

Thema der Bachelorarbeit
für
Herrn Timo Hamann

Automatisierte Auswertung von Programmier-Praktika

Aufgabenstellung

Der Einsatz von Continuous Integration Lösungen gehört mittlerweile zum bewährten Vorgehen in der Softwareentwicklung. Diese Arbeit soll den Einsatz von Continuous Integration Lösungen im Rahmen der Hochschullehre (Programmierausbildung) untersuchen.

Zur Unterstützung von Programmierlehrveranstaltung - insbesondere von Programmierpraktika - soll hierzu eine Lösung zur automatisierten Auswertung von wöchentlichen Abgabearbeiten auf Basis von existierenden Continuous Integration Lösungen geschaffen werden.

Auf diese Weise sollen Praktikumsbetreuer/innen in Zeiten stetig größer werdender Programmierkurse zeitlich entlastet werden und Studierenden eine ergänzende Möglichkeit des Feedback-gesteuerten Selbststudiums gegeben werden.

Erklärung zur Bachelorarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

(Datum)

Unterschrift

Zusammenfassung der Arbeit

Abstract of Thesis

Fachbereich: **Elektrotechnik und Informatik**

Department:

Studiengang: **Informatik / Softwaretechnik**

University course:

Thema: **Automatisierte Auswertung von Programmier-Praktika**

Subject:

Zusammenfassung:

Die vorliegende Arbeit beschäftigt sich mit dem Einsatz von Continuous Integration Lösungen im Rahmen der Hochschul-Programmierausbildung. Hierbei wurde Jenkins als die geeignetste Lösung für die ermittelten Anforderungen identifiziert. Das System wurde als Docker-Container konfiguriert und für unterschiedliche automatisierte Auswertungen von Quelltext vorbereitet. Anhand einiger konkreter Aufgaben wurde die Funktionsfähigkeit gezeigt. Dabei helfen vorbereitete Skripte bei der effizienten Einrichtung des Systems für viele Benutzer. Für die Anwender wurde eine Dokumentation zur Benutzung des Systems erstellt.

Abstract:

The following work analyses the use of Continuous Integration systems for programming training in college. Jenkins was identified as the most suitable system for the requirements. The System was configured as a Docker-Container and prepared to execute different automated evaluations of source code. The functionality of the system is shown by some selected exercises. Created scripts support an easy setup of the system for many users. A created documentation shows the usage of the system to the different users.

Verfasser: **Timo Hamann**

Author:

Betreuender Professor/in: **Prof. Dr. Nane Kratzke**

Attending Professor:

WS / SS : **SS 2016**

Inhaltsverzeichnis

Aufgabenstellung	2
Erklärung zur Bachelorarbeit	3
Zusammenfassung der Arbeit	4
Inhaltsverzeichnis	5
1 Einleitung	8
1.1 Motivation	8
1.2 Ziele der Arbeit.....	8
1.3 Gliederung des weiteren Dokumentes	9
2 Anforderungen an das System	10
3 Continuous Integration Systeme	11
3.1 Gewöhnliche Nutzungsweise	11
3.2 Besondere Anforderungen	12
3.3 Kandidatenfindung	12
3.4 Analyse der Kandidaten	14
3.4.1 Jenkins.....	14
3.4.2 Go	15
3.4.3 Travis CI	15
3.4.4 Buildbot.....	15
3.5 Vergleich der Kandidaten und Wahl eines Kandidaten.....	16
4 Realisierung des automatisierten Bewertungssystems	17
4.1 System-Übersicht.....	17
4.2 Benutzer- und Jobverwaltung.....	18

4.3	Auswertungen.....	18
4.3.1	Korrektheit	19
4.3.2	Code-Coverage	20
4.3.3	Programmierkonventionen und Komplexität	20
4.4	Leistungsübersicht und Feedback.....	22
4.5	Ablauf einer Auswertung	23
4.6	Docker-Container	24
5	Automatisierung von Aufgaben	25
5.1	Automatisierung konkreter Aufgaben	25
5.1.1	Schaltjahre bestimmen (Aufgabe 5.6)	25
5.1.2	Multiplikationstabelle (Aufgabe 7.1).....	27
5.1.3	Ausgaberroutine für Stack, List und Map (Aufgabe 10.1).....	28
5.1.4	Bestimmen von Baumeigenschaften (Aufgabe 13.2).....	28
5.1.5	Zusammenfassung	29
5.2	Analyse komplexerer Aufgaben	30
5.2.1	Telefonbuch (17.4)	30
5.2.2	Generische Warteschlange (20.1).....	31
5.2.3	GUI-Aufgaben (22)	32
5.2.4	Zusammenfassung	33
6	Anwender-Dokumentationen	34
6.1	Dokumentation für Administratoren.....	34
6.2	Dokumentation für Lehrende	38
6.3	Dokumentation für Studierende	47
7	Test und Nachweisführung.....	55

8	Fazit und Ausblick	58
9	Danksagungen.....	59
	Anhang A – Aufgabenstellungen und Quelltexte	60
	A.1 Schaltjahre bestimmen (Aufgabe 5.6).....	60
	A.1.1 Bisherige Aufgabenstellung.....	60
	A.1.2 Erweiterung zu der bisherigen Aufgabenstellung	60
	A.1.3 Quelltext der Tests	61
	A.2 Multiplikationstabelle (Aufgabe 7.1)	64
	A.2.1 Bisherige Aufgabenstellung.....	64
	A.2.2 Erweiterung zu der bisherigen Aufgabenstellung	64
	A.2.3 Quelltext der Tests	65
	A.3 Ausgaberroutine für Stack, List und Map (Aufgabe 10.1)	67
	A.3.1 Bisherige Aufgabenstellung.....	67
	A.3.2 Erweiterung zu der bisherigen Aufgabenstellung	67
	A.3.3 Quelltext der Tests	68
	A.4 Bestimmen von Baumeigenschaften (Aufgabe 13.2).....	70
	A.4.1 Bisherige Aufgabenstellung.....	70
	A.4.2 Erweiterung zu der bisherigen Aufgabenstellung	70
	A.4.3 Quelltext der Tests	71
	Abbildungsverzeichnis	73
	Tabellenverzeichnis	74
	Literaturverzeichnis.....	74

1 Einleitung

1.1 Motivation

Die Bewertung studentischer Abgaben in Programmierpraktika erfolgt durch Praktikumsbetreuer/innen. Diese führen gewisse Arbeitsschritte, wie zum Beispiel die Prüfung auf Korrektheit, für jede einzelne Lösung durch. In Zeiten stetig größer werdender Programmierkurse bedeutet dies eine Steigerung des ohnehin schon hohen Arbeitsaufwandes.

Durch eine automatisierte Auswertung der Lösungen können die sich wiederholenden Arbeitsschritte reduziert und dadurch die Lehrenden entlastet werden. Zudem wird den Studierenden durch Einsicht des Ergebnisses der automatisierten Auswertung die Möglichkeit des Feedback-gesteuerten Selbststudiums gegeben. [Kra16]

1.2 Ziele der Arbeit

Diese Arbeit soll den Einsatz von Continuous Integration (kurz CI) Lösungen im Rahmen der Programmierausbildung untersuchen. Hierfür soll ein CI-Server eingerichtet werden, welcher für Programmierabgaben bestimmte Auswertungen automatisiert durchführen kann und die Ergebnisse übersichtlich für Studierende und Lehrende darstellt. Für vier Aufgaben aus dem Aufgabenkatalog der Lehrveranstaltung „Programmieren I“ soll diese automatisierte Auswertung umgesetzt werden. Zudem soll die Machbarkeit der automatisierten Auswertung für offenere Aufgabenstellungen mit unterschiedlichsten Lösungsmöglichkeiten analysiert werden. Hierfür sollen drei Aufgaben aus dem Aufgabenkatalog der Lehrveranstaltung „Programmieren II“ betrachtet werden. [Kra16]

Die Arbeit beschränkt sich auf die Auswertung von Quelltexten in der Programmiersprache Java.

1.3 Gliederung des weiteren Dokumentes

In Kapitel 2 werden die Anforderungen an das fertige System definiert. Kapitel 3 beschäftigt sich mit den Grundlagen von CI-Lösungen sowie der Wahl eines konkreten CI-Servers als Ausgangsbasis. Auf die allgemeine Realisierung des automatisierten Bewertungssystems wird in Kapitel 4 eingegangen. In Kapitel 5 wird die Automatisierung mehrerer konkreter Aufgaben betrachtet. Kapitel 6 beinhaltet die Anwender-Dokumentation. Anschließend erfolgen in Kapitel 7 der Test und die Nachweisführung. Abschließend gibt es eine Zusammenfassung und ein Fazit sowie ein Ausblick mit möglichen Erweiterungen.

Das Kapitel 6 ist speziell auf die unterschiedlichen Anwenderrollen zugeschnitten. Dabei werden zum Teil Inhalte aus den Kapiteln 4 und 5 wiederholt. Diese beiden Kapitel wiederum verweisen hinsichtlich Erklärungen zur Verwendung auf die Anwender-Dokumentation.

2 Anforderungen an das System

Aus der Aufgabenstellung [Kra16] wurden Anforderungen an das System identifiziert und ihre Komplexität geschätzt. Anschließend wurde gemeinsam mit Herrn Prof. Dr. Kratzke eine Priorisierung vorgenommen.

#	Anforderung	Komplexität	Priorität
1	Als Student/Lehrender möchte ich Lösungen zu Programmieraufgaben hinsichtlich Korrektheit, Code Coverage, Komplexität und Programmierkonventionen auswerten.	hoch	hoch
2	Als Student möchte ich die Auswertung direkt aus der Entwicklungsplattform Eclipse heraus anstoßen.	mittel	niedrig
3	Als Student möchte ich Feedback in Relation zur Musterlösung erhalten.	niedrig	hoch
4	Als Student möchte ich Feedback in Relation zu anderen Studierenden erhalten.	hoch	niedrig
5	Als Student möchte ich eine Leistungsübersicht über alle meine Abgaben erhalten. Als Lehrender möchte ich eine Leistungsübersicht über alle Abgaben eines Studierenden erhalten.	hoch	hoch
6	Als Lehrender möchte ich eine Leistungsübersicht für eine Abgabe über alle Studierenden erhalten.	hoch	hoch
7	Als Lehrender möchte ich eine Leistungsübersicht über alle Abgaben aller Studierenden erhalten. (Semesterüberblick)	hoch	mittel
8	Als Lehrender möchte ich Aufgaben anhand eines Templates / Kochrezeptes für die automatisierte Auswertung vorbereiten.	mittel	hoch
9	Als Lehrender möchte ich das System komfortabel für eine Vielzahl von Studierenden einrichten.	mittel	hoch
10	Als Administrator möchte ich die Continuous Integration Lösung komfortabel als Docker-Container aufsetzen können.	mittel	hoch
11	Als Administrator möchte ich die Lösung anhand einer Dokumentation betreiben und konfigurieren.	niedrig	hoch
12	Als Student/Lehrender möchte ich durch geeignete Dokumentation bei der Benutzung der Lösung unterstützt werden.	niedrig	hoch

Tabelle 2-1: Anforderungen

3 Continuous Integration Systeme

In diesem Kapitel wird zunächst die gewöhnliche Nutzungsweise eines Continuous Integration Systems erläutert und die besonderen Anforderungen des konkreten Anwendungsfalles beschrieben. Anschließend werden verschiedene Systeme betrachtet und ein Kandidat als Basis für das zu konfigurierende System gewählt.

3.1 Gewöhnliche Nutzungsweise

Der renommierte Softwarearchitekt Martin Fowler beschrieb Continuous Integration vor einigen Jahren folgendermaßen:

„Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.“ [Fow06]

Dies ist auch heute noch der Kerngedanke hinter Continuous Integration. Daher werden CI-Systeme üblicherweise von Entwicklerteams verwendet, welche eine zusammenhängende Software entwickeln und auf einer gemeinsamen Quelltextbasis arbeiten. Hierfür werden Versionskontrollsysteme (VCS) wie zum Beispiel Subversion oder Git eingesetzt. Der CI-Server reagiert auf Änderungen im VCS und startet automatisiert den Build-Prozess. In der Regel beinhaltet das Build neben dem Kompilieren auch das Durchführen von Tests und Auswertungen zur Qualität des Quelltextes. Die Ergebnisse werden auf dem Server veröffentlicht und die Entwickler benachrichtigt. Die Ergebnisse werden zudem historisiert und durch eine Betrachtung über die Zeit werden Trends aufgezeigt. [Hei12]

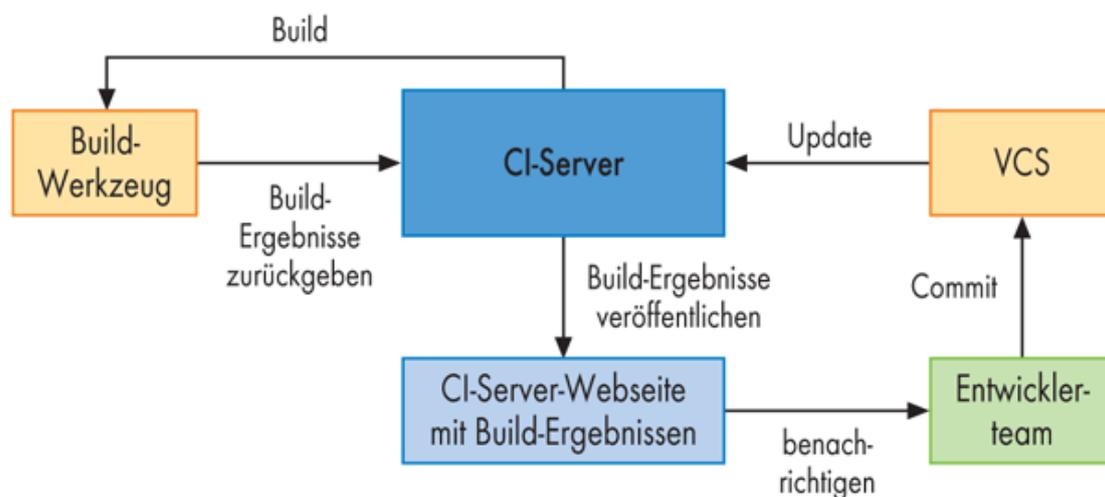


Abbildung 3-1: Bestandteile eines CI-Systems [Hei12]

3.2 Besondere Anforderungen

Das System soll von Programmieranfängern genutzt werden, wobei der Fokus der Studierenden weiterhin auf dem Lösen der Aufgaben liegen soll. Deshalb sollen möglichst keine weiteren Kenntnisse vorausgesetzt werden, wie zum Beispiel der Umgang mit einem VCS. Der Build-Prozess sollte nach Möglichkeit also nicht durch Änderungen im VCS ausgelöst werden, sondern die Studierenden starten den Prozess manuell und übergeben dabei ihre Lösung als Datei. Die Musterlösungen und Tests der Lehrenden hingegen werden in einem VCS verwaltet, aktuell über Bitbucket. Die Lösungen der Studierenden müssen im Prozess mit den Tests der Lehrenden zusammengeführt werden, wobei die Klassen der Lehrenden höchstens teilweise von den Studierenden eingesehen werden dürfen.

Zudem wird nicht auf einer gemeinsamen Quelltextbasis gearbeitet, sondern jeder Studierende fertigt eigene Lösungen an und ihm darf keine Einsicht in andere Lösungen gewährt werden. Lehrende hingegen müssen Einsicht in alle Lösungen haben.

Trends über die Zeit zu ermitteln ist ebenfalls nicht sinnvoll. Vielmehr interessiert der Vergleich der Auswertungsergebnisse unterschiedlicher Lösungen oder Abgabepflichten. Hierbei sind mehrere prüfungsübergreifende Auswertungen mit unterschiedlicher Gruppierung gewünscht (siehe Kapitel 2, Anforderungen 5, 6 und 7).

3.3 Kandidatenfindung

Aufgrund der von der gewöhnlichen Nutzungsweise abweichenden besonderen Anforderungen ist die Flexibilität und Erweiterbarkeit ein wichtiger Punkt bei der Wahl der Lösung. Zudem ist die Unterstützung von Java als Programmiersprache Pflicht. Ein weiteres Ausschlusskriterium ist die Art der Lizenz sowie die dadurch entstehenden Kosten der Software. Die Lösung darf keinerlei Lizenzkosten verursachen, sondern es muss sich um freie Software handeln. Grund hierfür ist, dass sich das Lizenzmodell ändern kann, die Kosten steigen oder beantragte Gelder gestrichen werden können.

Um einen generellen Überblick über die große Anzahl unterschiedlicher CI-Server zu erhalten, werden zunächst weit verbreitete Lösungen gesucht. Eine Umfrage von InfoQ.com [InQ14] hinsichtlich Bedeutung und Reife unterschiedlicher CI-Server ergab folgendes Ergebnis:

Option	Stimmen
Jenkins	569
TeamCity	192
Bamboo	164
Hudson	130
Cruise Control	129
Go	117
Team Foundation Server	108
TravisCI	98
Anthill	43
Buildbot	37
Continuum	32
QuickBuild	18
FinalBuilder Server/Continua CI	17
Mojo	11
Parabuild	11
Gump	11
Zed	11
easyCIS	10
Sin	9
Pulse	9

Tabelle 3-1: CI-Server Ranking (Umfrage) [InQ14]

Auf opensource.com wurde gezielt auf freie Softwarelösungen eingegangen. Jenkins, Buildbot, Travis CI, Strider, Go und Integrity wurden dabei als die besten Lösungen genannt. [Tiw15]

Die in beiden Quellen genannten Lösungen Jenkins, Go, Travis CI und Buildbot werden allesamt aktiv weiterentwickelt und können als eigener Server betrieben werden. Daher werden diese als mögliche Kandidaten genauer hinsichtlich der konkreten Anforderungen analysiert und verglichen.

Die beiden gut bewerteten Lösungen TeamCity und Bamboo sind kommerzielle Produkte und werden daher ausgeschlossen.

Jenkins entstand 2010 als Resultat auf interne Veränderungen bei Hudson, wobei die Softwarebasis übernommen wurde. Der Initiator Kohsuke Kawaguchi und das Entwicklerteam wechselten zu Jenkins und seitdem ist Jenkins das Projekt mit deutlich höherer Aktivität und Akzeptanz in der Community. [Kaw15] Daher wird nur Jenkins genauer betrachtet und Hudson nicht.

Cruise Control wird ebenfalls nicht genauer betrachtet, weil die aktuellste Version 2.8.4 aus dem September 2010 stammt und somit Kompatibilität zu neueren Technologien nicht sichergestellt ist.

