

## Masterarbeit

# Erstellung und Umsetzung eines Konzepts zur Verbesserung von automatischen Tests

Martin Locker

Ausgabedatum: 02. Mai 2016  
Abgabedatum: 02. November 2016

Betreuer: Dr. rer. nat. Dipl.-Inform. Nane Kratzke

(Prof. Dr. rer. nat. habil Schiffer)  
Vorsitzender des Prüfungsausschusses



# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Hintergrund . . . . .	1
1.2 Ziele der Arbeit . . . . .	2
1.3 Gliederung des weiteren Dokumentes . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 Workflow-Software . . . . .	3
2.2 Wirtschaftlichkeit der Testautomation . . . . .	4
2.3 Prüfebenen . . . . .	5
2.3.1 Unit-Tests . . . . .	6
2.3.2 Integrations-Tests . . . . .	7
2.3.3 Akzeptanztests . . . . .	8
2.4 Nichtfunktionale Tests . . . . .	8
2.5 Software Metriken . . . . .	10
2.6 Code Coverage . . . . .	12
2.7 Test Driven Development . . . . .	13
2.8 Continuous Integration (CI) . . . . .	14
2.9 Continuous Integration Server . . . . .	14
2.10 Continuous Delivery . . . . .	16
2.11 Gradle . . . . .	18
2.12 Virtualisierung . . . . .	19
2.12.1 Vollvirtualisierung . . . . .	19
2.12.2 Paravirtualisierung . . . . .	21
2.12.3 Containervirtualisierung . . . . .	21
2.12.4 Docker . . . . .	21
<b>3 Analyse des Ist-Zustandes</b>	<b>23</b>
3.1 Ist-Zustand Continuous Integration und Testausführung . . . . .	23
3.1.1 Verfügbare Server . . . . .	24
3.1.2 Laufzeiten . . . . .	26
3.1.3 Messung der Codequalität . . . . .	27
3.2 Nächtlicher Build . . . . .	28

## Inhaltsverzeichnis

3.3	Analyse des Testautomaten . . . . .	28
3.4	Performance- und Stresstests . . . . .	30
3.5	Ableitung zu lösender Probleme . . . . .	30
3.5.1	Ausführungszeit des CI-Builds - Problem I . . . . .	30
3.5.2	Lesbarkeit der Unit-Tests - Problem II . . . . .	30
3.5.3	Testbarkeit des Legacy-Codes - Problem III . . . . .	31
3.5.4	Fehleranalyse automatischer System-Tests - Problem IV . . . . .	32
3.5.5	Vereinfachung der Test-Erstellung - Problem V . . . . .	32
3.5.6	Einführung von Continuous Delivery - Problem VI . . . . .	33
<b>4</b>	<b>Konzepte zur Verbesserungen</b>	<b>35</b>
4.1	Verbesserung CI-Build . . . . .	35
4.1.1	Ausführungszeit der JUnit-Tests . . . . .	35
4.1.2	Lesbarkeit der Unit-Tests . . . . .	36
4.1.3	Testbarkeit des Quellcodes . . . . .	39
4.2	Verbesserungen des Testautomaten . . . . .	44
4.2.1	Log-Datei-Analyse . . . . .	44
4.2.2	Vereinfachte Erstellung neuer Tests . . . . .	44
4.2.3	Architektur-Anpassungen . . . . .	45
4.2.4	Parallelisierung . . . . .	51
4.2.5	Script-Server . . . . .	55
4.3	Konzept zur Erweiterung einer Continuous Delivery Pipeline . . . . .	56
4.3.1	Modulare Installer . . . . .	58
4.3.2	Remote System Update . . . . .	60
4.3.3	Hotfix Installer . . . . .	60
4.3.4	Microservices . . . . .	62
4.3.5	Aktualisierung des Test-Systems . . . . .	67
4.3.6	Erster Schritt für Continuous Delivery . . . . .	68
4.3.7	Skalierung der Jenkins-Slaves mit Docker . . . . .	69
<b>5</b>	<b>Umsetzung der Konzepte</b>	<b>73</b>
5.1	Beschleunigung Unit-Tests . . . . .	73
5.2	Beschleunigung durch Hardware . . . . .	74
5.3	Integrations-Tests separieren . . . . .	75
5.4	Testbarkeit des Codes sicherstellen . . . . .	75
5.5	Logdateien pro automatischem System-Test . . . . .	77
5.6	Verwendung von JUnit für automatische Tests . . . . .	78
5.7	Ausführung von automatischen System-Tests . . . . .	81
5.8	Parallelisierung der automatischen Tests . . . . .	82

5.9	Erhöhung parallel ausgeführter Script-Server . . . . .	84
5.10	Erweiterung zur Continuous Delivery Pipeline . . . . .	84
5.10.1	Pipeline-Prototyp mit Jenkins 2.x . . . . .	84
5.10.2	Erweiterung von Continuous Integration . . . . .	85
<b>6</b>	<b>Zusammenfassung und Ausblick</b>	<b>95</b>
<b>Appendix A</b>		<b>103</b>
1	Service-Registry . . . . .	103
2	Log-Datei Erweiterung . . . . .	103
3	FileMutex . . . . .	104
4	Verwendung JUnit in automatischen Tests . . . . .	107
5	Continuous Delivery Pipeline . . . . .	110
6	Artefact-Deployer . . . . .	112
7	Testauswertung . . . . .	117
<b>Abbildungsverzeichnis</b>		<b>119</b>
<b>Tabellenverzeichnis</b>		<b>121</b>
<b>Literaturverzeichnis</b>		<b>123</b>



# 1 Einleitung

## 1.1 Hintergrund

In großen Software-Projekten ist ein manueller Test der Software nach jeder Änderung am Quellcode nicht zu bewältigen. Solche Tests erfordern einen hohen personellen Aufwand und sind ökonomisch nicht sinnvoll. Ziel ist es daher durch eine Testautomatisierung auftretende Defekte nach einer Änderung an der Software so schnell wie möglich aufzudecken. Vergehen erst Monate nach einer Änderung, ist es für EntwicklerInnen schwer nachzuvollziehen, welche Änderung den Fehler verursacht hat. Spät entdeckte Fehler führen zu einem höheren Arbeitsaufwand und Kosten. EntwicklerInnen müssen sich lange nach der Implementierung mit dem Problem beschäftigen und mit mehr Aufwand den Fehler untersuchen. Zudem müssen Service-Mitarbeiter mit Kunden kommunizieren und Daten für die Fehlersuche und Fehlerreproduktion beschaffen. Erst beim Kunden gefundene Fehler können zu einem Verlust des Vertrauens in eine Software und ggf. zum Imageverlust des Herstellers führen.

Die Kosten der Fehlerbeseitigung bzw. der Fehlerverhütung sind in Abbildung 1.1 dargestellt. Sie steigen exponentiell von Phase zu Phase (Planung, Entwicklung, Arbeitsvorbereitung (AV), Fertigung, Endprüfung bis zum Kunden). Der Faktor '10' ist nicht als exakter Messwert zu verstehen, sondern verdeutlicht die extreme Ungleichverteilung der Kosten [Kre14].

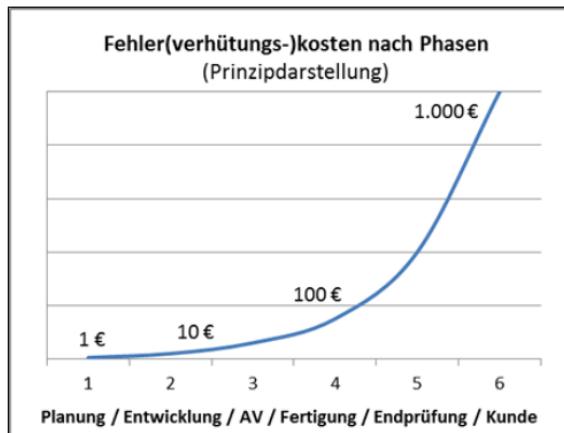


Abbildung 1.1: Zusammenhang zwischen Phasen der Fehlerverursachung und deren Fehlerkosten  
[SP10]

## 1.2 Ziele der Arbeit

Das Ziel ist die Verbesserung der Testautomatisierung eines Workflow-Systems, für das 2004 ein Testautomat entwickelt wurde, um automatisch Fehler zu entdecken. Der Testautomat genügt in der heutigen Zeit nicht mehr den Anforderungen einer agilen Entwicklung. Automatische Tests sollten so schnell wie möglich den EntwicklerInnen Feedback geben, ob Änderungen an der Software zu unerwünschten Nebeneffekten geführt haben. Um ein schnelles Feedback zu erhalten, wird für die Entwicklung des Workflow-Systems ein *Continuous Integration Server* eingesetzt, der die geänderte Software kompiliert und Modul-Tests ausführt. Die automatischen System-Tests erfolgen bisher am nächsten Tag. Ziel ist es, die Tests mehrmals am gleichen Tag ablaufen zu lassen. Dazu muss der Continuous Integration Build um Continuous Delivery erweitert werden. Für eine breitere Absicherung der Software soll die Möglichkeit geschaffen werden, automatische Tests einfacher zu erstellen. Weiterhin ist zu untersuchen, ob Containertechnologien unterstützend eingesetzt werden könnten.

## 1.3 Gliederung des weiteren Dokumentes

**Kapitel 2** gibt zunächst einen kurzen Überblick über das Workflow-System. Danach wird auf den generellen Einsatz von Testautomatisierung und die verschiedenen Arten von Tests eingegangen. Abschließend werden die Werkzeuge für eine Build- und Testautomatisierung wie Continuous Integration, Jenkins, Continuous Delivery, Gradle und Virtualisierungslösungen beschrieben. In **Kapitel 3** wird der Ist-Zustand von Continuous Integration und des Testautomaten analysiert und daraus zu lösende Probleme abgeleitet. **Kapitel 4** behandelt die Konzepte, um die Probleme zu lösen. Die Umsetzungen und die erzielten Ergebnisse sind in **Kapitel 5** aufgeführt. **Kapitel 6** fasst das Thema zusammen und liefert einen Ausblick.

# 2 Grundlagen

## 2.1 Workflow-Software

Die Workflow-Software wird in Druckereien zum Vorbereiten und Durchführen von Druckaufträgen eingesetzt. Durch ein Management-Informations-System (MIS) werden Auftragsdaten in das Workflow-System übergeben. Ein Mitarbeiter bearbeitet mit der Workflow-Software die zu druckenden Daten (PDFs) und platziert diese auf ein für die Druckmaschine geeignetes Ausschießschema. Der vereinfachte typische Ablauf in einer Druckerei ist in Abbildung 2.1 dargestellt. Für den Druck mit einer Offset-Druckmaschine werden durch die Workflow-Software, in einem Druckauftrag ohne Sonderfarben, vier Druckplatten (Cyan, Magenta, Yellow und Black) erzeugt. Ein Mitarbeiter im Drucksaal rüstet die Druckmaschine mit den erstellen Platten und startet den Druckauftrag. Die Berechnungen des Workflow-Systems, die den Preis beeinflussen (z. B. Farbdaten), werden wieder an das MIS zur Nachkalkulation und Rechnungserstellung zurückgesendet.

Die Kommunikation der verschiedenen Systeme einer Druckerei erfolgt über das Job Definition Format (JDF). Dies ist ein von dem CIP4-Konsortium<sup>1</sup> spezifiziertes und dokumentiertes Austauschformat, das alle Auftragsinformationen, Produktionsdaten, den Ablauf der Arbeitsschritte enthält sowie die Überwachung und Steuerung ermöglicht.

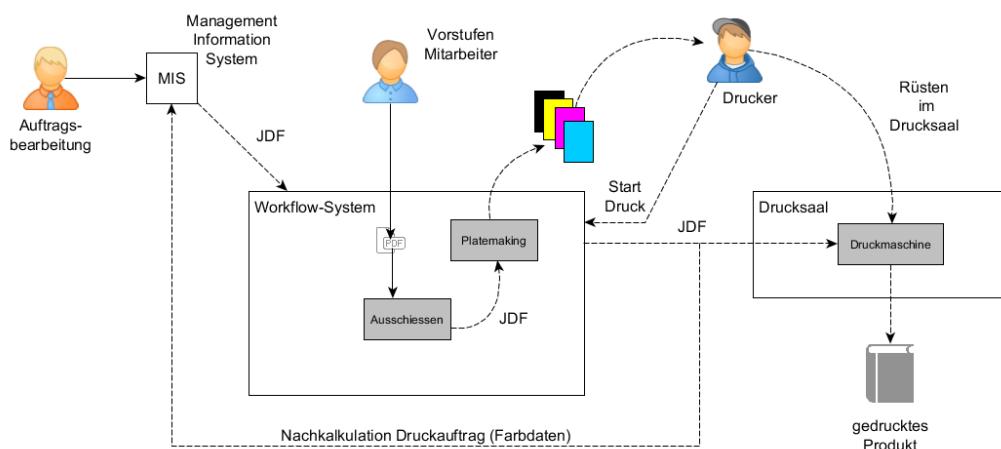


Abbildung 2.1: Ablauf in einer Druckerei

<sup>1</sup><https://cip4.org/>

## 2.2 Wirtschaftlichkeit der Testautomation

Die Automatisierung von Tests ist in der Regel zeitaufwendig und nicht immer lohnt es sich, einen Testfall zu automatisieren. Folgend wird der ökonomische Nutzen von automatischen Tests für das Workflow-System beleuchtet. Ein Softwaretest verfolgt zwei Ziele. Er soll erstens nachweisen, ob sich die Software konform zu den Anforderungen verhält und zweitens Fehler finden [Spi08]. Schon das Finden von Fehlern kann den Aufwand für den Test rechtfertigen. Der Testaufwand ist nicht gerechtfertigt, sollten die Fehlerkosten niedriger als die Testkosten sein.

Die Wirtschaftlichkeit des Tests wird durch die Berechnung des ROI (Return of Investment) bestimmt. Im Allgemeinen ist die Berechnung wie folgt:

$$ROI = \frac{(Nutzen - Kosten)}{Kosten}$$

Übertragen auf die Berechnung für den Test sieht die Berechnung des ROI wie folgt aus:

$$TestROI = \frac{(Testnutzen - Testkosten)}{Testkosten}$$

Zur Berechnung des Test-ROIs müssen der Testnutzen ermittelt und im Voraus die Testkosten bestimmt werden. Der Testnutzen ist die Vermeidung von Fehlerkosten, die durch Produktionsstörungen, Fehlerbehebungen und durch den Vertrauensverlust der Kunden entstehen. Produktionsstörungskosten entstehen durch Systemausfälle oder falsche Ergebnisse. Führt ein Fehler zehnmal 5 Minuten täglich zu Systemausfällen, kann das ein Unternehmen bis zu 10% der Produktivität seiner Mitarbeiter kosten. Dies übersteigt die Gewinnmarge mancher Unternehmen [PTvV02]. Die Fehlerbehebungskosten entstehen durch die Untersuchung des Fehlers, die Identifizierung der Ursache und deren Korrektur. Die Fehlerbehebung muss anschließend getestet werden und auf dem Kundensystem ggf. mehrfach installiert werden.

