Automatisierung der Durchführung und Auswertung von Mikrobenchmarks in Continuous-Integration-Systemen

Bachelorarbeit

Martin Zloch

27. März 2014

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
Dipl.-Inf. Jan Waller

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

In dieser Arbeit entwickle ich ein System zur automatischen Durchführung und Auswertung von Mikrobenchmarks. Dieses System baut auf Continuous-Integration-Systemen auf und ist durch seinen modularen Aufbau so flexibel gestaltet, dass es sich auf eine große Anzahl von Software anwenden lässt. Das entwickelte System wird dann für das CI-System Jenkins implementiert, um automatisch Mikrobenchmarks zur Berechnung des Overheads des Monitoring-Frameworks Kieker auszuführen. Die Implementierung lässt sich durch komponentenweisen Austausch einfach für andere Projekte, die ebenfalls Jenkins verwenden, anpassen und erweitern. Die abschließenden Tests des Systems und der Implementierung sind Bestandteil dieser Arbeit und verdeutlichen die Umsetzung der zu Beginn festgelegten Ziele.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziele . . . 2
1.2.1 Anforderungen an das System . . . 3
1.2.2 Anforderungen an die Implementierung . . . 4
1.2.3 Ziel der Evaluierung . . . 4
1.3 Aufbau . . . 4

2 Grundlagen und Technologien **7**
2.1 Grundlagen . . . 7
2.1.1 Monitoring-Frameworks . . . 7
2.1.2 Mikrobenchmarks . . . 7
2.1.3 Continuous Integration . . . 8
2.1.4 Analysemethoden . . . 8
2.2 Technologien . . . 9
2.2.1 Kieker . . . 9
2.2.2 MooBench . . . 10
2.2.3 Jenkins . . . 10
2.2.4 Plot-Plugin . . . 10
2.2.5 OpenCSV . . . 11
2.2.6 Commons Math . . . 11
2.2.7 Jelly . . . 11

3 Entwicklung eines Systems zum Einbinden von Benchmarks in CI-Systeme **13**
3.1 Voraussetzungen . . . 13
3.1.1 Phasen . . . 13
3.1.2 Abstraktion . . . 14
3.1.3 Vergleich der Ergebnisse von Benchmarks . . . 14
3.2 Komponenten . . . 15
3.3 Organisation von Daten . . . 17
3.4 Integration . . . 18
3.5 Ablauf . . . 19
3.6 Parallelisierung der Durchführung . . . 21
3.7 Vergleichbarkeit der Ergebnisse . . . 23
3.8 Zusammenfassung . . . 25

4 Beispielhafte Implementierung **27**
4.1 Einleitung . . . 27
4.2 Plug-in-Entwicklung für Jenkins . . . 27
4.3 Umsetzung der Komponenten . . . 29
4.3.1 Datenspeicher . . . 30
4.3.2 Durchführung . . . 32
4.3.3 Analyse . . . 38
4.3.4 Wächter . . . 39
4.3.5 Darstellung . . . 39
4.4 Erweiterung des Systems . . . 40
4.5 Installation und Konfiguration . . . 41
4.6 Zusammenfassung . . . 46

5 Evaluierung **47**
5.1 Entwicklung von Evaluierungsmetriken . . . 47
5.1.1 Evaluierungsmetriken für das System . . . 48
5.1.2 Evaluierungsmetriken für die Implementierung . . . 50
5.2 Evaluierung des Systems . . . 52
5.2.1 Plattformunabhängigkeit . . . 52
5.2.2 Konfigurierbarkeit . . . 52
5.2.3 Ausführung von Benchmarks . . . 53
5.2.4 Speichern der Ergebnisse . . . 53
5.2.5 Weiterverarbeitung der Ergebnisse . . . 53
5.2.6 Kontrolle der verarbeiteten Ergebnisse . . . 53
5.2.7 Präsentation der Ergebnisse . . . 54
5.2.8 Vergleichbarkeit der Ergebnisse . . . 54
5.3 Evaluierung der Implementierung . . . 54
5.3.1 Plattformunabhängigkeit . . . 55
5.3.2 Konfigurierbarkeit . . . 56
5.3.3 Ausführung von Benchmarks . . . 56
5.3.4 Speichern der Ergebnisse . . . 57
5.3.5 Weiterverarbeitung der Ergebnisse . . . 58
5.3.6 Kontrolle der verarbeiteten Ergebnisse . . . 58
5.3.7 Präsentation der Ergebnisse . . . 59
5.3.8 Vergleichbarkeit der Ergebnisse . . . 59
5.3.9 Installierbarkeit . . . 62
5.3.10 Automatische Ausführung von MooBench . . . 62
5.3.11 Portierbarkeit . . . 63
5.4 Zusammenfassung . . . 63

6 Verwandte Arbeiten **65**
6.1 BEEN 65
6.2 Performance Plugin 65
6.3 KoPeMe 66

7 Fazit und Ausblick **67**
7.1 Fazit 67
7.2 Ausblick 67

Bibliografie **69**

Kapitel 1

Einleitung

Diese Arbeit behandelt die Automatisierung der Durchführung und Auswertung von Mikrobenchmarks zur automatischen Messung der Performance einer Software während ihrer Entwicklung. In diesem Rahmen entwickle ich ein System, welches die Durchführung, Verarbeitung und Auswertung von Mikrobenchmarks in Continuous-Integration-Systeme ermöglicht und dabei so gestaltet ist, dass es für verschiedene CI-Systeme und Software anwendbar ist. Dieses System ist komponentenweise aufgebaut und ermöglicht die Durchführung von Mikrobenchmarks, die Analyse der Ergebnisse, das Speichern der Benchmark-Ergebnisse und der Analyseergebnisse, die automatische Kontrolle der Ergebnisse und die Präsentation der Ergebnisse.
Anschließend stelle ich eine Implementierung des Systems für das Continuous-Integration-System Jenkins vor. Diese Implementierung ist in der Lage, die Durchführung von MooBench-Benchmarks zur Messung des Monitoring-Overheads des Monitoring–Frameworks Kieker in den Build-Prozess einzubinden. Dazu integriere ich die bisher verwendeten MooBench-Skripte sowie andere Komponenten, die der Verarbeitung der Ergebnisse dienen, in ein Plug-in für Jenkins.
Die Evaluierung des Systems und der Implementierung zeigt dann, ob die gesetzten Ziele erreicht wurden. Für die Durchführung der Evaluierung entwickle ich geeignete Metriken mit Hilfe der Goal-Question-Metric-Methode [Basili u. a. 1994] und wende diese Metriken anschließend an.
Dieses Kapitel führt eine Motivation an, die den Nutzen der Arbeit vermittelt. Außerdem erläutere ich die Grundlagen dieser Arbeit und erkläre den Aufbau.

1.1. Motivation

Die Hauptaufgabe des Software-Engineering ist die Steigerung der Qualität von Software. Hierzu soll der Entwicklungsprozess verbessert werden, um dadurch das Produkt zu verbessern. Ein wichtiges Qualitätsmerkmal von Software ist Performance [ISO/IEC 2010; Ghezzi u. a. 2002], die häufig darüber entscheidet, ob Software verwendbar und konkurrenzfähig ist. Beispiele für stark Performance-abhängige Software sind Monitoring-Tools, welche eingesetzt werden können, um den Überblick über das Verhalten einzelner Komponenten einer Software zu überwachen. Diese Monitoring Tools, wie zum Beispiel das Kieker Framework [van Hoorn u. a. 2012a], erzeugen durch ihre Ausführung zusätzlich zur

beobachteten Software einen Overhead, welcher das Verhalten der überwachten Software beeinflussen kann. Dieser Overhead soll möglichst gering gehalten werden, um die Qualität der beobachteten Software nicht zu beeinträchtigen.
Um die Performance einer Software zu messen, können Mikrobenchmarks verwendet werden, die die Laufzeit einzelner Methoden messen. Da sich jedoch die Performance häufig während des Entwicklungsprozesses ändert, und dies auch Ziel von Änderungen sein kann, müssen zur Überwachung dieser Veränderungen wiederholt Benchmark-Experimente ausgeführt werden. Die manuelle wiederholte Ausführung verbraucht jedoch Ressourcen, die durch eine Automatisierung des Vorgangs eingespart werden können.
Um Vorgänge in der Software-Entwicklung zu automatisieren, können Continuous-Integration-Systeme eingesetzt werden, welche häufig verwendet werden, um Build- und Test-Vorgänge durchzuführen. Da diese Systeme weit verbreitet sind, bietet es sich an, die Automatisierung von Performance-Messungen in darüber verwaltete Build-Prozesse zu integrieren. Dadurch wird automatisch eine Versionshistorie der Performance erstellt, sodass zu jedem Zeitpunkt der Entwicklung ein einfacher Einblick in die Entwicklung der Performance möglich ist.

1.2. Ziele

Das Hauptziel des Software-Engineerings ist es, qualitativ hochwertige Software [ISO/IEC 2010] herzustellen. Zu diesem Zweck werden Mittel und Methoden entwickelt, um die interne Qualität, also den Aufbau und den Entwicklungsprozess, zu steigern. Die Annahme ist, dass dadurch auch die externe Qualität steigt, also zum Beispiel die Performance der entwickelten Software. Um die Performance einer Software zu messen, können unter anderem Benchmarks eingesetzt werden. Diese messen konkrete Laufzeiten einzelner Funktionen und liefern so eine quantitative Angabe über die Performance. Um also die Entwicklung der Qualität einer Software während des Entstehungsprozesses zu beobachten, können die Ergebnisse von Benchmarks verschiedener Revisionen verglichen werden. Dies erlaubt natürlich nur die Betrachtung der Performance. Diesen Vorgang zu automatisieren ist das Hauptziel dieser Arbeit.
Um dieses Ziel umzusetzen, wird eine Plattform benötigt, die das automatische Ausführen der Benchmarks ermöglicht. Hierzu passe ich Continuous-Integration-Systeme so an, dass nach jedem Build-Vorgang für die aktuelle Revision konfigurierbare Benchmark-Experimente durchgeführt werden und entwickle erst ein allgemeines System, wie die Automatisierung der Durchführung und Auswertung von Benchmarks in CI-Systemen umgesetzt werden kann. Dieses System implementiere ich dann exemplarisch für das CI-System Jenkins und bewerte anschließend den Entwurf des Systems und die Implementierung.

1.2.1. Anforderungen an das System

Plattformunabhängigkeit

Das System soll möglichst unabhängig von der Art des Continuous-Integration-Systems funktionieren. Dies erfordert eine starke Kapselung bei der Entwicklung des Systems und führt im Gegenzug zu hoher Flexibilität in der Einsetzbarkeit.

Konfigurierbarkeit

Um die Bedienung einer Implementierung des Systems zu erleichtern, soll es Möglichkeiten geben, das System innerhalb der Oberfläche des Continuous-Integration-Systems zu konfigurieren. Zu den Konfigurationsmöglichkeiten gehört insbesondere, welche Benchmarks ausgeführt werden und welche Methoden zur Verarbeitung, Kontrolle und Präsentation genutzt werden.

Ausführen von Benchmarks

Die erste funktionale Anforderung ist die automatische Durchführung von Benchmarks. Um hierbei eine hohe Flexibilität zu erreichen, soll das System unabhängig davon sein, welche Methode zum Ausführen der Benchmarks benutzt wird, solange bestimmte Kriterien eingehalten werden. Die Kriterien sollen hierbei möglichst tolerant sein, um die Flexibilität nicht unnötig einzuschränken. Das System soll demzufolge auch keine Methode beinhalten, wie Benchmarks durchzuführen sind.

Speichern der Ergebnisse

Die Ergebnisse von Benchmark-Experimenten sollen für spätere Vergleiche gespeichert werden. Dies führt dazu, dass einmal durchgeführte Experimente nicht für jeden Vergleich erneut durchgeführt werden müssen und ermöglicht eine starke Verbesserung der Performance des Systems.

Weiterverarbeitung der Ergebnisse

Falls die mit den Benchmark-Experimenten gewonnen Daten vor der weiteren Auswertung verarbeitet werden sollen, muss das System Möglichkeiten bereitstellen, solche Schritte einzufügen. Diese Weiterverarbeitungsmethoden sollen unabhängig von der Art der Benchmarks auf die Ergebnisse angewandt werden können.

Kontrolle der verarbeiteten Ergebnisse

Um die durch die Weiterverarbeitung entstandenen Ergebnisse hinsichtlich beliebiger Kriterien prüfen zu können, soll das System die Ergänzung des Benchmark-Prozesses um solche Kontrollen ermöglichen.

Präsentation der verarbeiteten Ergebnisse

Die verarbeiteten Ergebnisse sollen für menschliche Betrachter geeignet präsentiert werden, damit eine manuelle Kontrolle der Ergebnisse möglich ist.

Vergleichbarkeit der Ergebnisse

Da die Ergebnisse eines Benchmark-Experiments stark von dem Umfeld abhängen, in dem das Experiment durchgeführt wird, und die Daten meist nur im Vergleich eine Bedeutung haben, muss das System die Vergleichbarkeit der Ergebnisse gewährleisten können.

1.2.2. Anforderungen an die Implementierung

Grundsätzlich gelten für die Implementierung die gleichen Anforderungen wie für das System, da die Implementierung das System möglichst genau abbilden sollte. Zusätzlich stelle ich folgende weitere Anforderungen:

Installierbarkeit

Um eine einfache Benutzung zu ermöglichen, soll die Implementierung leicht zu installieren sein.

Automatische Ausführung von MooBench

Als Beispiel für die Methode der Benchmarks soll die Implementierung MooBench-Experimente ausführen.

Portierbarkeit

Die Implementierung soll sich möglichst leicht an beliebige Software-Typen anpassen lassen.

1.2.3. Ziel der Evaluierung

Das Ziel der Evaluierung ist die abschließende Bewertung des Systems und der Implementierung. Hierbei verwende ich die Goal-Question-Metric-Methode, um geeignete Metriken zur Beurteilung zu entwickeln. Insbesondere bewerte ich die Erfüllung der oben genannten Anforderungen. Diese Bewertung dient der Bestimmung von Schwachstellen des Systems und der Implementierung und soll so mögliche zukünftige Entwicklungsschritte aufzeigen.

1.3. Aufbau

Um die in Abschnitt 1.2 vorgestellten Ziele zu erreichen, sind bestimmte Grundlagen und Technologien erforderlich, wie zum Beispiel Continuous Integration und Kieker. Diese stelle ich in Kapitel 2 vor und erkläre deren Funktion in dieser Arbeit. Anhand der theoretischen Grundlagen entwickle ich in Kapitel 3 ein generelles System zur Automatisierung von Durchführung und Auswertung von Mikrobenchmarks. Dafür erkläre ich den Aufbau und gehe auf die Möglichkeiten zur Parallelisierung ein. Außerdem beschäftige ich mich mit dem Thema der Vergleichbarkeit von Benchmark-Ergebnissen. In Kapitel 4 beschreibe

ich eine Implementierung des Systems für das CI-System Jenkins zur Ausführung von MooBench zur Messung des Overheads von Kieker. Die Umsetzung der Datenspeicherung und eine Neuimplementierung der MooBench-Skripte sowie die Integration der Komponenten in Jenkins stehen hierbei im Vordergrund. Anschließend bewerte ich das entwickelte System sowie seine Implementierung in Kapitel 5 hinsichtlich der in Abschnitt 1.2 gesetzten Ziele und vergleiche in Kapitel 6 das von mir entwickelte System mit anderen, bereits existierenden. Abschließend ziehe ich ein kurzes Fazit aus der Arbeit und beschreibe einen Ausblick auf künftige Forschungsziele in Kapitel 7.

Kapitel 2

Grundlagen und Technologien

2.1. Grundlagen

Um die Automatisierung der Durchführung und Auswertung von Benchmarks zu ermöglichen, sind verschiedene Grundlagen erforderlich, auf welchen ich in dieser Arbeit aufbauen werde. Dazu gehören Monitoring-Frameworks, Mikrobenchmarks, Continuous Integration, Analyseverfahren und weitere. In diesem Kapitel beschreibe ich diese Grundlagen und wozu ich sie in der Arbeit verwende.

2.1.1. Monitoring-Frameworks

Monitoring-Frameworks [Gao u. a. 2000] werden eingesetzt, um bei immer größeren Softwarekomplexen einen Überblick über das Verhalten der Software zu bekommen, wenn die Dokumentation nicht vorhanden ist oder nicht ausreicht. Ein weiterer Anwendungsbereich ist die Überprüfung der Performance einer beobachteten Software, um Flaschenhälse identifizieren und beheben zu können. Bei der Anwendung wird die Monitoring-Software parallel zur beobachteten Software ausgeführt und misst verschiedene Aspekte, wie zum Beispiel Datendurchsatzraten, die Häufigkeit einzelner Methodenaufrufe und Laufzeiten. Durch die parallele Ausführung entsteht zusätzlicher Bedarf an Rechenkapazität, welcher die Performance der beobachteten Software verringern kann. Diesen Mehraufwand nennt man Monitoring-Overhead und möchte ihn möglichst gering halten, um die Qualität der beobachteten Software nicht zu stark zu beeinflussen. Hierzu ist es notwendig, dass das Monitoring-System möglichst performant arbeitet.
Ein Beispiel für ein solches Monitoring-System ist das Kieker-Framework, welches ich in Unterabschnitt 2.2.1 vorstelle. Die in Kapitel 4 vorgestellte Implementierung des in Kapitel 3 entwickelten Systems hat als Ziel, den Monitoring-Overhead des Kieker-Frameworks zu messen.

2.1.2. Mikrobenchmarks

Benchmarks [Jain 2008] werden verwendet, um Performance zu messen. Häufig haben Benchmarks im Computer-Bereich die Aufgabe, die Leistungsfähigkeit der Hardware eines Computersystems zu bestimmen, um die Geschwindigkeit vergleichend beurteilen zu können. Eine für solche Benchmarks bekannte Firma ist FutureMark welche eine ganze

Reihe von Benchmarks für verschiedenste Systeme anbietet.
Mikrobenchmarks hingegen werden zur Performance-Messung von Software verwendet. Sie tragen ihren Namen, da sie nur verhältnismäßig kleine Komponenten einer Software testen, und nicht ganze Systeme. Häufig werden auch nur Laufzeiten von einzelnen Methoden gemessen. Dies geschieht um die Qualität einer Software durch die Steigerung ihrer Performance zu heben. Durch Mikrobenchmarks können während der Entwicklung einer Software Veränderungen in der Performance gemessen werden, sodass gezielt auf die Steigerung derselben hingearbeitet oder zumindest eine ungewollte Verringerung vermieden werden kann.
Um diesen Vorgang zu vereinfachen, entwickle ich in dieser Arbeit ein System, um Mikrobenchmarks automatisch durchzuführen und auszuwerten. Als Beispiel für dieses System implementiere ich eine automatische Version von MooBench, einem Mikrobenchmark zur Bestimmung des Monitoring-Overheads des Monitoring-Frameworks Kieker.

2.1.3. Continuous Integration

Continuous Integration [Duvall u. a. 2007] beschreibt eine Methode der Software-Entwicklung, bei der die Software während des Entstehungsprozesses wiederholt zusammengefügt und gebaut wird, um inkrementelle Software-Entwicklung zu unterstützen. Systeme, die diese Methode implementieren, heißen Continous-Integration-Systeme, oder kurz CI-Systeme. CI-Systeme ermöglichen häufig die automatische Ausführung des Bauvorgangs sowie von anschließenden Tests, sodass sie sich als Ansatzpunkt zur Automatisierung von Vorgängen, die in diesen Prozess eingebunden werden sollen, gut eignen. Die meisten Continous-Integration-Systeme sind zudem erweiterbar, entweder dadurch, dass der Quellcode öffentlich zugänglich ist, oder durch Erweiterungskomponenten, die auch als Plug-ins bezeichnet werden.
Um die Durchführung von Mikrobenchmarks zu automatisieren, verwende ich das CI-System Jenkins, welches ich in Unterabschnitt 2.2.3 vorstelle. Zu diesem Zweck implementiere ich ein Plugin, welches Funktionen zur Durchführung und Auswertung von Mikrobenchmarks beinhaltet.

2.1.4. Analysemethoden

Die Ergebnisse von Mikrobenchmarks hängen stark von der Umgebung ab, in der sie durchgeführt werden. Selbst kleinste Veränderungen im Umfeld können dazu führen, dass sich die Laufzeit einer Methode stark ändert. Deshalb wird die Laufzeit einer Methode durch Mikrobenchmarks wiederholt gemessen, um dann mit statistischen Mitteln mögliche Laufzeiten berechnen zu können. Die folgenden Mittel verwende ich in dieser Arbeit, um Benchmark-Ergebnisse zu analysieren:

Median

Der Median ist der Wert, der bei der Sortierung der gesamten Eingabewerte in der Mitte steht. Er ist ein tatsächlich vorkommender Wert und lässt eine Aussage über die Verteilung der Ergebnisse zu, da eine Hälfte des Ergebnisses größer, und eine kleiner ist. Er ist als 50-Quantil ein Spezialfall der Quantilen.