3.4 Analyse der Kandidaten

Die Kandidaten werden hinsichtlich folgender Kriterien analysiert und verglichen:

- Flexibilität und Erweiterbarkeit
- VCS-Unterstützung und Umgang mit Code außerhalb einer Versionsverwaltung
- Benutzerverwaltung und Rechtemanagement
- Auswertung von Java-Code
 - Ausführung von Unit-Tests
 - Ermittlung von Code-Coverage (Quelltext-Abdeckung)
 - Ermittlung von Komplexität
 - Prüfung von Programmierkonventionen
- übergreifende Auswertungen
- Vorhandensein eines Eclipse-Plugins

Die Systeme bieten zum Teil einen Plugin- oder Addon-Mechanismus. Dieser Mechanismus verspricht die Möglichkeit, über leicht zu installierende Erweiterungen zusätzliche Funktionalität zu erhalten.

Die folgenden Daten wurden während der Bearbeitungszeit ermittelt. Sie bilden den Stand zum jeweiligen Zeitpunkt ab und wurden zuletzt am 04.06.2016 bearbeitet.

3.4.1 Jenkins

Die aktuellste Jenkins LTS-Version (Long-Term Support) 1.651.2 wurde am 11.05.2016 veröffentlicht. Jenkins ist mittels Plugin-Mechanismus erweiterbar. Dank der sehr großen Community stehen bereits über 1300 Plugins verschiedenster Kategorien frei zur Verfügung. Diese beinhalten unterschiedliche VCS-Unterstützung (inklusive Bitbucket) sowie die geforderten Code-Auswertungen, dessen Ergebnisse dadurch gezielt und spezifisch dargestellt werden können. Zusätzlich können beliebige Dateien als Artefakt einer Prüfung gespeichert werden. Jobübergreifende Auswertungen sind ebenfalls denkbar, zum Beispiel per „build-metrics-plugin“. Dank Steuerbarkeit per Kommandozeile sowie eingebauter Skript-Konsole ist eine Vielzahl von Benutzerkonten komfortabel erzeugbar. Mittels Matrix-Zugriffssteuerung können die jeweiligen Rechte sehr gezielt vergeben werden. Per Eclipse-Plugin „Hudson/Jenkins Mylyn Builds Connector“ können Jobs direkt aus Eclipse heraus gestartet und Resultate eingesehen werden. [ecm16]

Analyse von Code außerhalb einer Versionsverwaltung ist nicht vorgesehen. Jobs können allerdings Datei-Parameter haben, über welche die studentischen Lösungen übergeben werden könnten. Sollten sich diese Dateien zur Analyse in einer Versionsverwaltung befinden müssen, könnte der Server auch dies automatisiert erledigen. [jen16]

Die Anzahl der Jenkins-Installationen hat in den vergangenen Jahren stark zugenommen. [Neu14] Zusätzlich gibt es einen Release Candidate für die Version 2.0, welche unter anderem auch die gelegentlich kritisierte Usability verbessern soll. Es wird ein einfaches Upgrade von vorherigen Versionen versprochen.

3.4.2 Go

Die aktuellste Go Version 16.5.0 wurde am 24.05.2016 veröffentlicht. Go ist ebenfalls mittels Plugin-Mechanismus erweiterbar und es stehen rund 50 Plugins frei zur Verfügung. Es werden mehrere VCS-Systeme unterstützt (ohne Bitbucket). Zur Anzeige von Auswertungsergebnissen gibt es einen sehr allgemeinen Artefakt-Mechanismus, welcher unterschiedliche Dateien als Ergebnis anzeigen kann. Dadurch ist die Anzeige sehr flexibel, allerdings sollte es sich nicht um XML-Dateien handeln, sondern die Artefakte müssen in einem gut darstellbaren Format wie zum Beispiel HTML vorliegen.

Die Zugriffssteuerung basiert auf Benutzern und Rollen. Im Ursprungszustand existiert jedoch keine Möglichkeit, komfortabel eine Vielzahl von Benutzern einzurichten. Jobübergreifende Auswertungen sowie Analyse von Code außerhalb einer Versionsverwaltung sind nicht vorgesehen und Dateien sind als Parameter für Jobs nicht möglich. Ein Eclipse-Plugin ist nicht verfügbar. [god16]

3.4.3 Travis CI

Travis CI veröffentlicht keine speziellen Versionen, sondern besteht aus mehreren Teilprojekten, welche unabhängig voneinander neue Versionen veröffentlichen. Es gibt eine gehostete Variante, welche allerdings nur für quelloffene Projekte kostenlos ist. Travis CI ist stark mit GitHub verbunden, beispielsweise basieren die Zugriffsrechte auf den Rechten bei GitHub. Analyse von Code außerhalb einer Versionsverwaltung ist dementsprechend nicht vorgesehen. Die Ausführung von Tests ist möglich, allerdings bietet Travis CI keinen Plugin-Mechanismus, welcher die anderen geforderten Auswertungen liefert. Beispielsweise ist die Ermittlung und Darstellung von Code-Coverage schon deutlich komplizierter. [Vor15] Jobübergreifende Auswertungen sind nicht vorgesehen und es existiert kein Eclipse-Plugin. [tra16]

3.4.4 Buildbot

Die aktuellste Buildbot Version 0.9.0b9 wurde am 10.05.2016 veröffentlicht. Buildbot bezeichnet sich selbst als „job scheduling system“ sowie „framework with batteries included“ und zielt ab auf komplexe Anwendungsfälle wie zum Beispiel aus mehreren Sprachen bestehende Anwendungen. Buildbot ist damit sehr flexibel und kann sicherlich alle Anforderungen erfüllen, allerdings erfordert dies extrem viel Aufwand im Vergleich zu anderen Lösungen. [bui16]

3.5 Vergleich der Kandidaten und Wahl eines Kandidaten

Während der detaillierteren Betrachtung wurde schnell klar, dass Buildbot keine sinnvolle Alternative für den konkreten Anwendungsfall ist.

Mittels tabellarischer Darstellung erfolgt ein übersichtlicher Vergleich der anderen drei Kandidaten:

Kriterium	Jenkins	Go	Travis CI
Flexibilität und Erweiterbarkeit	sehr gut (über 1300 Plugins)	gut (rund 50 Plugins)	gering
VCS-Unterstützung	mehrere (inklusive Bitbucket)	mehrere (ohne Bitbucket)	nur GitHub
Code außerhalb einer Versionsverwaltung	nicht vorgesehen, Datei-Parameter möglich	nicht vorgesehen	nicht vorgesehen
Benutzerverwaltung, Rechtemanagement	sehr gut: gezielt, per Skript durchführbar	gut: gezielt, aber mühsam	via GitHub
Code-Auswertungen	alle, Plugin-Unterstützung	alle	Korrektheit leicht, andere komplex
übergreifende Auswertungen	per Plugin	nein	nein
Eclipse-Plugin	ja	nein	nein

Tabelle 3-2: Vergleich der Kandidaten

Bei jedem der Kriterien schneidet Jenkins am besten ab. Daher wird dieser als Basis für das zu konfigurierende System gewählt.

4 Realisierung des automatisierten Bewertungssystems

Dieses Kapitel beschreibt die Konfiguration des Jenkins-Servers. Hierbei wird zunächst ein Überblick über das System gegeben. Danach wird detailliert auf die Benutzerverwaltung, die automatischen Auswertungen, das Feedback für die Studierende, die Leistungsübersicht sowie den Ablauf einer Auswertung eingegangen. Abschließend wird beschrieben, wie das System als Docker-Container konfiguriert wird. Der Bezug auf konkrete Aufgaben folgt in Kapitel 5.

4.1 System-Übersicht

Die zentrale Komponente des automatisierten Bewertungssystems ist der Jenkins-Server. Dieser interagiert mit mehreren anderen Komponenten.

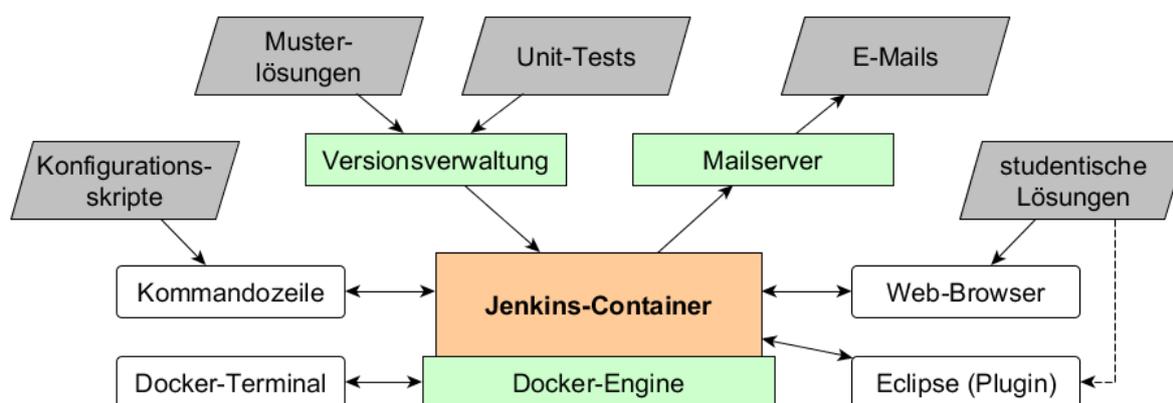


Abbildung 4-1: Komponenten-Übersicht

Der Jenkins-Server wird als Docker-Container betrieben. Docker ist eine Plattform um Anwendungen durch Virtualisierung in Containern zu isolieren. Dies bietet unter anderem die Vorteile, dass die Bereitstellung einer Anwendung vereinfacht wird und ihr Verhalten unabhängig von der Systemumgebung immer gleich ist. [doc16] Die Interaktion mit der Docker-Engine erfolgt per Docker-Terminal.

Der Jenkins-Server bietet eine Web-Oberfläche, welche mittels Browser nutzbar ist. Ein Teil der Interaktion kann ebenfalls per Eclipse-Plugin erfolgen. Es ist eine Benutzerverwaltung inklusive Rechtemanagement eingebaut. Prüfungen werden als sogenannte Jobs definiert und konkrete Prüfläufe werden Builds genannt. Während des Build-Prozesses werden Ressourcen aus einer Versionsverwaltung geladen. Durch die Konfiguration eines Mail-Servers können Benachrichtigungen per E-Mail erfolgen. Um all dies komfortabel einzurichten sind parametrisierte Konfigurationsskripte vorbereitet, welche per Kommandozeile ausgeführt werden.

4.2 Benutzer- und Jobverwaltung

Benutzerkonten können Berechtigungen sowohl Job-spezifisch als auch übergreifend für alle Jobs geltend erteilt werden. Die relevantesten Berechtigungen sind im konkreten Fall „Job read“, welche die Einsichtnahme in einen Job und dessen Build-Ergebnisse steuert, sowie „Job build“, welche das Starten eines Builds erlaubt. Build-spezifische Leserechte können nicht vergeben werden. Daraus resultiert, dass jeder Studierende pro Aufgabe einen separaten Job zugewiesen bekommen muss, damit er nicht die Ergebnisse anderer einsehen oder für diese eine Lösung abgeben kann. Den Lehrenden wird per übergreifendem Job-Leserecht die Einsicht in alle Lösungen ermöglicht.

Zusätzlich zu den Jobs für die studentischen Abgaben gibt es Jobs für die Musterlösungen. Die Buildergebnisse dieser Jobs darf jeder Benutzer einsehen, damit die Studierenden die Auswertung ihre Lösung mit der Musterlösung vergleichen können.

Diese Art der Einrichtung erfordert eine hohe Anzahl an Benutzerkonten und Jobs. Da eine manuelle Durchführung unzumutbar ist, wird dies per Skript ermöglicht. Per Skript erzeugten Benutzerkonten wird ein Passwort generiert und per E-Mail geschickt. Die Durchführung wird in der Dokumentation für die Lehrenden (Kapitel 6.2) beschrieben.

Die Übermittlung der studentischen Lösungsdateien erfolgt per Job-Parameter. Um mögliche Manipulationsversuche der Studierenden zu verhindern erfolgt eine manuelle Verarbeitung der Parameter. Hierbei werden die Musterlösungen gezielt durch die vom Studierenden erwarteten Dateien ersetzt. Wie die Jobs dafür konfiguriert werden ist in der Dokumentation für die Lehrenden (Kapitel 6.2) beschrieben. Die Abgabe wird in der Dokumentation für die Studierenden (Kapitel 6.3) erläutert.

4.3 Auswertungen

Die zu tätigen Auswertungen erfolgen per Ant-Build-Skript. Ant ist ein weit verbreitetes Tool zur Vereinfachung von Erstellungsprozessen. Zur Auswertung verwendet das Skript weitere Bibliotheken und die Ergebnisse werden in bestimmten Dateien und Formaten geschrieben. Mithilfe unterschiedlicher Jenkins-Plugins werden diese Ergebnisse schließlich visuell aufbereitet und den Benutzern bereitgestellt. Auf die konkreten Auswertungen hinsichtlich Korrektheit, Code Coverage, Komplexität und Programmierkonventionen wird in den folgenden Unterkapiteln detailliert eingegangen.

Die Auswertungen erfolgen für alle Aufgaben gleich und die Build-Skripte haben in der Regel keine aufgabenspezifischen Inhalte. Es müssen lediglich geringe Anpassungen vorgenommen werden, wenn das Projekt einer Aufgabe eine Abhängigkeit zu einem anderen Projekt hat, siehe „Aufgabe vorbereiten“ der Dokumentation für Lehrende (Kapitel 6.2).

4.3.1 Korrektheit

JUnit ist der Standard zum Testen von Java-Programmen und wird auch im Verlauf der Programmierausbildung der Studierenden behandelt. Mittels JUnit-Testfällen wird eine Aussage über die Korrektheit von Lösungen gemacht. Es ist nicht allgemein entscheidbar, ob ein Programm korrekt funktioniert, sondern für jede Aufgabe müssen spezifische Testfälle geschrieben werden.

Die Testfälle werden durch die Lehrenden geschrieben. Problematisch hierbei ist, dass die geschriebenen Tests nicht nur für die Musterlösung sondern auch für jede studentische Lösung lauffähig sein müssen. Dieses Ziel wird dadurch erreicht, dass genaue Benennungsvorschriften sowie Parameter- und Rückgabetypen in den Aufgabenstellungen vorgegeben werden.

Alternativ könnten Strukturen per Reflection-API untersucht werden und jeweils die Methode mit den erwarteten Parameter- und Rückgabetypen verwendet werden. Hierfür müssten weiterhin Parameter- und Rückgabetypen vorgegeben werden. Zudem ist dieser Ansatz softwaretechnisch unsauber, nicht robust und pro Klasse müssten die Methoden einzigartige Parameter- und Rückgabetypen besitzen, was bereits in den wenigen genauer betrachteten Aufgaben nicht der Fall ist. Die zuvor genannten Vorgaben hinsichtlich Methodennamen sowie Parameter- und Rückgabetypen könnten elegant per Interface vorgegeben werden. Eine Objekt-Instanziierung könnte per Reflection-API erfolgen, sodass die einzige notwendige Vorschrift ein parameterloser Konstruktor wäre. Interfaces werden allerdings erst im Laufe der Programmierausbildung behandelt, weshalb dieser Ansatz für die Aufgaben des ersten Semesters nicht verfolgt wird.

Im Ant-Build-Skript werden die Bibliothek „junit-4.12.jar“ sowie dessen Abhängigkeit „hamcrest-core-1.3.jar“ von „maven.org“ heruntergeladen. Mithilfe der Elemente „junit“ und „batchtest“ werden alle Tests ausgeführt, dessen Klassenname die Zeichenkette „Test“ beinhaltet und zugleich nicht „Abstract“ beinhaltet. Die Testergebnisse werden im Unterverzeichnis „report/tests“ in xml-Dateien abgelegt.

Für die Darstellung der Testergebnisse an der Oberfläche sorgt das „JUnit“-Plugin. Es sorgt dafür, dass zu Testfehlschlägen Details einsehbar sind, zum Beispiel welche Assert-Anweisung für den Fehlschlag verantwortlich ist oder wo eine Exception aufgetreten ist. Weitere Informationen hierzu befinden sich in der Dokumentation für die Studierenden (Kapitel 6.3).

4.3.2 Code-Coverage

Zur Ermittlung der Code-Coverage wird Jacoco verwendet. Alternativ wäre die Verwendung von Cobertura oder Clover denkbar, allerdings bieten diese einen vergleichbaren Funktionsumfang und keine ersichtlichen Vorteile.

Für die Ermittlung der Coverage werden die Tests ausgeführt und die dabei durchlaufenen Ausführungspfade und Anweisungen registriert. Im Ant-Build-Skript wird dies durch das „coverage“-Element erreicht, welches das in Kapitel 4.3.1 erwähnte „junit“-Element umschließt. Das Ergebnis wird in die Datei „/report/unitcoverage.exec“ geschrieben. Vorab wird die Jacoco-Bibliothek der Version 0.7.6 von „github.com“ heruntergeladen.

Für die Darstellung des Ergebnisses an der Oberfläche sorgt das „Jacoco“-Plugin. Dadurch werden die absolute Anzahl und der prozentuale Anteil abgedeckter Anweisungen angegeben. Zusätzlich kann der Quelltext inklusive farblicher Markierung der Abdeckung angezeigt werden. Per Tooltip können weitere Hinweise angezeigt werden, zum Beispiel wie viele der möglichen Pfade durchlaufen wurden. Für die Musterlösungen ist die Anzeige des Quelltextes deaktiviert, damit Studierende diese nicht für das Lösen der Aufgabe verwenden. Weitere Informationen hierzu befinden sich in der Dokumentation für die Studierenden (Kapitel 6.3).

4.3.3 Programmierkonventionen und Komplexität

Die Einhaltung von Programmierkonventionen wird mittels Checkstyle geprüft. Alternativ wäre die Verwendung von PMD, Findbugs oder Sonarqube denkbar. Diese Tools haben jeweils unterschiedliche Absichten. Der Schwerpunkt von PMD ist die Erkennung schlechter Praktiken, wie zum Beispiel doppelten oder überflüssigen Quelltext. Findbugs sucht nach potentiellen Fehlern, wie zum Beispiel eine voneinander abweichende Implementierung der Methoden „hashCode“ und „equals“. Checkstyle prüft definierte Regeln und Programmierkonventionen, wie zum Beispiel Formatierung und JavaDoc-Kommentare und erfüllt damit genau unsere Anforderungen. [Okt15] [Spr16] Sonarqube ist eine Lösung, die zwar alle genannten Bereiche abdeckt, aber als separater Server betrieben wird und dessen Verwendung die Pflege eines weiteren Systems voraussetzen würde.

Checkstyle wird per xml-Datei konfiguriert, sodass individuell definierte Regeln geprüft werden. Der verwendete Regelsatz basiert auf dem „Google Java Style“. Einige Regeln wurden etwas toleranter gestaltet, wie zum Beispiel die maximale Anzahl von Zeichen pro Zeile oder die Verwendung von Unterstrich-Zeichen in Paket- und Klassennamen. Andere Regeln wurden verschärft, um die Studierenden zu sensibilisieren. Beispielsweise verursachen fehlende Javadoc-Elemente wie „@param“ oder „@return“ eine Warnung.