Eine Studie hat gezeigt, dass eine Korrektur eines Fehlers im Betrieb beim Kunden rund 20-mal so viel kostet, als wenn der Fehler im Modultest identifiziert und behoben worden wäre [Boe75]. In der Studie von Endres [End03] ist die Wirtschaftlichkeit des Testens von besonderer Bedeutung. Er geht auf das Einsparpotential der rechtzeitigen Erkennung von Softwarefehlern ein und plädiert für einen strengen und automatisierten Testprozess [ER04]. Experten schätzen, dass die Korrektur von Fehlern auf Kundensystemen etwa 15% der Release-Kosten ausmachen [SHT04].

Sneed und Jungmayr [SJ11] berechnen für einen manuellen Softwaretest den ROI und stellen diesen einer Berechnung mit einer Testautomatisierung gegenüber. Durch die gesteigerte Anzahl der Testfälle ist der Test-ROI mit Testautomatisierung um das Dreifache höher. Dies zeigt die Bedeutung der Testautomatisierung. Ziel in einem Software-Projekt ist es, mit möglichst geringem Aufwand viele Fehler zu finden. Testbarkeit und Testproduktivität setzen

eine Vorinvestition voraus. Für eine Rechtfertigung muss ein Test-ROI berechnet werden.

Die stetig immer steigenden Qualitätsanforderungen in Software-Projekten setzen zum Nachweis und zur Sicherstellung der Qualitätsanforderungen einen professionellen Test voraus [KvdABV08]. Recardo Jackson [Jac09] weißt darauf hin, dass der Fokus nicht nur auf der Systemeinführung, sondern auch auf dem gesamten Anwendungszyklus liegen sollte. Der Testaufwand erreicht in Software-Projekten schnell 30-40% des Realisierungsaufwandes. Gerade bei regelmäßigen Releases größerer Anwendungen lassen sich Aufwand und Kosten durch Automatisierung senken.

Bei dem Workflow-System ist zu beachten, dass es sich nicht um ein einmaliges Projekt handelt. Die Software existiert schon viele Jahre und wird stetig weiter entwickelt. Einmal investierte Kosten zur Erstellung von automatischen Tests wirken sich über viele Software-Releases hin aus. Das System ist sehr komplex und erfordert nach Änderungen an der Software einen Regressionstest. EntwicklerInnen haben nicht die Möglichkeit zu überblicken, welche Auswirkungen ihre Änderungen auf das Gesamtsystem haben. Zu beachten ist, dass der Testnutzen bei der Berechnung des Test-ROI den geplanten Lebenszyklus eines Produkts berücksichtigt. Aus diesem Grund ist es für die Workflow-Software ökonomisch rentabel, einen hohen Testautomatisierungsgrad zu erreichen.

Regressionstests sollen das Auftreten von Fehlern in bereits getesteten Teilen der Software verhindern. Nach jeder Änderung durch Weiterentwicklung oder Fehlerkorrekturen werden Testfälle ausgeführt. Die Tests sollen sicherstellen, dass keine negativen Auswirkungen auf bestehende Funktionalitäten auftreten. Unbemerkt eingeschlichene Fehler sollen frühzeitig erkannt werden. In großen und langlebigen Software-Projekten nehmen Regressionstests oft einen großen Teil der Code-Basis ein. Um die Korrektheit der Software sicherzustellen, sind Regressionstests das primäre Mittel. In vielen Firmen ist es Pflicht, nach der Beseitigung eines Fehlers einen Testfall zu generieren, der das erneute Auftreten des Fehlers feststellen würde. Werden Regressionstests automatisiert durchgeführt, so führen die Tests stetig zu einer immer leistungsfähigeren und kostenökonomischen Sicherstellung der Software-Integrität [Hof13].

## 2.3 Prüfebenen

Tests für ein Software-System existieren idealerweise in verschiedenen Ebenen der Isolation. Mike Cohn hat an einer Testpyramide (Abbildung 2.2) verdeutlicht, dass die Mehrzahl der Fehler auf der Ebene der Unit-Tests gefunden werden können. Unit-Tests sind schnell erstellt und haben eine kurze Ausführungszeit. Der Grund für das Scheitern eines Unit-Tests ist deutlich, so dass Fehler sehr viel schneller lokalisiert werden können als bei Fehlern in Tests auf einer höheren Ebene. In der Regel sind Unit-Tests viel leichter zu warten als Tests der höheren Ebenen [DM11a]. Die oberen Ebenen sollen nur sicherstellen, dass sich einzelne

konkrete Komponenten in Interaktion korrekt verhalten. Manuelle Tests sind teuer und langwierig in der Durchführung. Sie geben wenig Aufschluss über die Quelle des Defekts [Sei12].

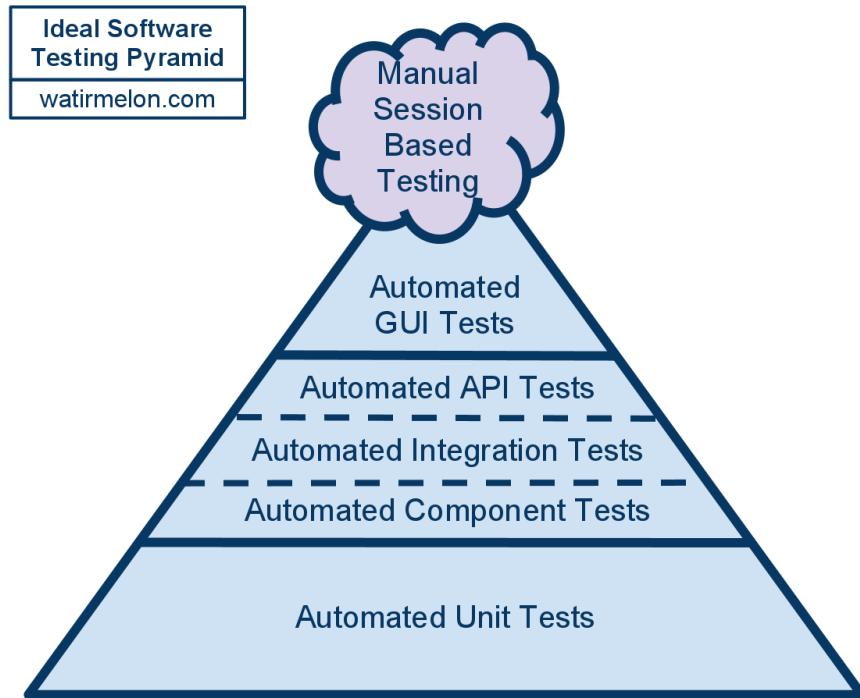


Abbildung 2.2: Ideale Test-Pyramide [Sco12]

### 2.3.1 Unit-Tests

Unit-Tests sind die unterste Ebene von automatisierten Tests, in den die Funktionalität des Programmcodes von Klassen oder Methoden überprüft wird. Idealerweise sollte nur der Produktionscode durchlaufen werden. Der Vorteil von Unit-Tests ist die sehr geringe Ausführungszeit von einigen Millisekunden. Nur so sind in großen Projekten mit einigen tausend Unit-Tests, Buildzeiten von einigen Minuten möglich [Tam13]. Um eine Klasse in Isolation zu testen, werden Abhängigkeiten zu anderen Objekten durch Stellvertreter-Objekte (Mock-Objekte) ersetzt. Unit-Tests ohne Mock-Objekte erlauben nur die Rückgabewerte von Methoden oder Zuständen von Klassen zu testen. Diese Art von Test wird als *State Verification* bezeichnet. Mit Mock-Objekten ist dagegen ein tieferer Einblick in den Produktionscode möglich. Es kann das Verhalten der aufgerufenen Methoden überprüft werden. In dieser sogenannten *Behavior Verification* ist die Kontrolle der Anzahl, Reihenfolge und der Parameter möglich [Tam13].

JUnit ist der de facto Standard für das Schreiben von Unit-Tests für Java. Testmethoden werden in der Testklasse mit der @Test-Annotation versehen. Mit den Annotationen @Before

und @After werden Methoden zum Auf- und Abbau von im Test benötigten Objekten definiert. In den Testmethoden werden Methoden aus dem Produktionscode mit definierten Parametern aufgerufen. Mit Assertion-Methoden werden die Rückgabewerte mit den Erwartungen verglichen. Durch die Assertion-Methoden wird bestimmt, ob ein Test erfolgreich oder fehlgeschlagen ist (siehe Abbildung 2.3).

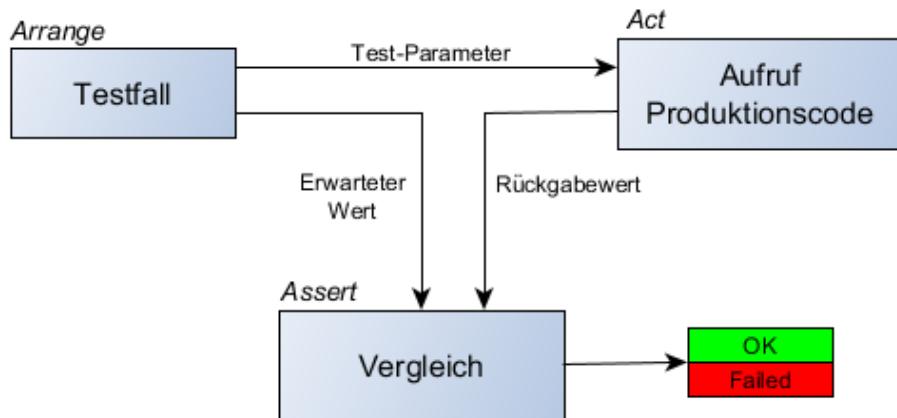


Abbildung 2.3: Ablauf eines Unit-Tests

### 2.3.2 Integrations-Tests

Es ist nicht ausreichend, einzelne Methoden oder Klassen durch Unit-Tests zu testen. Auf der nächsten Ebene wird das Zusammenspiel von mehreren Klassen in sogenannten Integrations-Tests überprüft. Typischerweise wird auf den Einsatz von Stellvertreterobjekten verzichtet und stattdessen eine Testumgebung aufgebaut, in der alle nötigen Abhängigkeiten zur Verfügung stehen. Durch Mocking wird das Zusammenspiel von Units ausgeblendet [Wol15a]. Sollen Komponenten explizit nicht mitgetestet werden, ist der Einsatz von Mocking dennoch sinnvoll. Wie bei Unit-Tests wird auf die Klassen des Produktionscodes direkt zugegriffen. Unit- und Integrationstests werden i.d.R. von EntwicklerInnen implementiert, um die Funktion des Produktionscodes zu testen [Tam13]. Kostis Kapelonis liefert einige Punkte, die einen Test als Integrationstest ausweisen [Kap13]:

1. Der Test verwendet eine Datenbank
2. Der Test verwendet Netzwerkzugriffe
3. Der Test verwendet externe Systeme (eine Warteschlange oder einen Mail-Server)
4. Der Test liest/schreibt Dateien oder führt andere I/O-Zugriffe aus

### 2.3.3 Akzeptanztests

Akzeptanztests dienen zur Absicherung der fachlichen Umsetzung von Anforderungen. Mit Akzeptanztests werden größere Teile des Systems oder sogar das ganze System getestet. Die Tests setzen auf einer anderen Ebene als Unit- oder Integrationstests an. Unit-Tests sind meistens White-Box-Tests und nehmen Bezug auf die Implementierung der Klasse. Aus dem Grund müssen Unit-Tests bei Refactorings, der getesteten Klasse, angepasst werden. Die Akzeptanztests sind Black-Box-Tests. Die Korrektheit der Implementierung wird mit Hilfe einer API oder der Oberfläche überprüft. Die Tests sind nicht von den Interna einer Klasse abhängig. Für den Test spielt es keine Rolle, in welchen Klassen die Funktionalität programmiert ist und wie diese aussehen. Akzeptanztests kommt eine große Bedeutung zu, da sie auf das Interesse der NutzerInnen abzielen. Sie haben den Fokus auf einer korrekten Umsetzung der Features [Wol15a].

## 2.4 Nichtfunktionale Tests

Neben funktionalen Anforderungen gibt es auch Anforderungen, die keine Funktion repräsentieren. Diese nichtfunktionalen Anforderungen sollten nicht vernachlässigt werden, da sie häufig wettbewerbsentscheidend sind [Emm10]. Sie sind oft erst erfahr-, test- und messbar, wenn das Gesamtsystem fertig gestellt wurde. In der Literatur gibt es keine einheitliche und allgemein akzeptierte Definition für nichtfunktionale Anforderungen [Emm10]. In einer Studie wurden aus 182 Quellen der letzten 30 Jahre die am meisten genannten nichtfunktionalen Anforderungen bestimmt [MZN10].

1. Performance (Leistung)
2. Reliability (Zuverlässigkeit)
3. Usability (Gebrauchstauglichkeit)
4. Security (Sicherheit)
5. Maintainability (Wartbarkeit)

Für die meisten dieser nicht funktionalen Anforderungen ist es kaum möglich automatische Tests zu schreiben. Die Gebrauchstauglichkeit erfordert meist manuelle Usability-Tests. In einigen Open Source Projekten wie web-accessibility-testing<sup>2</sup> gibt es erste Ansätze Usability-Tests zu automatisieren [Tam13].

Die Leistungsfähigkeit einer Anwendung kann mit Performance-, Last- und Kapazitätstest untersucht werden [Wol15b]. Bei einem Performance- bzw. Leistungs-Test wird die Antwortzeit

---

<sup>2</sup><https://code.google.com/archive/p/web-accessibility-testing/>

von definierten zeitkritischen Funktionen gemessen. Die Messung erfolgt unter optimalen Bedingungen, d.h. kein anderer Benutzer arbeitet mit dem System [Fra03]. Zu beachten ist, dass der Begriff Performance-Tests häufig nicht einheitlich verwendet wird [Vog09]. Tests zur Messung der Leistungsfähigkeit der Anwendung können, im Gegensatz zu den meisten anderen nicht funktionalen Anforderungen, automatisiert getestet werden und in eine Continuous Delivery Pipeline integriert werden. Bei den Performance-Tests ist, im Gegensatz zu Last- und Stress-Tests, auf eine sequentielle Abarbeitung zu achten, da parallele Tests um die Ressourcen konkurrieren und die Zeiten beeinflussen.

Mit Last-Tests wird der Grenzbereich der Spezifikation einer Software untersucht und in Stress-Tests wird dieser bewusst überschritten. Im Stress-Test kann das System durch Steigerung der zu verarbeitenden Daten oder durch Wegnahme von Ressourcen überansprucht werden. Durch Stress-Tests wird die reale Belastungsgrenze der Software ermittelt und überprüft, ob das System nach einer Überlast wieder in den Normalbetrieb zurückfindet [Hof13].