Durchschnitt

Der Durchschnitt ist das statistische Mittel der gesamten Werte. Im Gegensatz zu den Quantilen hat er eine vergleichbar höhere Aussagekraft, muss aber nicht als reales Ergebnis vorkommen. Eine hohe Schwankung der Vorkommen der Werte je Größe kann außerdem zu einem stark verfälschten Wert führen.

Konfidenzintervalle

Konfidenzintervalle des Durchschnitts geben an, welcher Wert ausreicht, sodass mit einer bestimmten Wahrscheinlichkeit alle jemals vorkommenden Werte um maximal diesen Wert vom Durchschnitt abweichen. Häufig wird als Wahrscheinlichkeitswert 95% gewählt, da dies als genau genug erachtet wird. Dieser Wert lässt eine Aussage über die Genauigkeit des Durchschnittswertes zu. Bei einem großen Konfidenzintervall schwanken die Werte sehr stark, sodass es unwahrscheinlicher ist, dass ein tatsächlicher Wert dem Durchschnitt entspricht.

Quantile

Die Quantile einer Testmenge wird anhand einer Zahl p bestimmt. Sie ermittelt einen Wert aus der Testmenge, sodass p% der Testmenge kleiner als das Ergebnis sind.

2.2. Technologien

In Abschnitt 2.1 sind die theoretischen Grundlagen für diese Arbeit beschrieben. Die dort beschriebenen Systeme und Methoden haben meist viele, sehr verschiedene, Repräsentationen in konkreter Technologie. In diesem Kapitel stelle ich die verwendeten Technologien vor und begründe die Entscheidungen, die zur Wahl des jeweiligen Vertreters eines Systems oder einer Methode geführt haben.

2.2.1. Kieker

Das Kieker-Framework [van Hoorn u. a. 2009; 2012, b; Rohr u. a. 2008] ist eine Software zur Beobachtung des Verhaltens von Software zur Laufzeit und gehört somit zu den in Unterabschnitt 2.1.1 beschriebenen Monitoring-Frameworks. Die Software zeichnet sich durch einen modularen Aufbau aus, der vielfältige Anwendungsgebiete ermöglicht und sowohl der Quellcode als auch Studien und Experimente sind frei zugänglich. Bei der Entwicklung von Kieker wird großen Wert auf einen geringen Monitoring-Overhead gelegt,

sodass die Automatisierung von Benchmarks zur Messung des Overheads einen großen Vorteil bietet, da der manuelle Aufwand stark reduziert werden kann.

2.2.2. MooBench

Moobench [Waller und Hasselbring 2013; MooBench Scripts] ist ein Benchmark-System, um Benchmarks für das Monitoring-Framework Kieker auszuführen. Es besteht aus einer Ausführbaren Java-Datei, welche nach dem Starten wiederholt eine Dummy-Methode ausführt, und verschiedenen Shell-Skripten, welche die Java-Datei von Kieker überwachen lassen. Dazu wird Kieker als Java-Agent gestartet. Die Dummy-Methoden haben hierbei in der Regel nur geringe Laufzeit, sodass die gemessenen Zeiten hauptsächlich durch den Overhead durch Kieker verursacht werden. Durch dieses Verfahren lassen sich relativ genaue Aussagen über den Overhead von Kieker treffen. In dieser Arbeit implementiere ich die Shell-Skripte neu, um eine bessere Integration in das in Kapitel 3 vorgestellte System zu ermöglichen. Die genauen Details hierzu, wie auch eine detailliertere Beschreibung der Skripte, sind Unterabschnitt 4.3.2 zu entnehmen.

2.2.3. Jenkins

Jenkins CI [Jenkins] ist ein webbasiertes Continuous Integration-System, welches nach der Übernahme durch Oracle aus dem bei Sun Microsystems entwickelten Hudson entstand. In Jenkins werden Buildvorgänge als Jobs bezeichnet, die sich aus verschiedenen Aufgaben, wie zum Beispiel Ant oder Maven Goals aufzurufen oder Windows Batch Dateien auszuführen, zusammenstellen lassen. Diese Aufgaben werden dann bei jeder Durchführung des Jobs ausgeführt. Die Durchführung von Jobs kann zeitgesteuert, manuell oder durch Abfrage eines Source Code Management Systems erfolgen. Die bereits integrierten Aufgabentypen können durch Plug-ins um weitere ergänzt werden.
Jenkins wurde als CI-System gewählt, weil es momentan als Build-System für das Monitoring-Framework Kieker eingesetzt wird und so eine gute Grundlage bietet, Benchmarks dafür zu automatisieren. Weitere Grundlagen zur Entwicklung von Plug-ins für Jenkins beschreibe ich in Abschnitt 4.2.

2.2.4. Plot-Plugin

Das Plot Plugin [PlotPlugin] für Jenkins fügt die Möglichkeit, Graphen aus verschiedenen Dateitypen wie XML, CSV oder Property Files zu generieren, zu den Aufgaben hinzu. So kann nach jedem Build ein Graph über beliebige Eigenschaften der verschiedenen Revisionen generiert werden. In dieser Arbeit verwende ich es, um die Analyseergebnisse von Benchmarks darzustellen, da es die nötigen Funktionen unterstützt. Die genaue Verwendung ist in Unterabschnitt 4.3.5 dokumentiert, dort befindet sich auch eine Beschreibung, wie eine CSV-Datei mit den anzuzeigenden Eingangsdaten aufgebaut sein muss.

2.2.5. OpenCSV

OpenCSV [OpenCSV] ist eine unter der Apache 2.0 Lizenz veröffentliche Java-Bibliothek, welche die Verwendung von CSV-Dateien unter Java erleichtert. Sie bietet Funktionen zum Auslesen und Editieren von CSV-Dateien, hierbei ist das Trennsymbol frei wählbar. Diese Bibliothek verwende ich, um den Datenspeicher zu realisieren, zur Verwaltung der Ergebnisse von MooBench und um die CSV-Dateien für das Plot-Plugin zu erstellen. Die Integration in Projekte erfolgt über Maven, hierzu wird die pom-Datei um die Abhängigkeit von openCSV ergänzt.

2.2.6. Commons Math

Commons Math [Commons Math] ist ebenfals eine unter der Apache 2.0 Lizenz veröffentliche Java-Bibliothek. Sie bietet verschiedene mathematische Funktionen, die nicht in Java enthalten sind. Für diese Arbeit verwende ich nur das Unterpaket *stat*, welches die Realisierung der in Unterabschnitt 2.1.4 vorgestellten Analysemethoden durch die Klassen *Percentile*, *Median*, *Mean* und *Variance* ermöglicht. Die Einbindung in das Projekt erfolgt ebenfalls über die Maven-Abhängigkeitsverwaltung.

2.2.7. Jelly

Jelly [Abiteboul u. a. 2003] ist eine auf Java und XML basierende Skript-Verarbeitungsmethode, deren Aufgabe es ist, aus XML-Skripten ausführbare Dateien zu erzeugen. Sie wird von Jenkins genutzt, um aus Oberflächenbeschreibungen in Form von Jelly-Skripten die Benutzeroberfläche des Systems zu generieren. Um die implementierten Komponenten des Systems in die Oberfläche zu integrieren, verwende ich Jelly-Skripte wie das folgende:

```
1  <j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler" xmlns:d="jelly:define"
2  xmlns:l="/lib/layout" xmlns:t="/lib/hudson" xmlns:f="/lib/form">
3    <f:entry title="ID of the benchmark" field="benchmarkIdentifier">
4      <f:textbox default="benchmark-id-1"/>
5    </f:entry>
6    <f:entry title="ID of the analysis" field="analysisIdentifier">
7      <f:textbox default="analysis-id-1"/>
8    </f:entry>
9    <f:entry title="ID of the confidence interval" field="ciIdentifier">
10     <f:textbox default="confidence95"/>
11   </f:entry>
12   <f:entry title="Number of warm-up results" field="warmUpCount">
13     <f:textbox default="75000"/>
14   </f:entry>
15 </j:jelly>
```

Der Name eines Jelly-Skripts muss zu diesem Zweck exakt dem der zugehörigen Klasse inklusive Paketname entsprechen. Eingabefelder werden wie im Beispiel ersichtlich durch Textboxen oder Checkboxen dargestellt, deren Eingabewerte dann in die Parameter der Klasse mit dem Namen übernommen wird, der dem *field*-Namen entspricht. In der darzustellenden Klasse sind Getter-Methoden für die entsprechenden Attribute erforderlich und die Eingaben können über zusätzliche Beschreibungsmethoden innerhalb der Klasse überprüft werden.

Kapitel 3

Entwicklung eines Systems zum Einbinden von Benchmarks in CI-Systeme

In diesem Kapitel entwickle ich ein generelles System, um die Durchführung und Analyse von Mikrobenchmarks zu automatisieren. Zu diesem Zweck beschreibe ich eine Methode, Benchmark-Experimente in Continuous-Integration-Systeme zu integrieren und gehe auf verschiedene Aspekte des Systems, wie den Aufbau und Funktionen, die benötigt werden, ein.

3.1. Voraussetzungen

3.1.1. Phasen

In *Kieker: a framework for application performance monitoring and dynamic software analysis* [Waller und Hasselbring 2013] unterteilen die Autoren die Entwicklung von Benchmarks in drei Phasen:

Design / Implementierung

Diese Phase beschreibt das Design der Benchmark-Methode und die Implementierung dieser für die zu testende Software. Häufig werden schon vorhandene Benchmark-Methoden verwendet und lediglich an die Software angepasst oder integriert.

Durchführung

In dieser Phase werden die eigentlichen Experimente durchgeführt. Hierzu wird die im vorherigen Schritt entwickelte Implementierung verwendet, um die Performance einer Software zu messen. Das Ergebnis dieser Phase sind meist Laufzeiten einzelner Funktionen, die pro Funktion untereinander vergleichbar sind und sich im Optimalfall nur geringfügig unterscheiden.

Analyse / Präsentation

Die Ergebnisse der Durchführungsphase werden anschließend mit geeigneten statistischen Methoden, wie zum Beispiel den in Unterabschnitt 2.1.4 beschriebenen, analysiert.

Diese Methoden fassen häufig eine Menge von Laufzeiten zu einem Wert zusammen, sodass für menschliche Betrachter leichter ersichtlich ist, wie sich diese Werte im Vergleich zu anderen verhalten. Diese Werte werden dann geeignet dargestellt, zum Beispiel vergleichend in Form einer Tabelle oder Grafik.

Diese Phasen umfassen die Entwicklung und die Durchführung des Experiments. Hierbei wird aber davon ausgegangen, dass vor jeder Durchführung erst das Komplette System zur Durchführung, Analyse und Präsentation entwickelt und implementiert werden muss. Um dies zu überprüfen und verallgemeinerbare Schritte zu finden, werde ich das oben genannte Vorgehen im folgenden abstrahieren.

3.1.2. Abstraktion

Das Design von Benchmarks ist stark von der getesteten Software und den Zielen des Experiments abhängig. Allgemein lässt sich nur sagen, dass Benchmarks eine bestimmte Zeit lang ausgeführt werden und dann eine Menge von Ergebnissen produziert haben. Um weitere Abstraktion zuzulassen ist es deshalb notwendig, weitere Bedingungen an Benchmarks zu stellen. Daher stelle ich als Anforderung an Benchmarks, die durch das System ausgeführt werden sollen, dass das Ergebnis eine Menge von Fließkommazahlen sein muss.
Auf Basis dieser Voraussetzung kann nun eine Verallgemeinerung der Analyse erfolgen. Während der Analyse wird aus der Menge von Fließkommazahlen ein Analyseergebnis erzeugt, welches leicht mit den Analyseergebnissen anderer Revisionen oder Konfigurationen verglichen werden kann. Dieses Analyseergebnis soll ebenfalls in Form einer Fließkommazahl vorliegen und kann für die Präsentation vergleichend dargestellt werden, zum Beispiel in Form eines Diagramms oder einer Tabelle.

Diese Abstraktion lässt nun zu, dass sich die Ergebnisse beliebiger Benchmarks mit beliebigen Analyseverfahren kombinieren lassen und die Analyseergebnisse auch verschiedener Benchmarks anschließend vergleichend darstellbar sind.

3.1.3. Vergleich der Ergebnisse von Benchmarks

Mit Hilfe von Benchmark-Experimenten können Laufzeiten von einer Software oder Teilen davon gemessen werden. Da die Ergebnisse aber stark von der Hardware des Durchführenden Systems abhängen, sind diese Werte allein oft nur wenig aussagekräftig. Viel mehr interessiert man sich für den Vergleich der Laufzeit unter verschiedenen Bedingungen. Die erste Möglichkeit ist, verschiedene Revisionen der selben Software zu vergleichen. Hierfür sollten unter gleichen Bedingungen für die Revisionen einzeln Benchmarks durchgeführt und anschließend verglichen werden. So lässt sich die Entwicklung der Performance einer Software beobachten. Die andere Möglichkeit ist, die Performance verschiedener Konfigurationen der selben Revision zu vergleichen. Hierbei werden Parameter der Software oder

das Umfeld geändert, in dem die Experimente durchgeführt werden.
Die im Unterabschnitt 3.1.2 vorgestellte Abstraktion lässt beide Arten von Tests zu.
Um den Vergleich für Menschen besser verständlich zu machen, werden im allgemeinen nicht die gesamten Ergebnisse eines Benchmark-Experiments analysiert. Stattdessen werden diese Ergebnisse zusammengefasst. Hierfür gibt es verschiedene statistische Verfahren, von denen einige in Unterabschnitt 2.1.4 beschrieben sind. Dieses Zusammenfassen hat einen zusätzlichen Nutzen neben dem Gewinn an Übersichtlichkeit: Der Vergleich von verschiedenen Versionen oder Revisionen geht so um einiges schneller, da eine Analyse nur ein mal auf einen Satz von Durchführungsergebnissen angewandt wird, und das Analyseergebnis dann beliebig oft verglichen werden kann.

3.2. Komponenten

Um die durch die Abstraktion gewonnen Kombinationsmöglichkeiten von Benchmarks, Analyseverfahren und Vergleich zu realisieren, verwende ich folgende Arten von Komponenten:

1. Die erste Art von Komponenten ist die **Durchführungskomponente**, welche die eigentlichen Benchmarks ausführt. Diese Komponenten sind daher stark vom jeweiligen Projekt abhängig und können nach belieben implementiert werden, sofern die Ergebnisse den Anforderungen in Abschnitt 3.3 genügen.

2. **Analysekomponenten** sind die zweite Art von Komponenten, sie wenden Analyseverfahren auf die Ergebnisse eines Benchmark-Experiments an und erstellen daraus Analyseergebnisse. Da die Analyseverfahren häufig ähnlich sind, lassen sich ein mal erstellte Analysekomponenten projektübergreifend verwenden, sodass der Aufwand für die Analyse neuer Benchmarks minimiert wird.

3. **Darstellungskomponenten** vergleichen die Analyseergebnisse und erzeugen daraus Tabellen, Diagramme oder ähnliches.

Als Schnittstelle zwischen den Komponenten dient eine **Datenspeicherkomponente**, welche Zugriff auf Benchmark-Ergebnisse und Analyseergebnisse liefert. Die Umsetzung dieses Datenspeichers ist in Abschnitt 3.3 beschrieben.
Eine **Steuerungskomponente** wird benötigt, um den Ablauf und die Konfiguration zu ermöglichen. Diese Komponente muss alle anderen Komponenten kennen und starten können. Die Aufgaben der Steuerung umfassen unter anderem die Initialisierung des Datenspeichers, das Starten der Benchmark-Durchführung, das Starten der Analyse, wenn die Benchmark-Durchführung abgeschlossen ist, und die abschließende Einleitung der Präsentation der Analysedaten.
Zusätzlich wird noch eine Komponente zur automatisierten Überprüfung der Analyseergebnisse benötigt, um verschiedene Überprüfungsverfahren mit beliebigen Analyseverfahren

kombinieren zu können. Diese **Wächterkomponenten** werden ebenfalls über die Steuerungskomponente konfiguriert und gestartet.

Die vollständige Liste der Komponenten besteht dann aus

1. Steuerung
2. Datenspeicher
3. Durchführung
4. Analyse
5. Wächter
6. Darstellung

wie in Abbildung 3.1 ersichtlich.

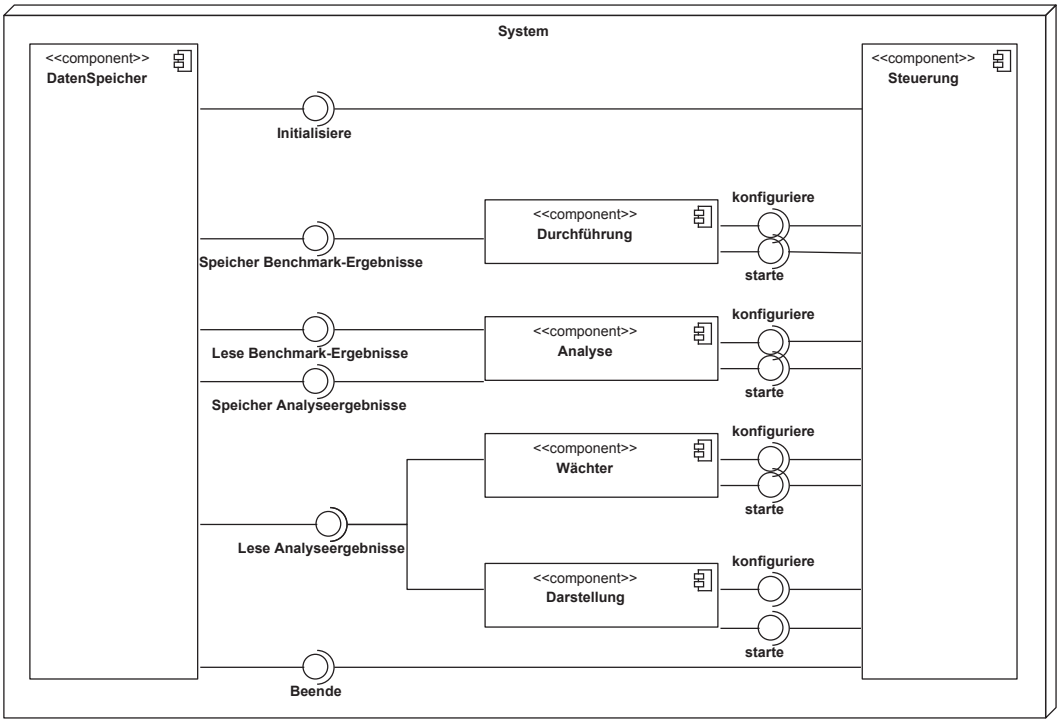

Abbildung 3.1. Interaktionen der Komponenten des Systems

3.3. Organisation von Daten

Bei der wiederholten Durchführung von Benchmark-Experimenten werden viele Daten produziert, wie zum Beispiel die Messergebnisse der Benchmarks und weiterverarbeitete Ergebnisse. Daher macht es Sinn, sich Gedanken um die Organisation der Daten zu machen.
Es sollten Rohdaten gespeichert werden, um spätere Analysen und Prüfungen zuzulassen, ohne die Benchmarks erneut ausführen zu müssen. Zusätzlich sollen Analyseergebnisse gespeichert werden, um Regressionstests zuzulassen, ohne die bisherigen Ergebnisse wiederholt analysieren zu müssen.
Bei der Durchführung von Experimenten erzeugt das System zuerst durch die Ausführung von Benchmarks einen Satz von Ergebnissen, der Quantitative Aussagen über die Performance einer Software zulässt. Diese Ergebnisse werden Benchmark-Ergebnisse genannt. Eine auf Benchmark-Ergebnissen basierende Analyse produziert Informationen, die weiterführende Vergleiche zulassen, indem, wie in Unterabschnitt 2.1.4 beschrieben, eine Menge von Ergebnissen auf unterschiedliche Weise verarbeitet wird; diese nenne ich im folgenden Analyseergebnisse.
Um die Zuordnung der Benchmark-Ergebnisse zu den zugehörigen Benchmarks zu ermöglichen, erhalten Benchmarks einen Identifikator, der frei konfiguriert werden kann, aber einmalig sein muss. Zu jedem Satz von Ergebnissen wird der Identifikator des zugehörigen Benchmarks sowie der Identifikator des Builds gespeichert, dies ermöglicht eine genaue Zuordnung. Ein Analyseergebnis bezieht sich immer auf einen bestimmten Benchmark und einen bestimmten Satz von Ergebnissen dieses Benchmarks, allerdings sollten auch verschiedene Analysen auf die Ergebnisse des selben Benchmarks angewandt werden können. Daher muss für die Zuordnung auch jeder Analyse ein für die Analysen einmaliger Identifikator zugewiesen werden, dieser Identifikator heißt im folgenden Analyse-Identifikator und die Identifikatoren von Benchmarks nenne ich Benchmark-Identifikatoren. Ein Analyseergebnis kann also anhand des Benchmark-Identifikators, eines Identifikators für den Build und des Analyse-Identifikators eindeutig einem Satz von Benchmark-Ergebnissen zugeordnet werden.
Hierbei gehe ich davon aus, dass ein Benchmark immer nur eine Sorte von Ergebnissen erzeugt, verschiedene Sorten müssen in diesem System auch von verschiedenen Durchführungskomponenten erzeugt werden. Die hier beschriebene Datenstruktur wird durch Abbildung 3.2 verdeutlicht.