Der verwendete Checkstyle-Regelsatz ist einsehbar unter <http://www.nkode.io/assets/programming/eclipse/programming-freshman-checkstyle-conventions.xml>.

Checkstyle kann mittels Regeln ebenfalls die Komplexität von Quelltext prüfen. Wie gut ein Quelltext tatsächlich von einem Menschen verstanden wird hat diverse Faktoren und ist nicht automatisiert durch eine Maschine ermittelbar, zum Beispiel wie aussagekräftig Kommentare oder der Name einer Methode sind. Die logische Struktur von Quelltexten kann hingegen sehr gut ausgewertet werden und wie viele unterschiedliche Ausführungspfade einer Methode existieren beeinflusst natürlich die Verständlichkeit. Genau dies bewertet die McCabe-Metrik, welche die sogenannte zyklomatische Komplexität als Anzahl linear unabhängiger Pfade auf dem Kontrollflussgraphen angibt. Diese Kennzahl erfüllt ihren Zweck und ist trotzdem leicht verständlich, weil sie nur von Verzweigungen abhängig ist. Alternativen wie die NPath-Komplexität berücksichtigen weitere Eigenschaften, werden dadurch aber auch schwieriger nachvollziehbar. Eine ermittelte zyklomatische Komplexität alleine sagt wenig aus, sondern sie muss mit anderen Komplexitätswerten verglichen werden, zum Beispiel können Studierende die Komplexität ihrer Lösung mit der Musterlösung vergleichen. Zudem können Grenzwerte definiert werden. In der Checkstyle-Dokumentation werden Komplexitätswerte von 1 bis 4 als leicht zu testen und Werte von 5 bis 7 als „ok“ gewertet. [Che16]

Durch Verwendung der Regel für die zyklomatische Komplexität ermittelt Checkstyle diesen Wert pro Methode. Ist die Methodenkomplexität höher als der angegebene Grenzwert, so resultiert dies in einer Checkstyle-Warnung. Der Wert der komplexesten Methode einer studentischen Abgabe wird als Gesamtkomplexität gewertet. Dieser Wert wird für erfolgreiche Builds durch ein Groovy-Skript und das „Groovy Postbuild Plugin“ als Buildbeschreibung gesetzt. Die Checkstyle-Regel bietet zudem die Option, „switch“-Blöcke als einen einzigen Entscheidungspunkt zu betrachten. Diese Option wird aktiviert, weil „switch“-Ausdrücke unabhängig von der Anzahl ihrer Fälle leicht verständlich sind.

Im Ant-Build-Skript werden die Bibliothek „checkstyle-6.16-all.jar“ sowie der Regelsatz von „nkode.io“ heruntergeladen. Mittels „checkstyle“-Element werden alle Klassen geprüft, dessen Namen nicht die Zeichenkette „Test“ beinhalten. Die Ergebnisse werden mittels eingebetteter „formatter“-Elemente in die Datei „report/checkstyle.xml“ geschrieben sowie auf der Konsole ausgegeben. Die Konsolenausgabe wird vom zuvor genannten Groovy-Skript verwendet.

Für die Darstellung der Warnungen an der Oberfläche sorgt das „Checkstyle“-Plugin. Neben einer Übersicht können auch detaillierte Informationen zu aufgetretenen Warnungen angezeigt werden. Weitere Informationen hierzu befinden sich in der Dokumentation für die Studierenden (Kapitel 6.3).

4.4 Leistungsübersicht und Feedback

Das Feedback zu Lösungen erfolgt über die Web-Oberfläche. Insbesondere für die Lehrenden sind sehr viele Jobs sichtbar. Um einen guten tabellarischen Überblick zu verschaffen sind spezielle Ansichten vorbereitet, welche Spalten zur Anzeige von Korrektheit, Code-Coverage, Programmierkonventionen und Komplexität beinhalten. Für die Anzeige von Testergebnissen sowie der Buildbeschreibungen wird das „Extra Columns Plugin“ verwendet. Bei erfolgreichen Builds enthält die Buildbeschreibung die Komplexität. Fehlschlagenden Builds erhalten eine mögliche Fehlerursache als Beschreibung, wie zum Beispiel „fehlender Parameter“ oder „Kompilierfehler - Benennung prüfen“. Der Text wird mittels Groovy-Skript gesetzt, welches die Konsolenausgabe des Builds auswertet und mithilfe des „Groovy Postbuild Plugins“ nach dem eigentlichen Buildverfahren ausgeführt wird. Neben einer Ansicht mit allen Jobs gibt es zwei Ansichten, welche eine Gruppierung nach Aufgabe beziehungsweise Person vornehmen. Die Gruppen können gezielt per Plus- und Minussymbol auf- und zugeklappt werden. Für die Funktionalität sorgt das Plugin „Categorized Jobs View“. In den Ansichten wird pro Job stets das zuletzt durchgeführte Build angezeigt.

Ein detailliertes Feedback gibt es durch Auswahl eines konkreten Builds. Über die weiterführenden Links „Testergebnis“, „Testabdeckung“ und „Checkstyle Warnungen“ ist zum Beispiel einsehbar, welche Testmethoden fehlschlagen, welche Quellcodezeilen der Lösung nicht durchlaufen wurden und in welchen Zeilen Checkstyle-Warnungen auftreten.

Weitere Informationen hierzu gibt es in den Bereichen „Leistungsübersicht ansehen“ und „Abgabe einer Lösung“ der Dokumentation für Lehrende und Studierende (Kapitel 6.2 und 6.3).

4.5 Ablauf einer Auswertung

Der Buildprozess eines Jobs umfasst neben der Ausführung des Ant-Build-Skriptes weitere Schritte.

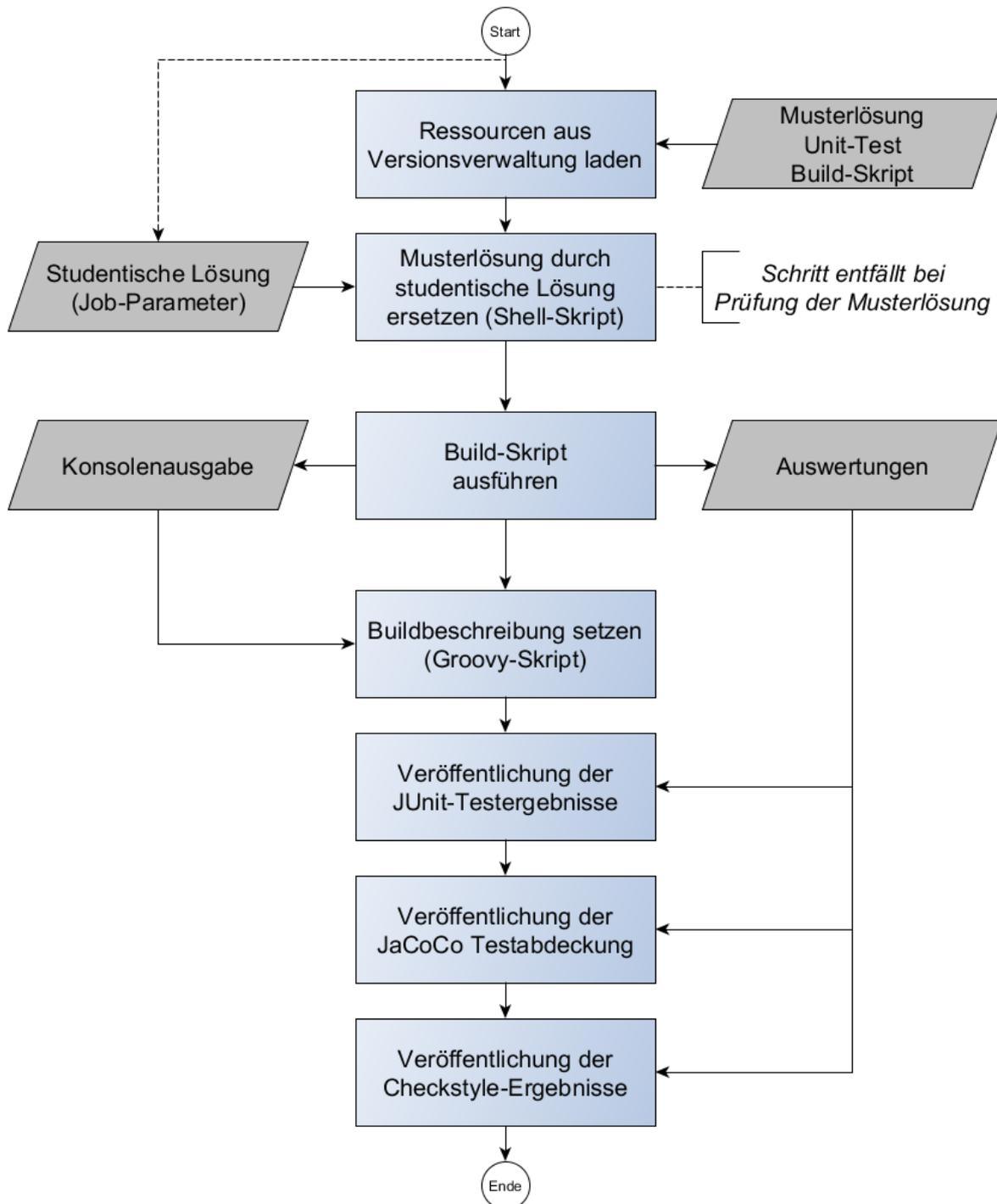


Abbildung 4-2: Ablauf des Buildprozesses

Der erste Schritt ist das Laden des Projektes zu einer konkreten Aufgabe aus der Versionsverwaltung. Dort werden vom Lehrenden die Musterlösungen, Unit-Tests und Build-Skripte gepflegt. Wenn es sich um einen Job zur Prüfung einer studentischen Lösung handelt, wird nun die Musterlösung durch die per Job-Parameter übergebene studentische Lösung ersetzt. Anschließend wird das Ant-Build-Skript aufgerufen, welches nun entweder die Musterlösung oder eine studentische Lösung auswertet. Dabei werden Dateien mit den Auswertungsergebnissen erzeugt und die Ausführung wird zusätzlich per Konsolenausgabe protokolliert. Die Konsolenausgabe wird genutzt, um mittels Groovy die Buildbeschreibung zu setzen. Abschließend werden nacheinander die Auswertungsergebnisse jeweils mittels zugehörigem Plugin veröffentlicht.

Die Konfiguration dieses Ablaufes wird Job-Definition genannt. Weitere Informationen zu den Job-Definitionen befinden sich in der Dokumentation für die Lehrenden (Kapitel 6.2).

4.6 Docker-Container

Ein Docker-Container ist eine lauffähige Instanz zu einem Docker-Image. Durch Erstellung eines eigenen Images können leicht Container auf dessen Basis erstellt werden. Das eigene Image wird mittels „Dockerfile“-Datei beschrieben. Es basiert auf dem derzeit aktuellsten Jenkins-Image der Version 2.7.1 vom 06.07.2016. Dabei handelt es sich um das erste LTS-Release der Jenkins Version 2.

Das Image beinhaltet eine initiale Jenkins „config.xml“, welche die drei konfigurierten Ansichten beinhaltet und die „Gruppierung nach Aufgabe“ als Standard-Ansicht definiert. Mittels Groovy-Skript wird eine Ant-Installation hinzugefügt, welche für die Ausführung der Ant-Skripte benötigt wird. Zudem werden die benötigten Plugins sowie deren Abhängigkeiten vorinstalliert. Dies geschieht mittels vorbereitetem Verfahren des offiziellen Jenkins-Images, die eindeutigen Namen der Plugins in eine Textdatei zu schreiben und anschließend das Skript „./usr/local/bin/plugins.sh“ mit der Textdatei als Parameter auszuführen. [DoH16]

Das Starten eines Docker-Containers unter Verwendung des Images per Docker Terminal wird in der Administrator-Dokumentation (Kapitel 6.1) beschrieben.

5 Automatisierung von Aufgaben

Dieses Kapitel beschäftigt sich mit der automatisierten Auswertung von Aufgaben des Aufgabenkatalogs der Lehrveranstaltungen „Programmieren I & II“. Hierbei werden zunächst vier konkrete Aufgaben aus dem Bereich „Programmieren I“ vollständig automatisiert. Anschließend wird die Machbarkeit der automatisierten Auswertung für offenere Aufgabenstellungen mit unterschiedlichsten Lösungsmöglichkeiten anhand von drei Aufgaben aus dem Bereich „Programmieren II“ analysiert.

5.1 Automatisierung konkreter Aufgaben

Für die vier betrachteten Aufgaben aus dem Kurs „Programmieren I“ sind jeweils eine Aufgabenstellung sowie eine Musterlösung gegeben. Die Automatisierung bestehender Aufgaben erfordert drei Bausteine. Es werden mindestens ein JUnit-Test, ein Build-Skript und zwei Job-Definitionen benötigt. Zudem müssen Benennungs- und Typisierungsregeln in die Aufgabenstellung aufgenommen werden und die Musterlösung gegebenenfalls überarbeitet werden, um gute Auswertungsergebnisse zu erzielen.

Die Musterlösungen wurden an die Checkstyle-Regeln angepasst und zum Teil die „main“-Methoden in JUnit-Tests überführt. Die Build-Skripte unterscheiden sich nur im Projektnamen. Zudem hat das Projekt der Aufgabe 13.2 eine Abhängigkeit, welche dort zusätzlich behandelt werden muss. Die Job-Definitionen unterscheiden sich in den Pfadangaben. Die Aufgabe 13.2 benötigt zudem einen weiteren Parameter, über welchen die Studierenden die abhängige Lösung hochladen. Weitere Informationen zu diesen Themen sowie der Erzeugung von Jobs per Skript anhand der Job-Definitionen befinden sich in der Dokumentation der Lehrenden (Kapitel 6.2).

Auf die spezifischen Unit-Tests wird in den folgenden Unterkapiteln gezielt eingegangen. Die bisherigen Aufgabenstellungen, Ergänzungen der Aufgabenstellungen für die Automatisierung sowie die Quelltexte der Tests befinden sich im Anhang A.

5.1.1 Schaltjahre bestimmen (Aufgabe 5.6)

Es soll eine Prüfung von Schaltjahren im Gregorianischen Kalender entwickelt werden. Es wird ein beispielhafter Programmablauf angegeben:

*Bitte geben sie eine beliebige Jahreszahl ein: 2016
2016 ist ein Schaltjahr.*

Es wird also über die Standardeingabe eine Jahreszahl eingegeben und anschließend textuell ausgegeben, ob es sich um ein Schaltjahr handelt.

Gerade im ersten Teil des Aufgabenkatalogs beinhalten viele Aufgaben die Interaktion mit der Standardein- und Ausgabe. Für diese Aufgabe wird beispielhaft diese Art der Interaktion in JUnit-Tests betrachtet. Ohne diese Interaktion könnte folgende Methodensignatur vorgeschrieben werden:

```
public static boolean isSchaltjahr(int jahr)
```

Stattdessen wird hier in der Klasse „de.fhl.prog.aufgabe_05_6.Aufgabe_05_6“ eine „main“-Methode mit der üblichen Signatur erwartet. Der JUnit-Test ruft diese „main“-Methode auf, schreibt die zu testende Jahreszahl in die Standardeingabe und erwartet anschließend einen gewissen Text auf der Standardausgabe. Ermöglicht wird dies durch die Umleitung der Datenströme via „java.lang.System.setIn(...)“ und „java.lang.System.setOut(...)“. Diese Logik wird in der Klasse „AbstractConsoleTest“ gekapselt. „TestSchaltjahr“ erweitert diese Klasse und beinhaltet die konkreten Testfälle.

Die Berechnung zur Bestimmung von Schaltjahren ist abhängig von der ganzzahligen Teilbarkeit der Jahreszahl durch 4, 100 und 400. Es werden also mehrere Testfälle benötigt, welche jeweils eine Jahreszahl und ein erwartetes Ergebnis haben. Das erwartete Ergebnis wird als boolean-Wert abgebildet, wobei je nach Wahrheitswert die textuelle Ausgabe „ist ein Schaltjahr“ beziehungsweise „ist kein Schaltjahr“ erwartet wird. Die unterschiedlichen Testfälle sind in der Art ihrer Ausführung identisch, weshalb ein parametrisierter JUnit-Test genutzt wird. Dadurch wird nur eine Testmethode geschrieben, welche die unterschiedlichen Testfall-Parameter per Klassenattribut erhält. Es werden folgende Werte getestet:

Jahreszahl (Eingabe)	Handelt es sich um ein Schaltjahr? (erwartetes Ergebnis)
2011	nein
2012	ja
2013	nein
2014	nein
2015	nein
2016	ja
2017	nein
1500	nein
1600	ja
1700	nein
1800	nein
1900	nein
2000	ja
2100	nein

Tabelle 5-1: Schaltjahr-Testfälle

In Anhang A.1 befinden sich Ergänzungen für die Aufgabenstellung sowie der Quelltext der Tests.

5.1.2 Multiplikationstabelle (Aufgabe 7.1)

Es soll die Erzeugung einer Multiplikationstabelle entwickelt werden. Es wird ein beispielhafter Programmablauf angegeben:

Geben Sie bitte eine Zahl i ein: 13

Geben Sie bitte eine Zahl j ein: 3

Die Multiplikationstabelle lautet:

```
1 2 3 4 5 6 7 8 9 10 11 12 13
2 4 6 8 10 12 14 16 18 20 22 24 26
3 6 9 12 15 18 21 24 27 30 33 36 39
```

Hierbei liegt der Schwerpunkt jedoch nicht auf der Interaktion mit der Kommandozeile, sondern es handelt sich um eine parametrisierte Weiterentwicklung einer vorangegangenen Aufgabe (6.5) und die Funktionalität soll nun lediglich in einer Methode „mtab“ gekapselt werden. Diese Art der Aufgabenstellung eignet sich gut zur automatisierten Auswertung.

Es wird die Rückgabe für folgende Werte getestet:

Wert für i	Wert für j	Erwartetes Ergebnis / Besonderheit
1	1	„1“
1	2	Tabelle mit nur einer Spalte
3	1	Tabelle mit nur einer Zeile
4	5	komplette 4x5-Tabelle
13	3	komplette 13x3-Tabelle
0	0	leere Zeichenkette
0	1	leere Zeichenkette
1	0	leere Zeichenkette
-1	-1	leere Zeichenkette

Tabelle 5-2: Multiplikationstabelle-Testfälle

Durch die gewählten Testfälle werden sowohl die grundsätzliche Funktionalität als auch Randfälle getestet. Der Wert von i ist mal kleiner, gleich oder größer als der Wert von j. Zudem wird der Umgang mit dem Wert 0 sowie negativem Wert getestet. Die Testfälle hätten ebenfalls als parametrisierbarer Test umgesetzt werden können. Hier wurde darauf verzichtet und stattdessen sinnvolle Methodennamen verwendet, welche bei Testfehlschlägen hilfreich sein können, wie zum Beispiel „testNegativeParameter“.