In Kapazitätstests wird die Performance und der Durchsatz eines Systems getestet. Die Performance gibt an, wie schnell ein Request von dem System verarbeitet wird. Bei Systemen mit einer schlechten Performance müssen BenutzerInnen auf eine Reaktion des Systems warten. Der Durchsatz bestimmt die Anzahl Requests, die ein System in einer bestimmten Zeit abarbeiten kann. An Systemen mit einem schlechten Durchsatz können nur wenige BenutzerInnen parallel arbeiten. Bei Kapazitätstests ist es vorteilhaft, Daten in Anzahl und Größe sowie eine Umgebung zu verwenden, wie sie in der Produktion vorkommen. Ein Problem bei Tests zur Messung der Leistungsfähigkeit ist es, dass eine Software erst fachlich und vollständig implementiert sein muss. Andernfalls kann ein Test zu unrealistischen Werten führen [Wol15a].

Die Testarten Stress-, Last-, und Kapazitätstests sind sehr schwer oder gar nicht manuell durchführbar. Die dafür notwendige Personenzahl steht meist nicht zur Verfügung bzw. ist nicht finanziert. Eine Automatisierung dieser Testarten ist damit unvermeidbar [KWT<sup>+</sup>06].

Wird eine Software wie das Workflow-System stetig weiter entwickelt, ist es möglich die Performance fortlaufend zu überprüfen und mit den Messwerten der Vorgängerversion zu vergleichen. Wichtig für den Vergleich ist die Verwendung von identischer Hardware. Es sollte keine virtuelle Maschine verwendet werden und keine anderen Applikationen auf der Maschine laufen.

Die Performance ist eine wichtige Eigenschaft des Workflow-Systems. Es beeinflusst die Effizienz einer Druckerei. Durch Fusionen und Firmenübernahmen entstehen gegenwärtig und in Zukunft größere Druckereien, deren Bedarf an höheren Durchsätzen von dem System beherrschbar sein müssen. Die Performance-Tests und deren Auswertung sollten künftig in einer Continuous Delivery Pipeline automatisiert werden (siehe Kapitel 2.10 Abbildung 2.7).

## 2.5 Software Metriken

Neben den funktionalen Tests existiert die Code-Qualität als eine ganz andere Dimension von Software-Qualität, die sich ebenfalls automatisch testen lässt. Schlechte Software entsteht meistens schleichend und sollte durch stetige Messungen kontrolliert werden. Die Kosten für schlechte Software-Qualität treten erst nach und nach auf. Mit der Zeit erhöht sich der Aufwand für Implementierungen und somit sinkt die Geschwindigkeit mit der Features geliefert werden können. Eine Continuous Delivery Pipeline sollte unterbrochen werden, wenn die Software der Anforderung an die Code-Qualität nicht mehr genügt.

Um die Qualitätsanforderungen von bereits geschriebenem Quellcode sicherzustellen, kann manuell oder automatisch eine statische Analyse durchgeführt werden. Die Quelltexte werden einer Überprüfung unterzogen für die eine Übersetzung des Programms nicht notwendig ist. Eine einfache Metrik, die Komplexität einer Software zu bestimmen, besteht darin die Codezeilen zu addieren (LOC-Metrik, LOC = Lines of Code). Für diese Metrik ist kein komplexes Werkzeug notwendig. Eine Erweiterung der LOC-Metrik ist das Aufsummieren ohne Berücksichtigung der Kommentarzeilen (NCSS-Metrik, NCSS = Non Commented Source Statements). Die Einfachheit der beiden Metriken geht zu Lasten der Aussagekraft. Allein durch Umformatierungen des Quellcodes ändert sich die Kennzahl der Metrik. Um die Aussagekraft zu verbessern wurden andere Zähltechniken eingesetzt, in der Kommentare und die visuelle Repräsentation des Codes nicht als Messgröße einfließen.

Maurice Howard Halstead [Hal77] postulierte 1977 Metriken, die aus der lexikalischen Struktur des Quelltextes hergeleitet werden. Er stellte einen Zusammenhang mit der Programmkomplexität und der Auftrittscharakteristik der lexikalischen Elemente her [Hof13]. Der Quellcode wird als eine Folge von Operatoren und Operanden interpretiert. Die Metrik wird häufig zur Messung der Wartbarkeit eingesetzt.

Eine der weit verbreitetsten statischen Software-Metriken wurde 1976 von Thomas McCabe eingeführt. Die Metrik ist von der Programmiersprache unabhängig und zeigt die Komplexität des Kontrollflusses im Code.  $v(G)$  ist die Anzahl von konditionellen Verzweigungen im Programmfluss. Der Wert von  $v(G) = 1$  beschreibt ein Programm, das lediglich aus sequenziellen Anweisungen besteht. Die Zahl wird bei zunehmender zyklomatischer Komplexität größer. Durch die ansteigende Anzahl an Pfaden wird es schwieriger, die Funktion zu verstehen und zu testen. Funktionen mit einer hohen zyklischen Komplexität benötigen mehr Testfälle als solche mit einem geringen McCabe-Wert. In der Regel sollte eine Funktion so viele Testfälle wie der errechnete  $v(G)$  Wert haben [CL07].

Der Kopplungsgrad des Systems sollte stets kontrolliert werden, denn je höher dieser ist, desto schwerer ist es i.d.R. das System zu ändern. Die von einer Codeänderung direkt oder indirekt betroffenen Codestellen, wachsen mit dem Kopplungsgrad. Dieser wird mit der ACD-Metrik (Average Component Dependency) [Lak96] gemessen. Zur Bestimmung des Wertes werden die direkten und indirekten Abhängigkeiten von Typcontainers (Packages, Jar-Files,

etc.) inklusive sich selbst gezählt. Die Summe wird durch die Anzahl aller Typcontainer dividiert [Zit07].

$$ACD = \frac{\sum \text{Abhängigkeiten}}{\sum \text{Knoten}}$$

Der Wert von 1 bedeutet, dass der Typcontainer nur von sich selbst abhängt. In Abbildung 2.4 ist ein Beispiel aufgeführt. Links führt die Berechnung eines zyklischen Graphs zu dem Wert  $ACD = 2,33$ . In jedem Typcontainer (als Knoten dargestellt) ist die Anzahl der Abhängigkeiten abgebildet. Der Wert sagt aus, dass jeder Knoten im Mittel von 2,33 Knoten abhängt. Bringt man nun eine zyklische Abhängigkeit mit in den Graphen (gestrichelter Pfeil), verschlechtert sich der Wert auf  $ACD = 3,67$ . Im rechten Graphen ist dieser Zyklus rot markiert.

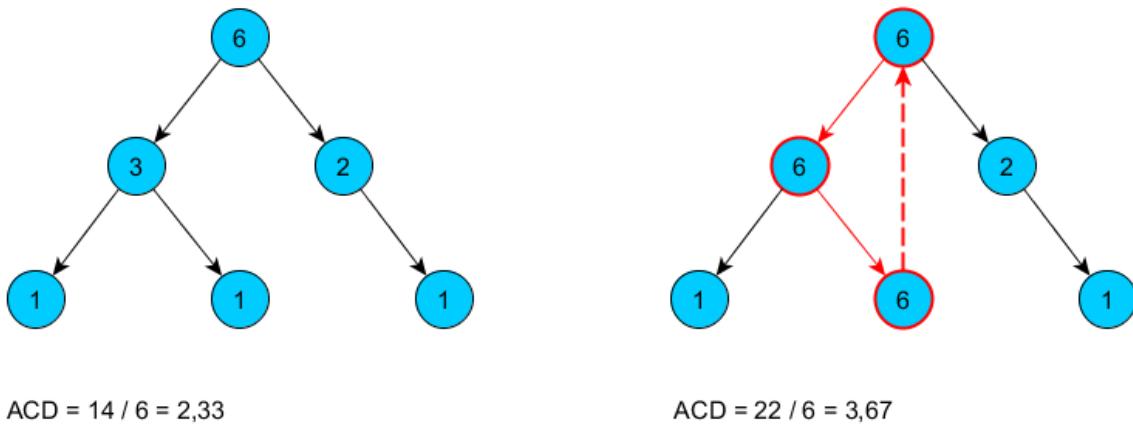
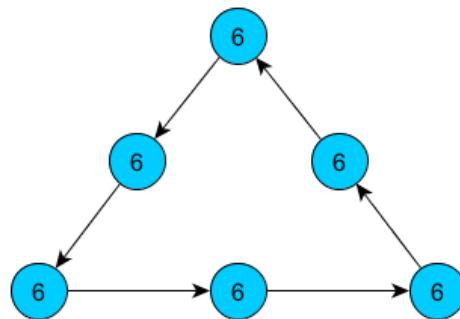


Abbildung 2.4: Berechnung des ACD-Werts mit und ohne Zyklen

Um vergleichbare Werte zu erhalten, muss noch einmal durch die Anzahl der Knoten dividiert werden, da größere Graphen wegen der Anzahl der Knoten einen höheren ACD-Wert aufweisen. Der Wert wird als rACD (relative Average Component Dependency) bezeichnet. Der schlechteste Kopplungsgrad ist, dass jeder Knoten jeden anderen Knoten kennt. Mit 6 Knoten ergibt sich:

$$ACD = \frac{6 * 6}{6} = 6 \quad \text{allgemein} \quad \frac{n * n}{n} = n$$

Wird noch einmal durch die Anzahl der Knoten  $n$  dividiert, erhält man den schlechtesten Wert für rACD von 1, unabhängig von der Anzahl der Knoten. In Abbildung 2.5 ist der schlechteste Zustand mit 6 Knoten dargestellt. Änderung in einem Knoten kann zu Änderungen in allen anderen Knoten führen, da jeder Knoten jeden anderen kennt.



$$rACD = 6 * 6 / 6 * 6 = 1$$

Abbildung 2.5: Beispiel für den schlechtesten rACD-Wert

## 2.6 Code Coverage

In einer Testabdeckungsanalyse (Code Coverage) wird überprüft, ob alle Codezeilen einer Methode durch einen Test abgedeckt sind. Wird Code geändert, der nicht durch einen Test überprüft wird, werden unerwünschte Verhaltensänderungen von den automatischen Tests nicht entdeckt. Ziel ist es daher, eine möglichst hohe Testabdeckung zu erreichen. Durch die Testabdeckungsanalyse werden Lücken aufgezeigt, die mit der Erstellung weiterer Testfälle geschlossen werden können. Die Code-Coverage-Analyse dient dazu, die Qualität der Tests zu verbessern.

Zur Bestimmung des Testabdeckungsgrades gibt es verschiedene Varianten wie Line-, Branch- und Path-Coverage. Bei der Line Coverage ist eine 100%ige Testabdeckung erreicht, wenn jede Zeile mindestens einmal durchlaufen wurde. Probleme bei Bedingungen im Code könnten auch bei einer Testabdeckung von 100% verborgen bleiben. Im Beispiel Listing 2.1 wird für einen Testparameter  $i \geq 0$  jede Zeile des Codes durchlaufen. Für  $i < 0$  tritt bei der Ausführung hingegen eine NullPointerException auf. Aus diesem Grund ist in der Praxis die Line Coverage nicht ausreichend.

In der Branch Coverage wird bestimmt, wie viele der Kanten im Kontrollflussgrafen ausgeführt werden. Eine Testabdeckung von 100% wird erreicht, wenn jede Entscheidung einmal mit *true* und einmal mit *false* durchlaufen wurde. In dem Beispiel von Listing 2.1 wird für  $i \geq 0$  nur eine Testabdeckung von 50% erreicht, da die Verzweigung nur mit *true* durchlaufen wird. Im Gegensatz zur Line Coverage ist ein 2. Testfall z. B.  $i = -5$  notwendig, um eine Testabdeckung von 100% zu erreichen. Die Bedingung wird einmal mit *true* und einmal mit *false* durchlaufen [Hof08].

Die Path-Coverage ist noch genauer, da alle möglichen Ablaufpfade berücksichtigt werden. Betrachtet man den Programmablauf einer Methode als Flow Chart, überprüft Path-Coverage,

ob jeder Weg von Beginn bis zum Ende im Diagramm abdeckt ist. Aufgrund der exponentiell ansteigenden Pfade wird im Allgemeinen die Path-Coverage von Coverage-Tools nicht unterstützt [Hof08].

Listing 2.1: Problem der Line Coverage

```
package de.fhbluebeck.lockemar.coverage;

public class LineCoverage
{
    public String convertIntToPositiveString(int i)
    {
        Integer posVal = null;
        if(i >= 0)
        {
            posVal = Integer.valueOf(i);
        }
        return posVal.toString();
    }
}
```

## 2.7 Test Driven Development

Oft werden parallel oder erst nach Abschluss einer Implementierung geeignete Testfälle generiert. Das Testen erfolgt sehr spät im Entwicklungsprozess und Fehler in den Anforderungen, dem Entwurf, dem Design und der Implementierung werden sehr spät erkannt.

Zur Verbesserung der Testerstellung ist die Idee der testgetriebenen Entwicklung (Test Driven Development) entstanden. Es werden Testfälle vor der Implementierung spezifiziert, mit denen das System entworfen wird. Die EntwicklerInnen machen sich bei der Testfallerstellung Gedanken über die Funktionalitäten der Komponenten und wie diese zu testen sind [SDW<sup>+</sup>10]. Neuer Code wird in einem 'Red-Green-Refactor'-Zyklus erstellt. Es wird erst ein Test geschrieben, der zunächst fehlschlägt. Anschließend wird Code implementiert, sodass dieser alle Tests besteht. Das Test-First Prinzip von TDD fördert eine hohe Testabdeckung. TDD ist keine Test-, sondern eine Designstrategie [Rös10]. Komponenten, die mit Test Driven Development entwickelt wurden, sind i.d.R. modularer und damit übersichtlicher aufgebaut. Vorhandene Unit-Tests helfen dem Entwickler, den Code leichter zu verstehen und Fehler im Programmcode einfacher zu finden. Unterschiedliche Bedingungen können von dem Unit-Test simuliert werden und somit den Code effizient überprüfen. Nach Michael Tamm ist ein Test sozusagen der erste Client für die API. Schnittstellen werden so entworfen, wie sie später einmal genutzt werden sollen. Nicht zu vergessen ist in dem TDD-Zyklus der 3. Schritt des Refactorings, nachdem der Test durch die Implementierung des Produktivcodes fehlerfrei (grün) gemacht wurde. Der Code soll gut lesbar und damit wartbar sein. In der Praxis wird das häufig vergessen, da die EntwicklerInnen es als ihre Arbeit ansehen, neue Anforderungen

## 2 Grundlagen

zu programmieren. Die Arbeit ist abgeschlossen, sobald der Test fehlerfrei ausgeführt wird [Tam13].