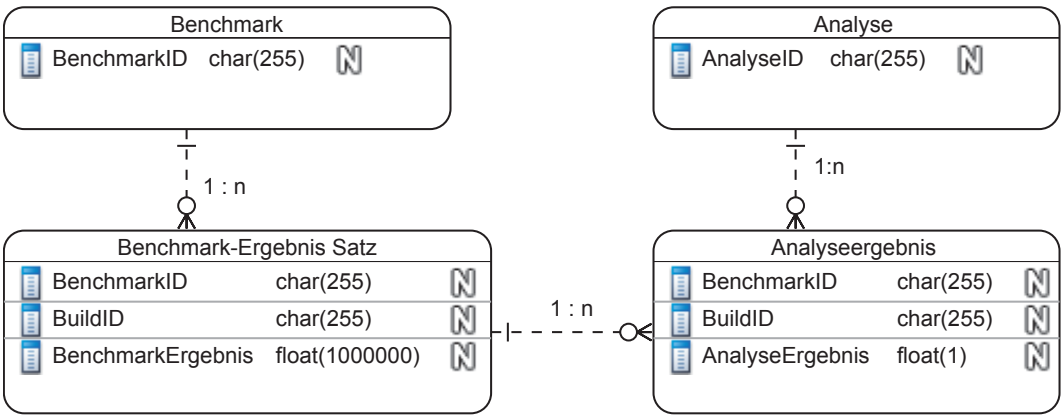

Abbildung 3.2. Datenstruktur

3.4. Integration

Die meisten Continuous-Integration-Systeme lassen sich durch Plug-ins anpassen und erweitern, daher bietet es sich an, die gewünschten Funktionen in ein Plug-in zu integrieren. Die in Abschnitt 3.2 beschriebenen Komponenten müssen dann die Schnittstelle des jeweiligen Continuous Integration Systems nutzen. Insbesondere die Steuerungskomponente ist stark von den Möglichkeiten der Programmierschnittstelle abhängig, da sie die Konfigurationsmöglichkeiten in die grafische Oberfläche des Continuous-Integration-Systems einbinden soll. Ebenso abhängig sind Darstellungskomponenten, falls die Darstellung innerhalb des CI-Systems erzeugt werden soll, sowie die Wächterkomponenten, da sie das Verhalten des CI-Systems abhängig von den Analyseergebnissen beeinflussen sollen.
Die Durchführung der Benchmarks kann auf einer separaten Maschine erfolgen, da sie keiner Einschränkung unterliegen, wie ihre Ergebnisse produziert werden. Das Auslagern kann aus verschiedenen Gründen erfolgen, einer ist die Vergleichbarkeit von Ergebnissen durch ein gleiches Umfeld erhalten zu wollen, wie in Abschnitt 3.7 beschrieben. Die Datenspeicherkomponente kann so implementiert werden, dass die Ergebnisse in einer Datenbank abgelegt werden, die auf einer anderen Maschine liegt oder von einer anderen Software verwaltet wird, sodass ein direkter Zugriff auf die Daten nicht möglich ist. Das System unterliegt auch hier keinen Einschränkungen, die dies verhindern würden. Bei der Implementierung der Datenspeicherkomponente sollte aber auf die Performance geachtet werden, da bei der Durchführung von Benchmarks oft sehr viele Ergebnisse entstehen, deren Speichern und Lesen je nach Art der Datenspeicherung unterschiedlich performant ist. Abbildung 3.3 verdeutlicht dies ebenso wie den oben beschriebenen Aufbau des Plug-ins.

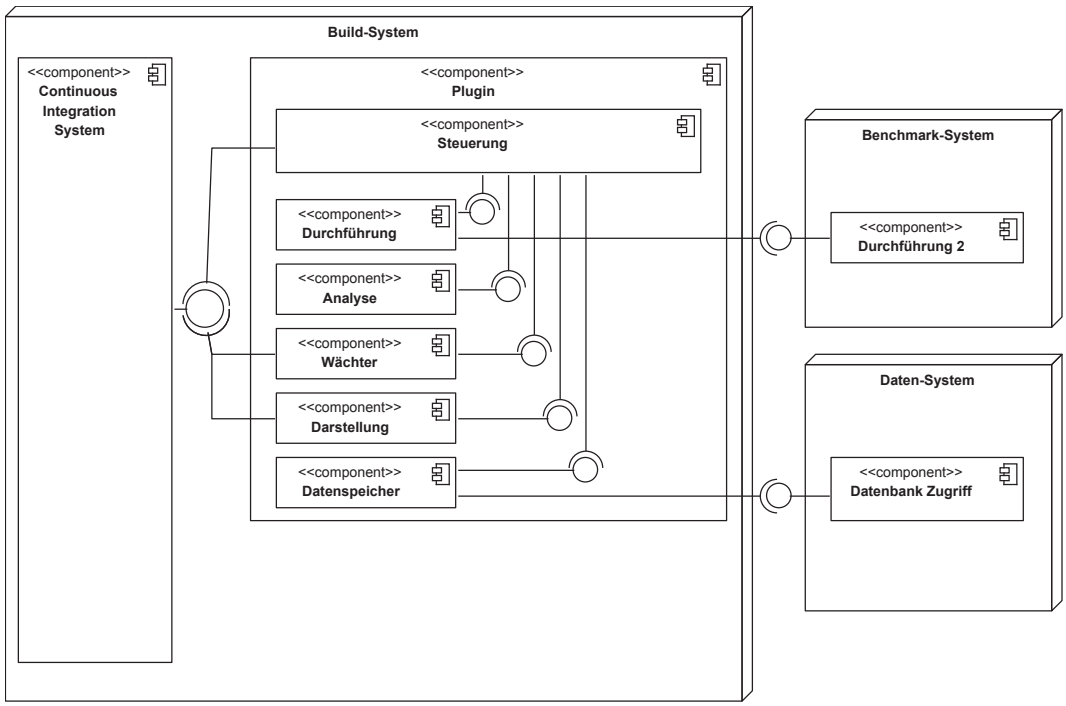

Abbildung 3.3. Aufbau des Plug-ins mit Auslagerung der Durchführung der Benchmarks und des Datenspeichers

3.5. Ablauf

Der genaue Ablauf ist durch die Flexibilität in der Konfiguration nicht fest bestimmt. Dies betrifft jedoch nur die Reihenfolge, in welcher die Komponenten innerhalb ihrer Kategorie arbeiten. Kategorie-übergreifend entsteht folgender Ablauf:

1. Im ersten Schritt erfolgt die Installation und Konfiguration des Plug-ins, dieser Schritt muss nur einmalig durchgeführt werden. Hier wird festgelegt, welche anderen Komponenten verwendet werden und in welcher Reihenfolge sie aktiviert werden. Zusätzlich wird die Konfiguration der Komponenten vorgenommen.

2. In der zweiten Phase wird der Build angestoßen und läuft durch, dann bekommt die Steuerungskomponente vom Continuous-Integration-System die Anweisung, die Durchführung der Benchmarks zu beginnen. Bevor dies geschieht, kann die Steuerungskomponente den Datenspeicher initialisieren, falls dies notwendig ist, oder andere Operationen tätigen. Die erforderlichen Schritte hängen von der Implementierung ab.

3. Nun startet die Steuerungskomponente die Durchführungskomponenten. Diese berechnen ihre Ergebnisse und benutzen dann die Datenspeicherkomponente, um ihre Ergebnisse zu sichern. Wie genau diese Ergebnisse produziert werden, ist nicht weiter eingeschränkt. Abschließend melden die Durchführungskomponenten die Fertigstellung der Steuerungskomponente.

4. Wenn eine Durchführungskomponente ihre Arbeit beendet hat, können die zugehörigen Analysekomponenten von der Steuerungskomponente gestartet werden. Die Analysekomponenten berechnen dann aus den aktuellsten zugehörigen Benchmark-Ergebnissen die Analyseergebnisse und geben diese ebenfalls an die Datenspeicherkomponente weiter.

5. Nachdem alle Durchführungs- und Analysekomponenten ihre Berechnungen beendet und die Ergebnisse gespeichert haben, können Wächter von der Steuerung aktiviert werden, um die Analyseresultate zu überprüfen. Dieser Schritt ist optional, da nicht in jedem Projekt die Kontrolle der Ergebnisse gewünscht sein muss. Die vom Wächter eingeleiteten Aktionen können beliebig gestaltet sein, wie auch die Art der Kontrolle.

6. Die Steuerungskomponente aktiviert zuletzt die Darstellungskomponente, welche die Analyseergebnisse ausliest und ihre Darstellungen erzeugt. Hiernach ist der gesamte Vorgang beendet und die Steuerung kann abschließende Operationen, wie das schließen des Datenspeichers, vornehmen.

Abbildung 3.4 verdeutlicht die Phasen und das Zusammenwirken der Komponenten. Die ersten beiden Phasen sind in dieser Abbildung nicht enthalten, da sie optional sind und nicht zwingend Interaktion zwischen den Komponenten benötigen.

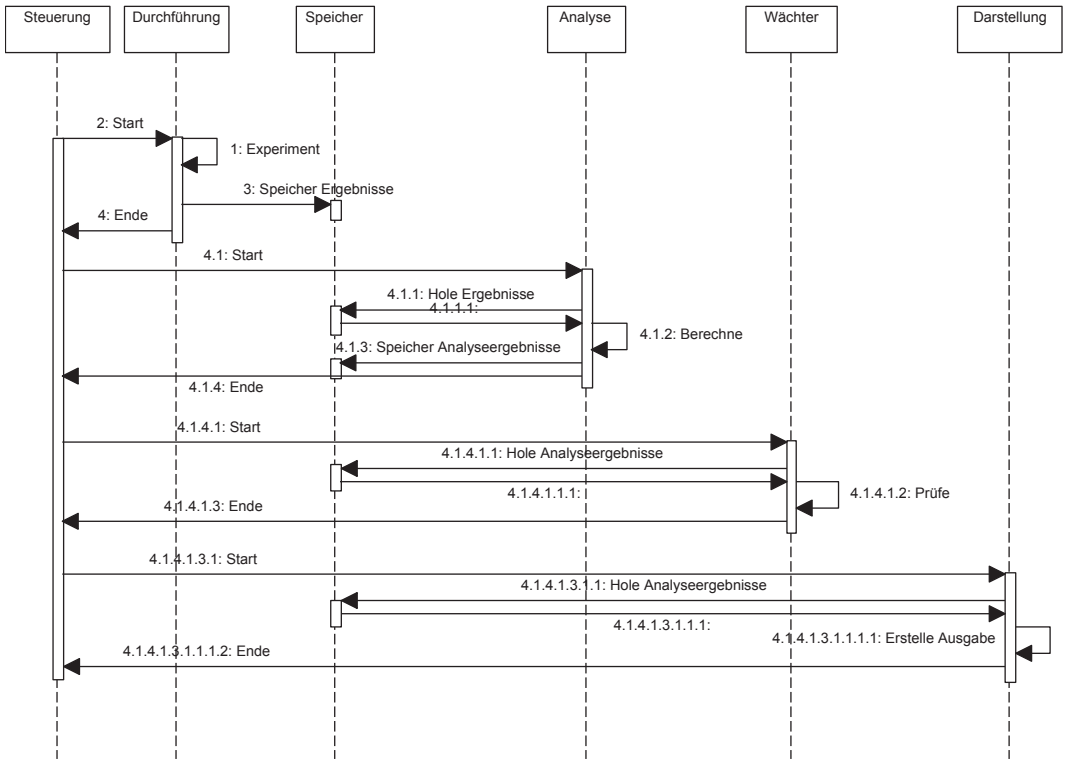

Abbildung 3.4. Ablauf des Benchmark-Experiments

3.6. Parallelisierung der Durchführung

In Abschnitt 3.5 ist der generelle Ablauf eines Benchmark-Experiments inklusive Durchführung, Analyse und Präsentation der Ergebnisse beschrieben. Dieser Ablauf ist sequentiell beschrieben, es gibt jedoch gute Gründe, eine Parallelisierung bestimmter Durchführungsschritte zu erwägen. Ein möglicher Grund ist, dass Benchmark-Experimente sehr lange dauern können und, je nach Implementierung der Durchführungskomponenten, bei sequentieller Ausführung Systeme mit mehreren Prozessoren oder Prozessorkernen nicht optimal auslasten, sodass die Fertigstellung zusätzlich verzögert wird.
Parallelisierbar sind insbesondere die Durchführungs- und Analysekomponenten, und gerade bei diesen Komponenten ist die Parallelisierung besonders interessant, weil sie die größten Laufzeiten aufweisen, da die größten Datenmengen von ihnen bearbeitet werden. Wächterkomponenten können ebenfalls parallelisiert werden, auch wenn dies in den meisten Fällen nicht erforderlich ist, da die Datenmengen, die von diesen Komponenten verarbeitet werden, vergleichsweise gering sind und sich die Laufzeiten daher im

Vergleich zu den Durchführungs- oder Analysekomponenten kaum bemerkbar machen. Um die Effizienzsteigerung des Systems durch die Parallelisierung nicht durch zeitliche Einschränkungen bei den Lese- und Schreibvorgängen von Ergebnissen zu beschränken, ist es empfehlenswert, auch die parallele Ausführung bestimmter Operationen der Datenspeicherkomponente zuzulassen.
Die Parallelisierung erfordert folgende Anpassungen der Komponenten:

1. Die Steuerungskomponente muss bei nebenläufig arbeitenden Komponenten so gestaltet werden, dass sie feststellen kann, wenn ein Benchmark beendet ist, damit erst dann die zu dem Benchmark gehörenden Analysen gestartet werden. Hierzu sind verschiedene Methoden zur Implementierung möglich, die ich hier nicht näher beschreibe. Zudem muss die Steuerung dafür sorgen, dass eine Wächterkomponente oder Darstellungskomponente nur dann ausgeführt wird, wenn alle zugehörigen Analysekomponenten ihre Ergebnisse in den Datenspeicher eingetragen haben.

2. Bei der Implementierung von Durchführungskomponenten sind bestimmte Kriterien zu beachten, die erfüllt sein müssen, damit die Ergebnisse vergleichbar bleiben, wie etwa ein gleiches Umfeld. Diese Kriterien sind in Abschnitt 3.7 näher beschrieben.

3. Für die Analysekomponenten liegt nur die Einschränkung vor, dass sie erst gestartet werden dürfen, wenn die zugehörige Durchführungskomponente die Durchführung ihrer Benchmarks beendet hat. Dies wird in der Regel von der Steuerungskomponente sichergestellt und bedarf keiner Anpassung der Analysekomponenten, sodass diese weiterhin unabhängig von der Art der Durchführungskomponenten nutzbar sind.

4. Für die Wächter- und Darstellungskomponenten gilt hingegen die Einschränkung, dass sie erst ausgeführt werden können, wenn die zugehörigen Analysen beendet sind. Für dies ist im Normalfall ebenfalls die Steuerungskomponente zuständig, sodass auch hier keine weiteren Anpassungen nötig sind.

5. Ob der Datenspeicher Anpassungen benötigt, ist von der Implementierung abhängig. Wenn paralleler Zugriff auf Funktionen, die Ergebnisse speichern, zu Datenverlust oder Verfälschung der Ergebnisse oder deren Zuordnung führen könnte, müssen diese Funktionen sich gegenseitig ausschließen. Die Durchführungsmöglichkeiten hierfür hängen von der Implementierung ab und werden hier nicht weiter erläutert.

Ein weiterer Aspekt des nicht-sequentiellen Ablaufs ist, dass ein Build-Vorgang nicht zwingend auf die Beendigung der Benchmarks warten muss, bevor der Build als Erfolg gewertet wird oder andere Operationen ausgeführt werden. Ob dies praktikabel ist, hängt von der jeweiligen Situation ab. Wenn zum Beispiel die Durchführung der Benchmarks sehr lange dauert und das Ergebnis nicht von entscheidender Bedeutung ist, sondern nur für statistische Zwecke oder für spätere Optimierungen benötigt wird, ist es nicht notwendig, auf die Fertigstellung der Benchmarks zu warten, bis mit dem Build-Prozess fortgefahren werden kann. Wenn hingegen die Verschlechterung der Benchmark-Ergebnisse für den

Erfolg eines Builds kritisch ist, muss darauf gewartet werden, damit die Wächterkomponenten eine Möglichkeit haben, das Ergebnis zu beeinflussen. Ein universales System zur Durchführung von Benchmark-Experimenten sollte beide Möglichkeiten anbieten, auch weil sich die Einstufung der Wichtigkeit der Performance einer Software während der Entwicklung stark verändern kann und andernfalls dann die Implementierung des Systems geändert werden müsste.

3.7. Vergleichbarkeit der Ergebnisse

Während der Entwicklung einer Software werden nicht nur immer neue Teile ergänzt, sondern häufig auch bestehende Teile umstrukturiert oder gar die gesamte Architektur geändert. Vergleichende Benchmarks erfüllen allerdings nur ihren Zweck, wenn die Testergebnisse auch vergleichbar sind. Hierzu ist es erforderlich, dass bestimmte Schnittstellen getestet werden, deren Definition nicht mehr verändert wird. So wird die Performance der Implementierungen verglichen und es ist sichergestellt, dass gleiche Funktionen des Programms miteinander verglichen werden. Diese Voraussetzung lässt sich aber schwer erfüllen, denn bei den meisten Software-Projekten verändern sich solche Schnittstellen auch in späten Entwicklungsphasen noch, weil sich die Anforderungen geändert haben, oder es sich gezeigt hat, dass die bisherige Definition unpraktisch ist.
Um also ein System zur Automatisierung von Benchmark-Experimenten an die praktischen Gegebenheiten anzupassen, muss es Veränderungen zwischen zwei Ausführungen ermöglichen. Die wichtigste Änderungsmöglichkeit ist das Hinzufügen neuer Benchmarks. Bei inkrementeller Entwicklung wächst der Funktionsumfang der Software stetig, was dazu führt, dass immer neue Anforderungen an Benchmarks hinzu kommen. Diese Benchmarks sollen sich später hinzufügen lassen, und das System muss bei der Verarbeitung damit umgehen können, dass nicht für jede Version der Software ein Benchmark-Ergebnis vorliegen muss. Hierzu empfiehlt es sich, die anderen Komponenten so aufzubauen, dass sie mit einer flexiblen, beschränkbaren Anzahl von Ergebnissen arbeiten. Beispielsweise sollte eine Darstellungskomponente nicht genau die 10 aktuellsten, sondern maximal die 10 aktuellsten Ergebnisse verwenden.
Ein weiterer Änderungsschritt ist das Deaktivieren von Benchmarks, sodass diese für neuere Versionen nicht mehr ausgeführt werden. Dies ist erforderlich, falls sich die Spezifikation einer getesteten Schnittstelle ändert, sodass die Vergleichbarkeit verloren gehen würde, oder falls Methoden entfallen, und deshalb auch nicht mehr getestet werden können. Das System sollte auch nach der Deaktivierung eines Benchmarks die bestehenden Ergebnisse speichern und anzeigen können, da andernfalls mit der Zeit immer mehr Ergebnisse verloren gingen.
Mit diesen beiden Änderungsschritten lassen sich Benchmarks zwischen zwei Ausführungen beliebig verändern, ohne dass zwei Durchführungsergebnisse verglichen werden, die auf verschiedenen Experimenten beruhen oder dass versucht wird, eine Komponente zu testen, die in der aktuellen Revision nicht mehr existiert. Die Vergleichbarkeit von

Benchmarks hängt aber nicht nur von der Vergleichbarkeit der getesteten Software ab, sondern auch von der Vergleichbarkeit der Umgebung.

Da bei Mikrobenchmarks konkrete Ausführungszeiten gemessen werden, ist es für die Vergleichbarkeit der Ergebnisse von Bedeutung, dass sich die Umgebung, in der die Experimente durchgeführt werden, nicht verändert. Beispielsweise könnte unterschiedliche Auslastung des Prozessors oder Speichers durch andere Programme während der Durchführung der Tests zu unterschiedlichen Ergebnissen führen, obwohl die getestete Software sich nicht geändert hat. So würden die Ergebnisse verfälscht und die Vergleichbarkeit damit verloren gehen.
Es ist also nötig, das System so zu gestalten, dass gleiche Bedingungen während der Experimente herrschen. Dies lässt sich nur erreichen, wenn sich die Hardware, auf der die Experimente durchgeführt werden, nicht ändert und keine anderen Prozesse die Laufzeiten beeinflussen. Hierzu ist es bei sequenzieller Ausführung der Tests hinreichend, eine Maschine zu haben, die von den Tests so blockiert wird, dass keine anderen Prozesse, wie etwa andere Build-Vorgänge in einem Continuous-Integration-System auf einem Build-Server, durchgeführt werden. Diese Lösung führt aber dazu, dass die Hardware abhängig von der Anzahl und der Dauer der Experimente sehr lange blockiert sein kann. Dadurch kann die Produktivität der Entwicklung eingeschränkt werden. Außerdem ist es häufig unpraktikabel, einen Build-Server nur sequentiell arbeiten zu lassen und mit den Benchmarks einer Software zu blockieren, wenn beispielsweise mehrere Prozessoren oder Prozessor-Kerne vorliegen, von denen dann nur einer genutzt würde. Dies lässt sich umgehen, indem die Durchführung der Benchmark-Experimente auf ein anderes System, das exklusiv für die Benchmarks verwendet wird, ausgelagert wird. Um die Reservierung von Hardware zu diesem Zweck zu verhindern, können virtuelle Maschinen verwendet werden.
Die genauen Bedingungen der Durchführung werden in dem hier entwickelten System durch die Definition der Durchführungskomponenten nicht weiter eingeschränkt, sodass sich beide Verfahren nach belieben anwenden lassen.