Für diese Aufgabe erfolgte im Vorfeld dieser Arbeit ein erstes Experiment der Automatisierung durch Herrn Prof. Dr. Kratzke. Sowohl die Testklasse mit sechs der neun Testmethoden als auch das Build-Skript wurden weiterverwendet und die Aufgabenstellung eignet sich daher bereits gut zur Automatisierung.

In Anhang A.2 befinden sich Ergänzungen für die Aufgabenstellung sowie der Quelltext der Tests.

5.1.3 Ausgaberroutine für Stack, List und Map (Aufgabe 10.1)

Es soll eine überladene Methode „collection_to_string“ zur Erzeugung gut lesbarer String-Repräsentationen für die Datentypen Stack, List und Map entwickelt werden. Ein beispielhafter Programmablauf zeigt, dass eine Liste und ein Stack mit den Werten 1, 2, 3 und 4 als „[1, 2, 3, 4]“ beziehungsweise „[4, 3, 2, 1]“ dargestellt werden sollen. Für eine Map sollen die Schlüssel-Wert-Paare mittels Pfeil dargestellt werden, also zum Beispiel „[65 -> A, 66 -> B, 67 -> C, 68 -> D]“.

Hierbei fällt auf, dass den Studierenden durch die vorgegebenen Methodensignaturen ein Teil der Arbeit abgenommen wird. Zuvor musste der entscheidende Hinweis „überladene Methode“ eigenständig verarbeitet und die geeigneten Methodensignaturen entsprechend gebildet werden.

Getestet werden die drei Methoden jeweils mit einer leeren Datenstruktur sowie mit einer fünf Elemente beinhaltenden Datenstruktur. Als Zusatzforderung wird geprüft, dass der Inhalt der übergebenen Datenstruktur nicht verändert wurde. Dies ist sinnvoll, da es sich lediglich um die Erzeugung von String-Repräsentationen handelt und die Objekte danach möglicherweise weiter verwendet werden. Insbesondere bei der Datenstruktur Stack besteht die Neigung die „pop“-Methode zu verwenden, welche das übergebene Objekt ändert, sofern zuvor keine Arbeitskopie erstellt wurde. Die Musterlösung wurde ebenfalls angepasst und erstellt für den Stack nun eine Arbeitskopie.

In Anhang A.3 befinden sich Ergänzungen für die Aufgabenstellung sowie der Quelltext der Tests.

5.1.4 Bestimmen von Baumeigenschaften (Aufgabe 13.2)

Es sollen die Höhe sowie die Anzahl an Knoten und Kanten eines Binärbaums ermittelt werden. Hierfür werden bereits drei Methodensignaturen vorgegeben. Hierbei wird als Parameter der Typ „Node“ aus der vorausgegangen Aufgabe 13.1 wiederverwendet.

Die drei Methoden werden anhand eines einzelnen Knoten sowie eines ebenfalls in der Aufgabenstellung angegebenen Binärbaumes getestet.

Die Besonderheit an dieser Aufgabe ist die erwähnte Abhängigkeit zur Node-Klasse der Vorgänger-Aufgabe. Damit diese Klasse in der Musterlösung nicht dupliziert wird, verwendet die Aufgabe 13.2 die Klasse wieder. Hierfür wird eine Projektabhängigkeit gesetzt. Beim Kompilieren per Build-Skript muss dies ebenfalls berücksichtigt werden. Zusätzlich müssen die studentischen Abgaben diese Klasse beinhalten. Hierfür wird entsprechend ein zweiter Parameter in der Job-Definition hinzugefügt. Die Verarbeitung des Parameters erfolgt analog zu dem ersten Parameter. In den Abschnitten „Aufgabe vorbereiten“ und „Job-Definitionen erstellen“ der Dokumentation für Lehrende (Kapitel 6.2) wird genau dieser Fall erklärt.

In Anhang A.4 befinden sich Ergänzungen für die Aufgabenstellung sowie der Quelltext der Tests.

5.1.5 Zusammenfassung

Die betrachteten vier Aufgaben waren leicht zu automatisieren. Dabei umfassten die gewählten Aufgaben mit Konsoleninteraktion sowie Abhängigkeiten zu anderen Aufgaben durch aufbauende Aufgabenstellungen zwei wiederkehrende Aspekte des Aufgabenkatalogs. Beides stellte kein Problem bei der Automatisierung dar. Es ist allerdings empfehlenswert, auf die Konsoleninteraktion zu verzichten und Funktionalitäten in einer Methode gekapselt einzufordern, weil dies wesentlich besser zu testen ist. Sobald das Interface-Konzept in der Vorlesung behandelt wurde, können Methodennamen sowie Parameter- und Rückgabetypen elegant per Interface vorgegeben werden.

Neben den genannten Anpassungen der konkreten Aufgabenbeschreibungen sollte dem Aufgabenkatalog ein einleitender Text hinzugefügt werden, in welchem verständlich die Wichtigkeit der Einhaltung von Benennungs- und Typisierungsvorschriften thematisiert wird. Hier sollte zudem angegeben werden, wo die Studierenden-Dokumentation für den Umgang mit dem Jenkins-System zu finden ist.

Durch die strengen Vorschriften, wie die Struktur der Klassen aussehen muss, entfällt beim Lösen allerdings ein Aspekt. Die Studierenden müssen sich nicht mehr selber Gedanken über sinnvolle Methodensignaturen machen, sondern bekommen diese vorgegeben. Die Automatisierung kann also auch einen negativen Einfluss auf die Aufgabenstellung haben. Welche Auswirkungen dies für komplexere Aufgaben mit offenen Aufgabenstellungen hat, wird im folgenden Kapitel betrachtet.

5.2 Analyse komplexerer Aufgaben

In diesem Kapitel werden Aufgaben des Kurses „Programmieren II“ betrachtet, wobei auch hier neben der Aufgabenstellung eine Musterlösung gegeben ist. Im Allgemeinen werden die Aufgaben des Aufgabenkatalogs zunehmend komplexer und die Aufgabenstellungen sind offener und bieten somit flexiblere Lösungsmöglichkeiten. Es erfolgt keine Automatisierung der Aufgaben, sondern es wird die Machbarkeit analysiert. Hierfür werden zunächst zwei konkrete Aufgaben detailliert betrachtet und anschließend eine Aufgabengruppe, dessen Schwerpunkt auf der grafischen Benutzeroberfläche liegt. Die Aufgaben werden grob erklärt, die ausführliche Aufgabenbeschreibung ist dem Aufgabenkatalog zu entnehmen.

5.2.1 Telefonbuch (17.4)

Ein vorgegebenes UML-Klassendiagramm mit einem Interface und vier Klassen soll in Quelltext überführt werden. Zusätzlich werden bestimmte Formatierungen durch die „toString“-Methoden erwartet und es soll eine Methode „telefonbuch“ implementiert werden, welche anhand eines erzeugten Objekt-Geflechtes eine Stringrepräsentation in einem vorgegebenen Format bildet.

Durch das UML-Klassendiagramm werden bereits Klassennamen und Methodensignaturen exakt vorgegeben. Hier muss lediglich eine Angabe hinzugefügt werden, in welchem Paket sich alle Klassen befinden sollen, wie zum Beispiel „de.fhl.prog.aufgabe_17_4“. Die Signatur der „telefonbuch“-Methode ist ebenfalls spezifiziert, allerdings fehlt hier die Angabe einer Klasse. Diese könnte man beispielsweise in einer alleinstehenden Klasse „Aufgabe_17_4“ fordern. Alternativ kann dem UML-Klassendiagramm auch die Klasse „Telefonbuch“ hinzugefügt werden, welche eine „0..*“-Beziehung zur Klasse „Mitarbeiter“ hat. Die Erzeugung der Stringrepräsentation würde dann per „toString“-Methode erfolgen.

Die Klassen beinhalten öffentliche „getter“-Methoden und die Methode zur Erzeugung der Stringrepräsentation liefert diese als Rückgabewert. Dadurch sind die korrekte Erzeugung von Objekt-Geflechtes sowie die erwartete Darstellung als Zeichenkette gut per JUnit-Test prüfbar.

Die Lösung dieser Aufgabe umfasst sechs Dateien. Eine Abgabe über sechs Job-Parameter ist denkbar. Alternativ könnte die Abgabe über eine Containerdatei wie „zip“ oder „gz“ erfolgen. Ant bietet die Funktionalität, diese zu entpacken. Bei der Weiterverarbeitung muss allerdings bedacht werden, dass die Container beliebige Dateien beinhalten können. Dies bietet zusätzliche Möglichkeiten zur Manipulation, wie zum Beispiel das Injizieren von Build-Skripten oder Testergebnissen. Um dies zu verhindern, könnte der Container zunächst in ein separates Verzeichnis entpackt werden und anschließend lediglich die Dateien mit der Endung „.java“ an die entsprechenden Stellen kopiert werden. Zusätzlich müssen Klassen mit der Zeichenkette „Test“ im Namen herausgefiltert werden, da ansonsten das

Auswertungsergebnis durch eigene Tests verfälscht werden könnte. Die beiden Varianten, mehrere Job-Parameter oder eine Containerdatei zu nutzen, sorgen für eine erhöhte Komplexität der Job-Definitionen.

5.2.2 Generische Warteschlange (20.1)

Es soll eine generische Warteschlange entwickelt werden, wobei der Klassenname sowie die Signaturen von fünf zu implementierenden Methoden vorgegeben werden. Zur automatisierten Auswertung der Datenstruktur fehlt lediglich die Vorgabe eines Paketes, wie zum Beispiel „de.fhl.prog.aufgabe_20_1“.

Die Aufgabenstellung beinhaltet zusätzlich Quelltext zum Testen der Datenstruktur durch Konsolenausgaben. Solange die Warteschlange nicht leer ist, wird innerhalb einer Schleife pro Durchlauf ein Element entfernt und mit einer gewissen Wahrscheinlichkeit ein neues Element hinzugefügt. Dieser Quelltext soll so angepasst werden, dass auch das Entfernen von Elementen nur mit einer gewissen Wahrscheinlichkeit erfolgt. Anschließend sollen zwei Fragen beantwortet werden: Ist es durch die zwei Zufallskomponenten noch sicher, dass das Programm terminiert? Welche Bedingung für die Wahrscheinlichkeiten muss gelten, damit das Programm höchstwahrscheinlich terminiert? Sowohl die Quelltextanpassung als auch die Beantwortung der Fragen kann in der aktuellen Struktur nicht automatisiert ausgewertet werden. Die Anpassung erfolgt innerhalb einer „while“-Schleife einer „main“-Methode. Was genau pro Schleifendurchlauf passiert, ist nicht von außerhalb testbar. Funktionalität mit Zufallskomponenten ist zudem nicht absolut zuverlässig per Unit-Test prüfbar. Die Antworten auf die Fragen erfordert eine Auswertung durch einen Lehrenden.

5.2.3 GUI-Aufgaben (22)

Die Aufgaben zur grafischen Benutzeroberfläche beinhalten die Programmierung eines Taschenrechners, die Darstellung von bewegten Figuren, „Conways Game of Life“ sowie das Spiel „Tic Tac Toe“. Teilweise wird die Berücksichtigung des „Model-View-Controller“-Konzeptes vorgegeben, was die Trennung der Verantwortlichkeiten beinhaltet. Das Model ist dabei unabhängig von View und Controller und kann daher gut per Unit-Test geprüft werden, sofern die entsprechende Struktur vorgegeben ist. Auf die Aufgaben bezogen bedeutet dies zum Beispiel, ob der Taschenrechner die Berechnungen korrekt durchführt oder ob das Spielfeld in „Conways Game of Life“ den erwarteten Zustand hat. Ob die Werte des Models auch korrekt dargestellt werden, ist deutlich schwieriger zu testen. Der Controller hat üblicherweise Abhängigkeiten zum Model und zur View. Mithilfe einer leeren Implementierung einer View als eine Art Attrappe kann zumindest die Manipulation des Models durch den Controller mittels Unit-Tests abgedeckt werden.

Um die View mit ihren Interaktionsmöglichkeiten sowie das gesamte Zusammenspiel mit Controller und Model zu testen, reichen gewöhnliche Unit-Tests jedoch nicht aus. Systemeingaben für „awt“- und „Swing“-Oberflächen können zum Beispiel mithilfe der Klasse `java.awt.Robot` generiert werden. Zudem gibt es viele Bibliotheken zum automatisierten Testen unterschiedlicher Oberflächen. Beispielsweise „QF-Test“ unterstützt ebenfalls „awt“- , „Swing“, und auch „JavaFX“-Oberflächen und bietet sogar ein Jenkins-Plugins an, wodurch die Integration in ein Jenkins-System sichergestellt wird. [QFT16] Es ist also grundsätzlich möglich, Aufgaben mit grafischer Benutzeroberfläche automatisiert auszuwerten. Vermutlich erfordern die automatisierten Oberflächentests jedoch sehr genaue Vorgaben an die Oberfläche, wodurch kaum Freiraum bei der Implementierung durch die Studierenden existiert, sondern die Lösung fast vorgegeben wird. Zudem können automatisierte Tests lediglich die Funktionalität testen. Aspekte wie Design und Benutzerfreundlichkeit einer Oberfläche erfordern die Bewertung durch einen Lehrenden. Des Weiteren sind die genannten Möglichkeiten für die Verwendung von Standard-Komponenten wie Textfelder und Schaltflächen ausgelegt. Ob beispielsweise Figuren richtig dargestellt werden und sich wie vorgegeben bewegen, ist nicht sinnvoll automatisiert auswertbar.

5.2.4 Zusammenfassung

Die betrachteten Aufgaben beinhalten einige neue Aspekte. Aufgaben, dessen Lösung mehrere Dateien umfassen, stellen kein Problem bei der Automatisierung dar. Sobald Aufgaben jedoch nicht mehr nur aus der Implementation von Quelltext bestehen, sondern weitere Teile beinhalten, wie zum Beispiel die Beantwortung von Fragen, ist die Auswertung durch einen Lehrenden notwendig. Teile der Lösungen von Aufgaben mit grafischer Benutzeroberfläche können mittels gewöhnlicher Unit-Tests geprüft werden. Es ist auch denkbar, die Funktionalität der Oberfläche automatisiert zu testen. Aspekte wie Design und Benutzerfreundlichkeit werden dadurch jedoch nicht berücksichtigt.

6 Anwender-Dokumentationen

Die Anwender-Dokumentation ist in drei Teile gegliedert, um gezielt auf die Rollen Administrator, Lehrender und Studierender einzugehen. Die allgemeinen Hinweise zur Konfiguration per Kommandozeile sowie die Erklärung der Leistungsübersicht betreffen jeweils zwei der genannten Rollen. Damit der Anwender genau die für sich wichtigen Informationen erhält, beinhalten beide Teildokumentationen den jeweiligen Abschnitt in leicht angepasster Form.

6.1 Dokumentation für Administratoren

Voraussetzungen

- Docker-Installation (<https://docs.docker.com/engine/installation/>)
- Jenkins-Verzeichnis von der CD oder aus dem Repository zu dieser Arbeit (<https://harry7e@bitbucket.org/harry7e/automatisierte-auswertung-von-programmier-praktika.git>, Zugriffsrechte benötigt)
- Zur Konfiguration per Kommandozeile (kann remote erfolgen):
 - Java-Installation
 - Ausführung von Bash-Skripten möglich

System starten

Das System wird als Docker-Container betrieben. Dafür wird zunächst per Docker Terminal das Image erstellt. Hierfür wird in das Verzeichnis der *Dockerfile*-Datei navigiert und dann der *build*-Befehl verwendet.

```
cd d:/Jenkins/docker
docker build -t aavpp .
```

Diese beispielhafte Eingabe erwartet das Jenkins-Verzeichnis unter „d:/“ und erstellt ein Docker-Image mit dem Namen „aavpp“.

Das Starten eines Containers mit diesem Image erfolgt ebenfalls per Docker Terminal.

```
docker run -d -p 8080:8080 -p 50000:50000 --name=myaavppinstance aavpp
```

Diese beispielhafte Eingabe startet einen neuen Container mit dem Namen „myaavppinstance“ unter Verwendung des Images „aavpp“. Neben dem http-Port 8080 wird der Port 50000 für die Konfiguration per Kommandozeile benötigt.

Die anfallenden Daten eines Docker-Containers werden direkt in diesem gespeichert. Dies bedeutet, dass durch Löschen eines Containers auch dessen Daten verloren gehen. Um dies zu verhindern, können Docker-Volumes verwendet werden. Hierfür kann der obige *run*-Befehl mit einem zusätzlichen Parameter (an beliebiger Stelle vor dem Namen des Images) versehen werden.

```
-v /c/Users/MyUser/aavpp_storage:/var/jenkins_home
```

Diese beispielhafte Parameter-Eingabe sorgt dafür, dass die Jenkins-Daten im Verzeichnis „c/Users/MyUser/aavpp_storage“ gespeichert werden. Das Löschen des Containers lässt dieses Verzeichnis bestehen. Bei der Erstellung eines neuen Containers kann dieses Volume mit den Daten wiederverwendet werden.

Die IP-Adresse der Docker-Maschine wird beim Starten des Docker Terminals angezeigt. Alternativ kann diese auch folgendermaßen ermittelt werden:

```
docker-machine ip
```

Über diese IP-Adresse ist die Jenkins-Oberfläche per Browser erreichbar, zum Beispiel unter „<http://192.168.99.100:8080/>“.

Zusätzliche Informationen zu den verwendeten Befehlen sind in der Docker-Dokumentation unter <https://docs.docker.com/engine/> sowie in der Beschreibung des offiziellen Jenkins-Images unter https://hub.docker.com/_/jenkins/ zu finden.

Allgemeine Hinweise zur Konfiguration per Kommandozeile

Die vorbereiteten Skripte setzen die Umgebungsvariable *JENKINS_URL* voraus, welche als Wert die URL des zu konfigurierenden Servers haben muss, also zum Beispiel „<http://192.168.99.100:8080/>“. Die Datei „*readme.txt*“ beinhaltet für alle Skripte eine Auflistung der erwarteten Parameter.

Sobald die Zugriffskontrolle aktiviert ist, muss vor der Verwendung weiterer Konfigurationsskripte eine Authentifizierung mittels *login.sh* durchgeführt werden. Das Skript erwartet zwei Parameter: Benutzername und Passwort.

```
.login.sh myUsername myPassword
```

Abschließend sollte sich mittels *logout.sh* ausgeloggt werden.

```
.logout.sh
```

Die grundlegenden Regeln bei der Verwendung der Kommandozeile sind zu beachten, zum Beispiel die Angabe von Zeichenketten mit Leerzeichen zwischen Anführungszeichen und die besondere Markierung bestimmter Zeichen, damit diese wie gewünscht interpretiert werden. Weitere Informationen hierzu befinden sich unter <http://tldp.org/LDP/abs/html/special-chars.html>.