### 2.8 Continuous Integration (CI)

Werden Programmkomponenten aus Aufwandsgründen erst im letzten Moment einer Auslieferung zusammengefasst, kann dies bei der Integration der Komponenten zu vielen unerwarteten Problemen führen. Häufig sind Korrekturen notwendig, damit das Projekt kompiliert und die Tests fehlerfrei durchlaufen. Nach den Ursachen muss zeitaufwendig geforscht werden. Mit Continuous Integration steht ein Werkzeug zur Verfügung, um die Risiken in der Softwareentwicklung zu minimieren und die Qualität zu steigern. Die Idee ist eine Integration nach jeder Codeänderung oder einmal pro Nacht im 'nightly build' durchzuführen. Der Integrationsaufwand sinkt, da durch das häufige Integrieren überschaubare Änderungen eingespielt wurden. Es sind keine umfangreichen Anpassungen unter Zeitdruck notwendig, um ein auseinander gelaufenes Projekt wieder auf einen gemeinsamen Kurs zu bringen. Die Fehlersuche wird vereinfacht, da die in Frage kommenden Fehlerquellen reduziert sind. Im Gegensatz zu einer manuellen Integration wird früher, öfter, gründlicher und umfangreicher getestet. Die Qualität des Softwareprodukts steigt erheblich. Die Tests sind gründlicher, weil stets derselbe Testparcours durchlaufen wird. Es werden aus Aufwandsgründen keine Tests ausgelassen, da in den zu testenden Bereich in der Software vermeintlich nichts geändert wurde. Die automatischen Tests sind umfangreicher, da eine Codeabdeckung erreicht werden kann, die beim manuellen Testen unwirtschaftlich wäre. In Abbildung 2.6 ist ein typischer CI-Zyklus abgebildet. Auf den Entwicklungsrechnern werden die Änderungen am Code durchgeführt und an ein zentrales Versionskontrollsystem übertragen (1). Der CI-Server bemerkt die Änderungen (2) und veranlasst einen neuen Build auf den Build-Rechnern (3). Ist der Build abgeschlossen, werden die Ergebnisse auf den CI-Server zur Auswertung und Archivierung übertragen (4). Die EntwicklerInnen werden über E-Mail, Issue-Tracker oder andere Kommunikationskanäle über den Ausgang des Builds informiert. Tritt ein Fehler auf, können die EntwicklerInnen gleich mit der Verbesserung des Codes beginnen. Auch andere Interessierte, wie TeamleiterInnen oder TesterInnen, können den aktuellen Stand verfolgen. In einem CI-Zyklus erfolgt ein vollautomatischer Build, in dem die entstehenden Produkte automatisch auf eine korrekte Funktionsweise überprüft werden. Ein CI-Zyklus sollte sehr schnell sein, idealerweise im Minutenbereich [Wie11].

### 2.9 Continuous Integration Server

Die zu bewältigenden Aufgaben im Continuous Integration Umfeld sollen automatisiert ablaufen. Dazu können diverse Tools eingesetzt werden. In dem Workflow-System kommt

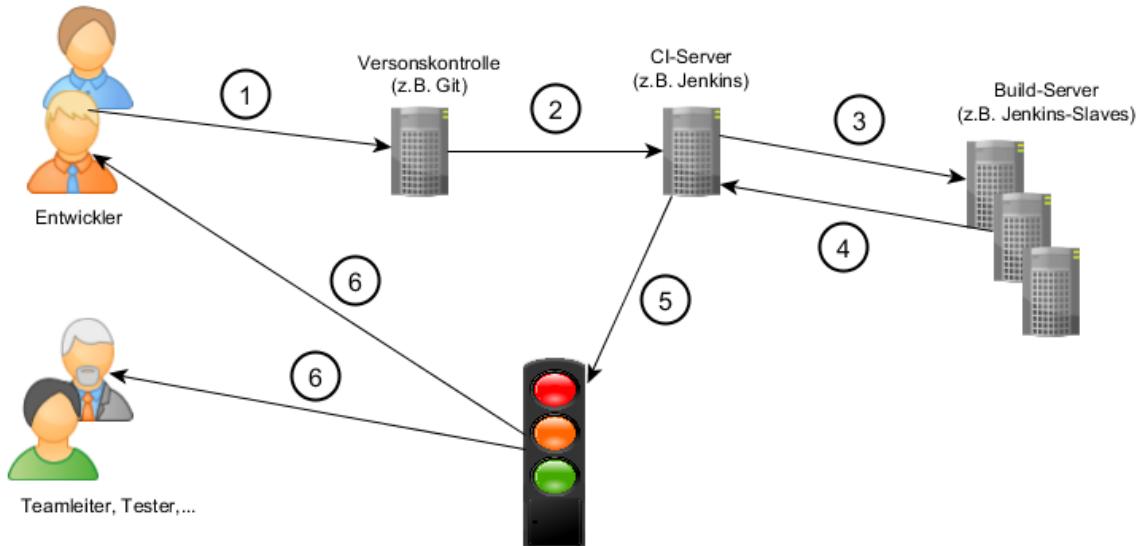


Abbildung 2.6: Typischer CI-Zyklus

Jenkins<sup>3</sup> zum Einsatz. Das Projekt ist frei nutzbar, bietet zudem noch weitere Vorteile. Jenkins lässt sich sehr schnell installieren und einrichten. Der Ressourcenverbrauch ist gering und mit der administrativen Weboberfläche lassen sich viele Dinge ohne Handbuch erledigen. Jenkins verfügt über eine Plugin-Schnittstelle, für die zahlreiche Plugins zur Verfügung stehen. Nicht unerheblich ist, dass Jenkins über eine große und hilfsbereite Anwender- und Entwicklergemeinde verfügt. Jenkins ist ein Java basierter CI-Server, der als Webapplikation realisiert ist. Dieser kann in diversen Webcontainern wie Tomcat<sup>4</sup> oder JBoss<sup>5</sup> ausgeführt werden. Alternativ bringt Jenkins einen einfachen Webcontainer 'Winstone' mit, der auch in Produktionssystemen eingesetzt werden kann. In einem minimalen CI-System überwacht der CI-Server das Versionskontrollsystem. Bei Änderungen an den Dateien holt sich der CI-Server alle Quelltexte zum Kompilieren. Die Ergebnisse stellt Jenkins auf einer Weboberfläche zur Verfügung. In der typischen Konfiguration besteht ein CI-System aus weiteren Komponenten. Durch einen E-Mail-Dienst können ProjektmitgliederInnen über den Ausgang eines Builds informiert werden. Auch wenn Jenkins weitere Informationskanäle unterstützt, ist E-Mail immer noch der Standard für Benachrichtigungen. Durch das Benutzerverzeichnis können BenutzerInnen authentifiziert werden und Rechte in Jenkins nach Anmeldung gesteuert werden. Auch die Integration eines Ticket-Systems ist möglich. In Verbindung mit dem Versionskontrollsystem werden die Informationen der geänderten Dateien des Entwicklers und ein Link auf den Jenkins-Build an dem Ticket eingetragen. Dadurch entsteht eine

<sup>3</sup><https://jenkins.io/><sup>4</sup><http://tomcat.apache.org/><sup>5</sup><http://www.jboss.org/>

## 2 Grundlagen

Transparenz, welche Änderungen für einen Fehler bzw. ein Feature gemacht wurden [Wie11]. Die Information ist sehr hilfreich, wenn die gleichen Änderungen später in einer älteren Version der Software benötigt werden, um z. B. einen Fehler auf einem Kundensystem zu beheben.

Jenkins kann die anstehenden Aufträge auf mehrere Rechner verteilen. Bei langen Buildzeiten können Builds parallel ablaufen. Gleichzeitiges Kompilieren in unterschiedlichen Umgebungen wie Mac und Windows sind ebenfalls möglich. Jenkins kann auf Software-Repositories lesend und schreibend zugreifen. Im Build kann auf Komponenten von Drittanbietern oder eigenen im Software-Repository abgelegten Versionen zugegriffen werden. Ein bekanntes Software-Repository ist Artifactory<sup>6</sup>, das in der Entwicklung des Workflow-Systems eingesetzt wird.

### 2.10 Continuous Delivery

Die Erstellung eines Releases hat das gleiche Problem wie die Integration von Änderungen in der Software. Mit Continuous Delivery wird der gleiche Ansatz wie beim Continuous Integration verfolgt. Statt eine große Änderung am Ende zum Laufen zu bringen, sollte eine Auslieferung der Software regelmäßig durchgeführt werden. Dafür ist ein automatischer Aufbau der Testumgebungen notwendig. Die Tests sollten nicht kurz vor Release durchgeführt werden. Eine regelmäßige Ausführung der Tests führt zu einer hohen Zuverlässigkeit. Änderungen an der auszuliefernden Software und der notwendigen Infrastruktur müssen nachvollziehbar sein. Dadurch ist gewährleistet, jeden Stand der Software und Infrastruktur zu rekonstruieren. Nach Modifikationen entsteht viel Aufwand Fehler in schon getesteten Softwareteilen zu vermeiden (Regressionen). Jede Modifikation kann in einer beliebigen Stelle des Systems einen Fehler verursachen. Aus dem Grund sollten nach jeder Änderung alle Tests ausgeführt werden. Dieser Anspruch ist nur mit automatischen Tests zu bewerkstelligen. Durch die häufigen automatischen Testdurchläufe wird die Qualität der Software gesteigert. Bei Continuous Delivery verkürzen sich die Feedback-Zyklen. Ein Entwickler-Team erhält Feedback nach jedem Durchlauf der Continuous Delivery Pipeline. Continuous Delivery erweitert das Vorgehen von Continuous Integration auf die in Abbildung 2.7 dargestellten weiteren Phasen. Nach dem Prinzip 'Stop the Line' wird nur bei einer erfolgreich absolvierte Phase die Nächste gestartet. Je weiter ein Software-Stand durch die Pipeline gelaufen ist, desto wahrscheinlicher ist es, dass eine Änderung keine Regressionen verursacht hat [BL16a].

---

<sup>6</sup><https://www.jfrog.com/open-source/>

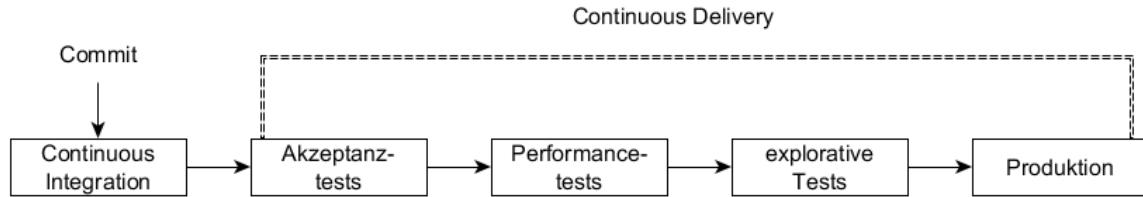


Abbildung 2.7: Erweiterung Continuous Integration

Die Commit Phase wird durch die Continuous-Integration-Infrastruktur abgedeckt. Nach einer Änderung an der Software wird der Code kompiliert, Unit-Tests und statische Code-Analysen werden ausgeführt. In der ersten Phase nach Continuous Integration werden Umgebungen aufgebaut. Dort werden Akzeptanztests durch eine automatische GUI-Interaktion oder automatische Tests durch Aufruf der API ausgeführt. In Akzeptanztests wird das Einhalten von Akzeptanzkriterien überprüft [BL16a]. Durch ständige Wiederholungen der Tests sollen Regressionen vermieden werden. Im Performancetest wird automatisch festgestellt, ob die Software ausreichend leistungsfähig ist. Im Performancetest und der statischen Code-Analyse werden die nicht funktionalen Anforderungen der Software getestet [BL16a]. Beim explorativen Testen gibt es keinen festen Testplan. Die TesterInnen sind Domänenexperten mit dem Fokus auf die neuen Features und deren Usability. In dem Schritt der Produktion wird der selbe Prozess wie schon im Aufbau der Testumgebung durchlaufen. Beim 'In-Produktion-Bringen' der Software ist das Risiko höher. Ein Rollback auf eine stabile Version sollte deshalb möglich sein. Eine Installation vor Ort beim Kunden sollte allerdings nicht nach jedem Continuous Delivery Durchlauf erfolgen, um die Kunden nicht zu oft zu einem Update zu zwingen. Für Akzeptanz-, Performance- und explorative Tests muss eine Infrastruktur zur Verfügung gestellt werden. Durch Installationsprogramme gibt es schon seit längerem die Möglichkeit, ein System zu konfigurieren und die Software zu installieren. Häufig treten Probleme auf, wenn sich ein System schon auf dem Rechner befindet. Ein Updateprozess muss mit vorhandenen Dateien und Verzeichnissen zurecht kommen und es müssen ggf. überflüssige Elemente, die nicht mehr benötigt werden, entfernt werden. Update-Skripte werden dadurch aufwendiger. Durch eine komplettene Neuinstallation kann dieses Problem umgangen werden. Soll eine Software nur neu installiert werden, eignet sich Docker hervorragend eine neue Umgebung aufzubauen. In Kapitel 2.12.4 wird auf die Vorteile von Docker gegenüber einer Virtuellen Maschine eingegangen. Alternativ kann zur Neuinstallation beschrieben werden, wie ein System nach der Installation aussehen soll statt anzugeben, welche Aktionen ausgeführt werden müssen. Zugriffsprobleme auf schon vorhandene Konfigurationsdateien werden bei einem Update vermieden, indem der Inhalt und die Zugriffsrechte verglichen werden. Ein Überschreiben mit notwendigem Neustart der Anwendung oder des Betriebssystems kann verhindert werden. Die Installation ist idempotent und kann beliebig wiederholt werden [Wol15a]. Bei ursprünglicher

## 2 Grundlagen

Vorgehensweise der Auslieferung bekommt ein Kunde eine Version erst nach Fertigstellung der Software. Erst dann stellt sich heraus, ob an den Kunden vorbei entwickelt oder die neuen Features angenommen wurden. Durch Continuous Delivery können kleine Features innerhalb von Tagen bei den Kunden installiert werden. Dies ermöglicht auch Designvarianten zu erproben. Die Entwicklung kann dichter am Kunden stattfinden und es kann schneller auf Änderungen am Markt reagiert werden. Der Gewinn einer Software kann dadurch gesteigert werden [BL16a].

