.StatsTest	x	x	-
$.store.routed.NodeValueTest	x	x	x
$.utils.UtilsTest	x	x	x
$.utils.ReflectUtilsTest	x	x	x
$.serialization.avro.AvroReflectiveSerializerTest	x	x	x
$.serialization.thrift.ThriftSerializerTest	x	x	x
$.serialization.avro.AvroGenericSerializerTest	x	x	x
$.utils.IoThrottlerTest	x	x	x
$.store.readonly.ExternalSorterTest	x	x	x
$.store.memory.CacheStorageEngineTest	x	x	x
$.store.readonly.chunk.DataFileChunkSetIteratorTest	x	x	-
$.serialization.SlopSerializerTest	x	x	x
$.serialization.protobuf.ProtoBufSerializerTest	x	x	x
$.store.slop.strategy.ConsistentHandoffStrategyTest	x	x	-
$.versioning.ClockEntryTest	x	x	x
$.store.readonly.ReadOnlyStorageMetadataTest	x	x	-
$.store.views.ViewStorageEngineTest	x	x	x
$.store.serialized.SerializingStoreTest	x	x	x
$.serialization.json.JsonTypeSerializerTest	x	x	x
$.store.logging.LoggingStoreTest	x	x	x
$.versioning.InconsistentDataExceptionTest	x	x	x
$.utils.CachedCallableTest	x	-	x
$.server.ServiceTest	x	x	x
$.serialization.json.JsonReaderTest	x	x	x
$.versioning.VersionedTest	x	x	x
$.cluster.failuredetector.BannagePeriodFailureDetectorTest	x	x	x
$.utils.NetworkClassLoaderTest	x	x	x
$.store.rebalancing.RebootstrappingStoreTest	x	x	-
$.utils.RebalanceUtilsTest	x	x	x
$.store.slop.SlopTest	x	x	-
$.scheduled.StreamingSlopPusherTest	x	x	-
$.serialization.avro.AvroSpecificSerializerTest	x	x	x
$.store.configuration.ConfigurationStorageEngineTest	x	x	x
$.versioning.VectorClockInconsistencyResolverTest	x	x	x
$.utils.ByteUtilsTest	x	x	x
$.routing.ConsistentRoutingStrategyTest	x	x	x

Tabelle A.1.: Verwendete Unit-Tests