Initiale Konfiguration (Zugriffskontrolle und Mailserver einrichten)

Als Erstes werden per Skript *0_init_server.sh* einige Grundeinstellungen des Servers vorgenommen. Es wird die Zugriffssteuerung aktiviert, ein Administrator-Benutzerkonto angelegt und der Mailserver konfiguriert. Vor der Ausführung ist die Zugriffskontrolle noch deaktiviert, sodass vorab kein Einloggen notwendig ist. Das Skript erwartet folgende Parameter:

1. Benutzername des Administrators
2. Passwort des Administrators
3. E-Mail-Adresse des Administrators
4. Adresse des SMTP-Mail-Servers
5. Benutzername zur SMTP Authentifizierung
6. Passwort zur SMTP Authentifizierung
7. Boolean, ob SSL verwendet werden soll
8. SMTP-Port

Der aktuelle Mail-Server der Fachhochschule Lübeck ist mit der Adresse „mail.fh-luebeck.de“ ohne SSL an Port 587 nutzbar. Informationen hierzu befinden sich im Lernraum-Kurs „IT-Services des Rechenzentrums“. Ein Aufruf des Skriptes mit der Verwendung dieses Mail-Servers könnte wie folgt aussehen:

```
./0_init_server.sh myJenkinsUsername myJenkinsPassword  
timo.hamann@stud.fh-luebeck.de mail.fh-luebeck.de 244251 myFHPasswrd false 587
```

Nach der Ausführung des Skriptes können nur noch authentifizierte Benutzer auf die Inhalte zugreifen. Die Konfiguration des SMTP-Mail-Servers ist zwingend erforderlich, weil bei der Erzeugung von Benutzerkonten Passwörter generiert und per Mail verschickt werden.

Benutzerkonten der Lehrenden hinzufügen

Das Anlegen von Benutzerkonten der Lehrenden erfolgt per Skript *1_add_tutor.sh*. Es wird die E-Mail-Adresse als Parameter erwartet.

```
./1_add_tutor.sh timo.hamann@stud.fh-luebeck.de
```

Als Benutzername wird die Zeichenkette vor dem „@“-Zeichen der E-Mail-Adresse verwendet. Für den Benutzer wird ein initiales Passwort generiert und ihm per E-Mail geschickt. Um die Lösungen aller Studierenden einsehen zu können, erhalten über dieses Skript angelegte Benutzer Zugriff auf alle Jobs. Zudem dürfen sie Zugangsdaten verwalten und Skripte ausführen, wodurch die Verwaltung der Aufgaben ermöglicht wird.

6.2 Dokumentation für Lehrende

Voraussetzungen

- Die URL zum Jenkins-Server
- Ein Jenkins-Benutzerkonto mit entsprechenden Rechten.
Die Zugangsdaten wurden per E-Mail geschickt.
- Zur Pflege von Aufgaben:
 - Java-Installation
 - Ausführung von Bash-Skripten möglich
 - Jenkins-Verzeichnis von der CD oder aus dem Repository zu dieser Arbeit (<https://bitbucket.org/harry7e/automatisierte-auswertung-von-programmier-praktika>, Zugriffsrechte benötigt)
 - Zugriff auf die Versionsverwaltung

Allgemeine Hinweise zur Konfiguration per Kommandozeile

Die vorbereiteten Skripte setzen die Umgebungsvariable `JENKINS_URL` voraus, welche als Wert die URL des zu konfigurierenden Servers haben muss, also zum Beispiel „`http://192.168.99.100:8080/`“. Die Datei „`readme.txt`“ beinhaltet für alle Skripte eine Auflistung der erwarteten Parameter.

Vor der Verwendung der Konfigurationsskripte muss eine Authentifizierung mittels `login.sh` durchgeführt werden. Das Skript erwartet zwei Parameter: Benutzername und Passwort.

```
./login.sh myUsername myPassword
```

Abschließend sollte sich mittels `logout.sh` ausgeloggt werden.

```
./logout.sh
```

Die grundlegenden Regeln bei der Verwendung der Kommandozeile sind zu beachten, zum Beispiel die Angabe von Zeichenketten mit Leerzeichen zwischen Anführungszeichen und die besondere Markierung bestimmter Zeichen, damit diese wie gewünscht interpretiert werden. Weitere Informationen hierzu befinden sich unter <http://tldp.org/LDP/abs/html/special-chars.html>.

Aufgabe vorbereiten

Die automatisierten Auswertungen von Code-Coverage, Komplexität und Programmierkonventionen erfordern keine weiteren Tätigkeiten. Für die Prüfung auf Korrektheit müssen jedoch aufgabenspezifische JUnit-Tests geschrieben werden. Bei der

Erstellung des JUnit-Tests verwendet man die Musterlösung als Testobjekt. Die Jenkins-Jobs zur Prüfung der studentischen Lösungen ersetzen bei ihrer Durchführung die Musterlösungs-Klasse durch die studentische Lösung, wodurch dann diese ausgewertet wird. Damit der Test dann noch lauffähig ist, müssen Package, Klassen- und Methodennamen sowie Parameter- und Rückgabetypen der studentischen Lösung mit der Musterlösung übereinstimmen. Um dies sicherzustellen, müssen die Aufgabenbeschreibungen im Aufgabenkatalog diese Regeln genau spezifizieren.

Damit die Tests bei den Auswertungen von Code-Coverage, Komplexität und Programmierkonventionen nicht berücksichtigt werden, müssen diese von den Lösungsklassen unterschieden werden können. Diese Unterscheidung erfolgt anhand des Klassennamens, wobei die Testklassen die Zeichenkette „Test“ im Namen enthalten müssen. Die Lösungsklassen dürfen „Test“ dementsprechend nicht im Namen beinhalten.

Unter <http://junit.org/junit4/> ist beschrieben, wie JUnit-Tests geschrieben werden.

Die bestehenden Musterlösungen beinhalten in der Regel eine main-Methode, welche für eine Eingabe das Berechnungsergebnis auf der Konsole ausgibt. Anhand dieser Ausgabe kann dann manuell überprüft werden, ob die Berechnung wie erwartet funktioniert. Diese main-Methoden können leicht in einen JUnit-Test umformuliert werden. Beispiel:

```
public abstract class Aufgabe_13_2 {

    public static void main(String[] args) {
        Node tree = new Node(5,
            new Node(3, new Node(1, null, null), new Node(4, null, null)),
            new Node(8, new Node(6, null, null),
                new Node(9, null, new Node(37,
                    new Node(17, null, null), new Node(42, null, null)))));

        System.out.println("Anzahl Knoten: " + countNodes(tree));
        System.out.println("Anzahl Kanten: " + countEdges(tree));
        System.out.println("Höhe des Baums: " + height(tree));
    }

    // ...
}
```

Diese main-Methode sorgt für folgende Ausgabe:

```
Anzahl Knoten: 10
Anzahl Kanten: 9
Höhe des Baums: 5
```

Diese drei Berechnungsergebnisse der Methoden „countNodes“, „countEdges“ und „height“ können beispielsweise in den folgenden JUnit-Test überführt werden.

```
public class TestBaumeigenschaften {

    private Node beispielbaum;

    @Before
    public void setUp() {
        beispielbaum = new Node(5,
            new Node(3, new Node(1, null, null), new Node(4, null, null)),
            new Node(8, new Node(6, null, null),
                new Node(9, null, new Node(37,
                    new Node(17, null, null), new Node(42, null, null)))));
    }

    @Test
    public void testCountNodes_Beispielbaum() {
        Assert.assertEquals(10, Aufgabe_13_2.countNodes(beispielbaum));
    }

    @Test
    public void testCountEdges_Beispielbaum() {
        Assert.assertEquals(9, Aufgabe_13_2.countEdges(beispielbaum));
    }

    @Test
    public void testHeight_Beispielbaum() {
        Assert.assertEquals(5, Aufgabe_13_2.height(beispielbaum));
    }
}
```

Beim Starten der Jenkins-Jobs wird ein Ant-Skript aufgerufen, welches die Auswertungen durchführt. Anschließend werden die Auswertungsergebnisse in bestimmten Dateien und Formaten erwartet und visuell aufbereitet. Die Skript-Datei „build.xml“ wird direkt im Projektverzeichnis erwartet. Das Skript kann von einer bereits automatisierten Aufgabe kopiert werden, wobei der Projektname angepasst werden sollte.

```
<project name="Aufgabe 13.2" default="all" basedir=".">
```

Weitere Änderungen sind dort nur vorzunehmen, wenn das Projekt Abhängigkeiten zu anderen Projekten hat. Dies muss im Ant-Target „compile“ zusätzlich berücksichtigt werden. Dies ist in Aufgabe 13.2 der Fall. Dort wird die Klasse „Node“ aus Aufgabe 13.1 verwendet.

```
<target name="compile" depends="prepare" description="Compile">
  <mkdir dir="bin"/>
  <javac debug="on" includeantruntime="false" srcdir="../ML-Aufgabe-13_1/src"
  destdir="bin" includes="**/Node*"/>
  <javac debug="on" includeantruntime="false" srcdir="src" destdir="bin">
    <classpath><pathelement location="lib/junit.jar"/></classpath>
  </javac>
</target>
```

Die Tests und das Ant-Skript müssen im VCS gespeichert werden, da der Jenkins-Job seine Ressourcen direkt aus diesem lädt.

Weitere Informationen zu Ant befinden sich unter <https://ant.apache.org/manual/>.

Job-Definitionen erstellen

Pro Aufgabe wird ein Job für die Musterlösung benötigt. Zudem ist pro Aufgabe und Studierenden ein separater Job erforderlich. Diese große Anzahl an Jobs kann durch Skripte generiert werden. Hierfür werden xml-Definitionen der Jobs benötigt, wobei diese für die Musterlösung und die studentischen Lösungen unterschiedlich sind. Die Jobs für studentische Abgaben benötigen zusätzlich Datei-Parameter und müssen sich um dessen Verarbeitung kümmern.

Ein Job kann an der Oberfläche konfiguriert werden und die Definition mittels Jenkins CLI (Command Line Interface) Befehl `get-job` gelesen und in eine Datei gespeichert werden.

```
java -jar jenkins-cli.jar get-job testjob > testjob.xml
```

Anmerkung: In der aktuellen CLI-Version verhindert ein Bug den Export eines Jobs, wenn „Anonym“ kein Leserecht für diesen Job hat. Siehe <https://issues.jenkins-ci.org/browse/JENKINS-12543>.

Beim Einrichten eines Jobs an der Oberfläche kann ein bestehender Job kopiert werden, wodurch viel Konfigurationsaufwand gespart werden kann. Für neue Aufgaben ist es jedoch noch einfacher, die Jobdefinitionen der bereits vorhandenen Aufgaben zu kopieren und anzupassen.

Um beispielsweise eine Jobdefinition für Aufgabe 7.1 anhand der Aufgabe 5.6 zu erstellen, müssen alle Vorkommnisse der Zeichenkette „05_6“ durch „07_1“ ersetzt werden. Dies kann in Eclipse komfortabel über den „Suchen und Ersetzen“-Mechanismus erfolgen. Es wird unter anderem der folgende Block angepasst:

```
<hudson.plugins.git.extensions.impl.SparseCheckoutPath>  
  <path>ML-Aufgabe-07_1</path>  
</hudson.plugins.git.extensions.impl.SparseCheckoutPath>
```

Dieser Block sorgt dafür, dass nicht das komplette Repository ausgecheckt wird, sondern nur der Unterordner zu der konkreten Aufgabe. Sollte das Projekt Abhängigkeiten zu anderen Projekten haben, müssen diese ebenfalls aufgeführt werden.

Die Jobdefinitionen für die Verarbeitung studentischer Lösungen beinhalten zusätzlich Parameter und dessen Verarbeitung. Der folgende Block gibt an, dass der Job eine Datei als Parameter erwartet.

```
<hudson.model.ParametersDefinitionProperty>  
  <parameterDefinitions>  
    <hudson.model.FileParameterDefinition>  
      <name>Aufgabe_07_1.java</name>  
      <description>de/fhl/prog/aufgabe_07_1/Aufgabe_07_1.java</description>  
    </hudson.model.FileParameterDefinition>  
  </parameterDefinitions>  
</hudson.model.ParametersDefinitionProperty>
```

Das Name-Element gibt an, wo die übermittelte Datei im Arbeitsbereich des Jobs abgelegt wird. Der Dateiname „Aufgabe_07_1.java“ sorgt dafür, dass die Datei im Wurzelverzeichnis des Arbeitsbereiches landet. Das Verzeichnis „ML-Aufgabe-07_1“ mit der Musterlösung, dem Test und dem Build-Skript wird ebenfalls in das Wurzelverzeichnis geladen.

Im ersten Build-Schritt der Jobs für studentische Lösungen wird die Parameter-Datei an die erwartete Stelle kopiert. Zuvor wird die Musterlösung aus dem Arbeitsbereich gelöscht.

```
<hudson.tasks.Shell>  
  <command>  
    rm ML-Aufgabe-07_1/src/de/fhl/prog/aufgabe_07_1/Aufgabe_07_1.java  
    cp Aufgabe_07_1.java  
      ML-Aufgabe-07_1/src/de/fhl/prog/aufgabe_07_1/Aufgabe_07_1.java  
  </command>  
</hudson.tasks.Shell>
```

Die manuelle Verarbeitung der Parameter ist notwendig, um mögliche Manipulationsversuche der Studierenden zu verhindern. Beispielsweise würde die Angabe des richtigen Pfades im Name-Element das Löschen und Kopieren überflüssig machen, jedoch resultiert das Starten eines Builds ohne Angabe einer Parameter-Datei dann in der Auswertung der Musterlösung.

Die Jobdefinitionen sollten ebenfalls im VCS gespeichert werden, da diese bei der Einrichtung neuer Server-Instanzen erneut benötigt werden.

Zugangsdaten hinzufügen

Der Jenkins-Server greift während der Durchführung von Jobs auf die Versionsverwaltung mit den Musterlösungen und Tests zu und benötigt daher die Zugangsdaten. Die Job-Definitionen beinhalten aus Sicherheitsgründen nicht die kompletten Zugangsdaten mit Benutzername und Passwort, sondern lediglich eine Referenz auf Zugangsdaten anhand einer ID. Somit müssen auf dem Server Zugangsdaten hinterlegt werden, welche die gleiche ID haben wie in den Job-Definitionen angegeben. Jobs können sich Zugangsdaten teilen, sodass dieser Schritt nicht für jede Aufgabe erneut durchgeführt werden muss.

Zugangsdaten können per Skript `2_add_credentials.sh` hinzugefügt werden. Es werden drei Parameter erwartet: Benutzername, Passwort und die Zugangsdaten-ID.

Eine Job-Definition beinhaltet beispielsweise folgende Zeilen:

```
<hudson.plugins.git.UserRemoteConfig>  
  <url>https://harry7e@bitbucket.org/harry7e/automatisierte-auswertung-von-progra  
mmier-praktika.git</url>  
  <credentialsId>bitbucket</credentialsId>  
</hudson.plugins.git.UserRemoteConfig>
```

Dann könnte das Hinzufügen der Zugangsdaten zu diesem Repository wie folgt aussehen:

```
./2_add_credentials.sh myBitbucketUsername myBitbucketPassword bitbucket
```

Jobs erzeugen

Das Skript *3_add_musterloesungen_jobs.sh* ist für die Erstellung der Jobs für die Musterlösungen verantwortlich. Hierbei werden für jede Zeile der Datei *musterloesungen_jobs.csv* Jobs erzeugt. Pro Zeile wird hier ein Präfix für den Namen des Jobs sowie eine Pfadangabe zur Job-Definition erwartet. Der Inhalt der Datei könnte wie folgt aussehen:

```
05_6;../ML-Aufgabe-05_6/jenkins_job_ml.xml
07_1;../ML-Aufgabe-07_1/jenkins_job_ml.xml
10_1;../ML-Aufgabe-10_1/jenkins_job_ml.xml
13_2;../ML-Aufgabe-13_2/jenkins_job_ml.xml
```

Nachdem die gewünschten Einträge in der Datei *musterloesungen_jobs.csv* vorgenommen wurden, wird das Skript ausgeführt. Das Skript erwartet keine Parameter.

```
./3_add_musterloesungen_jobs.sh
```

Analog dazu werden mittels *4_add_students_with_jobs.sh* die Jobs der Studierenden erzeugt. Hierbei wird für alle E-Mail-Adressen in der Datei *students.txt* ein Benutzerkonto erzeugt, sofern es noch nicht existiert. Als Benutzername wird die Zeichenkette vor dem „@“-Zeichen der E-Mail-Adresse verwendet. Es wird ein initiales Passwort generiert und dem Benutzer per E-Mail geschickt. Anschließend wird pro Studierenden für jede Zeile in der Datei *student_jobs.csv* ein Job erzeugt. Pro Zeile wird hier ebenfalls ein Präfix für den Namen des Jobs sowie eine Pfadangabe zur Job-Definition erwartet. Ein Eintrag in der Datei könnte wie folgt aussehen:

```
05_6;../ML-Aufgabe-05_6/jenkins_job_student.xml
07_1;../ML-Aufgabe-07_1/jenkins_job_student.xml
10_1;../ML-Aufgabe-10_1/jenkins_job_student.xml
13_2;../ML-Aufgabe-13_2/jenkins_job_student.xml
```

Nachdem die gewünschten Einträge in den Dateien *students.txt* und *student_jobs.csv* vorgenommen wurden, wird das Skript ausgeführt. Das Skript erwartet keine Parameter.

```
./4_add_students_with_jobs.sh
```

Die Erzeugung der Jobs für den ersten Studierenden dauert dabei etwas länger, weil für diesen die Job-Definitionen einzeln importiert werden. Für alle weiteren Studierenden wird der Job des ersten Studierenden kopiert und lediglich der Name sowie die Berechtigungen angepasst, was eine wesentlich geringere Laufzeit hat.

Leistungsübersicht ansehen

Nach der Anmeldung am System wird die Jobübersicht angezeigt. Als Standard-Ansicht ist „Gruppierung nach Aufgabe“ eingestellt.

Alle Jobs		Gruppierung nach Aufgabe	Gruppierung nach Person				
S	W	Categorized - Job ↓	Testergebnis	Zeilenabdeckung	# Checkstyle	Buildbeschreibung	Letzter Erfolg
+	🔴	Aufgabe 05_6	N/A				2 Tage 0 Stunden - #7
	🔴	05_6 - erika.musterfrau	N/A		N/A	-	Nicht anwendbar
	🔴	05_6 - hello.world	N/A		N/A	-	Nicht anwendbar
	🔴	05_6 - lorem.ipsum	N/A		N/A	-	Nicht anwendbar
	🔴	05_6 - max.mustermann	N/A		N/A	-	Nicht anwendbar
	🟢	05_6 Muster	0 von 14 fehlgeschlagen (±0)	90.91%	0	Zyklomatische Komplexität: <=5	2 Tage 0 Stunden - #7
+	🔴	Aufgabe 07_1	N/A				2 Tage 0 Stunden - #5
+	🔴	Aufgabe 10_1	N/A				2 Tage 1 Stunde - #5
+	🔴	Aufgabe 13_2	N/A				2 Tage 0 Stunden - #6

Abbildung 6-1: Gruppierung nach Aufgabe (Lehrende, Ausschnitt)

Über die Plus- und Minus-Symbole können Gruppen gezielt auf- und zugeklappt werden.