### 2.11 Gradle

Für die Java-Welt haben sich die Build-Werkzeuge Ant<sup>7</sup> und Maven<sup>8</sup> etabliert. Das Build-Werkzeug ermöglicht das Kompilieren des Source-Codes, Ausführen und Auswerten der Unit-Tests und Erzeugen der Artefakte (z. B. JAR-Dateien). In Ant wird imperativ der Build Schritt für Schritt von den EntwicklerInnen implementiert. Bei Maven wird hingegen ein deklarativer Ansatz verfolgt. Für viele Aspekte des Builds werden Konventionen angenommen, die eine Konfiguration sehr einfach machen. Bei abweichenden Konventionen oder geänderten Konfigurationen, werden Maven Build-Skripte oft komplexer als mit anderen Build-Werkzeugen. Das relativ neue Build-Werkzeug Gradle<sup>9</sup> versucht das Beste aus Ant und Maven zu vereinen. Gradle übernimmt den Convention-over-Configuration-Ansatz von Maven. Wird von den Konventionen abgewichen, können wie bei Ant flexibel eigene Abläufe definiert werden. Dies erfolgt mit einer Gradle-DSL (Domain Specific Language), die auf der Programmiersprache Groovy basiert. EntwicklerInnen haben die Möglichkeit, Groovy-Code einzubetten und somit beliebige Funktionen zu programmieren. Groovy lehnt sich an Java an. Jedes Java-Programm ist auch ein gültiges Groovy-Programm, wobei Groovy eine wesentlich einfachere Syntax ermöglicht. Gradle unterstützt inkrementelle Builds und führt nur Tasks aus, wenn Änderungen gemacht wurden. Für die Automatisierung von Builds ist der Gradle-Wrapper sehr nützlich. Die verwendete Gradle-Version kann mit in die Versionskontrolle aufgenommen werden. Wird das Projekt von einem Entwickler oder einem Continuous-Integration-Server ausgecheckt, sind alle notwendigen Werkzeuge zum Bauen des Projekts vorhanden. Das separate Installieren und Pflegen der Version ist nicht notwendig. Es wird sichergestellt, dass alle die gleiche Version des Werkzeugs verwenden. Der Build ist damit reproduzierbar [Wol15a]. Vor einem Build wird die Abhängigkeitsstruktur der Tasks als Directed Acyclic Graph (DAG) aufgebaut und danach erst ausgeführt. Dies ermöglicht ein deterministisch sequenzielles und inkrementelles Abarbeiten des Builds. Es wird verhindert, dass Tasks mehrfach ausgeführt werden. Gradle unterstützt Maven-Repositories und lädt Artefakte im POM (Project Object Model) ins zentrale Artifactory. Die Unit-Tests laufen in einer

---

<sup>7</sup><http://ant.apache.org/>

<sup>8</sup><https://maven.apache.org/>

<sup>9</sup><https://gradle.org/>

eigenen JVM (Java Virtual Machine), um Speicherproblemen bei wiederholter Ausführung zu umgehen. Dazu kann die JVM nach  $n$  ausgeführten Tests mit Setzen der Variable forkEvery neu gestartet werden [HG12].

```
test {
forkEvery=25
}
```

## 2.12 Virtualisierung

Durch eine Servervirtualisierung sollen Ressourcen effektiver und flexibler genutzt werden. Erreicht wird dies durch Konzepte und Technologien, mit denen eine abstrakte Sicht auf eine heterogene physische Ressourcenlandschaft erzeugt wird [BKL09]. Einer der Gründe für den Einsatz von Servervirtualisierung ist, dass häufig die CPUs auf Servern nur 2-5% ausgelastet sind. Wenn 10 Maschinen ihre Hardware zusammen nutzen könnten, kann die Auslastung auf 20-50% gesteigert werden. Es werden Kosten für Hardware und Energie von neun Rechnern eingespart. Bisher haben viele Firmen traditionell ihre Mailserver, Webserver, FTP-Server und andere Server auf separaten Computern laufen, manchmal mit unterschiedlichen Betriebssystemen. Weiterhin soll vermieden werden, dass der Absturz eines Systems die anderen Systeme mit beeinflusst [Tan09]. Neben den Einsparungen kann in virtuellen Maschinen ein System geklont und schnell etwas ausprobiert werden. Der Zustand nach dem Test kann übernommen oder ohne Probleme wieder entfernt werden.

Das Betriebssystems des Hostrechners behält den exklusiven Zugriff auf die Hardware. Das Gast-Betriebssystem kann deshalb, bis auf wenige Ausnahmen, keinen direkten Zugriff auf die Hardware erhalten. Das Betriebssystem der virtuellen Maschine findet eine andere Hardware als im tatsächlichen Rechner vor [Göp15]. Es gibt drei Konzepte der Virtualisierung:

- Vollvirtualisierung
- Paravirtualisierung
- Containervirtualisierung

### 2.12.1 Vollvirtualisierung

Bei der **Vollvirtualisierung** stellt das Host-Betriebssystem der virtuellen Maschine die physische Hardware durch Gerätetreiber als virtuelle Hardware zur Verfügung. Am Gast-Betriebssystem brauchen keine Änderungen vorgenommen zu werden, um in der isolierten Umgebung ausgeführt zu werden [BBKS08]. Das Gast-Betriebssystem erhält über den Hypervisor (Virtual Machine Monitor VMM) Zugriff auf die Ressourcen. Die Virtualisierung in der x86-Architektur lässt sich anhand des Schutzkonzepts der Ringe betrachten. Aktuelle Prozessoren erhalten zur Stabilität und des Schutzes vier Privilegienstufen zur Ausführung

## 2 Grundlagen

von Befehlen. Ein Prozess kann nur in einem Ring ausgeführt werden und ist nicht in der Lage diesen Ring zu verlassen. Nur der Ring 0 erhält den vollen Zugriff auf die Hardware. Die Zugriffsrechte nehmen mit den außen liegenden Ringen ab. Im Ring 3 werden in der Regel die Anwendungen ausgeführt. Da die meisten Prozessorarchitekturen nur 2 Ringe verwenden, nutzen weit verbreitete Betriebssysteme wie Linux nur den Ring 0 und Ring 3. Der Ring 0 wird häufig als Kernel-Bereich und der Ring 3 als Benutzerbereich bezeichnet. Ruft ein Prozess in einem weniger privilegierten Ring einen privilegierten Befehl auf, erzeugt der Prozessor eine Ausnahme. Diese wird im benachbarten inneren Ring behandelt. Kann eine Ausnahme nicht behandelt werden, führt dies zu einer allgemeinen Schutzverletzung und der aufrufende Prozess stürzt ab. Bei der Virtualisierung befindet sich der Hypervisor in Ring 0 auf der Ebene des Betriebssystemkerns. Das Gast-Betriebssystem befindet sich auf einem weniger privilegierten Ring. Der Hypervisor stellt für alle möglichen Ausnahmen eine Behandlung zur Verfügung, welche die privilegierten Befehle des Gast-Betriebssystems abfängt, interpretiert und ausführt. Das Gast-Betriebssystem erhält nur über den Hypervisor Zugriff auf die Ressourcen [BKL09]. Das Host-Betriebssystem läuft im Kernmodus (kernel mode) und erhält Zugriff auf die gesamte Hardware. Die restliche ausgeführte Software läuft im Benutzermodus (user mode), in der nur eine Teilmenge der Maschinenbefehle verfügbar ist. Das Betriebssystem einer virtuellen Maschine läuft ebenfalls nur im Benutzermodus. Für die Implementierung einer virtuellen Maschine muss die CPU virtualisierbar sein (Popkek und Goldberg 1974) [PG74]. Will das Gastbetriebssystem einen privilegierten Befehl im Kernmodus ausführen, dann muss die Hardware den Virtual Maschine Monitor aufrufen, damit der Befehl durch die Software nachgebildet werden kann. Diese Aufrufe werden von einigen CPUs z. B. dem Pentium und seinen Vorgängern ignoriert. AMD und Intel haben 2005 eine Virtualisierung auf ihren CPUs eingeführt. Für ein Gast-Betriebssystem wird ein Container erzeugt, in dem es so lange läuft, bis eine Ausnahmesituation erzeugt wird, die einen Sprung in den Hypervisor auslöst. Die Menge der Befehle, die einen solchen Sprung auslöst, wird von dem Hypervisor als Hardware-Bitmap gesetzt [Tan09]. Damit eine Virtualisierung mit x86-Prozessoren ohne Virtualisierungstechniken funktioniert, kommt ein sogenannter Typ-2-Hypervisor zum Einsatz. Kritische Befehle werden von dem Typ-2-Hypervisor erkannt und ausgetauscht. Die veränderten Befehle führen bei Ausführung zu einem Sprung in den Hypervisor, der die kritischen Befehle emulieren kann. Diese Art der Virtualisierung ermöglicht die Verwendung älterer Prozessoren, ist allerdings nicht so effizient [Man14].

Im Gegensatz zum Typ-2-Hypervisor, der auf einem vollwertigem Host-Betriebssystem aufsetzt, verwendet VMWare vSphere für ihre Virtualisierungslösung einen Typ-1-Hypervisor. Der auch als *Bare-Metal* bezeichnete Hypervisor läuft direkt auf der Hardware im Kernel-Modus. Damit wird nur eine Transaktion benötigt. Im Gegensatz dazu braucht der Typ-2-Hypervisor einen Zwei-Layer-basierten Prozess. Der Typ-1-Hypervisor muss die Hardware selbst durch Treiber unterstützen, da kein Host-Betriebssystem vorhanden ist. Damit wird

eine höhere Performance erreicht [PB11].

### 2.12.2 Paravirtualisierung

Bei der **Paravirtualisierung** werden auf einem Betriebssystem zusätzliche Betriebssysteme virtuell gestartet. Die Virtualisierung bzw. die Emulation von Hardware entfällt. Die Ressourcen werden effizient genutzt. Die Performance des virtualisierten Systems ist wie die der physischen Maschine. Der Nachteil ist, dass das Gast-Betriebssystem angepasst werden muss [BBKS08]. Das Gastsystem läuft im weniger privilegiertem Ring 1. Der Hypervisor befindet sich im Ring 0 und liefert dem Gast-Betriebssystem den Zugriff auf die physischen Ressourcen [BKL09].

### 2.12.3 Containervirtualisierung

Bei der **Betriebssystemvirtualisierung**, die auch als **Container** oder **Jail** bezeichnet wird, laufen unter einem und demselben Betriebssystemkern mehrere von einander abgeschottete Systemumgebungen. Es wird kein zusätzliches Betriebssystem erzeugt, sondern es entstehen isolierte Laufzeitumgebungen in virtuell geschlossenen Containern. Der Vorteil der Betriebssystemvirtualisierung ist der geringe Ressourcenbedarf und eine hohe Performance, da der Betriebssystemkern in seiner gewohnten Weise die Zugriffe auf die Ressourcen verwaltet. Der Nachteil ist, dass alle virtuellen System dasselbe Betriebssystem mit der gleichen Version verwenden müssen [BKL09].

### 2.12.4 Docker

Docker<sup>10</sup> ist eine Containervirtualisierung und bietet eine wesentlich effizientere Alternative der Virtualisierung als z. B. VMWare. Im Gegensatz zur virtuellen Maschine nutzen alle Docker-Container das darunter liegende Betriebssystem. Im System existiert nur eine Instanz des Kernels. Hingegen werden Prozesse, Dateisystem, Netzwerk und Benutzer in jedem Container getrennt verwaltet. Der Overhead ist im Gegensatz zur virtuellen Maschine wesentlich geringer. Ohne Probleme können mehrere hundert Container auf einem einfachen Rechner betrieben werden. Ein Docker-Container startet viel schneller, da kein Betriebssystem gebootet werden muss, sondern nur ein neuer Prozess gestartet wird. Ein Container benötigt nur seine eigene Konfiguration der Betriebssystem-Ressourcen und hat lesenden Zugriff auf das Dateisystem (Basis-Image). Im Container können weitere Dateisysteme eingeblendet werden, auf die schreibend zugegriffen werden kann. Es werden nur die Änderungen an den Dateien gespeichert. Der benötigte Speicherbedarf auf der Festplatte wird dadurch deutlich reduziert. Die geringe Trennung der Container kann sich auch negativ auswirken. Tritt ein Fehler im Kernel auf, würde sich das Problem auf alle Container auswirken [Wol15a].

---

<sup>10</sup><https://www.docker.com/>

## 2 Grundlagen

Jeder Docker-Container beruht auf einem Image. Um einen Container zu starten, muss ein Image aus einer öffentlichen Registry heruntergeladen oder ein Image selbst erstellt werden. Ein Docker-Image besteht aus mehreren Dateisystemlagern. Jeder Layer lässt sich für gewöhnlich den einzelnen Buildschritten des Images zuordnen. Mit Dockerfiles können die Images erstellt werden. Diese enthalten die erforderlichen Buildschritte zum Erzeugen eines Images. Die Images werden in öffentlichen oder privaten Registries gespeichert und ermöglichen den Docker-Hosts auf diese zuzugreifen [KK16].

Docker wurde bisher nur von Linux unterstützt. Mit Boot2Docker<sup>11</sup> ist es allerdings möglich, auf Windows oder Mac OSX Docker zu verwenden. Bei der Installation von Boot2Docker wird die Virtualisierungssoftware VirtualBox<sup>12</sup> auf dem System installiert, damit ein Basis-Docker Linux auf einer virtuellen Maschine installiert werden kann [Roß14].

Microsoft hat erkannt, dass die Container-Technologie immer wichtiger wird und hat in Zusammenarbeit mit den Docker-EntwicklerInnen an einer Docker-Einbindung für Windows Server 2016 gearbeitet. Hervorzuheben ist, dass es sich nicht um eine parallel entwickelte Technologie handelt, sondern die gleiche Codebasis wie bei Linux verwendet wird. Die Docker-EntwicklerInnen haben dafür den notwendigen Zugriff auf den Windows-Code erhalten [Joo15]. In der Windows-Containertechnologie wurden zwei Arten von Containern entwickelt: Windows-Server-Container und Hyper-V-Container. Die Instanzen der Windows-Server-Container teilen sich den Kernel des Hosts. Bei den Hyper-V-Containern findet zusätzlich eine Isolation auf Kernelebene statt, in dem jeder Container in einem speziellen virtuellen Computer ausgeführt wird [Pet16]. Der Hyper-V-Container wird von den Docker-EntwicklerInnen entwickelt. Der Vorteil dieser Art der Implementierung ist eine bessere Isolierung der Container. Die Hyper-V-Container sind sicherer und flexibler als die Windows-Server-Container [Joo15]. Beide Arten von Containern können auf die gleiche Weise erstellt und verwaltet werden. Sie unterscheiden sich nicht in der Funktionsweise [Pet16].

---

<sup>11</sup><http://boot2docker.io/>

<sup>12</sup><https://www.virtualbox.org/>

## 3 Analyse des Ist-Zustandes

In diesem Kapitel wird die Funktionsweise des Continuous Integration Builds des Workflow-Systems beschrieben. Ein wichtiger Aspekt für die Build-Pipeline sind die Laufzeiten des Kompilierens und der Unit-Tests. Die Funktionsweise des Testautomaten wird untersucht und der Ablauf der automatischen System-Tests geschildert. Am Ende werden zu lösende Probleme zum Erreichen der Ziele aufgeführt. Begonnen wird mit der tabellarischen Übersicht aller identifizierten Verbesserungspotentiale (Tabelle 3.1). Die Abbildung 3.1 stellt die Probleme des Ist-Zustandes an einer Continuous Delivery Pipeline grafisch dar.