Das Fazit aus den oben beschriebenen Umständen ist, dass die Definition des System wie folgt erweitert werden muss:

1. Die Durchführungskomponenten müssen entfernbar oder de-aktivierbar sein, ohne dass die Ergebnisse verloren gehen.

2. Alle Analysekomponenten müssen so gestaltet werden, dass sie mit einer variablen Anzahl von Ergebnissen umgehen können, da nicht für jede Revision Ergebnisse zu jedem Benchmark vorliegen müssen.

3. Die Durchführungskomponenten dürfen bezüglich der Art der Ausführung nicht weiter beschränkt werden, um die Flexibilität des Systems nicht unnötig einzuschränken. Dies ist bisher schon der Fall gewesen, da nur durch den Datenspeicher festgelegt ist, in welcher Form die Ergebnisse vorliegen müssen, nicht hingegen, wie sie erzeugt werden.

3.8. Zusammenfassung

In diesem Kapitel habe ich ein System entwickelt, welches die automatisierte Durchführung und Auswertung von Mikrobenchmarks umsetzt, indem ein entsprechendes Plug-in in ein Continuous-Integration-System integriert wird. Ein solches Plug-in besteht aus den Komponenten

1. Steuerung
2. Datenspeicher
3. Durchführung
4. Analyse
5. Wächter
6. Darstellung.

Dieses System ist in der Lage, Regressions- und Konfigurationstests durchzuführen und ist durch den komponentenweisen Aufbau sehr flexibel, leicht erweiter- und anpassbar. Im nächsten Kapitel werde ich eine beispielhafte Implementierung dieses Systems für das Continuous-Integration-System Jenkins vorstellen.

Kapitel 4

Beispielhafte Implementierung

4.1. Einleitung

In diesem Kapitel beschreibe ich die beispielhafte Implementierung des in Kapitel 3 beschriebenen Systems zur Automatisierung der Durchführung und Auswertung von Mikrobenchmarks. Das hierzu verwendete Continuous-Integration-System ist Jenkins, für welches ich ein Plug-in beschreibe, das die automatische Ausführung von MooBench-Benchmarks zur Ermittlung des Overheads von Kieker-Implementierungen ermöglicht.
Hierzu gehe ich zuerst auf den Aufbau eines Plug-ins für Jenkins ein, beschreibe dann die Implementierung der einzelnen Komponenten und gebe eine Anleitung sowie ein Beispiel zur Konfiguration. Abschließend diskutiere ich die Möglichkeiten, die Implementierung zu verändern und erweitern.

4.2. Plug-in-Entwicklung für Jenkins

Die Dokumentation der Plug-in-Entwicklung für Jenkins befindet sich im Web unter der Adresse `https://wiki.jenkins-ci.org/display/JENKINS/Extend+Jenkins`. Für die Entwicklung verwende ich die Netbeans IDE [Boudreau u. a. 2002] in der Version 7.4 mit dem Plug-in für die Plug-in-Entwicklung für Jenkins.
Jenkins kann durch Plug-ins erweitert werden, indem sogenannte *Extension Points*, also *Erweiterungspunkte*, genutzt werden. Dies sind Java-Klassen, von denen geerbt werden kann. Wenn das Plug-in installiert ist, erkennt Jenkins enthaltene Extension-Point Implementierungen und integriert diese automatisch.
Für die Beschreibung der grafischen Oberfläche wird die Skriptsprache Apache Jelly verwendet. Dadurch können ähnlich zu HTML Komponenten der Oberfläche beschrieben werden, welche dann von Jenkins eingefügt werden.
In Jenkins werden Build-Vorgänge durch so genannte *Jobs* beschrieben, diese enthalten die Konfiguration für einen Build-Vorgang. Ein Job besteht aus mehreren Abschnitten, zum Beispiel Vorbereitung, Build, Nachbereitung.
Für meine Implementierung verwende ich die Extension-Point Klasse *Builder* aus dem Paket *hudson.tasks*. Builder beschreiben in Jenkins Schritte eines Jobs, die zum bauen verwendet werden, wie zum Beispiel Skripte auszuführen oder Compiler zu starten. Implementierte Builder können in der Konfiguration des Projektes nach belieben hinzugefügt werden,

sodass das System in hohem Maß flexibel ist. Dies bietet sich für die Implementierung des von mir entwickelten Systems an, da die Komponenten in ihrer Verwendung sehr flexibel gehalten werden sollen, um möglichst viele Konfigurationen zu ermöglichen.
Konfigurationsparameter für Builder können durch Jelly-Skripte, die den gleichen Namen tragen wie die Builder-Klasse inklusive des Paketnamens, in die Oberfläche integriert werden. Ein Builder kann Methoden enthalten, um Inhalte der Felder während der Eingabe zu überprüfen. Ein simpler Builder sieht wie folgt aus:

```
1  public class MyBuilder extends Builder {
2
3      private final String attribute1;
4      public String getAttribute1() {
5          return attribute1;
6      }
7
8      @DataBoundConstructor
9      public MooBenchExecutor(String attribute1){
10                 this.attribute1 = attribute1;
11     }
12
13     @Override
14     public boolean perform(AbstractBuild build, Launcher launcher,
15                            BuildListener listener) {
16         listener.getLogger().println("Hello World!");
17         return true;
18     }
19
20     @Extension
21     public static final class DescriptorImpl extends BuildStepDescriptor<Builder> {
22         public FormValidation doCheckAttribute1(@QueryParameter String value)
23                 throws IOException, ServletException {
24             return FormValidation.validateRequired(value);
25         }
26         public String getDisplayName() {
27             return "Perform My Build";
28         }
29     }
30
31 }
```

Das Attribut *attribute1* wird hierbei über die Konfiguration des Builders in der Oberfläche von Jenkins gesetzt und über die Methode *doCheckAttribute1* getestet, falls ein geeignetes Jelly-Skript vorhanden ist. Während eines Builds wird dann die Methode *Perform* aufgerufen.

Mit Ausnahme des Datenspeichers sind alle Komponenten als Builder implementiert. Der Datenspeicher muss im Gegensatz zu den anderen Komponenten dauerhaft verfügbar sein und kann daher keine Builder-Implementierung sein, da diese nur sequentiell arbeiten und er sonst den anderen Komponenten nicht zur Verfügung stünde.
Da die Konfiguration von jedem Builder selbst über die ihm zugehörigen Jelly-Skripte geregelt wird, kann diese Aufgabe nicht von einer Steuerungskomponente durchgeführt werden. Das feste Anlegen von Konfigurationsparametern in einer Steuerungskomponente hätte zur Folge, dass Implementierungen der anderen Komponenten immer nur mit den dort aufgeführten Parametern arbeiten könnten und die Reihenfolge und Anzahl der Komponenten durch die Steuerungskomponente eingeschränkt wären. Aus diesem Grund implementiere ich keine Steuerungskomponente. Die Konfiguration der anderen Komponenten wird jeweils direkt festgelegt und die Ablaufreihenfolge wird durch Jenkins bestimmt. Jenkins nimmt also die Rolle der Steuerungskomponente ein.

4.3. Umsetzung der Komponenten

In diesem Abschnitt beschreibe ich die Umsetzung der einzelnen Komponenten. Die verwendeten Klassen sind in Abbildung 4.1 gezeigt und werden in der jeweiligen Komponenten-Beschreibung weiter erklärt. Die Komponenten sind in verschiedene Pakete geordnet. Die Analysekomponenten befinden sich im Paket *autobenchmark.analyse*, die Durchführungskomponenten in *autobenchmark.execute*, Wächterkomponenten befinden sich in *autobenchmark.guard*, und Darstellungskomponenten im Paket *autobenchmark.view*. Die Datenspeicherkomponente befindet sich mit weiteren für die Daten repräsentativen Klassen im Paket *autobenchmark.storage*.

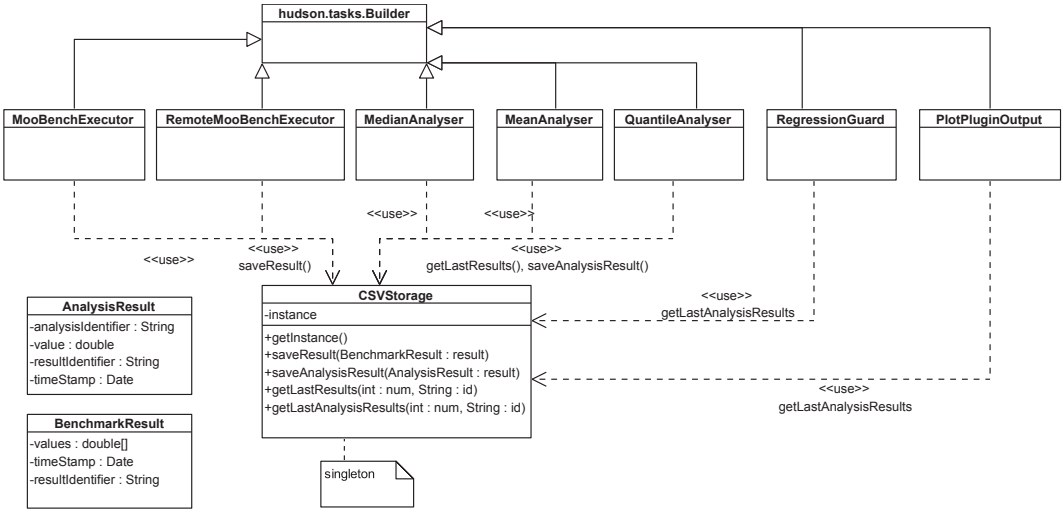

Abbildung 4.1. Die wichtigsten Klassen der Implementierung

4.3.1. Datenspeicher

Das grundlegende Datenmodell habe ich bereits in Abschnitt 3.3 vorgestellt. Die Klasse *CSVStorage* implementiert dieses Modell mit CSV-Dateien. Dieses Format bietet gegenüber Datenbanksystemen den Vorteil, dass die ausgegebenen Dateien sehr leicht von anderen Programmen dargestellt werden können und kein separates System erfordern. Diese Dateien werden unter einem konfigurierbaren Pfad abgelegt, welcher im Folgenden Datenpfad genannt wird. Da der Datenspeicher keine eigene Oberfläche bietet, muss der zugehörige Datenspeicher durch seinen Pfad in jeder Komponente, die einen Datenspeicher verwendet, angegeben werden. Der CSVStorage folgt dem Singleton-Entwurfsmuster [Gamma u. a. 1994], um die Daten für andere Klassen einfach zugänglich zu machen. Es gibt es vier Arten von Dateien, die im folgenden weiter erläutert werden.

Index

Die Index-Datei speichert den Pfad von Benchmark-Ergebnis-Satz-Dateien sortiert nach Benchmark-Identifikator und Datum. Diese Dateien sind aufgebaut wie in Tabelle 4.1 gezeigt. Im Datenmodell entspricht diese Datei der Relation zwischen Benchmark und Benchmark-Ergebnis-Satz. Diese Datei wird immer unter dem Namen *index.csv* im Datenpfad gespeichert.

Tabelle 4.1. Aufbau der Index-Datei

	Benchmark-ID 1	Benchmark-ID 2	...
Datum 1	Pfad 1.1	Pfad 2.1	...
Datum 2	Pfad 1.2	Pfad 2.2	...
...	...	...	...

Analyse-Index

Die Datei Analyse-Index speichert entsprechend den Pfad von Analyseergebnis-Dateien wie in Tabelle 4.2 gezeigt. Hierbei entfällt eine Zuordnung zum Datum, die Begründung hierfür liegt im Aufbau der Analyseergebnis-Dateien. Der Analyse-Index entspricht der Relation zwischen Benchmark-Ergebnis-Satz und Analyseergebnis im Datenmodel und ist unter dem Namen *analysis-index.csv* im Datenpfad zu finden.

Tabelle 4.2. Aufbau der Analyse-Index-Datei

	Benchmark-ID 1	Benchmark-ID 2	...
Analyse-ID 1	Pfad 1.1	Pfad 2.1	...
Analyse-ID 2	Pfad 1.2	Pfad 2.2	...
...	...	...	...

Benchmark-Ergebnis-Satz

Benchmark-Ergebnisse werden in den in der Index-Datei aufgelisteten Dateien nach dem in Tabelle 4.3 beschriebenen Muster gespeichert. Eine Benchmark-Ergebnis-Satz-Datei entspricht einem Eintrag der Benchmark-Ergebnis-Satz-Tabelle im Datenmodell. Diese Tabelle wird nicht in einer Datei zusammengefasst, da Zugriffe meistens nur einzelne Datensätze benötigen und nicht alle. Das getrennte Speichern erhöht so die Performance, da nicht alle Ergebnisse gelesen werden, sondern nur die benötigten. Diese Dateien werden in Ordnern mit dem Namen des Benchmark-Identifizierers zusammengefasst und tragen eine Bezeichnung, die den Benchmark-Identifizierer und den Build-Identifizierer enthält.

Tabelle 4.3. Aufbau einer Benchmark-Ergebnis-Satz-Datei

Ergebnis 1
Ergebnis 2
Ergebnis 3
...

Analyseergebnis

Im Gegensatz zu den Benchmark-Ergebnis-Sätzen werden die Analyseergebnisse für eine Analyse und einen Benchmark immer in einer Datei zusammengefasst, da diese Ergebnisse häufig gleichzeitig benötigt werden und die Anzahl vergleichsweise viel geringer ist, als die Anzahl der Ergebnisse in einer Benchmark-Ergebnis-Satz-Datei, sodass die Ladezeiten im Regelfall in tolerierbaren Grenzen bleiben. Die wie in Tabelle 4.4 ersichtlich aufgebauten Dateien werden unter dem Datenpfad in einem Ordner mit dem Identifizierer des Benchmarks als Namen unter einer Bezeichnung, die Benchmark- und Analyse-Identifizierer enthält, abgelegt.

Tabelle 4.4. Aufbau einer Analyseergebnis-Datei

Datum 1	Ergebnis 1
Datum 2	Ergebnis 2
Datum 3	Ergebnis 3
...	...

Der Datenspeicher bietet Methoden zum Speichern von Benchmark- und Analyseergebnissen und zum Auslesen der Ergebnisse des letzten Benchmark-Experiments mit einem bestimmten Benchmark-Identifizierer und einer Anzahl der aktuellsten Ergebnisse einer bestimmten Analyse zu einem bestimmten Benchmark. Diese Methoden werden gebraucht, um die Funktionen der anderen Komponenten wie in Abbildung 4.1 gezeigt zu ermöglichen und sind durch das Interface *Storage* wie folgt definiert:

```
1 public interface Storage {
2     public void saveBenchmarkResult(BenchmarkResultSet result) throws IOException;
3     public void saveAnalysisResult(AnalysisResult result) throws IOException ;
4     public AnalysisResult[] getAnalysisResults(int max, String analysisIdentifier,
5         String benchmarkIdentifier) throws IOException;
6     public BenchmarkResultSet[] getBenchmarkResults(int max,
7         String benchmarkIdentifier) throws IOException;
8 }
```

Wie in Abschnitt 3.7 beschrieben, lässt sich die Anzahl der Ergebnisse durch die abrufenden Methoden nur begrenzen, da bei dem Aufruf nicht feststeht, wie viele Ergebnisse zur Verfügung stehen.

4.3.2. Durchführung

Die Aufgabe der Durchführungskomponente ist es, MooBench-Benchmarks auszuführen und die Ergebnisse anschließend dem Datenspeicher zu übergeben. Um dies zu erreichen,

können bisher verwendete MooBench-Shell-Skripte [MooBench Scripts] ausgeführt werden, und nach deren Ausführung die Ergebnisse aus den Ergebnis-Dateien ausgelesen und in den Datenspeicher eingetragen werden. Ein Beispiel für ein solches Skript ist in stark gekürzter Fassung hier zu sehen:

```
1  #!/bin/bash
2  SLEEPTIME=30            ## 30
3  NUM_LOOPS=10            ## 10
4  THREADS=1               ## 1
5  RECURSIONDEPTH=10       ## 10
6  TOTALCALLS=2000000      ## 2000000
7  METHODTIME=0            ## 0
8  MOREPARAMS=""#MOREPARAMS="--quickstart"
9  JAVAARGS="-server"
10 JAVAARGS="${JAVAARGS} -d64"
11 JAVAARGS_NOINSTR="${JAVAARGS}"
12 JAVAARGS_LTW="${JAVAARGS} -javaagent:${BASEDIR}lib/kieker-1.8-SNAPSHOT_aspectj.jar"
13 JAVAARGS_KIEKER_LOGGING2="${JAVAARGS_LTW}
14 -Dkieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncBinaryZipWriter"
15 ## Execute Benchmark
16 for ((i=1;i<=${NUM_LOOPS};i+=1)); do
17     j=${RECURSIONDEPTH}
18     k=0
19     ${JAVABIN}java  ${JAVAARGS_KIEKER_LOGGING2} ${JAR} \
20         --output-filename ${RAWFN}-${i}-${j}-${k}.csv \
21         --totalcalls ${TOTALCALLS} \
22         --methodtime ${METHODTIME} \
23         --totalthreads ${THREADS} \
24         --recursiondepth ${j} \
25         ${MOREPARAMS}
26     sync
27     sleep ${SLEEPTIME}
28 done
```

Diese Methode hat den Nachteil, dass für die Anfertigung von Benchmarks zusätzlich zur Durchführungskomponente noch Skripte in einer anderen Sprache geschrieben werden müssen, sodass das System unnötig fragmentiert wird. Außerdem enthalten diese Skripte viele Teile, die bei der Ausführung in dem hier entwickelten System überflüssig sind, wie zum Beispiel das Erstellen von Dateien zur Präsentation der Ergebnisse. Die elegantere Methode ist, die benötigten Funktionen dieser Skripte direkt in eine Durchführungskomponente zu integrieren.
Die wesentliche Funktion der Skripte ist es, ein ausführbares Java-Archiv wiederholt zu

starten und mittels Java-Agent eine Implementierung des Kieker-Frameworks Dummy-Methoden, die von der Archiv-Datei gestartet werden, überwachen zu lassen. Die Dummy-Methoden selbst haben keine weiteren Funktionen und benötigen, falls nicht anders konfiguriert, keine Zeit, sodass die gemessene Zeit nur durch das Kieker-Framework verursacht wird. Dadurch lässt sich der Overhead des Frameworks messen. In vielen der Skripten wird die Ausführung mit verschiedenen Argumenten durchgeführt, um unterschiedliche Aspekte zu testen. Daher würde es sich anbieten, auch dies in die Durchführungskomponenten zu integrieren. Da die Zahl aber schwankt, ist es praktischer, immer nur eine Konfiguration zuzulassen. Dies erhöht zwar den Konfigurationsaufwand, schränkt aber die Funktionalität nicht ein und führt zu einer größeren Flexibilität.

Die Implementierung der Skripte als Durchführungskomponente bildet die Klasse *MooBenchExecutor*. Sie erbt wie in Abschnitt 4.3 beschrieben von *hudson.tasks.Builder* und kann so wie jeder andere Builder in Jenkins verwendet werden. Die ersten Konfigurationsparameter entsprechen denen der Skripte:

- Der Parameter *numLoops* gibt die Anzahl der Schleifendurchläufe an. In jedem Schleifendurchlauf werden von jedem der *threads* vielen Threads *totalCalls* Aufrufe der Dummy-Methode gestartet, sodass die Zahl der Ergebnisse insgesamt durch $numLoops \cdot threads \cdot totalCalls$ gegeben ist.
- Die Anzahl der zu verwendenden Threads wird durch *threads* bestimmt.
- Die folgenden Parameter werden der ausführbaren MooBench-Java-Datei als argumente mitgegeben. *TotalCalls* ist der erste dieser Parameter. Er gibt die Anzahl der Aufrufe der Dummy-Methode pro Schleifendurchlauf an.
- *MethodTime* bestimmt, wie lange die Dummy-Methode in der Ausführung verbleibt. Diese Zeit muss von den Ergebnissen abgezogen werden, falls nur der Overhead ermittelt werden soll.
- Mittels des Parameters *quickStart* kann festgelegt werden, ob die MooBench-Java-Datei mit einem Schnellstart ausgeführt werden soll. Dies führt dazu, dass bestimmte vorbereitende Speicheroperationen ausgelassen werden.