In der Ansicht „Gruppierung nach Person“ erfolgt die Gruppierung nach Studierenden. Die Musterlösungen werden in der Gruppe „Muster“ aufgelistet.

Alle Jobs		Gruppierung nach Aufgabe	Gruppierung nach Person				
S	W	Categorized - Job ↓	Testergebnis	Zeilenabdeckung	# Checkstyle	Buildbeschreibung	Letzter Erfolg
+	🔴	erika.musterfrau	N/A				Nicht anwendbar
+	🔴	hello.world	N/A				Nicht anwendbar
+	🔴	lorem.ipsum	N/A				Nicht anwendbar
+	🔴	max.mustermann	N/A				Nicht anwendbar
+	🟢	Muster	N/A				2 Tage 0 Stunden - #6
	🟢	05_6 Muster	0 von 14 fehlgeschlagen (±0)	90.91%	0	Zyklomatische Komplexität: <=5	2 Tage 0 Stunden - #7
	🟢	07_1 Muster	0 von 6 fehlgeschlagen (±0)	85.71%	0	Zyklomatische Komplexität: <=5	2 Tage 1 Stunde - #5
	🟢	10_1 Muster	0 von 6 fehlgeschlagen (±0)	95.0%	0	Zyklomatische Komplexität: <=5	2 Tage 1 Stunde - #5
	🟢	13_2 Muster	0 von 3 fehlgeschlagen (±0)	88.24%	0	Zyklomatische Komplexität: <=5	2 Tage 0 Stunden - #6

Abbildung 6-2: Gruppierung nach Person (Lehrende, Ausschnitt)

In der Ansicht „Alle Jobs“ erfolgt keinerlei Gruppierung. In den drei Ansichten werden das Testergebnis, die Zeilenabdeckung, die Anzahl der Checkstyle-Warnungen sowie die Komplexität übersichtlich dargestellt.

Bei fehlerhaften Builds enthält die Buildbeschreibung Hinweise zur Fehlerursache, wie zum Beispiel „fehlender Parameter“ oder „Kompilierfehler – Benennung prüfen“. Bei erfolgreichen Builds gibt die Buildbeschreibung die zyklomatische Komplexität der komplexesten Methode an. Ist diese höher als der per Checkstyle-Regel geforderte Maximalwert, wird dieser Wert exakt angegeben, ansonsten lediglich das dieser kleiner gleich dem geforderten Maximalwert ist.

Lösungen der Studierenden einsehen

Die Lösungen der Studierenden werden per Job-Parameter übergeben. Durch Auswahl eines Jobs und Builds können die Parameter betrachtet werden.

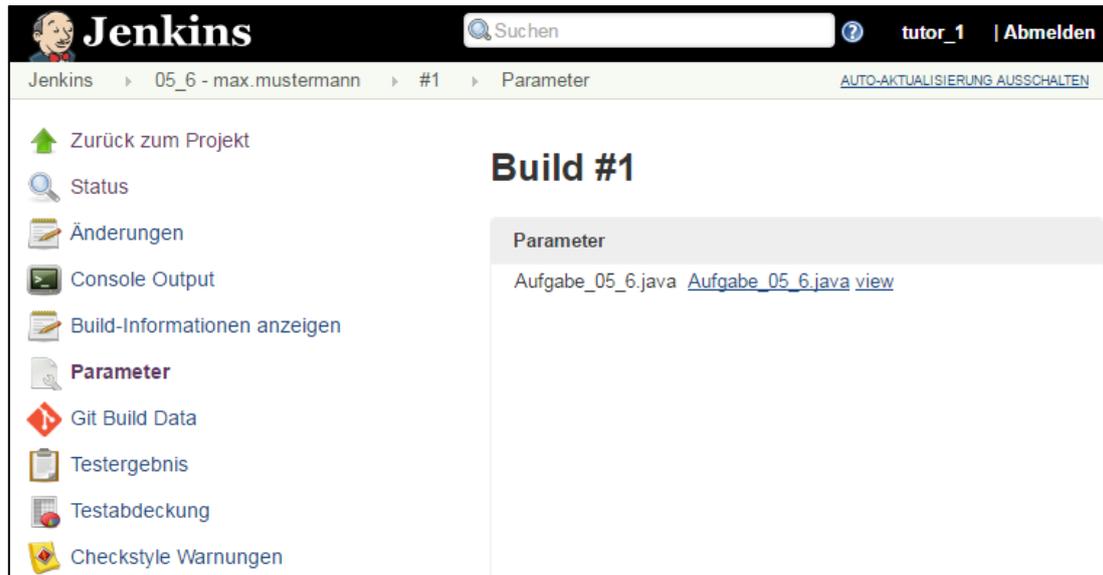


Abbildung 6-3: Parameter einsehen

Jobs aktivieren oder deaktivieren

Jobs können aktiviert und deaktiviert werden. Nur aktive Jobs können gestartet werden. Durch die Deaktivierung kann beispielsweise eine studentische Abgabe nach einer Frist verhindert werden.

Die Aktivierung und Deaktivierung erfolgt mittels Skript `5_set_jobs_enabled.sh`. Das Skript erwartet zwei Parameter. Der erste Parameter dient zur Filterung der Jobs. Es werden nur Jobs bearbeitet, deren Titel die übergebene Zeichenkette beinhalten. Der zweite Parameter gibt an, ob die Jobs aktiviert oder deaktiviert werden. Hierbei sorgt die Eingabe „true“ (unabhängig von Groß- und Kleinschreibung) für eine Aktivierung und alle anderen Eingaben für eine Deaktivierung der gefilterten Jobs.

Der folgende beispielhafte Aufruf sorgt dafür, dass alle Jobs zur Aufgabe 5.6 deaktiviert werden:

```
./5_set_jobs_enabled.sh 05_6 false
```

Es ist auch denkbar, diesen Befehl mithilfe des Linux-Kommandos „at“ zu einem bestimmten Zeitpunkt auszuführen. Informationen zur Verwendung des Kommandos befinden sich unter <http://linux.die.net/man/1/at>.

6.3 Dokumentation für Studierende

Voraussetzungen

- Die URL zum Jenkins-Server
- Ein Jenkins-Benutzerkonto. Die Zugangsdaten wurden per E-Mail geschickt.

Leistungsübersicht ansehen

Nach der Anmeldung am System wird die Jobübersicht angezeigt. Als Standard-Ansicht ist „Gruppierung nach Aufgabe“ eingestellt. Diese Ansicht eignet sich für den Vergleich der eigenen Lösung mit der Musterlösung.

Alle Jobs		Gruppierung nach Aufgabe	Gruppierung nach Person				
S	W	Categorized - Job ↓	Testergebnis	Zeilenabdeckung	# Checkstyle	Buildbeschreibung	Letzter Erfolg
+	🔴	Aufgabe 05_6	N/A				2 Tage 1 Stunde - #7
		<ul style="list-style-type: none"> 🔴 05_6 - max.mustermann N/A N/A - Nicht anwendbar 🟢 05_6 Muster 0 von 14 fehlgeschlagen (±0) 90.91% 0 Zyklomatische Komplexität: <=5 2 Tage 1 Stunde - #7 					
+	🔴	Aufgabe 07_1	N/A				2 Tage 1 Stunde - #5
+	🔴	Aufgabe 10_1	N/A				2 Tage 1 Stunde - #5
+	🔴	Aufgabe 13_2	N/A				2 Tage 0 Stunden - #6

Abbildung 6-4: Gruppierung nach Aufgabe (Studierende, Ausschnitt)

Über die Plus- und Minus-Symbole können Gruppen gezielt auf- und zugeklappt werden.

In der Ansicht „Gruppierung nach Person“ werden alle eigenen Aufgaben in einer Gruppe angezeigt. Die Musterlösungen werden in der Gruppe „Muster“ aufgelistet. Diese Ansicht eignet sich für den aufgabenübergreifenden Überblick der eigenen Leistungen.

Alle Jobs		Gruppierung nach Aufgabe	Gruppierung nach Person				
S	W	Categorized - Job ↓	Testergebnis	Zeilenabdeckung	# Checkstyle	Buildbeschreibung	Letzter Erfolg
+	🔴	max.mustermann	N/A				Nicht anwendbar
		<ul style="list-style-type: none"> 🔴 05_6 - max.mustermann N/A N/A - Nicht anwendbar 🔴 07_1 - max.mustermann N/A N/A - Nicht anwendbar 🔴 10_1 - max.mustermann N/A N/A - Nicht anwendbar 🔴 13_2 - max.mustermann N/A N/A - Nicht anwendbar 					
+	🟢	Muster	N/A				2 Tage 0 Stunden - #6

Abbildung 6-5: Gruppierung nach Person (Studierende, Ausschnitt)

In der Ansicht „Alle Jobs“ erfolgt keinerlei Gruppierung. In den drei Ansichten werden das Testergebnis, die Zeilenabdeckung, die Anzahl der Checkstyle-Warnungen sowie die Komplexität übersichtlich dargestellt.

Bei fehlerhaften Builds enthält die Buildbeschreibung Hinweise zur Fehlerursache, wie zum Beispiel „fehlender Parameter“ oder „Kompilierfehler – Benennung prüfen“. Bei erfolgreichen Builds gibt die Buildbeschreibung die zyklomatische Komplexität der komplexesten Methode an. Ist diese höher als der per Checkstyle-Regel geforderte Maximalwert, wird dieser Wert exakt angegeben, ansonsten lediglich das dieser kleiner gleich dem geforderten Maximalwert ist.

Abgabe einer Lösung

Durch die Auswahl eines Jobs (zum Beispiel „05_6 – max.mustermann“ für die Aufgabe 5.6) und „Bauen mit Parametern“ erfolgt die Aufforderung zum Hochladen der eigenen Lösungsdateien.

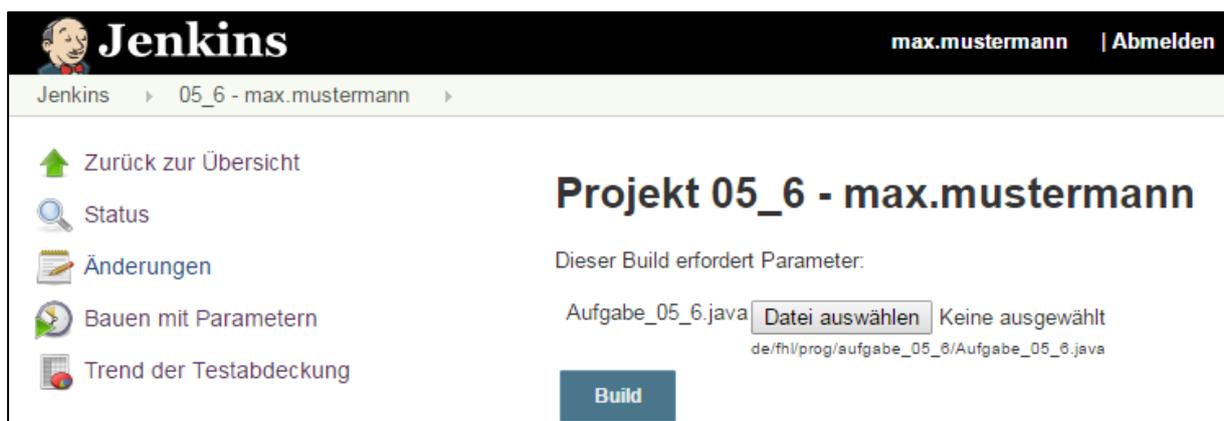


Abbildung 6-6: Abgabe eines Studierenden

Mittels „Build“-Schaltfläche wird die Prüfung gestartet.

Wird keine Datei angegeben, beinhaltet das Buildergebnis die Meldung „fehlender Parameter“. Schlägt das Kompilieren fehl, wird "Kompilierfehler - Benennung prüfen" angezeigt. Dieser Fehler tritt auf, wenn sich nicht exakt an die vom Lehrenden vorgegebene Benennungen, Parameter- und Rückgabetypen gehalten wird. Dies ist essenziell, um die Lösung automatisiert mittels JUnit-Tests auf Korrektheit prüfen zu können.

Ist der Prüfdurchlauf erfolgreich abgeschlossen, können die Auswertungsergebnisse eingesehen werden. Durch Auswahl des konkreten Prüflaufes wird eine Übersicht angezeigt.

The screenshot shows the Jenkins web interface for a specific build. The main content area displays the following information:

- Build #3 (07.07.2016 11:49:48)**: Started 2 minutes and 2 seconds ago, duration 26 seconds.
- Zyklomatische Komplexität: 8**: Cyclomatic complexity of the build.
- No changes**: Indicated by a notepad icon.
- Gestartet durch Benutzer max.mustermann**: Started by user max.mustermann.
- Revision: 0429b1c903d29176a12ad9c9d1b5661a8c5a3025**: The specific commit hash.
- git**: The version control system used.
- Testergebnis (4 fehlgeschlagene Tests / -10)**: Test results showing 4 failed tests out of 10. Links are provided for each test:
 - [de.fhl.prog.test.TestSchaltjahr.test\[2016=true\]](#)
 - [de.fhl.prog.test.TestSchaltjahr.test\[1600=true\]](#)
 - [de.fhl.prog.test.TestSchaltjahr.test\[2000=true\]](#)
 - [de.fhl.prog.test.TestSchaltjahr.test\[2012=true\]](#)
- Jacoco - Overall Coverage Summary**: A table showing coverage percentages for different code elements.

INSTRUCTION	88%
BRANCH	82%
COMPLEXITY	69%
LINE	85%
METHOD	50%
CLASS	100%
- Checkstyle: 37 Warnungen**: 37 warnings from a static code analysis.

Abbildung 6-7: Build-Ergebnis eines Studierenden – Übersicht

Die Komplexität wird als Buildbeschreibung gesetzt und ist dadurch direkt im oberen Bereich aufgeführt. Eine Erläuterung der zyklomatischen Komplexität befindet sich unter <http://www.guru99.com/cyclomatic-complexity.html>.

Über die weiterführenden Links „Testergebnis“, „Testabdeckung“ und „Checkstyle Warnungen“ sind jeweils detaillierte Informationen zu den einzelnen Bereichen abrufbar.

Das Testergebnis macht eine Aussage über die Korrektheit der Lösung. Hierbei werden vom Lehrenden definierte Tests auf die Lösung angewendet. Zu Testfehlschlägen sind ebenfalls Details einsehbar, zum Beispiel welche Prüfung für den Fehlschlag verantwortlich ist oder wo eine Exception aufgetreten ist.

Jenkins Suchen admin | Abmelden

Jenkins > 05_6 - erika.musterfrau > #3 > Testergebnis [AUTO-AKTUALISIERUNG EINSCHALTEN](#)

Zurück zum Projekt
 Status
 Änderungen
 Console Output
 Build-Informationen editieren
 Verlauf
 Parameter
 Git Build Data
 No Tags
Testergebnis
 Testabdeckung
 Checkstyle Warnungen
 Vorheriger Build

Testergebnisse

Fehlschläge Tests
Dauer: 0.19 Sekunden
[Beschreibung hinzufügen](#)

Alle fehlgeschlagenen Tests

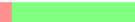
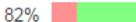
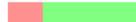
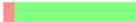
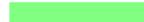
Testname	Dauer	Alter
de.fhl.prog.test.TestSchaltjahr.test[2016=true]		
Fehlerdetails expected:<...szahl ein: 2016 ist [ein] Schaltjahr. > but was:<...szahl ein: 2016 ist [] Schaltjahr. >	30 ms	1
de.fhl.prog.test.TestSchaltjahr.test[1600=true]	1 ms	1
de.fhl.prog.test.TestSchaltjahr.test[2000=true]	4 ms	1
de.fhl.prog.test.TestSchaltjahr.test[2012=true]	1 ms	1

Alle Tests

Package	Dauer	Fehlgeschlagen (Diff.)	Übersprungen (Diff.)	Pass (Diff.)	Summe (Diff.)
de.fhl.prog.test	67 ms	4 +4	0	10 +10	14 +14

Abbildung 6-8: Darstellung von JUnit-Testergebnissen

Die Testabdeckung gibt an, welche Teile des Quelltextes dabei durchlaufen werden, wodurch überflüssige Stellen identifiziert werden können. Es werden die absolute Anzahl und der prozentuale Anteil abgedeckter Anweisungen angegeben. Zusätzlich kann der Quelltext inklusive farblicher Markierung der Abdeckung angezeigt werden. Per Tooltip können weitere Hinweise angezeigt werden, zum Beispiel wie viele der möglichen Pfade durchlaufen wurden. Für die Musterlösungen ist die Anzeige des Quelltextes deaktiviert.

Aufgabe_05_6					
Name	instruction	branch	complexity	Zeilen	Methoden
Aufgabe_05_6()	M: 3 C: 0 0% 	M: 0 C: 0 0%	M: 1 C: 0 0% 	M: 1 C: 0 0% 	M: 1 C: 0 0% 
main(String[])	M: 7 C: 72 91% 	M: 4 C: 18 82% 	M: 3 C: 9 75% 	M: 1 C: 11 92% 	M: 0 C: 1 100% 

Coverage

```

1: package de.fhl.prog.aufgabe_05_6;
2:
3: import java.util.Scanner;
4:
5: /**
6:  * @author Max Mustermann
7:  */
8: public abstract class Aufgabe_05_6 {
9:
10:     /**
11:      * Hauptprogramm.
12:      * @param args Kommandozeilenparameter
13:      */
14:     public static void main(String[] args) {
15:         if(args != null && args.length > 0){
16:             throw new IllegalArgumentException("3 of 4 branches missed.");
17:         }
18:         Scanner in = new Scanner(System.in);
19:         System.out.print("Bitte geben sie eine beliebige Jahreszahl ein: ");
20:         int jahr = in.nextInt();
21:
22:         boolean durch4Teilbar = jahr % 4 == 0;
23:         boolean durch100Teilbar = jahr % 100 == 0;
24:         boolean durch400Teilbar = jahr % 400 == 0;
25:
26:         boolean schaltjahr = durch4Teilbar && !durch100Teilbar
27:             || durch4Teilbar && durch100Teilbar && durch400Teilbar;
28:
29:         System.out.println(jahr + " ist " + (schaltjahr ? "" : "kein") + " Schaltjahr.");
30:
31:         in.close();
32:     }
33: }

```

Abbildung 6-9: Darstellung der Code-Coverage (Ausschnitt)

Die Checkstyle Warnungen zeigen auf, wo gegen Programmierkonventionen verstoßen wurde. Es wird ein leicht modifizierter Google-Regelsatz angewendet, welcher unter <http://www.nkcode.io/assets/programming/eclipse/programming-freshman-checkstyle-conventions.xml> einsehbar ist. Eine Konfigurationsdatei für den Eclipse-Formatter zur automatischen Formatierung des Quelltextes im „Java-Google-Style“ ist verfügbar unter <https://github.com/google/styleguide>.