Problem	Beschreibung
I	Die Ausführungszeit der Unit-Tests muss optimiert werden, da die CI-Builds zu lange dauern
II	Die Lesbarkeit der Unit-Tests ist zu verbessern, damit Fehlerursachen schneller analysiert werden können
III	Eine Continuous Delivery Pipeline erfordert eine breite Basis an Unit-Tests. Für vorhandenen Code existieren noch zu wenige Unit-Tests (Testabdeckungsgrad 28 %). Der Legacy-Code ist von der Architektur schlecht für Unit-Tests geeignet.
IV	Die Analyse aufgetretener Fehler in den automatischen Tests ist zu aufwendig
V	Die Erstellung neuer automatischer Tests muss vereinfacht werden
VIa,b	Die Paketierung und die Installation dauern zu lange
VIc	Die Ausführungszeit der automatischen Tests ist zu hoch
VID	Erweiterung der Continuous Integration- auf eine Continuous Delivery Pipeline

Tabelle 3.1: Verbesserungspotentiale für Continuous Integration und Testautomatisierung

### 3.1 Ist-Zustand Continuous Integration und Testausführung

Kernstück des Workflow-Systems ist historisch bedingt das Prepress-Projekt, mit der die Software vor über 15 Jahren begonnen wurde. Für das Projekt existiert ein automatischer Continuous Integration Build, der nach jeder Änderung des Quellcodes ausgeführt wird. Die System-Tests werden dann allerdings erst am nächsten Tag für alle getätigten Änderungen ausgeführt, da der Installer des Workflow-Systems nachts gebaut wird. Für alle anderen

### 3 Analyse des Ist-Zustandes

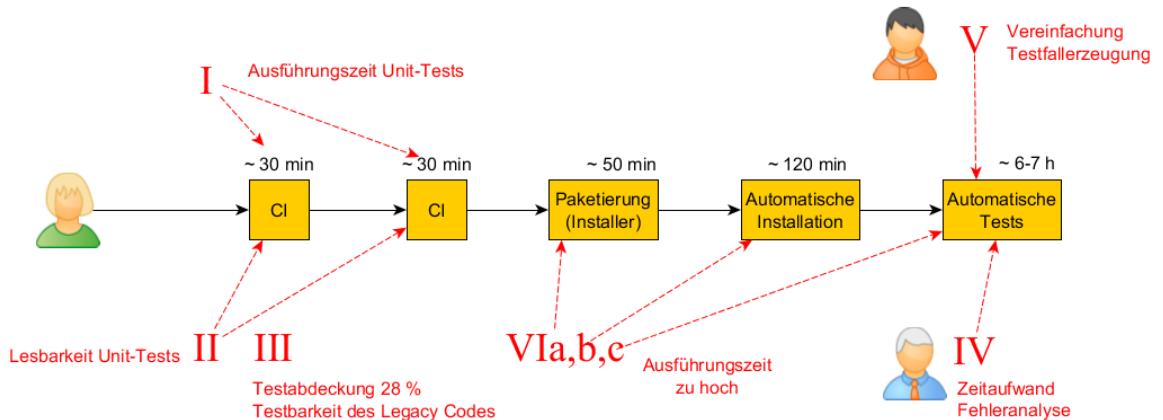


Abbildung 3.1: Identifizierte Probleme bei Einführung einer Continuous Delivery Pipeline

Komponenten existiert ebenfalls ein automatischer CI-Build mit Ausführung der Unit-Tests. Im Gegensatz zum Prepress-Projekt werden die Build-Artefakte nicht automatisch, sondern nach Entscheidung des zuständigen Entwicklers in das Artifactory geladen. Im CI-Build des Prepress-Projekts erfolgt die Integration aller Komponenten. Einmal am Tag wird ein CI-Build mit den neuen Versionen aus dem Artifactory durchgeführt. Die BuildmanagerInnen entscheiden welche Änderungen übernommen und somit in das Prepress-Projekt integriert werden. Der aktuelle Ablauf von Continuous Integration ist in Abbildung 3.2 dargestellt. Die 4 EntwicklerInnen (links unten) committen ihre Änderungen in das Git-Repository. Der Jenkins-Master bemerkt die Änderung und startet den CI-Build der 1. Stufe. Anschließend werden die Änderungen auf dem Developer-Branch eingecheckt und alle anderen EntwicklerInnen sehen den neuen Softwarestand. Mit den Änderungen auf dem Developer-Branch wird ein CI-Build (2. Stufe) durchgeführt. Läuft dieser Build fehlerfrei durch, werden die Änderungen für den Master-Branch und damit für den nächtlichen Build übernommen. Während dieses CI-Builds laufen die Änderungen 2 und 3 durch die 1.CI-Stufe. Nachdem der 1.CI-Build (CI 1) der 2. Stufe durchgelaufen ist, startet der 2.CI-Build (CI 2,3) der 2. Stufe mit den Änderungen 2 und 3. Der CI-Build (CI 4) wird durch die Übernahme von geänderten Komponenten aus dem Artifactory durch die BuildmanagerInnen ausgelöst. Die externen Komponenten werden in das Prepress-Projekt integriert und damit in den Installer des Workflow-Systems übernommen. Am Ende der Pipeline muss ein Entwickler am nächsten Tag die gemeldeten Fehler der automatischen Tests analysieren und herausfinden, welche Änderungen zu dem Fehlverhalten führen.

#### 3.1.1 Verfügbare Server

Die gesamte Serverlandschaft für Build- und Testrechner besteht aus 20 Servern mit jeweils 512 GB Ram. Festplattenzugriff bekommen die Server über ein mit Fibre Channel verbundenes

### 3.1 Ist-Zustand Continuous Integration und Testausführung

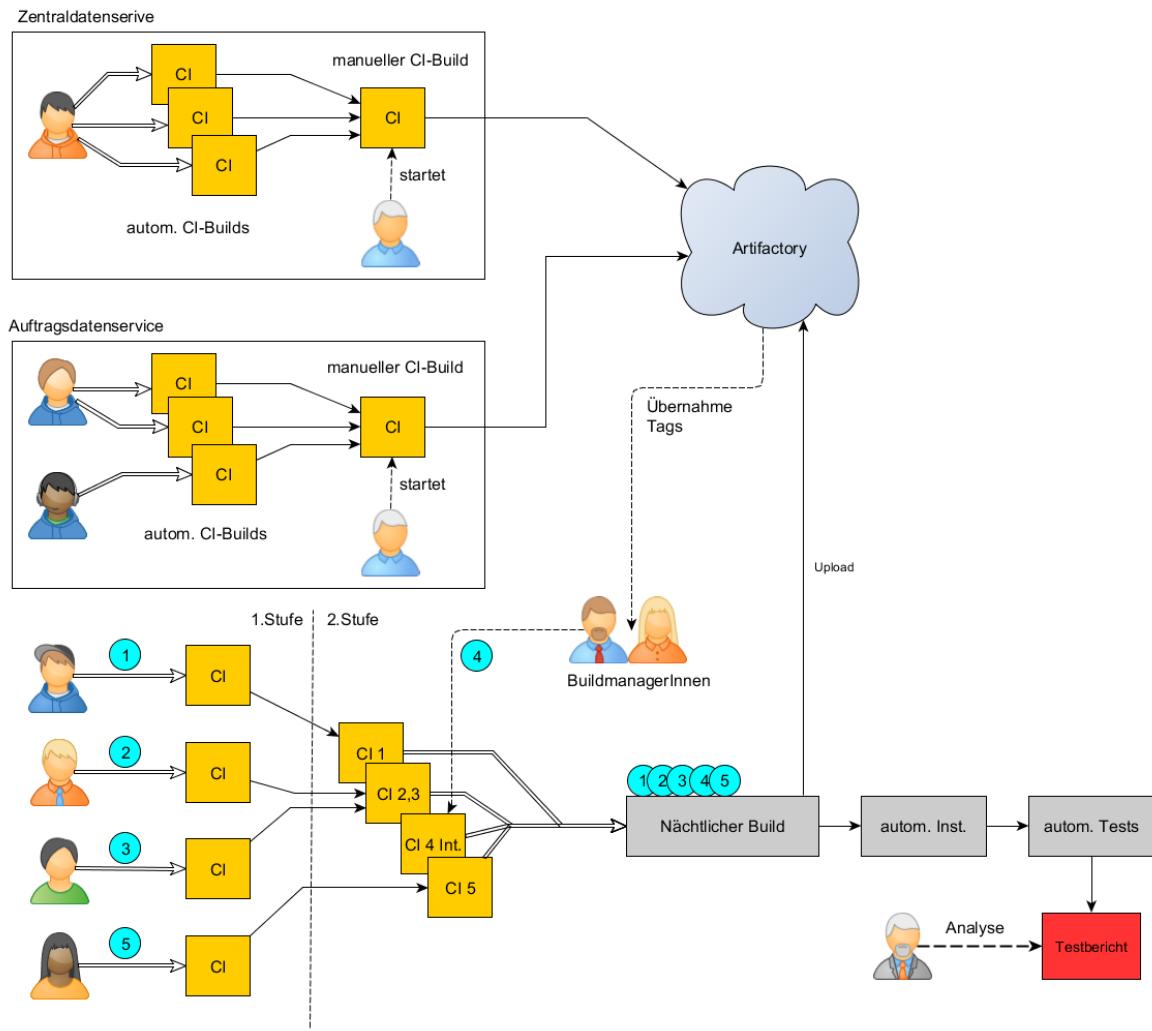


Abbildung 3.2: Ist-Zustand Continuous Integration

Storage Area Network (SAN). Es sind ca. 1000 virtuelle Maschinen konfiguriert, die mit Virtualcenter<sup>1</sup> (vCenter) von VMWare verwaltet werden. 400-500 virtuelle Maschinen sind gleichzeitig gestartet. Für Continuous Integration stehen ein Jenkins-Master (Linux), vier Jenkins-Windows-Slaves (Windows Server 2012) und zwei Jenkins-Mac-Slaves zur Verfügung. Der Jenkins-Master verteilt die Jenkins-Jobs auf die Slaves. Der Build-Server für nächtliche, vollständige Builds und einige Testrechner sind eigenständige physikalische Server außerhalb des Virtualcenters.

<sup>1</sup><https://www.vmware.com/de/products/vcenter-server>

### 3.1.2 Laufzeiten

Das Prepress-Projekt besteht aus 133 einzelnen Komponenten. Die restlichen Komponenten der Systemlandschaft werden von dem Continuous-Integration-Server als Artifactory angezogen. Die durchschnittliche Gesamtdauer des Continuous Integration Builds beträgt etwa 30 Minuten (Abbildung 3.3) und enthält nur die Änderungen eines Commits. Erst wenn der CI-Build fehlerfrei durchgelaufen ist, werden die Änderungen auf dem Developer-Branch eingecheckt und damit für alle EinwicklerInnen sichtbar. Somit ist gewährleistet, dass ein fehlgeschlagener Build die restlichen EntwicklerInnen nicht stört. Die Ausführung der Unit-Tests erfolgt parallel und hat eine Gesamtlaufzeit von ca. 2 Stunden. Dies ist der kumulierte Wert von allen CPU-Kernen des Build-Servers. In Jenkins kann der Anteil der Unit-Tests von der gesamten Buildzeit nicht einfach abgelesen werden. Das Build-Tool Gradle führt die Unit-Tests für jede fertig kompilierte Komponente aus. Die Reihenfolge der Komponenten ist durch deren Abhängigkeiten zueinander definiert und erfolgt teilweise ebenfalls parallel. Alle Komponenten verfügen zusammen über 4349 Unit-Tests. Bei Erfolg werden alle bisher aufgelaufenen Änderungen mit einem weiteren CI-Build kompiliert und in den Master-Branch eingecheckt. Die Build-Dauer ohne Unit-Tests beträgt 17-18 Minuten.

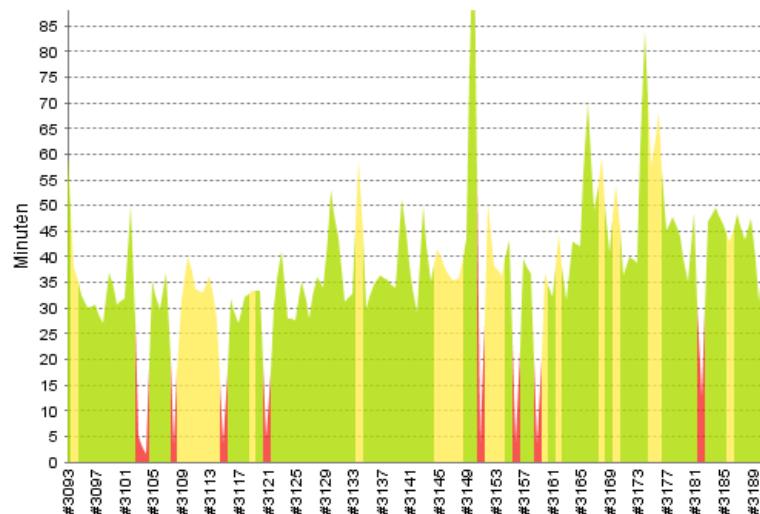


Abbildung 3.3: Durchschnittliche Dauer der CI-Builds

Nach Änderungen am Code wird die Auswertung der Softwaremetriken auf einem Jenkins-Slave gestartet. Die Ausführungszeit ist von den geänderten Komponenten abhängig und liegt zwischen 23 und 95 Minuten. Da die Metriken keinen Einfluss auf die Übernahme von Änderungen in den nächtlichen Build haben, werden diese alle 2 Stunden ausgeführt und befinden sich somit zurzeit außerhalb einer Pipeline.

### 3.1.3 Messung der Codequalität

Alle zwei Stunden werden Codeanalysen in einem separaten Jenkins-Slave durchgeführt. Es werden die Jenkins Plugins Checkstyle<sup>2</sup>, Findbugs<sup>3</sup> und Crap4J<sup>4</sup> für die statische Codeanalyse und JaCoCo<sup>5</sup> für die Bestimmung der Testabdeckung verwendet.

Checkstyle wurde ursprünglich entwickelt, um die Einhaltung eines einheitlichen Layouts zu überprüfen. Im Laufe der Weiterentwicklung kann u.a. ein problematisches Klassen-Design erkannt werden. Findbugs sucht in den kompilierten .class-Dateien nach über 400 typischen Fehlermustern, wie die Verwechselung von Variablen oder den falschen Einsatz von boolschen Variablen [Wie11]. Crap4J (Change Risk Analysis and Predictions) kombiniert die zyklomatische Komplexität mit dem Testabdeckungsgrad. Danach ist das Änderungsrisiko einer komplexen Methode ohne Testabdeckung am höchsten. Die Analyse liefert eine Liste, in der die Klassen im absteigenden Änderungsrisiko aufgeführt sind.

Die Bestimmung der Testabdeckung erfolgt mit der Open-Source-Bibliothek JaCoCo. Die Testabdeckung wird in einem HTML-Report dargestellt (siehe Abbildung 3.4). Es wird der Testabdeckungsgrad u. a. von Zeilen, Klassen und Methoden aufgeführt. Der Wert M steht für vom Test nicht abgedeckt und C für abgedeckt.



Abbildung 3.4: Ergebnis einer Testabdeckungsanalyse

In dem CI-Build des Workflow-Systems haben die Metriken nur einen informativen Charakter. Die Ergebnisse werden nicht dazu genutzt den Build bei Verschlechterung der Metriken scheitern zu lassen. Jenkins verschickt nur eine Mail an die betroffenen EntwicklerInnen.