Zusätzlich werden folgende Parameter benötigt, um die Flexibilität der Durchführungskomponente zu ermöglichen:

jar

Dieser Parameter gibt den Pfad der MooBench-Datei absolut an.

javaArgs

Über diesen Parameter können Argumente für die JVM, die Java Virtual Machine, übergeben werden, welche beim Start von MooBench verwendet werden. Diese Argumente können wie im Beispiel für MooBench-Skripte in Zeile 12 bis 19 sein.

args

Dieser Konfigurationsparameter kann zusätzliche Argumente enthalten, die am Ende eingefügt werden, so können MooBench weitere Argumente angefügt werden.

benchmarkIdenfitifer

Über den *benchmarkIdentifier* können die Benchmark-Ergebnisse identifiziert werden und ihnen können Analysekomponenten zugeordnet werden. Dieser Identifizierer sollte einmalig sein, um die eindeutige Zuordnung zu gewährleisten.

Ein Beispiel für die Belegung der Argumente befindet sich in Abschnitt 4.5. Die Methode *Perform* der Klasse *MooBenchExecutor* sieht dann in stark gekürzter Form so aus:

```
1  public boolean perform(AbstractBuild build, Launcher launcher,
2  BuildListener listener) {
3      //Get storage
4      CSVStorage storage = CSVStorage.getInstance(storagePath);
5      //Initialise
6      final String buildIdentifier = build.getId();
7      double[] values = new double[Integer.parseInt(numLoops) * numResultsPerLoop];
8      //Execute Experiment
9      for (int i = 1; i <= Integer.parseInt(numLoops); i++) {
10         //Create output file
11         String outputFilename = /*[...] +*/ buildIdentifier + "-loop" + i + ".csv";
12         //Set the command line
13         List<String> arg = new LinkedList<String>();
14         arg.add("java");
15         if (!javaArgs.equals(""))
16             arg.addAll(Arrays.asList(javaArgs.split(" ")));
17         if (!kiekerJar.equals(""))
18             arg.add("-javaagent:" + kiekerJar);
19         arg.add("-jar");
```

```
20          arg.add(jar);
21          arg.add("-o");
22          arg.add(outputFilename);
23          arg.add("-t");
24          arg.add(totalCalls);
25          arg.add("-m");
26          arg.add(methodTime);
27          arg.add("-h");
28          arg.add(threads);
29          arg.add("-d");
30          arg.add(recursionDepth);
31          if (isQuickStart())
32              arg.add("-q");
33          if (!args.equals(""))
34                  arg.addAll(Arrays.asList(args.split(" ")));
35          //Start the moobench-jar in a separate system process
36          ProcessBuilder pb = new ProcessBuilder(arg);
37          Process proc = pb.start();
38          //Retreive the process output
39          InputStream in = proc.getInputStream();
40          InputStream err = proc.getErrorStream();
41          //Wait for experiment to finish
42          while (isRunning(proc)) {
43              Thread.sleep(1000);
44          }
45          //Save output to log
46          String logFileName = buildIdentifier + "-loop" + i + "-log.txt";
47          File log = new File(logFileName);
48          FileOutputStream logOut = new FileOutputStream(log);
49          IOUtils.copy(in, logOut);
50          //Print error-output
51          boolean error = false;
52          int j = err.read();
53          while (j != -1) {
54              listener.getLogger().print((char) j);
55              j = err.read();
56          }
57          //Read results
58          List<String[]> index;
59          CSVReader reader = new CSVReader(new FileReader(outputFilename), ';');
60          index = reader.readAll();
```

```
61          reader.close();
62          //Copy results
63          for (j = 0; j < index.size() && j < numResultsPerLoop; j++) {
64              values[(i - 1) * numResultsPerLoop + j] =
65                  Double.parseDouble(index.get(j)[1]);
66          }
67      }
68      storage.saveBenchmarkResult(
69          new BenchmarkResultSet(values, benchmarkIdentifier, buildIdentifier));
70      return true;
71  }
```

In Zeile 3 bis 4 wird der Datenspeicher initialisiert. Danach wird in den Zeilen 6 und 7 die ID des aktuellen Builds ausgelesen und Speicher für die Ergebnisse reserviert. Die Schleife ab Zeile 9 entspricht einer Schleife aus dem MooBench-Skript. Hier wird zuerst die Datei für die MooBench-Ergebnisse angelegt, dann werden die Argumente zusammengesetzt und schließlich wird in Zeile 37 die Ausführung der Benchmarks über einen *ProcessBuilder* gestartet. Die Methode wartet anschließend, bis der Prozess beendet ist. Dann wird die Ausgabe der Ausführung in eine Log-Datei geschrieben und falls eine Error-Ausgabe vorhanden ist, diese in den Build-Log von Jenkins geschrieben. Ab Zeile 57 werden die Ergebnisse aus der Ergebnisdatei von MooBench ausgelesen und in Zeile 63 in den reservierten Speicher geschrieben. Nach dem Durchlauf aller Schleifen werden die Ergebnisse durch Zeile 68 in den Datenspeicher übermittelt.

Jenkins unterstützt ein verteiltes Build-System, indem einem Master-System verschiedene Slave-Systeme untergeordnet werden können. Je nach Konfiguration kann Jenkins dann dynamisch oder nach manueller Konfiguration Builds auf diese Systeme auslagern. Um die Vergleichbarkeit der Ergebnisse zu fördern, implementiere ich eine weitere Version des *MooBenchExecutors*, welche die Ausführung von MooBench auf einen Slave auslagert. Das soll ermöglichen, die Benchmarks auf einer separaten Maschine auszuführen, die nur für die Benchmarks reserviert ist, sodass andere Prozesse, wie etwa parallel laufende Builds, die Ergebnisse nicht beeinflussen.
Der neue Builder *RemoteMooBenchExecutor* verfügt zusätzlich das Attribut *node* welches den Namen des Slaves, auf dem das Experiment ausgeführt werden soll, angibt. Zusätzlich bietet der Builder an, die benötigten Jar-Dateien auf den Slave zu kopieren. Diese werden auf dem Slave unter dem Pfad *remoteDatapath* abgelegt, falls *copyFiles* gesetzt ist. Für den Kopier- und Ausführungsvorgang werden *Callable*-Objekte angelegt, die dann, wie im unteren Beispiel in Zeile 20 ersichtlich, auf dem Slave ausgeführt werden.

```
1  Callable<List<String>, IOException> copyJars =
2          new Callable<List<String>, IOException>() {
3      public List<String> call() throws IOException {
4      LinkedList<String> output = new LinkedList<String>();
5      FilePath kiekerSource =
6          new FilePath(SlaveComputer.getChannelToMaster(), kiekerFileMaster);
7      FilePath jarSource =
8          new FilePath(SlaveComputer.getChannelToMaster(), jarFileMaster);
9      FilePath kiekerDest = new FilePath(
10         new File(remoteDatapath + File.separatorChar + kiekerFileName));
11     FilePath jarDest =
12         new FilePath(new File(remoteDatapath + File.separatorChar + jarFileName));
13     try {
14         kiekerSource.copyToWithPermission(kiekerDest);
15         jarSource.copyToWithPermission(jarDest);
16     } catch (InterruptedException ex) {
17         output.add(ex.toString());
18     }
19     return output;}};
20 if (copyFiles) Jenkins.getInstance().getNode(node).getChannel().call(copyJars);
```

Das *Callable*-Objekt zur Ausführung der Experimente gibt als Ergebnis *MooBenchResult*-Objekte zurück, welche ein Feld mit Ergebniswerten und eine Liste von Log-Ausgaben beinhalten. Die Ergebnisse werden dann wie im lokalen MooBenchExecutor gespeichert und die Logs werden in Jenkins ausgegeben.
Jenkins verwaltet die Aufteilung der Builds normalerweise direkt. Dies hat jedoch den Nachteil, dass der gesamte Build auf der gleichen Maschine ausgeführt werden muss. Der von mir implementierte *RemoteMooBenchExecutor* führt jedoch nur die benötigten Aufgaben auf dem Slave aus. Dies führt im Gegenzug dazu, dass Jenkins die Verwendung des Slaves in der Oberfläche nicht anzeigt, und auch nicht vor dem Starten des Builds darauf wartet, dass der Slave einsatzbereit ist. Falls der Slave nicht verfügbar ist, während das Experiment gestartet werden soll, kommt es zu einer Fehlerausgabe und der Build schlägt fehl.

4.3.3. Analyse

Die Implementierten Analyseverfahren sind die in Unterabschnitt 2.1.4 vorgestellten. Diese Verfahren sind jeweils als ein Builder implementiert und verwenden einen konfigurierbaren *BenchmarkIdentifier*, um vom Datenspeicher den aktuellsten Datensatz von Ergebnissen mit diesem Identifizierer abzurufen. Anschließend werden die Benchmark-Ergebnisse nach dem jeweiligen statistischen Mittel analysiert und die Ergebnisse mit einem ebenfalls konfigurierbaren *AnalysisIdentifier* als Schlüssel dem Datenspeicher übergeben.

Alle Analysemethoden haben als zusätzlichen Parameter die Zahl *warumUpCount*, welche bestimmt, wie viele der Ergebnisse nicht berücksichtigt werden, da sie noch in der Aufwärmphase des Systems erzeugt wurden.
Die Implementierung der Quantilen erfolgt in der Klasse *QuantileAnalyser*, die des Medians in der Klasse *MedianAnalyser*, und die Implementierung des Durchschnitts ist in der Klasse *MeanAnalyser* enthalten.
Die Klasse *MeanAnalyser* berechnet außerdem die Abweichung des 95%-Konfidenzintervalls, hierfür wird ein separater Schlüssel, der *ciIdentifier* benötigt, welcher als Schlüssel für diese Ergebnisse verwendet wird. Dieser Wert wird hierbei als getrennte Analyse dem Datenspeicher übergeben.

4.3.4. Wächter

Eine beispielhafte Implementierung für eine Wächterkomponente ist die Klasse *RegressionGuard*, welche das aktuelle Ergebnis einer beliebigen Analyse mit dem zweit-aktuellsten Ergebnis der Analyse eines erfolgreichen Builds vergleicht und wahlweise den Build fehlschlagen lässt oder nur eine Warnung ausgibt, falls der prozentuale Unterschied einen konfigurierbaren Grenzwert überschreitet.
Der Wächter benötigt hierzu den *analysisIdentifier* der Analyse und den *benchmarkIdentifier* des Benchmarks, um die Analyseergebnisse zu bestimmen, die kontrolliert werden sollen. Als Grenzwert wird das *regressionLimit* angegeben. Falls die Formel

$$alterWert / neuerWert > 1 + regressionLimit / 100$$

erfüllt ist, ist das gegebene Regressionslimit überschritten. Dann wird eine entsprechende Ausgabe im Log des Builds hinterlegt und falls *isFailureCritical* gesetzt ist, wird der Build abgebrochen. Der neue Wert ist hierbei der Wert des aktuellsten Builds, und der alte Wert ist der Wert des aktuellsten erfolgreichen Builds. Dieser wird anhand der *buildID* bestimmt, welche als *buildIdentifier* den Analyseergebnissen zugeordnet ist.

4.3.5. Darstellung

Um eine grafische Darstellung der Analyseergebnisse in Jenkins zu erreichen, verwende ich das Plot Plugin von Nigel Daley und Eric Nielsen, welches über die Plug-in-Quellen von Jenkins erhältlich ist. Dieses Plug-in liest mit jedem Build die Werte aus einer CSV-Datei, im folgenden als Eingangsdatei bezeichnet, und speichert sie zusammen mit den bisherigen Daten in einer separaten CSV-Datei, im folgenden Ausgangsdatei genannt, ab. Diese Ausgangsdatei wird anschließend als Grafik dargestellt.
Die Eingangsdatei muss dabei immer wie folgt aufgebaut sein:

Tabelle 4.5. Aufbau der Eingangs-Datei für das Plot Plugin

Beschriftung 1	Beschriftung 2	Beschriftung 3	...
Datum 1	Datum 2	Datum 3	...

Die Spalten stehen hierbei immer für einen zusammengehörigen Datenstrang. Die Eingangsdatei ist nach jedem Build die selbe, daher müssen die Daten überschrieben werden.
Die Klasse *PlotPluginOutput* implementiert einen Builder, der die Daten von bis zu 5 Analysen für einen Grafen vorbereitet. Dafür nimmt der Builder 5 *analysisIdentifier* und 5 *benchmarkIdentifier* als Parameter entgegen, um die Analysen zu identifizieren, deren Ergebnisse angezeigt werden sollen. Leer gelassene Felder werden hierbei ignoriert, sodass es auch möglich ist, weniger Datensätze anzuzeigen. Als zusätzliches Argument der Klasse muss der Ausgabename der Datei gewählt werden. Diese wird im Workspace des Jobs erstellt. Der Name muss anschließend genau so im Plot Plugin eingegeben werden.
Für die Einrichtung empfiehlt es sich, bereits eine leere Eingangsdatei anzulegen, da die Eingabekontrolle des Plot Plugins nur Pfade akzeptiert, die schon existieren. Diese Datei wird von *PlotPluginOutput* bei der Verwendung überschrieben, sodass der Inhalt nicht relevant ist.
Neben der grafischen Ausgabe bietet das Plot Plugin auch die Möglichkeit, zusätzlich die zugrunde liegende Ausgangsdatei anzuzeigen. Dies kommt dann einer tabellarischen Gegenüberstellung der Ergebnisse gleich, sodass die Analyseergebnisse auch in Zahlen innerhalb von Jenkins ablesbar sind.

4.4. Erweiterung des Systems

Die in dieser Arbeit entwickelten und vorgestellten Implementierungen der Komponenten können um weitere Varianten ergänzt werden, um die Implementierung den individuellen Anforderungen von Projekten anzupassen. Insbesondere die Durchführungskomponenten müssen für andere Softwaretypen angepasst oder neu implementiert werden, da die hier entwickelte Komponente nur für das Ausführen von MooBench-Benchmarks ausgelegt ist. Da die Komponenten untereinander nur über den Datenspeicher kommunizieren, ist es möglich, einzelne Komponenten auszutauschen, ohne die Funktionalität der anderen Komponenten zu beschränken. Bei einer Neuimplementierung der Durchführungskomponente muss lediglich darauf geachtet werden, dass der in einem *Builder* mit *build.getId()* abfragbare Build-Identifizierer beim Speichern der Ergebnisse in der Datenbank als *buildIdentifier* angegeben wird, um die Funktionalität des in Unterabschnitt 4.3.4 vorgestellten Wächters nicht einzuschränken.
Bei der Ersetzung der Datenspeicherkomponente müssen alle Komponenten so angepasst werden, dass sie den neuen Datenspeicher verwenden. Dies ist am leichtesten zu erreichen,

wenn der Datenspeicher das in Unterabschnitt 4.3.1 vorgestellte Interface *Storage* implementiert, welches alle Methoden enthält, die zum Speichern und Auslesen der Daten benötigt werden.

4.5. Installation und Konfiguration

Die Installation des Plug-ins geschieht durch das Einfügen der Datei *AutoMicroBenchmark.hpi* in Jenkins unter *Jenkins Verwalten -> Plug-ins Verwalten -> Erweiterte Einstellungen*. Hier kann in der Rubrik *Plug-in Hochladen* die Datei ausgewählt und anschließend installiert werden. Danach stehen Jobs die Komponenten als Build-Schritte im Konfigurationsmenü des Jobs zur Verfügung. Zusätzlich muss das Plot Plugin installiert werden, falls dies noch nicht geschehen ist. Dies kann unter *Jenkins Verwalten -> Plug-ins Verwalten -> Verfügbar* in der Jenkins-Oberfläche von den offiziellen Plug-in-Quellen heruntergeladen und installiert werden.
Die Konfigurationsparameter sind in den einzelnen Beschreibungen der Komponenten in Abschnitt 3.2 beschrieben. Zusätzlich sind Hilfetexte und Eingabekontrollen eingerichtet, die bei der Konfiguration helfen sollen. Ein Beispiel für die Konfiguration unter Windows sieht wie folgt aus:

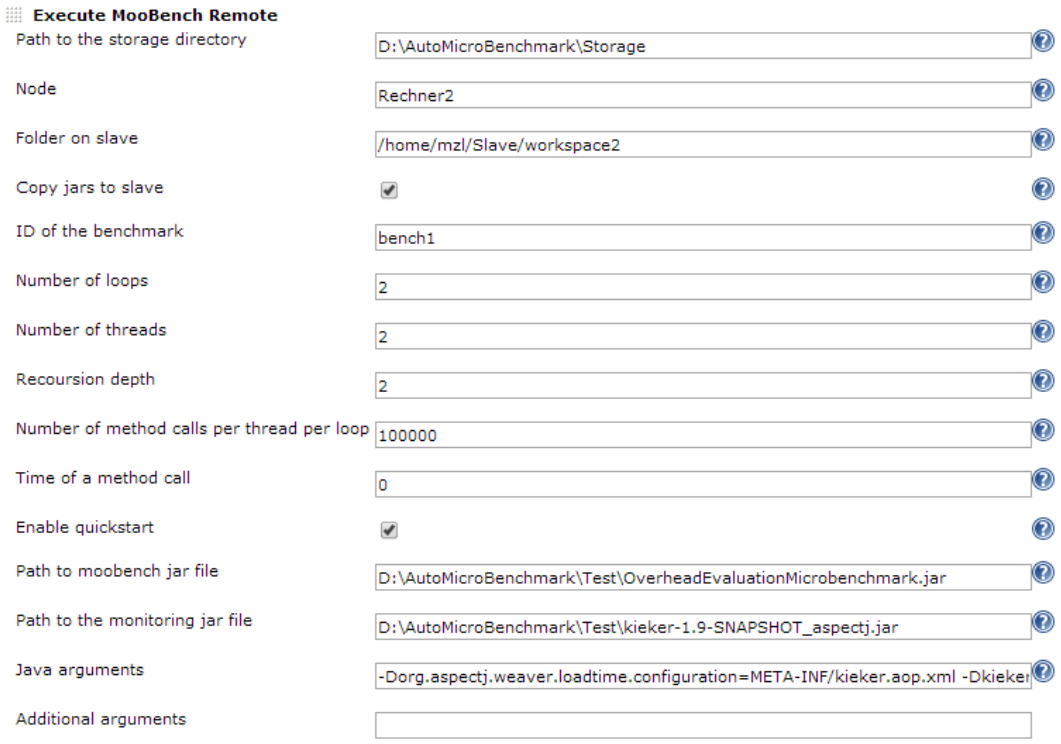

Abbildung 4.2. Ein Konfigurationsbeispiel für die Durchführungskomponente *MooBenchExecutorRemote*