An der Oberfläche wird zunächst eine Übersicht der aufgetretenen Warnungen angezeigt.

CheckStyle Ergebnis

Vergleich mit letzter Analyse

Alle Warnungen	Neue Warnungen	Behobene Warnungen
37	37	0

Zusammenfassung

Gesamt	Hohe Priorität	Normale Priorität	Niedrige Priorität
37	0	37	0

Details

Kategorie	Gesamt	Verteilung
Indentation	14	<div style="width: 38%;"></div>
Javadoc	1	<div style="width: 3%;"></div>
Metrics	1	<div style="width: 3%;"></div>
Whitespace	21	<div style="width: 57%;"></div>
Gesamt	37	

Abbildung 6-10: Darstellung der Checkstyle-Warnungen

Durch weiterführende Links sind zudem detaillierte Informationen einsehbar, zum Beispiel welche Zeile die Warnung verursacht sowie Erklärungen zu den Regeln und Warnungen.

Details

[Aufgabe_05_6.java:14](#), CyclomaticComplexityCheck, Priorität: Normal

Cyclomatic Complexity is 8 (max allowed is 5).

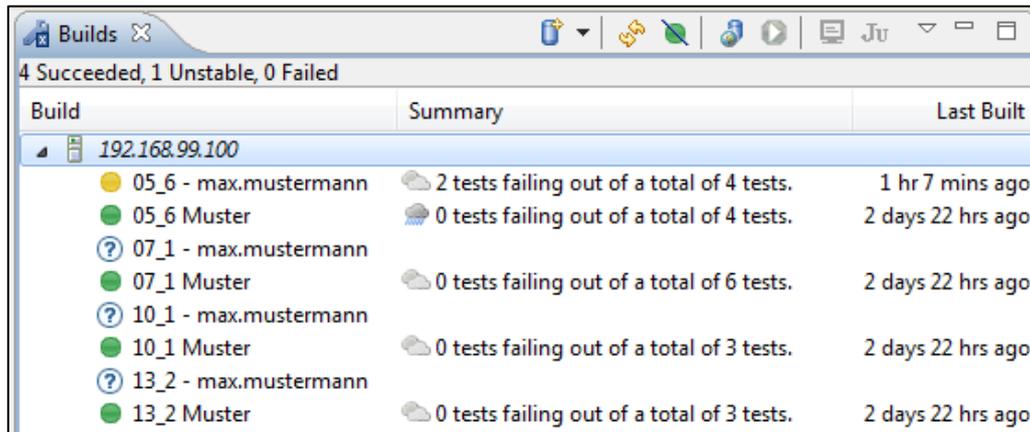
Checks cyclomatic complexity against a specified limit. It is a measure of the minimum number of possible paths through the source and therefore the number of required tests, it is not a about quality of code! It is only applied to methods, c-tors, [static initializers](#) and [instance initializers](#). The complexity is equal to the number of decision points + 1 Decision points: `if`, `while`, `do`, `for`, `?:`, `catch`, `switch`, `case` statements, and operators `&&` and `||` in the body of target. By pure theory level 1-4 is considered easy to test, 5-7 OK, 8-10 consider re-factoring to ease testing, and 11+ re-factor now as testing will be painful. When it comes to code quality measurement by this metric level 10 is very good level as a ultimate target (that is hard to archive). Do not be ashamed to have complexity level 15 or even higher, but keep it below 20 to catch really bad designed code automatically. Please use Suppression to avoid violations on cases that could not be split in few methods without damaging readability of code or encapsulation.

Abbildung 6-11: Darstellung der Checkstyle-Details (Ausschnitt)

Die Prüfung der eigenen Lösung kann beliebig oft wiederholt werden, somit können die Detail-Informationen ideal zur Verbesserung genutzt werden. Die Lösung des letzten durchgeführten Prüflaues wird als Abgabe gewertet. Nach Ablauf der durch die Lehrenden definierte Abgabefrist können Jobs deaktiviert werden, sodass die Durchführung weiterer Prüfläufe nicht mehr möglich ist.

Verwendung des Eclipse-Plugins

Durch Verwendung des Eclipse-Plugins „Mylyn Builds Connector: Hudson/Jenkins“ (<https://marketplace.eclipse.org/content/hudsonjenkins-myllyn-builds-connector>) kann die Anzahl fehlschlagender Tests direkt in der Eclipse-View „Builds“ angesehen und per Kontextmenü direkt zum detaillierteren Ergebnis im Browser gesprungen werden.



Build	Summary	Last Built
192.168.99.100		
05_6 - max.mustermann	2 tests failing out of a total of 4 tests.	1 hr 7 mins ago
05_6 Muster	0 tests failing out of a total of 4 tests.	2 days 22 hrs ago
07_1 - max.mustermann	0 tests failing out of a total of 6 tests.	2 days 22 hrs ago
07_1 Muster	0 tests failing out of a total of 6 tests.	2 days 22 hrs ago
10_1 - max.mustermann	0 tests failing out of a total of 3 tests.	2 days 22 hrs ago
10_1 Muster	0 tests failing out of a total of 3 tests.	2 days 22 hrs ago
13_2 - max.mustermann	0 tests failing out of a total of 3 tests.	2 days 22 hrs ago
13_2 Muster	0 tests failing out of a total of 3 tests.	2 days 22 hrs ago

Abbildung 6-12: Build-Übersicht in Eclipse

Das Starten einer Prüfung direkt aus Eclipse ist nicht möglich, weil das Plugin Datei-Parameter aktuell nicht unterstützt, siehe https://bugs.eclipse.org/bugs/show_bug.cgi?id=407158.

Zur Einrichtung des Plugins müssen die eigenen Zugangsdaten sowie die URL zum Jenkins-Server angegeben werden.

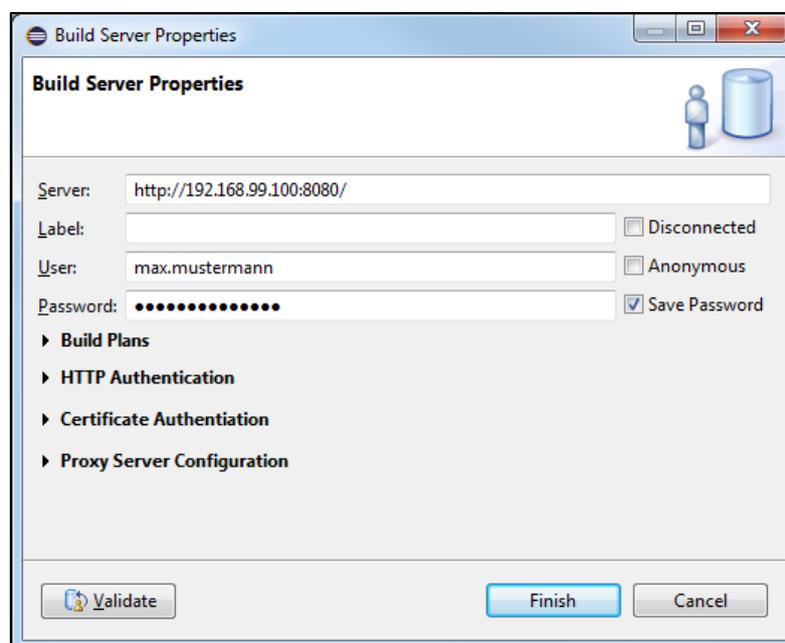


Abbildung 6-13: Plugin-Konfiguration in Eclipse

7 Test und Nachweisführung

In diesem Kapitel werden das Testen sowie der Erfüllungsgrad der Anforderungen an das System aus Kapitel 2 beschrieben. Nach der Übersicht folgt eine Erklärung zu den einzelnen Status-Werten.

#	Anforderung	Priorität	Status
1	Als Student/Lehrender möchte ich Lösungen zu Programmieraufgaben hinsichtlich Korrektheit, Code Coverage, Komplexität und Programmierkonventionen auswerten.	hoch	erfüllt
2	Als Student möchte ich die Auswertung direkt aus der Entwicklungsplattform Eclipse heraus anstoßen.	niedrig	nicht erfüllt
3	Als Student möchte ich Feedback in Relation zur Musterlösung erhalten.	hoch	erfüllt
4	Als Student möchte ich Feedback in Relation zu anderen Studierenden erhalten.	niedrig	nicht erfüllt
5	Als Student möchte ich eine Leistungsübersicht über alle meine Abgaben erhalten. Als Lehrender möchte ich eine Leistungsübersicht über alle Abgaben eines Studierenden erhalten.	hoch	erfüllt
6	Als Lehrender möchte ich eine Leistungsübersicht für eine Abgabe über alle Studierenden erhalten.	hoch	erfüllt
7	Als Lehrender möchte ich eine Leistungsübersicht über alle Abgaben aller Studierenden erhalten. (Semesterüberblick)	mittel	bedingt erfüllt
8	Als Lehrender möchte ich Aufgaben anhand eines Templates / Kochrezeptes für die automatisierte Auswertung vorbereiten.	hoch	erfüllt
9	Als Lehrender möchte ich das System komfortabel für eine Vielzahl von Studierenden einrichten.	hoch	erfüllt
10	Als Administrator möchte ich die Continuous Integration Lösung komfortabel als Docker-Container aufsetzen können.	hoch	erfüllt
11	Als Administrator möchte ich die Lösung anhand einer Dokumentation betreiben und konfigurieren.	hoch	erfüllt
12	Als Student/Lehrender möchte ich durch geeignete Dokumentation bei der Benutzung der Lösung unterstützt werden.	hoch	erfüllt

Tabelle 7-1: Anforderungserfüllung

Es wurde eine gezielte Anwender-Dokumentation für die Rollen Administrator, Lehrender und Studierender erstellt. Die Inhalte der Dokumentation wurden getestet, indem das System genau anhand der Dokumentation verwendet wurde. So wurden nur genau die genannten Befehle verwendet (Ausnahme: tatsächliche Passwörter statt Platzhalter), um beispielsweise den Server als Docker-Container aufzusetzen. Die Dokumentation der Lehrenden erklärt zudem ausführlich, wie Aufgaben für die automatisierte Auswertung vorbereitet werden. Damit werden die Anforderungen 8, 10, 11 und 12 als erfüllt betrachtet.

Anhand des Skriptes `4_add_students_with_jobs.sh` wird das System komfortabel für eine Vielzahl von Studierenden eingerichtet. Getestet wurde dieses Skript mit 100 Studierenden und 32 Aufgaben, wobei die vier umgesetzten Aufgaben jeweils acht Mal mit unterschiedlichem Präfix genutzt wurden. Die Messung der Laufzeit erfolgte händisch mittels Stoppuhr. Die Erzeugung der 100 Benutzerkonten ohne Mailversand dauerte circa 14 Sekunden. Der Mailversand wurde separat in geringerem Umfang gemessen und dauerte pro Mail zwischen 600 und 800 Millisekunden. Diese Laufzeit wurde mittels „`System.currentTimeMillis()`“ direkt im Groovy-Skript gemessen. Das Anlegen der ersten 32 Jobs dauerte rund 32 Sekunden. Für diese Jobs werden die Jobdefinitionen per CLI importiert. Die restlichen 3168 Jobs werden durch Kopieren der zuvor erzeugten Jobs erstellt. Dies dauerte lediglich 26 Sekunden. Die Gesamtlaufzeit des Skriptes ohne Mailversand betrug 72 Sekunden. Inklusive Mailversand ist eine Laufzeit von circa 142 Sekunden zu erwarten. Das vollständige Laden der Seite dauert einen kleinen Moment länger als mit wenigen Jobs, aber die Oberfläche lässt sich weiterhin gut bedienen. Damit wird die Anforderung 9 als erfüllt betrachtet.

Mit dem System können Lösungen hinsichtlich Korrektheit, Code-Coverage, Komplexität und Programmierkonventionen ausgewertet werden. Das Starten eines Prüflaufes ohne Angabe einer Lösung per Parameter oder die Nichteinhaltung der Benennungsregeln resultierten in sinnvollen Meldungen. Die Musterlösungen werden ebenfalls ausgewertet und die Ergebnisse sind durch die Studierenden zum Vergleich einsehbar. Mithilfe der nach Aufgabe oder Person gruppierten Ansichten erhalten Studierende und Lehrende eine Leistungsübersicht über mehrere Abgaben. Die tabellarischen Ansichten beinhalten Spalten zu den vier geforderten Auswertungen. Damit werden die Anforderungen 1, 3, 5 und 6 als erfüllt betrachtet.

Neben den gruppierten Ansichten gibt es auch eine Ansicht aller Jobs, welche die gleichen Tabellenspalten beinhaltet. Dies ist zwar eine Art Semesterüberblick, während die Anzahl der angezeigten Jobs durch die Gruppierung jedoch überschaubar gehalten wird, beinhaltet die Ansicht aller Jobs im genannten Testszenario über 3000 Zeilen. Die Anforderung 7 wird daher als nur bedingt erfüllt betrachtet.

Studierende dürfen die Lösungen der anderen Studierenden nicht einsehen. Es kann nicht genauer über Berechtigungen gesteuert werden, ob Job-Parameter eingesehen werden dürfen. Dadurch wurden eine Art der Jobeinrichtung und des Rechtemanagements gewählt, die Studierenden lediglich die Einsicht in die eigenen Jobs sowie die Auswertungsergebnisse der Musterlösungen erlaubt. Dadurch ist die Anforderung 4 nicht erfüllt.

Über das Eclipse-Plugin können zwar Testergebnisse angezeigt werden, weil Datei-Parameter aber nicht unterstützt werden, ist das Anstoßen der Auswertung nicht möglich. Somit ist die Anforderung 2 ebenfalls nicht erfüllt.

8 Fazit und Ausblick

Programmieraufgaben können mithilfe einer CI-Lösung automatisiert ausgewertet werden. Es wurden vier Aufgaben automatisiert und die Machbarkeit für drei weitere Aufgaben analysiert. Außerdem wurden alle hoch priorisierten Anforderungen an das System erfüllt. Dadurch lässt sich zusammenfassen, dass das Ziel der Arbeit erreicht wurde.

Das System kann grundsätzlich sofort eingesetzt werden, wobei natürlich noch weitere Aufgaben automatisiert werden sollten. Es gibt zudem noch Potential für mögliche Erweiterungen, welche während der Bearbeitung identifiziert wurden. So würde beispielsweise ein Einrichtungstool mit grafischer Benutzeroberfläche die Konfiguration des Systems vereinfachen. Dieses Tool könnte intern weiterhin die vorbereiteten Skripte aufrufen, die Angabe der Parameter jedoch deutlich komfortabler machen. Es ist ebenfalls wünschenswert, dass die Einrichtung des Administrator-Benutzerkontos sowie des Mailservers bereits im Docker-Image enthalten sind und nicht erst nachträglich erfolgen. Dadurch würde die Erstellung mehrerer Container-Instanzen vereinfacht werden. Ferner bietet Jenkins keinen Mechanismus, um das Passwort eines Benutzerkontos zurückzusetzen, falls dieses vergessen wurde. Sollte dieser Fall eintreten, müsste der Administrator manuell ein neues Passwort vergeben. Die erzeugten E-Mails mit den Zugangsdaten enthalten aktuell nur den Benutzernamen und ein generiertes Passwort. Die zusätzliche Angabe der Server-URL würde den erstmaligen Aufruf vereinfachen. Der Server kann seine eigene URL jedoch nicht zuverlässig ermitteln, sodass hierfür zusätzlicher Konfigurationsaufwand entsteht. Darüber hinaus bietet Jenkins noch weitere Funktionalität, die bisher nicht genutzt wird. So ist es beispielsweise denkbar, Quelltexte in anderen Programmiersprachen auszuwerten, was sicherlich für andere Lehrveranstaltungen interessant ist. Für die Programmiersprachen C und C++ werden beispielsweise direkt auf der Jenkins-Hauptseite fünf Plugins zur Unterstützung aufgelistet. [JeC16]

9 Danksagungen

Ich danke Herrn Prof. Dr. Kratzke für die interessante und fordernde Aufgabenstellung sowie für die gute Betreuung. Termine konnten auch kurzfristig vereinbart werden und ich habe stets hilfreiches Feedback zu meinen Zwischenergebnissen erhalten.

Ebenfalls danke ich meinem Vater Günter Hamann sowie meinen Kommilitonen Max Golubew und Florian Richter für das Korrekturlesen der Arbeit.