Bevor die Metriken mehr Beachtung finden, sollte zuerst ein Ziel definiert werden, was mit dem Erfassen bestimmter Daten erreicht werden soll. Andernfalls könnte man durch Erfassen aller denkbaren Kennzahlen in einen 'Metrikwahn' verfallen. Wird nur gemessen um

<sup>2</sup><http://checkstyle.sourceforge.net/>

<sup>3</sup><http://findbugs.sourceforge.net/>

<sup>4</sup><http://www.crap4j.org/>

<sup>5</sup><http://eclemma.org/jacoco/>

### 3 Analyse des Ist-Zustandes

die Zahlen verfügbar zu haben, führt dies zum Aufwand ohne Nutzen. Die Akzeptanz sinkt und es kann der Eindruck der Überwachung entstehen [BK13].

Eines der wichtigsten Schwerpunkte der statischen Codeanalyse sollte die Überwachung der tatsächlichen Struktur sein. Denn strukturelle Probleme sind ein häufiger Grund für das Scheitern von Softwareprojekten. Probleme wie unerwünschte Abhängigkeiten (z. B. direkter Zugriff von der GUI- in die Datenpersistenzschicht) werden vom Compiler nicht erkannt. In großen und komplexen Systemen ist es für viele EntwicklerInnen sehr schwierig zwischen erwünschten und unerwünschten Abhängigkeiten zu unterscheiden. Mit wachsender Systemgröße ist es immer schwieriger Architekturverletzungen zu erkennen [Zit07].

### 3.2 Nächtlicher Build

Nachdem tagsüber alle Änderungen stetig integriert wurden, werden in einem nächtlichen Build die Prepress-Komponenten erzeugt und ins Artifactory hochgeladen. Mit den neuen Versionsständen wird dann ein Installer gebaut. Dieser enthält die über den Tag eingespielten Änderungen des Continuous-Integration-Builds und die zuvor integrierten Komponenten (siehe Abbildung 3.5). Die Installation setzt einen Windows Server 2008RC2 oder 2012 voraus, wobei 2012RC2 empfohlen wird. Nachts wird die neue Version automatisch auf zwei Test-Systemen installiert und anschließend werden alle automatischen System-Tests ausgeführt.

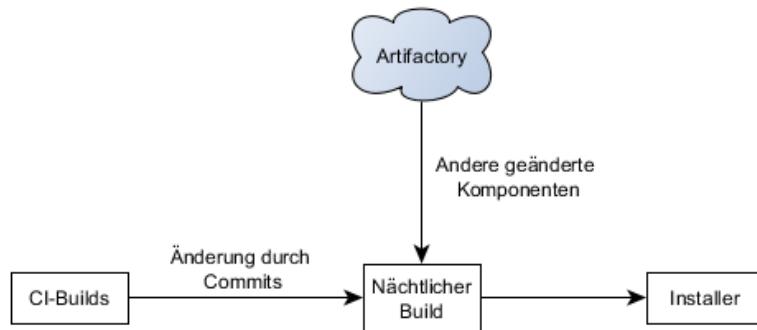


Abbildung 3.5: Erstellung des Installers im nächtlichen Build

### 3.3 Analyse des Testautomaten

Zur Ausführung von Autotests existiert ein in Java programmiertes Script-Server. Dieser kommuniziert mit allen notwendigen Services im System. Der Script-Server kann von der Funktionalität wie ein Client agieren. Ein automatischer Test wird mit der Programmier-

sprache Python geschrieben. Der Script-Server verwendet das OpenSource-Projekt Jython<sup>6</sup>, um Python-Code auf der Java-Plattform auszuführen. Java-Klassen können importiert und ausgeführt werden. In einem Projekt für die automatischen Tests (AutoTestLib.jar) werden alle notwendigen Java-Funktionen, die zum Ausführen eines Tests benötigt werden, implementiert. Das Projekt dient als Schnittstelle zwischen der API für die Python-Scripte und dem Java-Produktionscode des Workflow-Systems. Ein automatischer Test wird ausgeführt, in dem das Python-Script in das Hotfolder-Verzeichnis des Script-Servers kopiert wird. Das Verzeichnis wird von dem Script-Server überwacht und bei eintreffen einer neuen Datei wird diese geladen, interpretiert und ausgeführt. Die Ergebnisse der ausgeführten Autotests werden in einer HTML-Datei tabellarisch aufgelistet. Die Zeilen sind je nach Status (Fehler: rot, Warnung: gelb und OK: grün) farblich hinterlegt. Detaillierte Informationen zu einem Test werden durch Klicken auf den Namen in einer weiteren HTML-Seite dargestellt. Es sind zwei Script-Server gestartet, die die Tests teilweise parallel ausführen. Einige in den Tests verwendeten Komponenten können konfiguriert werden. Die Parameter der Konfiguration werden für unterschiedliche Testfälle verändert. Damit sich die Testfälle nicht gegenseitig beeinflussen, werden diese nach einander ausgeführt. Im Script-Server ist eine Logik programmiert, die anhand der Zeilen des Python-Scripts entscheidet, welche Tests nicht parallel ausgeführt werden können.

Nach einem erfolgreichen nächtlichen Build werden die Autotest-Scripte in den Hotfolder des Script-Servers kopiert und der Reihe nach abgearbeitet. Um die Verlässlichkeit der Test-Ausführung zu untersuchen, wurden 30 der nächtlichen Buildläufe kontrolliert und von diesen sind nur 16 Builds fehlerfrei durchgelaufen. Bei den fehlerfreien Builds musste häufig manuell eingegriffen werden, um die automatische Installation abzuschließen. Die Konsequenz daraus ist, dass die automatischen Tests nicht über Nacht durchgelaufen sind. Dies ist für eine zeitige Rückmeldung nach Änderungen vom Vortag nicht ausreichend. Manuelle Eingriffe sind für eine automatisch ablaufende Continuous Delivery Pipeline nicht tragbar.

Das Ausführen der automatischen Tests dauert ca. 7 Stunden und beginnt nach einem fehlerfreien Build um 2 Uhr nachts. Kommt es bei der Installation der neuen Version zu Verzögerungen, werden nur 100 von 239 Tests ausgeführt. Der Testlauf mit reduzierten Autotests enthält nicht die zeitaufwendigen sondern nur die Tests, die sehr schnell sind oder essentiell für das Testen der Basisfunktionalität sind. Dadurch wird gewährleistet, dass möglichst schnell Aussagen zur Qualität der neuen Version getroffen werden können. Bei schwerwiegenden Software-Problemen kann es vorkommen, dass automatische Tests gar nicht ausgeführt werden können bzw. gleich beim Starten scheitern. Dieser Zustand muss zum nächsten Tag beseitigt werden. Weitere Probleme wurden ggf. noch gar nicht entdeckt, so dass die Probleme um einen weiteren Tag verzögert sichtbar werden.

Die Untersuchung der in automatischen Tests aufgetretenen Fehler ist sehr zeitintensiv

---

<sup>6</sup><http://www.jython.org/>

### *3 Analyse des Ist-Zustandes*

und erfordert sehr gute Kenntnisse über das Gesamtsystem.

## **3.4 Performance- und Stresstests**

Die Performance des Workflow-Systems wird mit einem Client manuell im Scrum-Sprint-Rhythmus getestet, indem Zeiten für definierte Aktionen festgehalten und mit vorherigen Versionen verglichen werden. Für die Tests wird immer der selbe Server ohne Virtualisierung verwendet. In der virtualisierten Serverlandschaft müsste mit probaten Mitteln für eine gleichmäßige Verfügbarkeit der Ressourcen gesorgt werden. Wichtigstes Mittel ist die exklusive Verfügbarkeit der I/O-Performance.

In unregelmäßigeren Abständen werden Stresstests durchgeführt. Dazu wird neben der Schnittstelle für die automatischen Tests auch die automatisierte Bedienung des Webportals verwendet. Letztes simuliert BedienerInnen unter Verwendung des Selenium WebDrivers<sup>7</sup>. Für die erste Version einer Continuous Delivery Pipeline werden die nichtfunktionalen Tests nicht berücksichtigt und sollten aber in der Zukunft integriert werden.

## **3.5 Ableitung zu lösender Probleme**

### **3.5.1 Ausführungszeit des CI-Builds - Problem I**

Ausführungszeiten der Commit Stage (Build + Unit-Tests) sollte so schnell sein, dass EntwicklerInnen sich noch nicht in die nächste Aufgabe vertieft haben [BL16b]. Die Ausführungszeit des CI-Builds ist mit fast 30 Minuten deutlich zu lang. Angestrebt werden Ausführungszeiten von 5-10 Minuten. Das Kompilieren des Quellcodes kann nur durch Einsatz besserer Hardware beschleunigt werden. Potential liegt bei den Ausführungszeiten der JUnit-Tests. Problematisch sind JUnit-Tests, die Dateien laden statt die Abhängigkeiten zu anderen Klassen durch Stellvertreter-Objekte zu ersetzen. Zu untersuchen ist, ob der Einsatz besserer Hardware bzw. eine Optimierung der Virtuellen Maschine zu einer Steigerung der Performance führt. Die Ausführung und ggf. die Implementierungen der JUnit-Tests müssen für eine kürzere Ausführungszeit optimiert werden.

### **3.5.2 Lesbarkeit der Unit-Tests - Problem II**

Die in den Komponenten der Workflow-Software existierenden Unit-Tests sind teilweise schwer zu lesen. Die Test-Methoden sind sehr lang und die Vorbereitung der Testdaten, das Ausführen der zu testenden Methoden und die Überprüfung sind in keiner klaren Reihenfolge implementiert. Der Testzweck ist schwer nachzuvollziehen. Vermutlich wurden einige Tests unter dem Zwang erstellt, eine Funktionalität absichern zu müssen.

---

<sup>7</sup><http://www.seleniumhq.org/projects/webdriver/>

Unit-Tests können neben ihrem Testnutzen auch als Dokumentation für EntwicklerInnen dienen. An übersichtlichen und einfachen Tests mit einem klaren Fokus, lässt sich durch das Lesen der Tests die Funktionalität der getesteten Klasse gut erkennen. Nach Markus Gärtner [Gär16] können Teams ihr Wissen in Tests festhalten. Klassische Anforderungsdokumente wie Architektur- und Designspezifikationen können mit der Zeit veralten und haben nicht mehr viel mit dem Anwendungsverhalten zu tun. Unit-Tests dokumentieren das Softwaredesign und in automatisierten Akzeptanztests wird das Domänenwissen und die Kenntnis der Architektur festgehalten.

### 3.5.3 Testbarkeit des Legacy-Codes - Problem III

Wie in Kapitel 2.3 beschrieben, sollten Unit-Tests die größte Basis in den Testebenen einnehmen. Ein im Unit-Test gefundener Fehler tritt schon frühzeitig im CI-Build auf und kann direkt einer Änderung eines Entwicklers zugeordnet werden. Alle Fehler, die gleich am Anfang der Continuous Delivery Pipeline durch Unit-Tests gefunden werden, brauchen nicht aufwendig analysiert zu werden (siehe Kapitel 3.5.4). Dadurch kann der Aufwand der Analyse (**Problem IV**) entschärft werden. Um den Vorteil der Unit-Tests besser zu nutzen, muss der Testabdeckungsgrad erhöht werden. Dazu wird es nicht ausreichen, nur für neu implementierten Code, Unit-Tests zu erstellen. Für die vorhandene Codebasis ist es allerdings häufig nicht einfach Tests zu schreiben. Oft ist der Code nicht modular programmiert. Eine Klasse bzw. Methode erledigt zu viele Dinge. Nach dem *Single-Responsible-Principle* [Mar13] sollte eine Klasse nur für einen Zweck zuständig sein. Tests, die nur einen Zweck überprüfen sind einfacher zu implementieren und zu verstehen. Tests für den vorhandenen Code zu schreiben ist zeitaufwendig und verschlechtern die Lesbarkeit der Unit-Tests (**Problem II**). Werden Unit-Tests zu einem späteren Zeitpunkt geschrieben, wird es oft zum Problem, dass diese sehr groß und schlecht wartbar werden. Durch eine zu hohe Kopplung lässt sich der Code schlecht isoliert testen [DM11b].

Durch statische Codeanalysen wie CheckStyle und FindBugs werden Fehler im Quelltext gefunden. Wichtig ist es aber auch, die zyklischen Abhängigkeiten laufend zu kontrollieren. Während eines Projekts kann die Architektur durch unerwünschte zyklische Abhängigkeiten 'Degenerieren'. Man spricht auch von der 'Erosion' der Architektur. Zyklische Abhängigkeiten verschlechtern die Testbarkeit des Codes, da die Zyklenteilnehmer nicht mehr getrennt voneinander getestet werden können. Da alle Zyklenteilnehmer berücksichtigt werden müssen, ist das Erreichen einer guten Testabdeckung erheblich schwieriger. Das Codeverständnis wird erschwert, da alle Zyklenteilnehmer verstanden werden müssen. Das von 'unten nach oben' Hocharbeiten wird verhindert [Zit07].

### 3.5.4 Fehleranalyse automatischer System-Tests - Problem IV

Die Analyse von den in Autotests entdeckten Fehlern ist sehr zeitaufwendig. Der personelle Aufwand wird von der Entwicklung geleistet und reduziert die Zeit für Feature- und Testimplementierungen. Gesucht wird eine Möglichkeit, die Zeit für eine Analyse zu reduzieren. Es wäre weiterhin vorteilhaft, dass die analysierende Person über kein so fundiertes Fachwissen für das Gesamtsystem verfügen müsste.

Ein Grund für den hohen Zeitaufwand bei der Analyse von gescheiterten Tests ist die Aussagefähigkeit der Fehlermeldung. Wird ein Test nach der Fertigstellung eines Features implementiert, so wird dieser für den funktionierenden Fall angewendet. Häufig wird keine weitere Zeit investiert, um mögliche Fehlerfälle zu testen und aussagekräftige Meldungen zu generieren.

Änderungen an externen Komponenten aus dem Artifactory sind oft einige Tag alt (siehe Kapitel 3.1) wenn diese integriert werden. Da die automatischen Test nur einmal am Tag ausgeführt werden, werden alle geänderten externen Komponenten auf einmal in die Software eingebunden. Diese Art der Integrationsstrategie wird Big-Bang-Integration genannt. Beobachtete Fehler lassen sich schwer lokalisieren, da viele Änderungen auf einmal in die Software eingeflossen sind [Hof13]. Das zeitnahe Ausführen der System-Tests nach Änderungen ist Gegenstand von **Problem VI**. Dies vereinfacht eine Analyse extrem, weil die Ursache eines Fehlers genauer und schneller eingegrenzt werden kann.

### 3.5.5 Vereinfachung der Test-Erstellung - Problem V

Die Erstellung von automatischen Tests ist nur den wenigen EntwicklerInnen möglich, die sich mit der Programmiersprache Python und dessen Verwendung in Java beschäftigt haben. Auch diese EntwicklerInnen können mit Java effizienter Implementierungen als mit Python umsetzen. Es ist ein Konzept zu entwickeln, bei dem automatische Tests einfacher und von allen EntwicklerInnen implementiert werden können. Nach Erfahrungen von Tobias Getrost [Get16] ist einer der zentralen Aspekte bei der Einführung einer Continuous Delivery Pipeline, dass alle EntwicklerInnen im Team schnell und einfach einen Testfall generieren können.