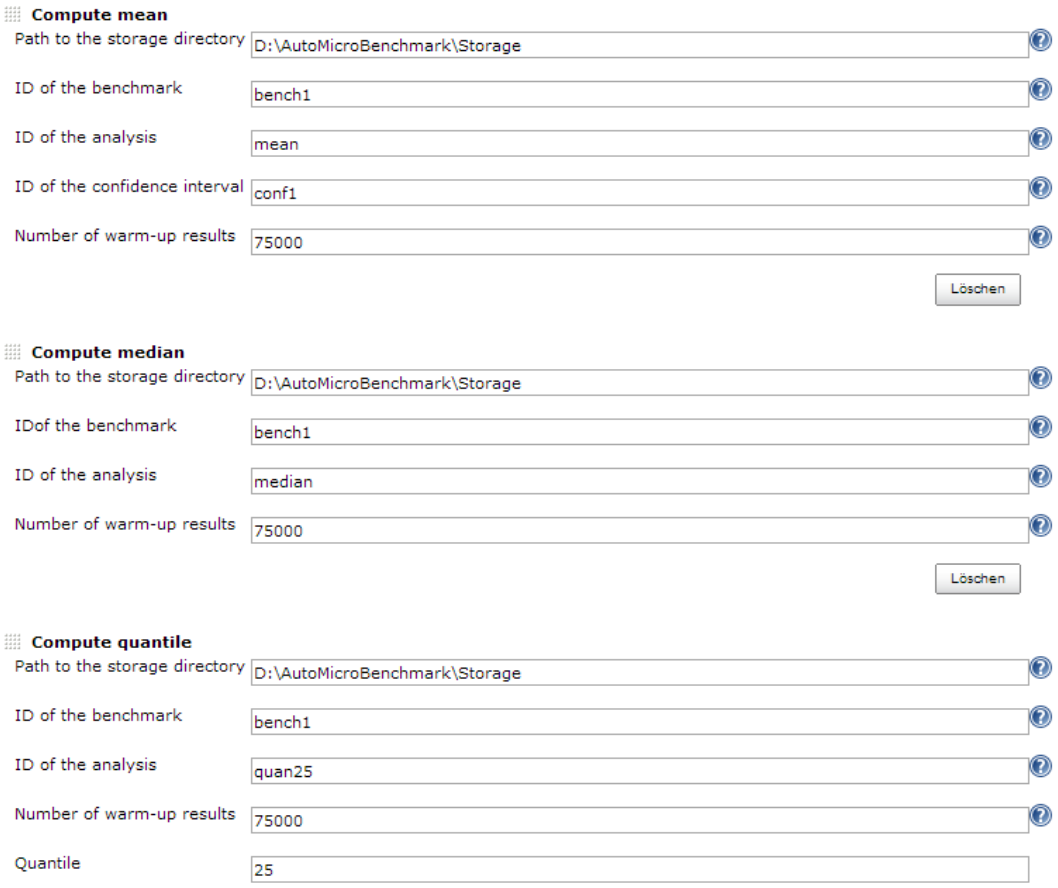

Abbildung 4.3. Konfigurationsbeispiel für die Analysekomponenten

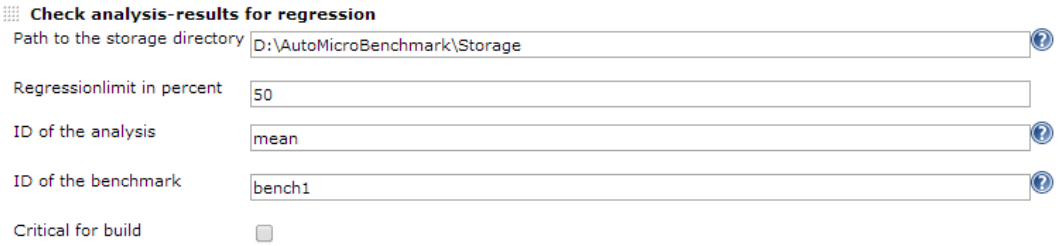

Abbildung 4.4. Konfigurationsbeispiel für die Wächterkomponente

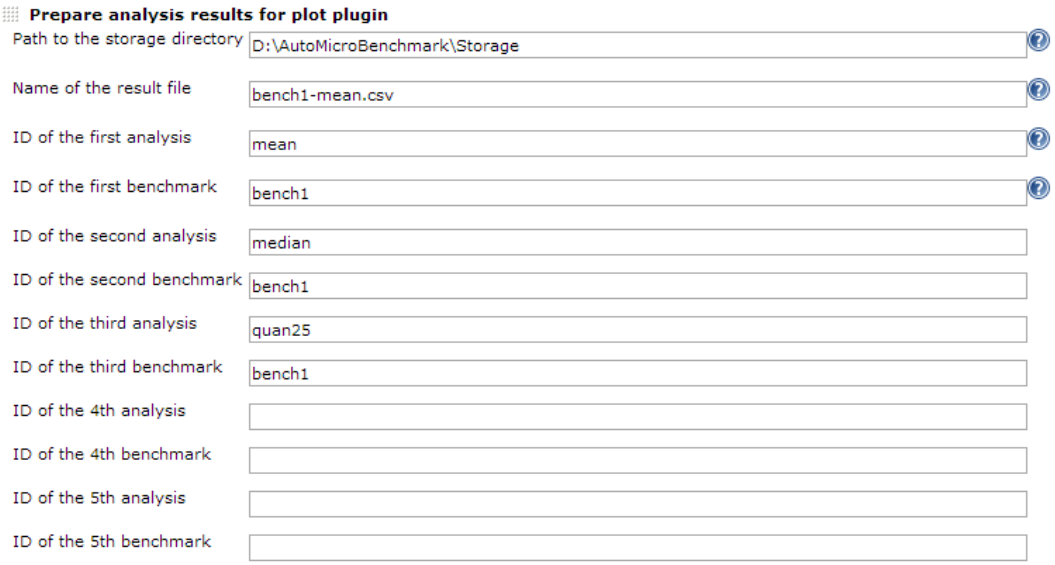

Abbildung 4.5. Konfigurationsbeispiel für die Präsentation-Vorbereitungs-Komponente

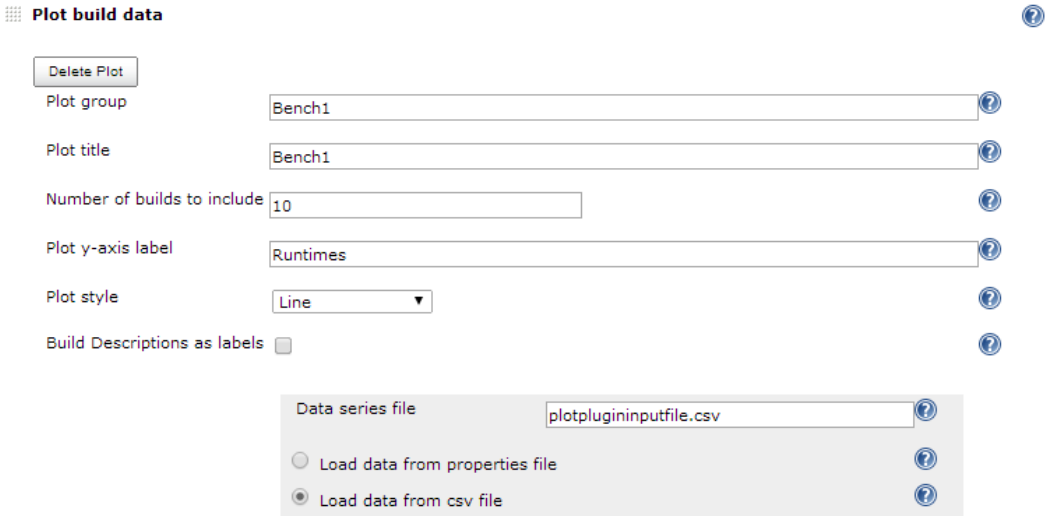

Abbildung 4.6. Konfigurationsbeispiel für das Plot Plugin

Die Ergebnisse werden dann durch das Plot Plugin wie folgt dargestellt:

Build #	mean	median	quan25
87	258496.61861540604	37670.0	28260.0
88	128366.45679999751	57671.0	37873.0
89	125181.39904615386	61299.0	40044.0
90	131105.58144615445	52101.5	38524.75
91	153389.94707690863	53539.0	31677.25
92	179755.90375386583	45342.5	30890.0
93	59779.20067692772	31174.0	23909.0
94	53979.64972921303	33690.0	24442.0
95	56790.24189843658	28642.0	23442.0
96	61157.50661231424	29604.0	23904.0
97	51923.385255404915	36735.0	27209.0

Abbildung 4.7. Tabelle in der Anzeige des Plot Plugin

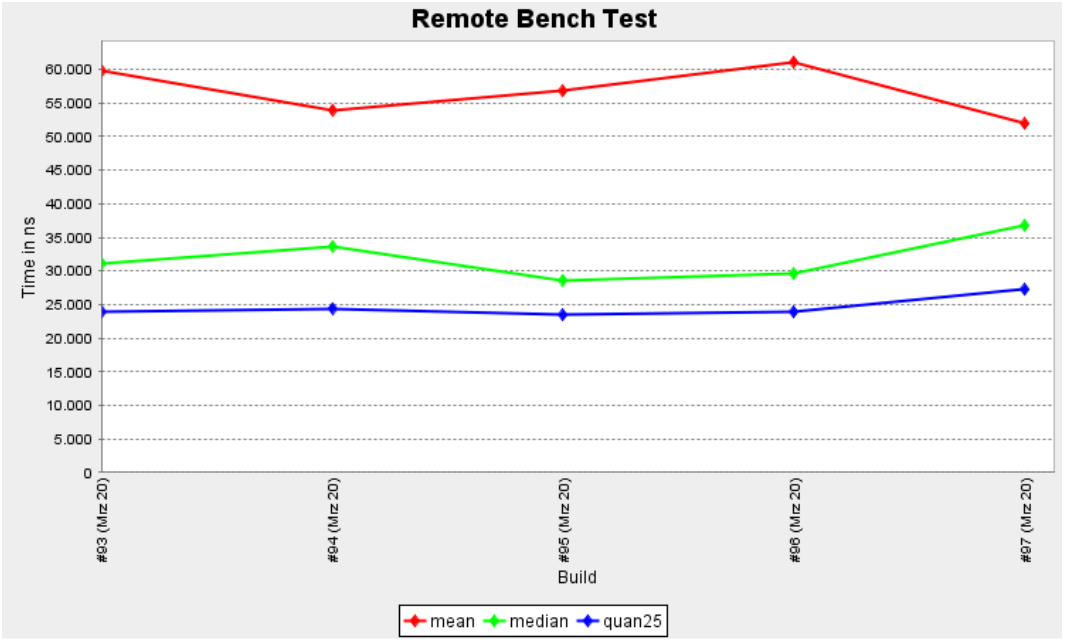

Abbildung 4.8. Diagramm in der Anzeige des Plot Plugin

4.6. Zusammenfassung

Dieses Kapitel beschreibt die Implementierung des in Kapitel 3 vorgestellten Systems. Die Komponenten finden ihre Entsprechungen in wie in Tabelle 4.6 dargestellt.

Tabelle 4.6. Implementierungen der Komponenten

Steuerung	entfällt, ersetzt durch Jenkins
Datenspeicher	*CSVStorage*
Durchführung	*MooBenchExecutor*, *RemoteMooBenchExecutor*
Analyse	*MeanAnalyser*, *MedianAnalyser*, *QuantileAnalyser*
Wächter	*RegressionGuard*
Darstellung	*PlotPluginOutput* in Kombination mit Plot Plugin

Hierbei speichert der Datenspeicher die gesamten Daten in CSV-Dateien unter einem konfigurierbaren Pfad, die Durchführungskomponenten implementieren MooBench-Skripte, die Analysekomponenten bilden die in Unterabschnitt 2.1.4 beschriebenen Methoden zur Analyse ab, die Wächterkomponente überprüft die Performance von Builds auf Regression und die Klasse *PlotPluginOutput* bereitet Analyseergebnisse zur Darstellung mittels Plot Plugin auf.
Diese Komponenten sind ausreichend, um automatisch MooBench-Benchmarks für Kieker durchzuführen. Ein ausführlicher Test und die Bewertung folgen in Kapitel 5.

Kapitel 5

Evaluierung

Abschließend zur Entwicklung des Systems in Kapitel 3 und seiner Implementierung folgt in diesem Kapitel die Bewertung beider. Zur Bestimmung von Evaluierungsmetriken wird die Goal-Question-Metric-Methode [Basili u. a. 1994] verwendet. Als Grundlage für die Bewertung dienen hier die Anforderungen aus Abschnitt 1.2.

5.1. Entwicklung von Evaluierungsmetriken

Beim Goal-Question-Metric-Verfahren werden zuerst Ziele festgelegt, die erreicht werden sollen. Im Falle dieser Arbeit entsprechen die Ziele denen aus Abschnitt 1.2, also:

1. Plattformunabhängigkeit
2. Konfigurierbarkeit
3. Ausführen von Benchmarks
4. Speichern der Ergebnisse
5. Weiterverarbeitung der Ergebnisse
6. Kontrolle der verarbeiteten Ergebnisse
7. Präsentation der Ergebnisse

Sowie zusätzlich für die Implementierung

1. Installierbarkeit
2. Automatische Ausführung von MooBench
3. Portierbarkeit

Zu diesen Zielen werden Fragen gestellt, die die Erfüllung der Ziele betreffen, also zum Beispiel "Wie Plattformunabhängig ist das System?", für diese Fragen werden dann Metriken entworfen, welche die Fragen beantworten können, also in diesem Beispiel die Anzahl der Plattformen, auf denen das System eingesetzt werden kann.
Um die Evaluierungsmetriken zu ermitteln, gehe ich alle Ziele durch und stelle die

möglichen Fragen und für deren Beantwortung anwendbare Metriken vor. System und Implementierung werden getrennt behandelt, da ihre Entwicklung und Bewertung ebenfalls separat aufgeführt ist.

5.1.1. Evaluierungsmetriken für das System

Plattformunabhängigkeit

Hiermit ist die Plattformunabhängigkeit im Bezug auf die Wahl des Continuous-Integration-Systems gemeint, wie der Beschreibung des Ziels zu entnehmen ist.

- ▻ Wie Plattformunabhängig ist das System?
  1. Messen des Anteils der CI-Systeme, auf denen sich das System implementieren lässt.

Konfigurierbarkeit

- ▻ Ist das System innerhalb der Oberfläche konfigurierbar?
  1. Prüfen, ob eine solche Konfigurationsmöglichkeit vorgesehen ist.
  2. Anteil der Konfigurationsmöglichkeiten, die über die Oberfläche bedienbar sind, bestimmen.
- ▻ Lassen sich die in der Beschreibung des Ziels genannten Teile in der Oberfläche konfigurieren?
  1. Prüfen der Konfigurationsmöglichkeit der auszuführenden Benchmarks in der Oberfläche.
  2. Prüfen der Konfigurationsmöglichkeit der Verarbeitungsmethoden in der Oberfläche.
  3. Prüfen der Konfigurationsmöglichkeit der Kontrollmethoden in der Oberfläche.
  4. Prüfen der Konfigurationsmöglichkeit der Präsentationsmethode in der Oberfläche.

Ausführung von Benchmarks

- ▻ Können Benchmarks ausgeführt werden?
  1. Prüfen, ob die Ausführung von Benchmarks durch das System möglich ist.
- ▻ Geschieht die Ausführung automatisch?
  1. Prüfen, ob die automatische Ausführung durch das System möglich ist.
- ▻ Ist das System von der Art der Benchmarks unabhängig?
  1. Prüfen, welche Einschränkungen für auszuführende Benchmarks vorliegen.

Speichern der Ergebnisse

- ▻ Werden die Ergebnisse gespeichert?
  1. Prüfen, ob das Speichern der Ergebnisse in das System integriert wurde.

Weiterverarbeitung der Ergebnisse

- ▻ Können die Ergebnisse weiter verarbeitet werden?
  1. Prüfen, ob die Weiterverarbeitung der Ergebnisse durch das System möglich ist.
  2. Prüfen, ob die automatische Weiterverarbeitung der Ergebnisse durch das System möglich ist.
- ▻ Ist die Verarbeitung unabhängig von der Art der Benchmarks?
  1. Prüfen, welche Einschränkungen für verarbeitende Methoden vorliegen.
  2. Prüfen, ob verschiedene Verarbeitungsmethoden integriert werden können.

Kontrolle der verarbeiteten Ergebnisse

- ▻ Können Ergebnisse nach der Weiterverarbeitung hinsichtlich beliebiger Kriterien überprüft werden?
  1. Prüfen, welche Einschränkungen für Kontrollen gelten.
  2. Prüfen, was kontrolliert werden kann.
  3. Verschiedene Kontrollmöglichkeiten integrieren.

Präsentation der Ergebnisse

- ▻ Wie werden die Ergebnisse präsentiert?
  1. Prüfen, ob die Präsentation der Ergebnisse in die Oberfläche integriert ist.
- ▻ Ist die Form der Präsentation geeignet für menschliche Betrachter?
  1. Prüfen, ob die Präsentation einen schnellen Überblick ermöglicht.
  2. Prüfen, ob die Präsentation vergleichend Ergebnisse darstellt.
  3. Prüfen, ob sich die Ergebnisse in der Präsentation eindeutig den zugehörigen Tests zuordnen lassen.

Vergleichbarkeit der Ergebnisse

- ▻ Ist die Vergleichbarkeit der Ergebnisse gewährleistet?
  1. Prüfen, ob Maßnahmen getroffen wurden, um die Erhaltung der Vergleichbarkeit zu ermöglichen.

5.1.2. Evaluierungsmetriken für die Implementierung

Viele der für das System entwickelten Metriken lassen sich in abgewandelter Form auch für die Implementierung anwenden, an anderer Stelle kommen aber auch neue hinzu, deshalb wiederhole ich die Ziele für die Implementierung noch einmal, da sich die Redundanz der Metriken auf ein Minimum beschränkt.

Plattformunabhängigkeit

Die Plattformunabhängigkeit im Bezug auf Continuous-Integration-Systeme ist durch die Umsetzung für Jenkins nicht gegeben, stattdessen untersuche ich die Plattformunabhängigkeit im Bezug auf Betriebssysteme.

- ▻ Auf welchen Betriebssystemen funktioniert die Implementierung?
  1. Funktioniert ein Test der Implementierung unter Windows?
  2. Funktioniert ein Test der Implementierung unter Linux?

Konfigurierbarkeit

- ▻ Ist die Implementierung innerhalb der Oberfläche konfigurierbar?
  1. Anteil der Konfigurationsmöglichkeiten, die über die Oberfläche bedienbar sind.
- ▻ Lassen sich die in der Beschreibung des Ziels genannten Teile in der Oberfläche konfigurieren?
  1. Prüfen der Konfigurationsmöglichkeit der auszuführenden Benchmarks in der Oberfläche.
  2. Prüfen der Konfigurationsmöglichkeit der Verarbeitungsmethoden in der Oberfläche.
  3. Prüfen der Konfigurationsmöglichkeit der Kontrollmethoden in der Oberfläche.
  4. Prüfen der Konfigurationsmöglichkeit der Präsentationsmethode in der Oberfläche.

Ausführung von Benchmarks

- ▻ Können Benchmarks ausgeführt werden?
  1. Prüfen, ob die Ausführung von Benchmarks durch die Implementierung funktioniert.
- ▻ Geschieht die Ausführung automatisch?
  1. Prüfen, ob die Ausführung durch die Implementierung automatisch geschieht.
- ▻ Ist die Implementierung von der Art der Benchmarks unabhängig?
  1. Prüfen, welche Einschränkungen für auszuführende Benchmarks vorliegen.
  2. Testen, ob sich weitere Benchmark-Methoden ausführen lassen.

Speichern der Ergebnisse

- ▷ Werden die Ergebnisse gespeichert?
  1. Messen, welcher Anteil der produzierten Ergebnisse gespeichert wird.
  2. Bestimmung des Fehlers durch das Speichern der Ergebnisse.

Weiterverarbeitung der Ergebnisse

- ▷ Können die Ergebnisse weiterverarbeitet werden?
  1. Prüfen, ob die Weiterverarbeitung der Ergebnisse funktioniert.
  2. Prüfen, ob die Weiterverarbeitung der Ergebnisse automatisch geschieht.
- ▷ Ist die Verarbeitung unabhängig von der Art der Benchmarks?
  1. Prüfen, welche Einschränkungen für die verarbeitenden Methoden vorliegen.
  2. Testen, ob verschiedene Verarbeitungsmethoden verwendbar sind.

Kontrolle der verarbeiteten Ergebnisse

- ▷ Können Ergebnisse nach der Weiterverarbeitung hinsichtlich beliebiger Kriterien überprüft werden?
  1. Prüfen, welche Einschränkungen für Kontrollen gelten.
  2. Prüfen, was kontrolliert werden kann.
  3. Verschiedene Kontrollmöglichkeiten integrieren.
- ▷ Funktionieren implementierte Kontrollen?
  1. Test mit absichtlicher Überschreitung der Grenzwerte.

Präsentation der Ergebnisse

- ▷ Wie werden die Ergebnisse präsentiert?
  1. Prüfen, ob die Präsentation der Ergebnisse in die Oberfläche integriert ist.
- ▷ Ist die Form der Präsentation geeignet für menschliche Betrachter?
  1. Prüfen, ob die Präsentation einen schnellen Überblick ermöglicht.
  2. Prüfen, ob die Präsentation Ergebnisse vergleichend darstellt.
  3. Prüfen, ob sich die Ergebnisse in der Präsentation eindeutig den zugehörigen Tests zuordnen lassen.

Vergleichbarkeit der Ergebnisse

- ▷ Ist die Vergleichbarkeit der Ergebnisse gewährleistet?
  1. Vergleich der Messergebnisse des gleichen Experiments unter verschiedenen Bedingungen.

Installierbarkeit

- ▻ Lässt sich die Implementierung einfach installieren?
  1. Messung der Zeit für die Installation.
  2. Bewertung des nötigen Wissens für die Installation.

Automatische Ausführung von MooBench

- ▻ Führt die Implementierung MooBench-Benchmarks aus?
  1. Test der Implementierung.
  2. Vergleich der Ergebnisse mit durch die Skripte produzierten Ergebnissen.

Portierbarkeit

- ▻ Wie aufwändig ist es, die Implementierung an Benchmarks für eine andere Software anzupassen?
  1. Anzahl der auszutauschenden Komponenten bestimmen.
  2. Bestimmung des weiterhin verwendbaren Anteils.
  3. Bewertung anhand subjektiver Erfahrungen.

5.2. Evaluierung des Systems

Nach der Entwicklung der Evaluierungsmetriken in Abschnitt 5.1 wende ich nun diese Metriken an, um die Qualität des entwickelten Systems zu testen.

5.2.1. Plattformunabhängigkeit

Die Plattformunabhängigkeit lässt sich als konkrete Zahl nur durch den Anteil der Continuous-Integration-Systeme ausdrücken, die eine Implementierung des Systems ermöglichen. Dieser Anteil ist aber durch die große Zahl und Vielfalt existierender CI-Systeme nur durch Implementierung für jedes einzelne exakt bestimmbar.
Da CI-Systeme aber meist darauf ausgelegt sind, vielfältige Build-Prozesse zu unterstützen, bieten die meisten Systeme auch Möglichkeiten, diese Systeme zu erweitern. Durch die sehr abstrakte Beschreibung des Systems sollte es dadurch möglich sein, es in die meisten der CI-Systeme einzufügen. Ich bewerte daher die Plattformabhängigkeit als gegeben.