Anhang A – Aufgabenstellungen und Quelltexte

A.1 Schaltjahre bestimmen (Aufgabe 5.6)

A.1.1 Bisherige Aufgabenstellung

Sie sollen nun ein Programm zur Prüfung von Schaltjahren (Gregorianischer Kalender) entwickeln. Ein typischer Programmablauf kann wie folgt aussehen:

*Bitte geben sie eine beliebige Jahreszahl ein: 2016
2016 ist ein Schaltjahr.*

Wikipedia hilft ihnen weiter, wenn Sie nicht wissen, wie man ein Schaltjahr im Gregorianischen Kalender bestimmt. <https://de.wikipedia.org/wiki/Schaltjahr>

A.1.2 Erweiterung zu der bisherigen Aufgabenstellung

Es werden folgende Klassen und Methoden erwartet:

- de.fhl.prog.aufgabe_05_6.Aufgabe_05_6
 - public static void main(String[] args)

Innerhalb der main-Methode soll eine Zahl von der Standardeingabe gelesen werden. Wenn es sich bei der eingegebenen Jahreszahl x um ein Schaltjahr handelt, soll der Text „x ist ein Schaltjahr.“ in die Standardausgabe geschrieben werden, andernfalls „x ist kein Schaltjahr.“. (x ist durch die jeweils eingegebene Zahl zu ersetzen)

A.1.3 Quelltext der Tests

```
package de.fhl.prog.test;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

import org.junit.After;
import org.junit.Before;

/**
 * Abstrakte Basisklasse zum Testen mit Konsoleninteraktion.
 *
 * @see #writeToConsole(String)
 * @see #readFromConsole()
 *
 * @author Timo Hamann
 */
public abstract class AbstractConsoleTest {

    private final ByteArrayOutputStream outByteStream = new
    ByteArrayOutputStream();

    @Before
    public void setUp() {
        System.setOut(new PrintStream(outByteStream));
    }

    @After
    public void cleanUp() {
        System.setIn(null);
        System.setOut(null);
    }

    /**
     * Schreibt einen Text in die Konsoleneingabe.
     *
     * @param text Text
     */
    protected void writeToConsole(String text) {
        System.setIn(new ByteArrayInputStream(text.getBytes()));
    }

    /**
```

```
* Liest einen Text von der Konsolenausgabe.
*
* @return Text
*/
protected String readFromConsole() {
    return new String(outByteArray.toByteArray());
}

}

package de.fhl.prog.test;

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameter;
import org.junit.runners.Parameterized.Parameters;

import de.fhl.prog.aufgabe_05_6.Aufgabe_05_6;

/**
 * Test zur Aufgabe 5.6
 *
 * @author Timo Hamann
 */
@RunWith(value = Parameterized.class)
public class TestSchaltjahr extends AbstractConsoleTest {

    @Parameter
    public Entry<Integer, Boolean> parameter;

    /**
     * @return Testdaten des {@link Parameterized parametrisierten} Tests
     * @see #test()
     */
    @Parameters(name = "{0}")
    public static Set<Entry<Integer, Boolean>> data() {
```

```
Map<Integer, Boolean> data = new HashMap<>();
data.put(2011, false);
data.put(2012, true);
data.put(2013, false);
data.put(2014, false);
data.put(2015, false);
data.put(2016, true);
data.put(2017, false);

data.put(1500, false);
data.put(1600, true);
data.put(1700, false);
data.put(1800, false);
data.put(1900, false);
data.put(2000, true);
data.put(2100, false);
return data.entrySet();
}

/**
 * Testet einen Testparameter.
 *
 * @see #parameter
 */
@Test
public void test() {
    int year = parameter.getKey();
    boolean expectedResult = parameter.getValue();

    writeToConsole(String.valueOf(year));
    Aufgabe_05_6.main(null);
    String consoleOutput = readFromConsole();
    StringBuilder expectedStringBuilder = new StringBuilder();
    expectedStringBuilder.append(year);
    expectedStringBuilder.append(" ist ");
    expectedStringBuilder.append(expectedResult ? "ein" : "kein");
    expectedStringBuilder.append(" Schaltjahr.");

    Assert.assertTrue(consoleOutput.trim().endsWith(expectedStringBuilder.toString()));
}
}
```

A.2 Multiplikationstabelle (Aufgabe 7.1)

A.2.1 Bisherige Aufgabenstellung

Diese Aufgabe ist eine parametrisierte Weiterentwicklung der Aufgabe 6.5. Schreiben Sie nun ein Programm, das zwei ganze Zahlen abfragt und aus diesen eine Multiplikationstabelle erstellt. Die erste der beiden Zahlen, darf dabei nicht größer als 15 und nicht kleiner als null sein, die zweite nicht kleiner als null. Dies soll vor Aufruf einer zu definierenden Methode zur Berechnung der Multiplikationstabelle abgefragt werden. Wird gegen eine dieser beiden Bedingungen verstoßen, soll das Programm mit einer Fehlermeldung terminieren.

Nachfolgend ein beispielhafter Programmablauf für korrekte Eingaben.

Geben Sie bitte eine Zahl i ein: 13

Geben Sie bitte eine Zahl j ein: 3

Die Multiplikationstabelle lautet:

```
1 2 3 4 5 6 7 8 9 10 11 12 13
2 4 6 8 10 12 14 16 18 20 22 24 26
3 6 9 12 15 18 21 24 27 30 33 36 39
```

Die Multiplikationstabelle soll durch folgendes Statement ausgegeben werden:

```
System.out.println(mtab(i, j));
```

Sie müssen also die in Aufgabe 6.5 implementierte Funktionalität in einer Methode `mtab` kapseln und aufrufen.

A.2.2 Erweiterung zu der bisherigen Aufgabenstellung

Es werden folgende Klassen und Methoden erwartet:

- `de.fhl.prog.aufgabe_07_1.Aufgabe_07_1`
 - `public static String mtab(int i,int j)`

Die Leerräume zwischen den Zahlen sollen weiterhin per Tabulator "\t" abgebildet werden.

A.2.3 Quelltext der Tests

```
package de.fhl.prog.tests;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import de.fhl.prog.aufgabe_07_1.Aufgabe_07_1;

/**
 * Test zur Aufgabe 7.1
 *
 * @author Nane Kratzke, Timo Hamann
 */
public class TestMtab {

    @Test
    public void testEinMalEins() {
        assertEquals("1", Aufgabe_07_1.mtab(1, 1));
    }

    @Test
    public void testEinZeilig() {
        assertEquals("1\t2\t3", Aufgabe_07_1.mtab(3, 1));
    }

    @Test
    public void testEinSpaltig() {
        assertEquals("1\n2", Aufgabe_07_1.mtab(1, 2));
    }

    @Test
    public void testLeer() {
        assertEquals("", Aufgabe_07_1.mtab(0, 0));
    }

    @Test
    public void testLeereSpalte() {
        assertEquals("", Aufgabe_07_1.mtab(1, 0));
    }

    @Test
    public void testLeereZeile() {
        assertEquals("", Aufgabe_07_1.mtab(0, 1));
    }
}
```

```
}

@Test
public void testNegativeParameter() {
    assertEquals("", Aufgabe_07_1.mtab(-1, -1));
}

@Test
public void test4x5() {
    assertEquals(
        "1\t2\t3\t4\n" + "2\t4\t6\t8\n" + "3\t6\t9\t12\n" + "4\t8\t12\t16\n"
+ "5\t10\t15\t20",
        Aufgabe_07_1.mtab(4, 5));
}

@Test
public void test13x3() {
    assertEquals("1\t2\t3\t4\t5\t6\t7\t8\t9\t10\t11\t12\t13\n"
        + "2\t4\t6\t8\t10\t12\t14\t16\t18\t20\t22\t24\t26\n"
        + "3\t6\t9\t12\t15\t18\t21\t24\t27\t30\t33\t36\t39",
        Aufgabe_07_1.mtab(13, 3));
}

}
```

A.3 Ausgaberroutine für Stack, List und Map (Aufgabe 10.1)

A.3.1 Bisherige Aufgabenstellung

Sie sollen nun eine überladene Methode schreiben, die für die abstrakten Datentypen Stack, List und Map eine gut lesbare Stringrepräsentation erzeugt. So soll für die folgenden Datenstrukturen

```
List v = new LinkedList(); for (int i = 1; i < 5; i++) v.add(i);
Stack s = new Stack(); for (int i = 1; i < 5; i++) s.add(i);
Map t = new HashMap(); for (int i = 65; i < 69; i++) t.put(i, (char)i);
// Die letzte erzeugt einen kleinen Teil der sogenannten ASCII Tabelle
```

die folgende Anweisungen folgende Ausgaben erzeugen:

```
System.out.println(collection_to_string(v));
[1, 2, 3, 4]
```

```
System.out.println(collection_to_string(s));
[4, 3, 2, 1]
```

```
System.out.println(collection_to_string(t));
[65 -> A, 66 -> B, 67 -> C, 68 -> D]
```

Implementieren Sie nun bitte die Methode `collection_to_string` und testen Sie diese mit den oben genannten Werten aus.

Hinweis: Beachten Sie bitte, dass bei der Datenstruktur Map die Reihenfolge der Key/Value Paare nicht notwendig spezifiziert ist. Sie müssen in der Methode `collection_to_string` daher nur sicherstellen, dass alle in der Map befindlichen Key-Value Paare ausgegeben werden. Die Reihenfolge der Ausgabe muss durch Sie nicht programmiertechnisch beeinflusst werden.

A.3.2 Erweiterung zu der bisherigen Aufgabenstellung

Es werden folgende Klassen und Methoden erwartet:

- `de.fhl.prog.aufgabe_10_1.Aufgabe_10_1`
 - `public static String collection_to_string(List list)`
 - `public static String collection_to_string(Stack stack)`
 - `public static String collection_to_string(Map map)`

Der Inhalt der Datenstruktur darf durch die Bildung der String-Repräsentation nicht verändert werden.

A.3.3 Quelltext der Tests

```
package de.fhl.prog.test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.Stack;
import java.util.TreeMap;

import org.junit.Assert;
import org.junit.Test;

import de.fhl.prog.aufgabe_10_1.Aufgabe_10_1;

/**
 * Test zur Aufgabe 10.1
 *
 * @author Timo Hamann
 */
public class TestAusgaberoutinen {

    @Test
    public void testEmptyList() {
        Assert.assertEquals("[]",
Aufgabe_10_1.collection_to_string(Collections.emptyList()));
    }

    @Test
    public void testList() {
        List<Integer> list = Arrays.asList(1, 4, 3, 2, 5);
        List<Integer> workingCopy = new ArrayList<>(list);

        Assert.assertEquals("[1, 4, 3, 2, 5]",
Aufgabe_10_1.collection_to_string(workingCopy));
        Assert.assertEquals("Der Inhalt der Datenstruktur wurde veraendert!",
list, workingCopy);
    }

    @Test
    public void testEmptyStack() {
        Assert.assertEquals("[]", Aufgabe_10_1.collection_to_string(new
Stack<>()));
    }
}
```

```
}

@Test
public void testStack() {
    Stack<Integer> stack = new Stack<>();
    stack.addAll(Arrays.asList(1, 4, 3, 2, 5));
    Stack<Integer> workingCopy = new Stack<>();
    workingCopy.addAll(stack);

    Assert.assertEquals("[5, 2, 3, 4, 1]",
Aufgabe_10_1.collection_to_string(workingCopy));
    Assert.assertEquals("Der Inhalt der Datenstruktur wurde veraendert!",
stack, workingCopy);
}

@Test
public void testEmptyMap() {
    Assert.assertEquals("[]",
Aufgabe_10_1.collection_to_string(Collections.emptyMap()));
}

@Test
public void testMap() {
    Map<Integer, Character> map = new TreeMap<>();
    for (int i = 65; i < 70; i++) {
        map.put(i, (char) i);
    }
    Map<Integer, Character> workingCopy = new TreeMap<>(map);

    Assert.assertEquals("[65 -> A, 66 -> B, 67 -> C, 68 -> D, 69 -> E]",
        Aufgabe_10_1.collection_to_string(workingCopy));
    Assert.assertEquals("Der Inhalt der Datenstruktur wurde veraendert!", map,
workingCopy);
}
}
```

A.4 Bestimmen von Baumeigenschaften (Aufgabe 13.2)

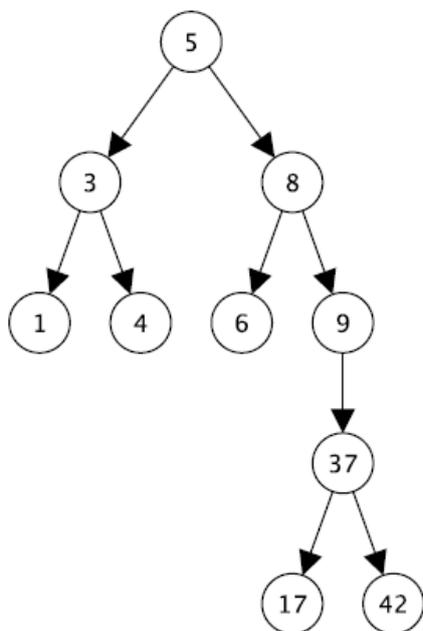
A.4.1 Bisherige Aufgabenstellung

Nutzen Sie für diese Aufgabe ihren Datentyp für einen rekursiven Binärbaum aus Aufgabe 13.1. Implementieren Sie nun folgende Methoden zur Bestimmung von Baumeigenschaften:

- `public static int countNodes(Node n)` zum Zählen von Knoten in einem Binärbaum
- `public static int countEdges(Node n)` zum Zählen von Kanten in einem Binärbaum
- `public static int height(Node n)` zum Bestimmen der Höhe eines Binärbaums

Testen Sie ihre Routinen anhand des folgenden Binärbaums.

Hinweis: Für die Methode `height` benötigen Sie vermutlich eine Maximumbestimmung. In JAVA existiert hierzu die Methode `Math.max(int v, int w)` die für zwei Werte den größeren zurückgibt.



A.4.2 Erweiterung zu der bisherigen Aufgabenstellung

Es werden folgende Klassen und Methoden erwartet:

- `de.fhl.prog.aufgabe_13_2.Aufgabe_13_2`
 - `public static int countNodes(Node n)`
 - `public static int countEdges(Node n)`
 - `public static int height(Node n)`
- `de.fhl.prog.aufgabe_13_1.Node`
 - `public Node(int wert, Node links, Node rechts)`

A.4.3 Quelltext der Tests

```
package de.fhl.prog.test;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import de.fhl.prog.aufgabe_13_1.Node;
import de.fhl.prog.aufgabe_13_2.Aufgabe_13_2;

/**
 * Test zur Aufgabe 13.2
 *
 * @author Timo Hamann
 */
public class TestBaumeigenschaften {

    private Node beispielbaum;

    @Before
    public void setUp() {
        beispielbaum = new Node(5, new Node(3, new Node(1, null, null), new Node(4,
null, null)),
            new Node(8, new Node(6, null, null),
                new Node(9, null, new Node(37, new Node(17, null, null), new
Node(42, null, null)))));
    }

    @Test
    public void testCountNodes_Beispielbaum() {
        Assert.assertEquals(10, Aufgabe_13_2.countNodes(beispielbaum));
    }

    @Test
    public void testCountEdges_Beispielbaum() {
        Assert.assertEquals(9, Aufgabe_13_2.countEdges(beispielbaum));
    }

    @Test
    public void testHeight_Beispielbaum() {
        Assert.assertEquals(5, Aufgabe_13_2.height(beispielbaum));
    }

    @Test
```

```
public void testCountNodes_EinzelnerKnoten() {
    Assert.assertEquals(1, Aufgabe_13_2.countNodes(new Node(42, null,
null)));
}

@Test
public void testCountEdges_EinzelnerKnoten() {
    Assert.assertEquals(0, Aufgabe_13_2.countEdges(new Node(42, null,
null)));
}

@Test
public void testHeight_EinzelnerKnoten() {
    Assert.assertEquals(1, Aufgabe_13_2.height(new Node(42, null, null)));
}

}
```

Abbildungsverzeichnis

Abbildung 3-1: Bestandteile eines CI-Systems [Hei12]	11
Abbildung 4-1: Komponenten-Übersicht.....	17
Abbildung 4-2: Ablauf des Buildprozesses.....	23
Abbildung 6-1: Gruppierung nach Aufgabe (Lehrende, Ausschnitt).....	45
Abbildung 6-2: Gruppierung nach Person (Lehrende, Ausschnitt).....	45
Abbildung 6-3: Parameter einsehen	46
Abbildung 6-4: Gruppierung nach Aufgabe (Studierende, Ausschnitt).....	47
Abbildung 6-5: Gruppierung nach Person (Studierende, Ausschnitt).....	47
Abbildung 6-6: Abgabe eines Studierenden.....	48
Abbildung 6-7: Build-Ergebnis eines Studierenden – Übersicht.....	49
Abbildung 6-8: Darstellung von JUnit-Testergebnissen	50
Abbildung 6-9: Darstellung der Code-Coverage (Ausschnitt).....	51
Abbildung 6-10: Darstellung der Checkstyle-Warnungen.....	52
Abbildung 6-11: Darstellung der Checkstyle-Details (Ausschnitt)	52
Abbildung 6-12: Build-Übersicht in Eclipse.....	54
Abbildung 6-13: Plugin-Konfiguration in Eclipse	54

Tabellenverzeichnis

Tabelle 2-1: Anforderungen	10
Tabelle 3-1: CI-Server Ranking (Umfrage) [InQ14]	13
Tabelle 3-2: Vergleich der Kandidaten	16
Tabelle 5-1: Schaltjahr-Testfälle	26
Tabelle 5-2: Multiplikationstabelle-Testfälle	27
Tabelle 7-1: Anforderungserfüllung	55

Literaturverzeichnis

- [bui16] (2016) buildbot.net. [Online]. <http://buildbot.net/index.html#/framework>
- [Che16] Checkstyle-Dokumentation (Zyklomatische Komplexität). [Online]. http://checkstyle.sourceforge.net/config_metrics.html#CyclomaticComplexity
- [doc16] docker.com. [Online]. <https://www.docker.com/what-docker>
- [DoH16] (2016, July) Docker-Hub. [Online]. <https://hub.docker.com/r/library/jenkins/>
- [ecm16] (2016, Feb.) Eclipse Marketplace. [Online]. <https://marketplace.eclipse.org/content/hudsonjenkins-mylyn-builds-connector>
- [Fow06] Martin Fowler. (2006, May) martinfowler.com. [Online]. <http://martinfowler.com/articles/continuousIntegration.html>
- [god16] (2016) go.cd. [Online]. <https://docs.go.cd/current/index.html>
- [Hei12] Steffen Schluff and Björn Feustel. (2012, Feb.) Heise. [Online]. <http://www.heise.de/developer/artikel/Continuous-Integration-in-Zeiten-agiler-Programmierung-1427092.html>
- [InQ14] Charles Humble. (2014, Mar.) InfoQ. [Online]. <http://www.infoq.com/research/ci-server#>

- [JeC16] Jenkins and C/C++. [Online]. <https://jenkins.io/solutions/c/>
- [jen16] (2016, Apr.) jenkins.io. [Online]. <https://jenkins.io/>
- [Kaw15] Kohsuke Kawaguchi. (2015, Dec.) jenkins-ci.org. [Online]. <https://wiki.jenkins-ci.org/pages/viewpage.action?pageId=53608972>
- [Kra16] Nane Kratzke. (2016, Mar.) nkode.io. [Online]. <http://www.nkode.io/thesis/2016/03/01/thesis-automatisierte-bewertung-von-prog-praktika.html>
- [Neu14] Alexander Neumann. (2014, Nov.) heise. [Online]. <http://www.heise.de/developer/meldung/Zahl-der-aktiven-Jenkins-Installationen-se-it-Ende-2010-vervierfacht-2442975.html>
- [Okt15] Piotr Oktaba. (2015, Aug.) continuousdev.com/. [Online]. <http://continuousdev.com/2015/08/checkstyle-vs-pmd-vs-findbugs/>
- [QFT16] QF-Test. [Online]. <https://www.qfs.de/qf-test/testautomatisierung-mit-qf-test.html>
- [Spr16] Markus Sprunck. sw-engineering-candies.com. [Online]. <http://www.sw-engineering-candies.com/blog-1/comparison-of-findbugs-pmd-and-checkstyle>
- [Tiw15] Nitish Tiwari. (2015, July) opensource.com. [Online]. <https://opensource.com/business/15/7/six-continuous-integration-tools>
- [tra16] travis-ci.org. [Online]. <https://docs.travis-ci.com/>
- [Vor15] Paul Vorbach. (2015, July) vorba.ch. [Online]. <http://vorba.ch/2015/java-gradle-travis-jacoco-codecov.html>

Alle URLs wurden zuletzt am 27.07.2016 geprüft.