5.2.2. Konfigurierbarkeit

Die Konfigurierbarkeit über eine Benutzeroberfläche wird durch die Steuerungskomponente hergestellt. Wie genau sie umgesetzt wird, ist nicht vorgegeben. So können

Konfigurationsmöglichkeiten beispielsweise in die Oberfläche des Continuous-Integration-Systems integriert werden oder die Steuerungskomponente kann über eine eigene grafische Oberfläche verfügen. Es sind also Möglichkeiten gegeben, die Konfiguration über eine Benutzeroberfläche zu realisieren.
Durch die Anbindung der anderen Komponenten an die Steuerung wird diese Möglichkeit ebenfalls für die Wahl der auszuführenden Benchmarks, der Verarbeitungsmethoden, der Kontrollmethoden und der Präsentationsmethode hergestellt.
Somit ist die Konfigurierbarkeit nach den Metriken aus Unterabschnitt 5.1.1 gegeben, auch wenn die Umsetzung von der Implementierung abhängt.

5.2.3. Ausführung von Benchmarks

Das System sieht die Durchführung von Benchmarks durch Durchführungskomponenten vor, wie in Abschnitt 3.2 beschrieben. In Abschnitt 3.4 beschreibe ich, wie dieser Vorgang durch die Einbindung des Systems in den Build-Vorgang durch Continuous-Integration-Systeme automatisiert wird. Die zu verwendenden Benchmarks unterliegen hierbei nur den Einschränkungen, durch die Steuerung konfigurier- und startbar zu sein und müssen die Datenformate des Datenspeichers verwenden. Mit welcher Methode die Ergebnisse produziert werden, ist für das System nicht relevant.

5.2.4. Speichern der Ergebnisse

Abschnitt 3.3 beschreibt die Struktur der Daten, welche von der in Abschnitt 3.2 beschriebenen Datenspeicherkomponente genutzt wird, um die Ergebnisse der Durchführungs- und Analysekomponenten zu speichern. Dadurch, dass die Komponenten nur über den Datenspeicher kommunizieren, wird sichergestellt, dass alle Daten gespeichert werden.

5.2.5. Weiterverarbeitung der Ergebnisse

Ebenso wie die Durchführung von Benchmarks mit der Durchführungskomponente ist das Ausführen unterschiedlicher Analysen durch das System in Abschnitt 3.2 beschrieben und geschieht durch die Integration in Continuous-Integration-Systeme automatisch.
Die verarbeitenden Methoden werden nur durch die Datenstruktur eingeschränkt, dies gilt sowohl für das Auslesen der Ergebnisse der Durchführung sowie das Speichern der Analyseergebnisse. Zudem muss die Konfiguration und das Starten über die Steuerungskomponente möglich sein. Jede Analysemethode, die diesen Anforderungen entspricht, lässt sich in das System integrieren und kann automatisch ausgeführt werden.

5.2.6. Kontrolle der verarbeiteten Ergebnisse

Die in Abschnitt 3.2 vorgestellten Wächterkomponenten ermöglichen durch ihre Integration in Continuous-Integration-Systeme die automatische Kontrolle von Benchmark- oder

Analyseergebnissen hinsichtlich verschiedener Kriterien. Welche Aktionen Wächterkomponenten ausführen, falls sie ausgelöst werden, ist nicht eingeschränkt. Die Konfiguration von Grenzwerten sowie die Ausführung der Kontrolle muss lediglich durch die Steuerungskomponente möglich sein. Zudem dürfen Wächterkomponenten nur über den Datenspeicher auf Ergebnisse anderer Komponenten zugreifen und unterliegen so auch den Einschränkungen der in Abschnitt 3.3 beschriebenen Datenstruktur.

5.2.7. Präsentation der Ergebnisse

Die Präsentation der Ergebnisse wird durch die in Abschnitt 3.2 beschriebenen Darstellungskomponenten realisiert. Diese unterliegen den selben Einschränkungen wie die Wächterkomponenten, können aber aus den Daten beliebige Darstellungen erzeugen, die dann entweder in der Oberfläche des Continuous-Integration-Systems oder einer separaten Oberfläche angezeigt werden können. Die Übersichtlichkeit der Daten hängt dann von der Implementierung ab.
Darstellungskomponenten ist es hierbei möglich, auf beliebig viele Datensätze zuzugreifen, da der Zugriff auf die Daten nur durch die Datenstruktur beschränkt ist. So können vergleichende Darstellungen erzeugt werden. Die eindeutige Zuordnung ist durch die Schlüssel *BenchmarkID* und *AnalyseID* gegeben, sodass es möglich ist, Analyseergebnisse eindeutig der Analyse und dem Benchmark zuzuordnen, zu denen sie gehören. Durch die *BuildID* ist außerdem sichergestellt, dass der Build, zu dem die Ergebnisse gehören, bekannt ist. Für die Durchführungsergebnisse gilt das gleiche durch *BenchmarkID* und *BuildID*.

5.2.8. Vergleichbarkeit der Ergebnisse

Maßnahmen zur Erhaltung der Vergleichbarkeit der Ergebnisse erläutere ich in Abschnitt 3.7. Die Umsetzung muss durch eine Implementierung geschehen und kann nicht durch das System garantiert werden. Die geringen Einschränkungen der Durchführungskomponenten ermöglichen jedoch die Wahrung der Vergleichbarkeit der Ergebnisse, sodass diese Anforderung für das System als erfüllt zu werten ist.

5.3. Evaluierung der Implementierung

Im Gegensatz zu der Evaluierung des Systems können bei der Evaluierung der Implementierung konkrete Tests ausgeführt werden. Hierzu verwende ich die folgende Konfiguration, sofern ich nichts anderes angebe:
Als Plattform dient ein Rechner mit Windows 8.1 Pro in der 64-Bit Variante, der mit einem AMD Phenom II X4 965 Prozessor mit einer Taktrate von 3.4GHz und 12GB Arbeitsspeicher ausgestattet ist. Auf einem Notebook mit Ubuntu-Linux in der Version 13.10 und einem Intel-Core i5-4200U Prozessor mit bis zu 2.6GHz und 4GB Ram läuft der vom

RemoteMooBenchExecutor benötigte Slave. Ich verwende die Jenkins-Version, die durch das Netbeans-Plug-in zur Entwicklung von Jenkins-Plug-ins bereitgestellt wird.
Zum Testen verwende ich einen extra dafür angelegten Job, welcher in der Build-Phase aus den Schritten

1. Execute MooBench Remote
2. Compute mean
3. Compute median
4. Compute quantile
5. Check analysis results for regression
6. Prepare analysis results for plot plugin
7. Plot build data

besteht. Die Konfiguration der Komponenten entspricht der in Abschnitt 4.5 gezeigten. Der Moobench-Test hat die Java-Argumente

```
1 -Dorg.aspectj.weaver.loadtime.configuration=META-INF/kieker.aop.xml
2 -Dkieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter
3 -Dkieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=/home/mzl/tmp
```

Die anderen Parameter sind 2 Schleifendurchläufe, 2 Threads, Rekursionstiefe 2, 100.000 Methoden-Aufrufen mit der Methodenzeit 0 und mit Schnellstart. Der Warm-Up der Analysekomponenten ist auf 75.000 gesetzt.

5.3.1. Plattformunabhängigkeit

Zur Prüfung der Plattformunabhängigkeit im Bezug auf das Betriebssystem führe ich den oben genannten Test mit dem lokalen *MooBenchExecutor* auf den oben genannten Systemen durch.
Der Test verlief unter Windows und unter Ubuntu erfolgreich, der Build wurde ohne Fehler abgeschlossen und erzeugte die folgenden Analyseergebnisse, welche auch durch das Plot-Plugin dargestellt wurden:

Tabelle 5.1. Ergebnisse des Tests unter Ubuntu 13.10 und Windows 8.1 auf verschiedenen Rechnern

	Windows	Ubuntu
Median	12712	30697
Konfidenz-Intervall	1738411	10012642
Durchschnitt	31213	48452
25-Quantil	12222	23295

Die starken Unterschiede in den Endergebnissen zwischen den Plattformen sind auf die verwendete Hardware zurückzuführen. Der Test hat gezeigt, dass die Implementierung auf beiden Systemen funktioniert.

5.3.2. Konfigurierbarkeit

Die Konfigurierbarkeit ist durch die Oberfläche von Jenkins implementiert. Die Komponenten bieten die Konfigurationsmöglichkeiten, die in den einzelnen Beschreibungen in Abschnitt 4.3 aufgeführt und in Abschnitt 4.5 zu sehen sind. Was nicht über die Oberfläche konfigurierbar ist, ist die zugrunde liegende Dateistruktur, der Pfad hingegen wird durch die *storagePath*-Attribute jeder Komponente bestimmt.
Die Zusammenstellung der zu verwendenden Komponenten kann innerhalb der Jenkins-Oberfläche durch das Hinzufügen der entsprechenden Builder geschehen. Da diese Builder vollkommen unabhängig voneinander sind, kann die Wahl der auszuführenden Benchmarks, Verarbeitungsmethoden, Kontrollmethoden und Präsentationsmethoden beliebig erfolgen. Somit ist die Konfigurierbarkeit innerhalb der Oberfläche soweit möglich, dass die Anforderungen diesbezüglich erfüllt sind.

5.3.3. Ausführung von Benchmarks

Die Durchführung von Benchmarks wurde bereits durch die Tests in Unterabschnitt 5.3.1 gezeigt. Die Durchführung geschieht durch das Einfügen von Buildern in den Build-Prozess automatisch mit jedem bis zu diesem Build-Schritt erfolgreichen Build, sodass auch die Automatisierung dieses Vorgangs gegeben ist.
Die Einschränkungen für Benchmarks sind dadurch gegeben, dass sie als Builder implementiert werden müssen und die Methoden des Datenspeichers zum Speichern der Ergebnisse nutzen müssen. Um zu testen, ob sich weitere Benchmark-Methoden ausführen lassen, verwende ich eine neue Durchführungskomponente, welche durch die Klasse *TestBenchmark* implementiert ist und deren *Perform*-Methode lediglich die Zahlen 1 bis 10 in den Datenspeicher schreibt. Dieser Builder ist hinsichtlich der Funktion minimal, genügt aber den Anforderungen an Durchführungskomponenten.

Die *Perform*-Methode dieses Builders ist wie folgt aufgebaut:

```
1  //Get storage
2  CSVStorage storage;
3  storage = CSVStorage.getInstance(storagePath);
4  //Get buildID
5  final String buildIdentifier = build.getId();
6  //Save results
7  double[] d = new double[]{1d,2d,3d,4d,5d,6d,7d,8d,9d,10d};
8  storage.saveBenchmarkResult(
9          new BenchmarkResultSet(d, benchmarkIdentifier, buildIdentifier));
10 return true;
```

Ich verwende die gleichen Test-Einstellungen wie in Abschnitt 5.3 und ersetze lediglich den *MooBenchExecutor* durch den *TestBenchmark* und setze die Warm-Up Parameter der Analysekomponenten auf 2, sodass alle Ergebnisse ab 3 gewertet werden. Der Build verläuft erfolgreich und produziert die folgenden Ergebnisse:

Tabelle 5.2. Ergebnisse der Durchführungskomponente *TestBenchmark*

Median	6.5
Konfidenz-Intervall	4.8
Durchschnitt	6.5
25-Quantil	4.25

Dass die Ergebnisse für den Median und das 25-Quantil nicht in den tatsächlichen Ergebnissen vorkommt, liegt an der Implementierung der Commons-Math-Bibliothek. Die dort verwendeten Methoden sind für große Datenmengen optimiert und berechnen nur einen Schätzwert.
Das Ergebnis des Tests ist, dass die automatische Durchführung von Benchmarks implementiert ist und die Anforderungen aus Abschnitt 1.2 erfüllt sind.

5.3.4. Speichern der Ergebnisse

Zur Evaluierung des Datenspeichers verwende ich die Dateien, die durch einen Test mit 100 Methodenaufrufen entstanden sind und vergleiche die Anzahl der Ergebnisse und berechne anschließend mit Hilfe der Software LibreOffice Calc [Seimert 2012; LibreOffice] in der Version 4 den Fehler.
Der Test ergibt, dass die Zahl der Ergebnisse mit 400 identisch und korrekt ist und die die vom Datenspeicher gespeicherten Werte nicht von den von MooBench produzierten abweichen. Daraus folgt, dass der Datenspeicher die von MooBench produzierten Werte

zuverlässig speichert.
Zum weiteren Test des Datenspeichers ist ein JUnit-Test für diese Komponente integriert, welcher die Methoden des Datenspeichers verwendet, um das Speichern und Lesen von Analyse- sowie Durchführungsergebnissen zu testen, indem erst Ergebnisse geschrieben, diese dann wieder ausgelesen und mit den ursprünglichen Daten verglichen werden. Dieser Test verläuft ebenfalls erfolgreich, sodass der Datenspeicher die Anforderungen aus Abschnitt 1.2 erfüllt.

5.3.5. Weiterverarbeitung der Ergebnisse

Die Weiterverarbeitung der Ergebnisse ist durch die Durchführung der Tests in Unterabschnitt 5.3.1 und Unterabschnitt 5.3.3 bereits gezeigt. Die Implementierung beinhaltet zudem 3 verschiedene Analysemethoden, die auch bei den Tests benutzt wurden. So ist sichergestellt, dass die Implementierung verschiedene Analysemethoden zulässt. Die Einschränkungen für Analysemethoden sind im wesentlichen die gleichen, die auch für die anderen Komponenten mit Ausnahme des Datenspeichers gelten: Analysemethoden müssen als Builder implementiert werden und ihre Ergebnisse in den Datenspeicher überführen. Die Ausgangsdaten können sie ebenfalls aus dem Datenspeicher beziehen.

5.3.6. Kontrolle der verarbeiteten Ergebnisse

Die Kontrolle der Ergebnisse erfolgt über Wächterkomponenten, die als Builder implementiert sind. Ein Wächter kann sich dabei beliebig verhalten, er muss nur als Builder implementiert werden. Er kann Daten aus dem Datenspeicher verwenden, könnte aber auch anhand anderer Kriterien agieren, wie etwa durch extern gemessene Systemauslastung. Durch diese geringen Einschränkungen der Wächter lassen sich beinahe beliebige Wächterkomponenten integrieren.
Die implementierte Wächterkomponente fragt mit Hilfe zweier Benchmark-Identifizierer die Ergebnisse des letzten erfolgreichen und des aktuellen Builds ab und vergleicht diese. Wird ein konfigurierbarer Grenzwert überschritten, kann der Build wahlweise fehlschlagen oder nur eine Ausgabe erzeugen. Um dieses Verhalten zu testen, erweitere ich die Klasse *TestBenchmark* aus Unterabschnitt 5.3.3 um zwei Felder in der Konfiguration, die den Wertebereich angeben, in dem die Benchmark-Ergebnisse liegen sollen. Diese Durchführungskomponente verwende ich, um drei Tests durchzuführen.
Im ersten Test prüfe ich, ob in dem Fall, dass sich die Veränderung des Ergebnisses innerhalb der gegebenen Grenze liegt, eine Ausgabe oder das Fehlschlagen des Builds fälschlicherweise ausgelöst wird. Hierzu verwende ich den *TestBenchmark* um die Ergebnisse 1-10 zu produzieren, lasse von der Klasse *MeanAnalyser* den Durchschnitt mit 2 Ergebnissen als Warm-Up bilden und überprüfe mit dem Wächter, ob sich das Ergebnis um mehr als 5% verändert hat. Hierbei ist die Option *Critical for build* aktiviert, sodass eine Überschreitung der Grenze das Fehlschlagen des Builds zur Folge haben würde. Die zweimalige Durchführung des Tests zeigt, dass weder der Build fehlschlägt noch eine

Warnung ausgegeben wurde.
Der zweite Test soll ermitteln, ob bei einer Überschreitung des Grenzwertes tatsächlich eine Aktion ausgelöst wird. Hierzu verwende ich die im ersten Test verwendeten Einstellungen für einen Durchlauf und führe danach noch einen Build mit den Werten 6-15 für den *TestBenchmark* aus, sodass sich der Durchschnitt um etwa 75% von 6,5 auf 11,5 erhöht. Das Resultat ist, dass der Build beim zweiten Durchlauf fehlschlägt und die Ausgabe *Regressiontest failed: Regression=0.7692307692307692%, regressionlimit=0.1%* in der Konsolenausgabe angezeigt wird. Ein dritter Durchlauf mit unveränderten Einstellungen erzeugt ebenfalls einen fehlschlagenden Build. Dies zeigt, dass wirklich mit dem Ergebnis des letzten erfolgreichen Builds verglichen wird.
Der letzte Test gleicht dem zweiten, jedoch ist die Option *Critical for build* deaktiviert. Dies soll zeigen, dass der Build trotz Überschreitung des Regressionslimits nicht fehlschlägt, aber eine Fehlerausgabe erzeugt wird. Auch dieser Test verläuft erfolgreich, beide Builds schlagen nicht fehl und im zweiten Build wird die Ausgabe *Regressiontest failed: Regression=0.7692307692307692%, regressionlimit=0.1%* ausgegeben.
Mit Hilfe der oben genannten Tests konnte ich zeigen, dass sich die implementierte Wächterkomponente verhält wie erwartet und so ihrer funktionalen Anforderung entspricht.

5.3.7. Präsentation der Ergebnisse

Die bisher durchgeführten Tests zeigen, dass die Ergebnisse der Analysen mit Hilfe der implementierten Darstellungskomponente möglich sind. Dadurch entstehen Ausgaben wie in Abbildung 4.8 ersichtlich.
Die manuelle Konfiguration der anzuzeigenden Daten pro Diagramm durch die Klasse *PlotPluginOutput* ermöglicht es, beliebige Analyseergebnisse in einem Diagramm zu kombinieren. Somit ist von der Konfiguration abhängig, wie gut sich die Daten vergleichen lassen. Wie durch die bisher durchgeführten Experimente gezeigt wurde und in Abbildung 4.8 ersichtlich ist, werden diese Ergebnisse dann gegeneinander und pro Build vergleichend dargestellt, sodass sowohl der Vergleich der Ergebnisse verschiedener Analysen als auch der Vergleich zwischen verschiedenen Revisionen möglich ist. Die Möglichkeit, die zugrunde liegende Tabelle über das Plot Plugin anzuzeigen, lässt außerdem die Anzeige der genauen Werte zu. Da die Anzahl der Graphen pro Diagramm durch die Klasse *PlotPluginOutput* auf 5 beschränkt ist, kann die Übersichtlichkeit eines Diagramms nicht durch die Anzahl der Graphen in einem Diagramm vermindert werden. Die Beschriftung der x-Achse mit der *BuildID* lässt die eindeutige Identifizierung des zugehörigen Builds zu und aus der angezeigten Legende ist ersichtlich, zu welcher Analysemethode welches Ergebnis gehört.

5.3.8. Vergleichbarkeit der Ergebnisse

In Abschnitt 3.7 sind Maßnahmen zur Wahrung der Vergleichbarkeit der Ergebnisse beschrieben. Die Implementierung setzt die Nutzung einer variablen Anzahl von Ergebnissen durch das Interface *Storage* um, damit Änderungen in der Build-Konfiguration ermöglicht

werden.
In Unterabschnitt 4.3.2 beschreibe ich einen lokalen *MooBenchExecutor*. Dieser führt die Experimente immer auf dem Build-System aus. Dadurch unterliegen die Ergebnisse den Einflüssen anderer Prozesse , wie etwa parallel laufender Builds, auf die Hardware. Dies umgeht der im gleichen Abschnitt vorgestellte *RemoteMooBenchExecutor*, indem die Experimente auf ein Slave-System ausgelagert werden. Um den Nutzen dieser Maßnahme zu demonstrieren, führe ich den eingangs genannten Test zum Vergleich 15 mal mit dem lokalen *MooBenchExecutor* auf dem Notebook aus und 15 mal mit der Remote-Variante auf dem Windows-Rechner mit dem Notebook als Slave. So werden die Experimente beide male auf dem gleichen System durchgeführt und die Ergebnisse werden nicht durch verschiedene Hardware verfälscht. Um den Einfluss durch äußere Belastung zu testen verwende ich das Programm Prime95 [Prime95], welches zur verteilten Berechnung von Primzahlen oder für Stresstests verwendet wird. Der verwendete Modus ist der *Blend*-Modus, welcher nach Angaben des Programms alle Komponenten etwas und den Speicher sehr belastet. Test 1 bis 5 laufen ohne Prime95, um Schwankungen zu messen, die bei alleiniger Ausführung auftreten. Bei Test 6 bis 10 starte ich Prime95 mit 2 Threads, bei Test 11 bis 15 mit 4 Threads, um die Belastung zu erhöhen. Dabei wird Prime95 immer auf dem System ausgeführt, auf dem der Jenkins-Master läuft. Die Ergebnisse sind in Abbildung 5.1 abgebildet.
Die Ergebnisse für die Tests ohne weitere Belastung liegen bei der lokalen Ausführung zwischen 53700 und 57787. Somit treten Schwankungen von ungefähr 10% auch ohne Belastung des Systems auf. Die Belastung durch Prime95 mit zwei Threads liefert bei der lokalen Ausführung von MooBench Ergebnisse zwischen 102036 und 126032 und zeigen damit eine Steigerung von ungefähr 100%. Dies ist eindeutig nicht mehr im Rahmen der normalen Schwankungen und zeigt daher den Einfluss des Umfelds. Noch deutlicher wird der Unterschied bei der nebenläufigen Ausführung von Prime95 mit 4 Threads: Hier liegen die Ergebnisse zwischen 219476 und 284973 und sind damit ungefähr 4 mal so hoch, wie die Ergebnisse ohne Belastung. Die Ergebnisse des *RemoteMooBenchExecutors* liegen für die ersten 5 Tests ohne weitere Belastung zwischen 49425 und 57676 und zeigen somit eine Schwankungsbreite von ungefähr 20% des niedrigsten Wertes. Die Ergebnisse der anderen beiden Tests liegen in der Remote-Variante insgesamt zwischen 48787 und 58651 und sind somit nicht signifikant höher, als die Ergebnisse, die ohne Belastung des Systems erzielt wurden.

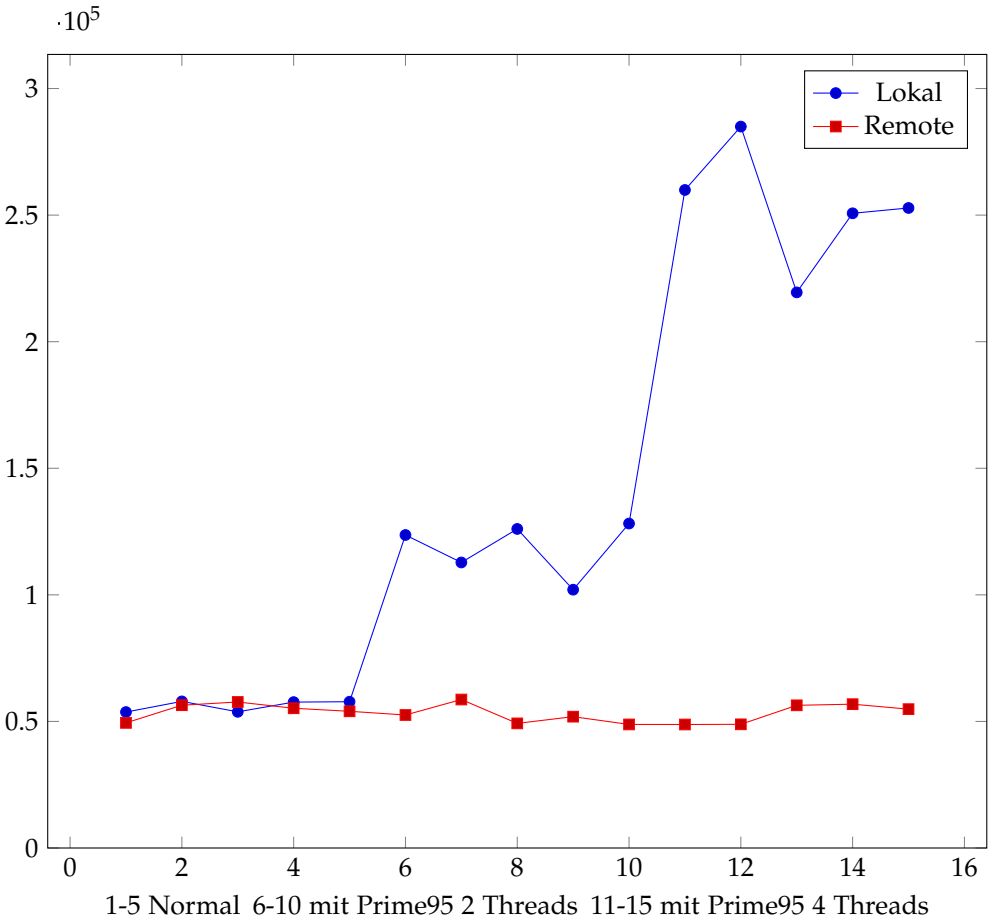

Abbildung 5.1. Ergebnisse der Benchmarks lokal und auf einem Slave

Der Test zeigt, dass die Ergebnisse der lokale Variante stark von der Auslastung des Systems beeinflusst werden, während die ausgelagerte Variante von diesen Einflüssen unberührt ist. Dadurch wird deutlich, dass die Auslagerung der Benchmarks auf ein Slave-System ein Weg ist, die Vergleichbarkeit der Ergebnisse im Bezug auf äußere Einflüsse zu gewährleisten. Die Ursache der verbleibenden Schwankungen in der Größe der Ergebnisse ist unklar und durch die große Komplexität der verwendeten Hard- und Software nicht eindeutig bestimmbar.
Die Implementierung setzt somit einige Maßnahmen um, die die Vergleichbarkeit der Ergebnisse fördern. Die verbleibenden Schwankungen sind vorhersehbar und bleiben in tolerierbaren Grenzen, sodass ich diese Anforderung als erfüllt ansehe. Weitere Methoden zur Sicherstellung der Vergleichbarkeit können das Ziel späterer Forschung werden.

5.3.9. Installierbarkeit

Die Installierbarkeit teste ich, indem das fertige Plugin auf einem Jenkins-Server installiert wird. Die Installation erfolgt über die Jenkins-Oberfläche und lässt sich ohne weiteres Wissen innerhalb weniger Minuten durchführen. Eine Installation über die offiziellen Plugin-Quellen wäre noch simpler.
Ein Test der Installation auf einer nicht über Netbeans betriebenen Jenkins-Installation zeigte, dass ein Neustart von Jenkins notwendig ist, damit die Installation von Plug-ins, die nicht über die offiziellen Plug-in-Quellen geschehen, beendet wird, erst danach ist das Plugin installiert und kann verwendet werden.
Die Einrichtung von Slaves in Jenkins kann je nach System sehr aufwändig sein. Dies ist aber kein Bestandteil dieser Arbeit und wird daher nicht weiter beachtet.

5.3.10. Automatische Ausführung von MooBench

Für die bereits genannten Tests verwende ich in den meisten Fällen die Durchführungskomponente *MooBenchExecutor* oder *RemoteMooBenchExecutor*, welche MooBench ausführen. Dadurch ist bereits gezeigt, dass die Ausführung von MooBench funktioniert. Um die Korrektheit zu zeigen, führe ich den zu Beginn des Kapitels genannten Test mit 100 Methodenaufrufen zuerst durch die Implementierung mit der lokalen Variante aus und verwende danach folgendes Skript, welches die gleichen Einstellung beinhaltet:

```
1  java -javaagent:.\kieker-1.9-SNAPSHOT_aspectj.jar
2  -Dorg.aspectj.weaver.loadtime.configuration=META-INF/kieker.aop.xml
3  -Dkieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter
4  -Dkieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=.\tmp
5  -jar .\OverheadEvaluationMicrobenchmark.jar --output-filename .\test.csv
6  --totalcalls 100 --methodtime 0 --totalthreads 2 --recursiondepth 2
7  java -javaagent:.\kieker-1.9-SNAPSHOT_aspectj.jar
8  -Dorg.aspectj.weaver.loadtime.configuration=META-INF/kieker.aop.xml
9  -Dkieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter
10 -Dkieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=.\tmp
11 -jar .\OverheadEvaluationMicrobenchmark.jar --output-filename .\test.csv
12 --totalcalls 100 --methodtime 0 --totalthreads 2 --recursiondepth 2
```

Der Vergleich der Ergebnisse erfolgt wie auch bei der Evaluierung des Datenspeichers mit LibreOffice Calc [Seimert 2012; LibreOffice]. Der Durchschnitt der über das Skript produzierten Werte liegt bei 86016, der der über das Plug-in produzierten Werte bei 94078. Die Differenz lieft bei 8062 und somit bei etwa 10% des geringeren Wertes. Diese Abweichung entspricht der im Test der Vergleichbarkeit gemessenen und liegt somit im Rahmen der normalen Schwankungen zwischen zwei Ausführungen. Da für den Vergleich deutlich weniger Methodenaufrufen verwendet wurden und kein Warm-Up berücksichtigt wurde, wäre eher eine höhere Differenz zu erwarten gewesen. Der Test zeigt somit, dass die

Ergebnisse den von den MooBench-Skripten produzierten bis auf normale Abweichungen gleichen. Das verdeutlicht, dass die Durchführung von MooBench-Experimenten durch die lokale Durchführungskomponente korrekt implementiert wurde.

5.3.11. Portierbarkeit

In Unterabschnitt 5.3.8 und Unterabschnitt 5.3.6 zeige ich, dass sich Durchführungskomponente austauschen lässt, ohne andere Komponenten zu ändern. Da nur diese Komponente notwendig ist, um die Implementierung für Benchmarks einer anderen Software anzupassen, beläuft sich die Zahl der dafür auszutauschenden Komponenten auf 1. Der weiterverwendbare Anteil ist somit der gesamte Rest, also Datenspeicher-, Analyse-, Wächter-, und Darstellungskomponente. Die Portierbarkeit kann also als sehr gut angesehen werden, da 6 der 7 Klassen weiterhin verwendet werden können.

5.4. Zusammenfassung

Dieses Kapitel zeigt, dass das in Kapitel 3 entwickelte System die in Abschnitt 1.2 gestellten Anforderungen vollkommen erfüllt. Viele der Ziele werden hierbei durch die abstrakte Sichtweise in Kombination mit dem komponentenweisen Aufbau erfüllt. Die in Kapitel 4 vorgestellte Implementierung konnte durch vielfältige Tests ihre Funktionalität zeigen und stellt bis auf Abweichungen bezüglich der Steuerungskomponente eine gelungene Implementierung des Systems dar. Die Evaluierung konnte somit erfolgreich die Erfüllung der Ziele bestätigen.

Kapitel 6

Verwandte Arbeiten

6.1. BEEN

BEEN [Kalibera u. a. 2004; 2006] ist ein Projekt der Arbeitsgruppe Distributed Systems Research Group der Fakultäten Mathematik und Physik der Charles Universität von Prag und dient der automatischen Durchführung von Benchmarks mit verteilten Systemen. Es bietet einen modularen Aufbau, bei dem einzelne Komponenten verteilt ausgeführt werden können, sodass die Software gut skaliert werden kann. Hauptziel von BEEN ist es, Regressionstest durchzuführen, also Veränderungen der Performance zu messen.
BEEN bietet wie Jenkins eine Web-Oberfläche zur Steuerung und unterstützt das Bauen von Software. Konkrete Benchmarks können über Plug-ins integriert werden. Im Gegensatz zu dem in Kapitel 3 vorgestellten System ist BEEN eine eigenständige Software. Das ermöglicht einen Aufbau, der optimal an die Anforderungen angepasst ist. Im Gegenzug stehen Funktionen, die in gängigen Continuous-Integration-Systemen enthalten oder über Plug-ins verfügbar sind, unter Umständen nicht zur Verfügung. Bei der nachträglichen Entscheidung für BEEN muss zudem ein komplettes zusätzliches System eingerichtet werden, wogegen die Installation eines Plug-ins deutlich einfacher ist.

6.2. Performance Plugin

Das Testframework JUnit bietet über *@Timeout*-Notationen die Möglichkeit, Grenzwertüberprüfungen durchzuführen. Die hierdurch entstandenen Werte kann das Performance Plugin für Jenkins [PerformancePlugin] speichern und daraus Graphen erzeugen, die den Vergleich der Messwerte zulassen. Dieses Plug-in bietet den Vorteil, dass ziemlich einfach Laufzeittests in Projekte integriert werden können und die Ergebnisse in der Jenkins-Oberfläche angezeigt werden. Andere Performance-Eigenschaften können jedoch nicht gemessen werden, was in dem von mir vorgestellten System durch die Implementierung eigener Durchführungskomponenten möglich ist.

6.3. KoPeMe

Das Framework KoPeMe (**Ko**ntinuierliche **Pe**rformanz**me**ssung) [Reichelt und Braubach 2014] ermöglicht ebenfalls automatische Performance-Tests durch die Integration von Benchmarks in den Build-Prozess. Hierzu werden vorher in Java definierte Benchmarks in den Build-Prozess mit Maven oder Ant integriert und anschließend die Ergebnisse in Jenkins durch ein eigenes Plug-in angezeigt.
KoPeMe erfüllt somit ebenfalls die Anforderungen dieser Arbeit, die Konfiguration der Benchmarks über Maven oder Ant ist aber in der Konfiguration und bei Änderungen aufwändiger als die Einrichtung über ein einzelnes Plug-in.

Kapitel 7

Fazit und Ausblick

7.1. Fazit

Das entwickelte System hat sich hinsichtlich der gesetzten Ziele als sehr gut herausgestellt und konnte in der Evaluierung zeigen, dass es kontinuierliche Performance-Messungen und -Auswertungen zulässt. Die Implementierung ist eine erste Umsetzung des Systems, die es schafft, den Anforderungen gerecht zu werden und die automatische Durchführung und Auswertung von MooBench-Benchmarks für das Monitoring-Framework Kieker ermöglicht. Die Evaluierung verlief erfolgreich und konnte Stärken und Schwächen von System und Implementierung aufzeigen.
Abschließend lässt sich sagen, dass Mikrobenchmarks und die Automatisierung von Vorgängen durch CI-Systeme ein spannendes Thema sind, auch wenn die Generalisierung von Benchmarks durch die Vielfältigkeit der getesteten Software nur mit starken Einschränkungen möglich ist. Ein allgemeines System für Benchmarks, das auf beliebige Software angewandt werden kann, würde die Automatisierung erheblich vereinfachen, ist aber kaum umsetzbar. Somit bleibt trotz aller Abstraktion für jedes Projekt ein unvermeidbarer Einrichtungsaufwand, der bei kleineren Projekten, bei denen Performance nicht kritisch ist, zu hoch sein kann.

7.2. Ausblick

Die Evaluierung hat gezeigt, dass die Ergebnisse der Benchmarks trotz Auslagerung Schwankungen von etwa 20% unterliegen. Die Ursachen dafür zu finden und Methoden zu entwickeln, um diese Schwankungen weiter einzugrenzen, könnte das Ziel weiterer Forschungen werden. Dazu sind tiefer gehende Analysen der laufenden Prozesse, des Betriebssystems und der Auslastung des Systems und deren Einfluss auf die Benchmarks nötig. Auch der Einfluss durch die Java Virtual Machine auf die Benchmark-Ergebnisse sind ungewiss und könnten in diesem Rahmen studiert werden.
Eine Möglichkeit, die Implementierung zu verbessern, ist die nebenläufige oder verteilte Ausführung von Benchmarks zu ermöglichen. Dies erfordert eine Anpassung der Durchführungskomponenten und würde Skalierbarkeit des Benchmark-Prozesses ermöglichen. Momentan kann nur durch parallele Ausführung von Jobs, welche *RemoteMooBenchExecutor* auf verschiedenen Slaves verwenden, eine parallele Ausführung erreicht werden.

Einschränkungen dafür sind in Abschnitt 3.6 beschriebenen.
Um die Übersichtlichkeit der Oberfläche zu verbessern, kann durch zukünftige Arbeit die Möglichkeit untersucht werden, die Konfiguration in einen Builder zusammenzufassen und die Durchführungskomponente so zu gestalten, dass das Plug-in bei Änderungen der Durchführungskomponente nicht neu gebaut werden muss, sodass das Plug-in ohne Änderung für verschiedene Projekte verwendet werden kann.
Die Erkennung von Regression erfolgt in der bisherigen Implementierung nur auf dem Vergleich zweier Revisionen anhand eines festgelegten Grenzwertes. Dieser Grenzwert sollte aber eine gewisse Verschlechterung der Werte zulassen, da kleinere Schwankungen der Ergebnisse von Benchmarks normal sind. Dadurch kann eine stetige Senkung der Performance unbemerkt bleiben, falls die Verschlechterung zwischen zwei Revisionen immer unterhalb des Grenzwertes bleibt. Daher ist es Wünschenswert, eine geeignetere Anomalieerkennung in eine Wächterkomponente zu integrieren. Hierzu kann untersucht werden, ob die von Frotscher entwickelten Verfahren zur Anomalie-Erkennung [Frotscher 2013] angewandt werden können.

Literaturverzeichnis

[Abiteboul u. a. 2003] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu und T. Milo. Dynamic xml documents with distribution and replication. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, Seiten 527–538. (Siehe Seite 11)

[Basili u. a. 1994] V. R. Basili, G. Caldiera und H. D. Rombach. The goal question metric approach. In: *Encyclopedia of software engineering*. AJohn J. Marciniak, Feb. 1994, Seiten 528–532. (Siehe Seiten 1 und 47)

[Boudreau u. a. 2002] T. Boudreau, J. Glick, S. Greene, V. Spurlin und J. J. Woehr. NetBeans: the definitive guide. O'Reilly Media, Inc., 2002. (Siehe Seite 27)

[Commons Math] Commons Math Apache Commons Math library. Apachei. URL: http://www.http://commons.apache.org/proper/commons-math/. (Siehe Seite 11)

[Duvall u. a. 2007] P. M. Duvall, S. Matyas und A. Glover. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007. (Siehe Seite 8)

[Frotscher 2013] T. Frotscher. Architecture-Based Multivariate Anomaly Detection for Software Systems. Masterarbeit. Kiel University, 2013. URL: http://eprints.uni-kiel.de/21346/. (Siehe Seite 68)

[Gamma u. a. 1994] E. Gamma, R. Helm, R. Johnson und J. Vlissides. Design patterns: elements of reusable object-oriented software. Pearson Education, 1994. (Siehe Seite 30)

[Gao u. a. 2000] J. Gao, E. Y. Zhu, S. Shim und L. Chang. Monitoring software components and component-based software. In: *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*. IEEE. 2000, Seiten 403–412. (Siehe Seite 7)

[Ghezzi u. a. 2002] C. Ghezzi, M. Jazayeri und D. Mandrioli. Fundamentals of software engineering. Prentice Hall PTR, 2002. (Siehe Seite 1)

[ISO/IEC 2010] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technischer Bericht. 2010. (Siehe Seiten 1, 2)

[Jain 2008] R. Jain. The art of computer systems performance analysis. John Wiley & Sons, 2008. (Siehe Seite 7)

[Jenkins] Jenkins continuous integration system. Kohsuke Kawaguch. URL: http://www.jenkins-ci.org/. (Siehe Seite 10)

[Kalibera u. a. 2004] T. Kalibera, L. Bulej und P. Tuma. Generic environment for full automation of benchmarking. In: *SOQUA 2004*. unknown, Mai 2004, Seiten 1–10. (Siehe Seite 65)

[Kalibera u. a. 2006] T. Kalibera, L. J., M. D., R. B., T. M., T. A., T. P. und U. J. Automated benchmarking and analysis tool. In: *VALUETOOLS 2006 (preliminary version)*. unknown, Mai 2006, Seiten 1–5. (Siehe Seite 65)

[LibreOffice] LibreOffice Open Source Office-Suite. The Document Foundation. URL: http://de.libreoffice.org/. (Siehe Seiten 57 und 62)

[MooBench Scripts] MooBench Scripts Scripts for executing MooBench. Jan Waller. URL: https://build.se.informatik.uni-kiel.de/gitlab/kieker/moobench.git. (Siehe Seiten 10 und 33)

[OpenCSV] OpenCSV CSV library for java. Sean Sullivani. URL: http://www.opencsv.sourceforge.net/. (Siehe Seite 11)

[PerformancePlugin] PerformancePlugin jenkins plugin for capturing performance-tests performed with JUnit. Manuel Carrasco Monino. URL: https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin. (Siehe Seite 65)

[PlotPlugin] PlotPlugin jenkins plugin for graph-plotting. Nigel Daley. URL: https://wiki.jenkins-ci.org/display/JENKINS/Plot+Plugin. (Siehe Seite 10)

[Prime95] Prime95 distributed computing project dedicated to finding new Mersenne prime numbers. GIMPS. URL: http://www.mersenne.org/freesoft/. (Siehe Seite 60)

[Reichelt und Braubach 2014] D. G. Reichelt und L. Braubach. Sicherstellung von performanzeigenschaften durch kontinuierliche performanztests mit dem kopeme framework. In: *Software Engineering Ideas Track on Software Engineering Conference (SE 2014)*. Universität Hamburg, 2014, Seiten 1–6. URL: http://vsis-www.informatik.uni-hamburg.de/vsis/publications/lookpub/505. (Siehe Seite 66)

[Rohr u. a. 2008] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke und W. Hasselbring. Kieker: continuous monitoring and on demand visualization of java software behavior. In: *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*. Herausgegeben von C. Pahl. Anaheim, CA, USA: ACTA Press, 2008, Seiten 80–85. URL: http://eprints.uni-kiel.de/14496/. (Siehe Seite 9)

[Seimert 2012] W. Seimert. LibreOffice 3.5; BP. Hüthig Jehle Rehm, 2012. (Siehe Seiten 57 und 62)

[Van Hoorn u. a. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey und D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Forschungsbericht. Kiel University, 2009. URL: http://eprints.uni-kiel.de/14459/. (Siehe Seite 9)

[Van Hoorn u. a. 2012a] A. van Hoorn, J. Waller und W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, Seiten 247–248. (Siehe Seite 1)

[Van Hoorn u. a. 2012b] A. van Hoorn, J. Waller und W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, 2012, Seiten 247–248. URL: http://eprints.uni-kiel.de/14418/. (Siehe Seite 9)

[Waller und Hasselbring 2013] J. Waller und W. Hasselbring. Data for: A Benchmark Engineering Methodology to Measure the Overhead of Application-Level Monitoring. 2013. URL: http://eprints.uni-kiel.de/22648/. (Siehe Seiten 10 und 13)