Die Erstellung neuer Testfälle sollte wie das Entwickeln neuer Features möglich sein. In der Regel sollten bei Implementierungen neuer Features diese auch durch automatische System-Tests überprüft werden. Nach Birk und Lukas [BL16c] sollte das Erstellen von Tests mit im Entwicklungsprozess verankert und Know-How zur Erstellung von automatischen Tests aufgebaut werden, um mit einer Continuous Delivery Pipeline erfolgreich zu sein. Dies betrifft neben der einfacheren Erstellung der automatischen System-Tests auch die Lesbarkeit (**Problem II**) und die Realisierbarkeit (**Problem III**) von Unit-Tests. Das Schreiben von Tests sollte integraler Bestandteil auf allen Stufen von Continuous Delivery sein. Dabei ist es von Vorteil, wenn die Erstellung von automatischen Tests so einfach wie möglich ist. Nur so kann der Testabdeckungsgrad der Software nachhaltig gesteigert werden.

### 3.5.6 Einführung von Continuous Delivery - Problem VI

Der jetzige Ablauf der automatischen Tests muss erweitert werden, um am selben Tag nach Änderungen Feedback der Testdurchläufe zu erhalten. Dazu ist es notwendig, das Paketieren der Software (**Problem VIa**) und die Installation der Software (**Problem VIb**) deutlich zu beschleunigen. Nach dem aktuellen CI-Durchlauf müssen ein oder mehrere Systeme auf den aktuellen Softwarestand gebracht werden, um dort die automatischen Tests auszuführen. Die Ablaufgeschwindigkeit der automatischen Tests muss ebenfalls deutlich gesteigert werden (**Problem VIc**), da das Feedback der Testergebnisse für die Änderungen an der Software sonst viel zu spät kommt. Der Ablauf der einzelnen Stufen der Pipeline soll in Jenkins visualisiert werden. Am Ende der automatischen Tests muss Jenkins das Ergebnis zur Auswertung der Pipeline mitgeteilt werden. Nur wenn die Continuous Delivery Pipeline bis einschließlich der automatischen Tests fehlerfrei durchgelaufen ist, wird die Software-Version als geprüft freigegeben (**Problem VIId**).



# 4 Konzepte zur Verbesserungen

## 4.1 Verbesserung CI-Build

### 4.1.1 Ausführungszeit der JUnit-Tests

Wie in Kapitel 3.5.1 beschrieben, ist die Ausführungszeit der Unit-Tests (**Problem I**) im CI-Build für einen schnellen Feedback zu lange. Um das Problem zu entschärfen, wird die Ausführung der Unit-Test und deren Inhalte untersucht. Unit-Tests in Modul- und Integrationstests zu unterscheiden, ist gar nicht so wichtig, da der Übergang fließend ist. Ziel ist es, die langsamen von den schnellen Tests zu unterscheiden. Angestrebgt wird eine Buildzeit von 10 Minuten. Länger dauernde Unit-Tests sollten in der Continuous Delivery-Pipeline in einer weiter hinten liegenden Phase ausgeführt werden. Ein Unit-Test sollte nur wenige Millisekunden dauern.

In dem Workflow-System existieren Unit-Tests, die lesend und teilweise schreibend auf Dateien auf der Festplatte zugreifen. Es handelt es sich um Auftragsdaten im JDF-Format (Job Definition Format siehe Kapitel 2.1). Für einen Unit-Test sind solche Aktionen zu zeit-aufwendig. Einige Tests laden die gesamte Test-Systemkonfiguration. Damit werden im Test eine große Anzahl von abhängigen Klassen getestet. Die Spitzenreiter in der Ausführungszeit benötigen bis zu einer Minute.

Damit es bei konkurrierenden Zugriffen auf statische Variablen zu keinen Problemen kommt, werden alle Unit-Tests in einer eigenen Java-VM gestartet. Dadurch lädt jeder Unit-Test die Systemkonfiguration neu, falls diese benötigt wird. Das war bisher eine schnelle und einfache Lösung, die Probleme mit den konkurrierenden Zugriffen zu beheben. Die steigende Anzahl der Unit-Tests führt aber durch das mehrfache Laden der Test-Systemkonfiguration zur deutlichen Verlangsamung der Gesamtlaufzeit.

Im Continuous-Integration-Server werden die Unit-Tests mit Gradle isoliert vom Buildprozess in einer eigenen JVM ausgeführt. Durch Properties (gradle.properties) kann der Testprozess beeinflusst werden (siehe Kapitel 2.11). Per Default werden die Tests nicht parallel ausgeführt, dadurch braucht bei der Teststellung auf die gegenseitige Beeinflussung keine Rücksicht genommen zu werden. Mit der Property *maxParallelForks* wird die maximale Anzahl gleichzeitig laufender Testprozesse gesetzt. Durch die Property *forkEvery* wird die Anzahl, der in einem Prozess ausgeführten Tests, definiert bis eine neue JVM gestartet wird. Nach Erreichen der maximal ausgeführten Tests wird eine neue JVM gestartet und statische

## 4 Konzepte zur Verbesserungen

Variablen werden neu initialisiert. Per Default werden alle Tests in der selben JVM ausgeführt (`forkEvery = ∞`). Die Gradle-Properties können auch dynamisch zur Laufzeit des Builds gesetzt werden. Zur Beschleunigung der Tests wird in dem Gradle-Buildscript die maximale Anzahl der parallel ausgeführten Unit-Tests auf die Anzahl der verfügbaren CPU-Kerne minus 2 gesetzt. In einer Untersuchung sollten für `forkEvery` unterschiedliche Werte gesetzt werden, um zu prüfen wie sich diese auf die Ablaufgeschwindigkeit und die Stabilität der Unit-Tests auswirken. Für Unit-Tests mit konkurrierenden Zugriffen verbleibt der Wert bei `forkEvery = 1`.

Um die Unit-Tests im CI-Build zu beschleunigen, sollten langsame Unit-Tests weiter nach hinten in die Continuous Delivery Pipeline verschoben werden. Bei den Unit-Tests handelt es sich überwiegend um Tests, die mehr als eine Klasse testen. Nach dem Bauen der Build-Artefakte durch den CI-Build können die langsamsten Tests (Integrations-Tests) ausgeführt werden. Für die Tests werden die Build-Artifakte auf einen anderen Jenkins-Slave kopiert, damit sichergestellt ist, dass der Code nicht noch einmal übersetzt werden muss. Parallel können mit den Build-Artefakten auch die Metriken und die Paketierung ausgeführt werden (siehe auch Kapitel 4.3). Ein Teil der Unit-Tests wird aus dem sequentiellen Bereich der Continuous Delivery Pipeline in den parallelen Bereich verschoben. Damit kann die Gesamtaufzeit der Pipeline verkürzt werden.

Unit-Tests, die ihre Abhängigkeiten nicht mocken, sollten in Zukunft eingeschränkt werden, damit sich die Ausführungszeit nicht signifikant verschlechtert. Besser ist es, die Zeit zu spendieren und Abhängigkeiten durch Stellvertreter-Objekte zu ersetzen. Andernfalls werden die im Unit-Test referenzierten Klassen implizit mit durchlaufen, ohne die Ergebnisse direkt zu überprüfen. Sollte ein Fehler im Unit-Test durch eine referenzierte Klasse auftreten, ist die Fehlerursache nicht direkt erkennbar. Da es sich bei den referenzierten Klassen häufig um Basisfunktionalität handelt, werden diese Klassen sehr häufig von den Unit-Tests aufgerufen und somit unnötig mehrfach durchlaufen. Die Testabdeckung durch reine Unit-Tests wird daher geringer als der gemessene Wert von 28% ausfallen.

Die Umsetzung zur Verbesserung der Ausführungszeit wird in Kapitel 5 bis 5.3 behandelt.

### 4.1.2 Lesbarkeit der Unit-Tests

Eine Verbesserung der Lesbarkeit (**Problems II**) kann durch den internen Aufbau der Unit-Tests und die Verwendung einer Bibliothek für Assertions erreicht werden.

Eine gute Lesbarkeit lässt sich mit der 'Arrange-Act-Assert'-Konvention erreichen. Ein Test wird in drei Bereiche aufgeteilt:

- Arrange: Vorbereitung der Testdaten und Definition des Verhaltens der Stellvertreter-Objekte
- Act: Ausführung der zu testenden Funktion
- Assert: Überprüfung, ob am Ende das erwartete Ergebnis zurückgegeben wurde

Werden kleine, klar definierte Funktionalitäten getestet, können neu hinzugekommene EntwicklerInnen diese leicht nachvollziehen [Wol15a].

JUnit ist so aufgebaut, dass ein Test erfolgreich ist oder scheitert. Zunächst wird der Produktionscode aufgerufen und anschließend mit Assertion-Methoden das erwartete Verhalten überprüft. Neben den Assertion-Methoden `assertTrue` und `assertFalse`, die einen boolschen Ausdruck erwarten, wird in der `assertEquals`-Methode ein erwarteter und ein im Test erstellter Rückgabewert verglichen. Im folgenden Beispiel werden 2 Gleitkommazahlen verglichen. Wegen Rundungsfehlern wird ein Toleranzwert von 0,01 angegeben.

```
Assert.assertEquals(25.3, 21.0, 0.01);
```

liefert die Fehler-Ausgabe:

```
expected<25.3> but was <21.0>
```

Zur Verbesserung der Lesbarkeit wurde Hamcrest<sup>1</sup> entwickelt. Es handelt sich um eine Open-Source Assert-Bibliothek. Assertions werden mit der Methode `assertThat` angegeben. Als letzten Parameter der Methode wird immer ein Hamcrest-Matcher erwartet. Jeder Matcher ist in der Lage sich selbst zu beschreiben und eine aussagekräftige Meldung bei `assertThat` zu erzeugen. Der Matcher beschreibt neben der Erwartungshaltung auch den Grund, warum ein Wert nicht erfüllt wurde. Der Aufruf

```
assertThat(21.0, IsCloseTo(25.3, 0.01))
```

liefert eine aussagekräftigere Ausgabe (siehe Abbildung 4.1).

```
J| java.lang.AssertionError:  
  Expected: a numeric value within <0.01> of <25.3>  
    but: <21.0> differed by <4.290000000000001>  
  | at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)  
  | at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:8)  
  | at de.fhlubeck.lockemar.paper.impl.PaperHamcrestTest.testPaperWidth(PaperHamcrestTest.java:24)
```

Abbildung 4.1: Fehlermeldung des IsCloseTo-Matchers

Zum besseren Verständnis sollte die Assert-Methode von JUnit mit einem zusätzlichen String verwendet werden, in dem der Entwickler des Tests Hinweise beim Scheitern hinzufügen

---

<sup>1</sup><http://hamcrest.org/>

## 4 Konzepte zur Verbesserungen

kann. Wird die Methode `assertTrue` ohne den beschreibenden Text verwendet, steht im Ergebnis des Tests nur, dass dieser fehlgeschlagen ist. Der Hamcrest-Matcher liefert automatisch, ohne weitere Parametrierung, den aktuellen und den erwarteten Wert. In Abbildung 4.2 werden die Ausgaben im Fehlerfall von JUnit und Hamcrest gegenüber gestellt. An der Ausgabe von JUnit ist der Grund des Scheiterns nicht zu erkennen. Der Hamcrest-Matcher liefert hingegen, ohne weitere Parametrierung, eindeutig den Grund für das Scheitern des Tests.

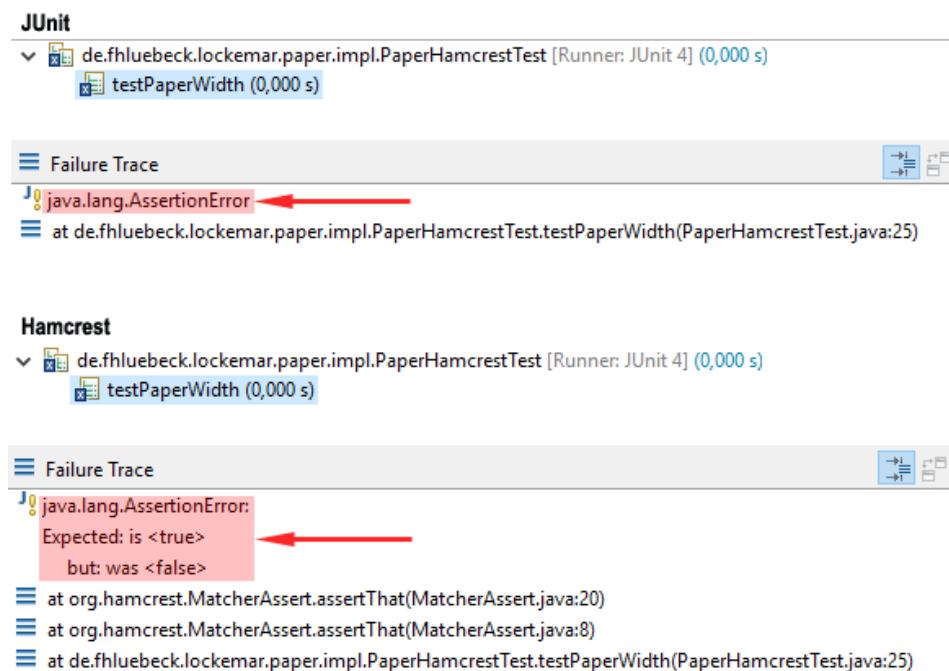


Abbildung 4.2: Gegenüberstellung der Ausgaben von JUnit und Hamcrest

Die Hamcrest Matcher liefern als Ergebnis einen boolschen Wert zurück. Dadurch können die Matcher im Gegensatz zu den assert-Methoden von JUnit negiert und kombiniert werden. In Hamcrest gibt es zahlreiche Matcher, die bei der Bestimmung des Testergebnisses hilfreich sind. Da es Hamcrest-Matcher für viele Zwecke gibt, können diese auch für den Produktionscode eingesetzt werden. Die Funktionalität der Matcher ist ausgiebig getestet und muss nicht aufwendig selbst implementiert werden. U.a. gibt es einen Matcher, der das Vorhandensein von Objekten in einem Array unabhängig von der Reihenfolge überprüft `arrayContainingInAnyOrder(E... items)`.

Zur Verbesserung der Lesbarkeit wird keine Lösung erarbeitet, die das Problem direkt löst. In Kapitel 6 werden Maßnahmen erläutert, wie die Implementierung von Unit-Tests durch die Entwicklung-Teams in Zukunft verbessert werden könnte.

### 4.1.3 Testbarkeit des Quellcodes

Der Code der Workflow-Software ist häufig nicht geeignet, um saubere Unit-Tests zu schreiben und den alten Code mit in die automatische Überprüfung zu integrieren (**Problem III**). Daher stellt sich die Frage, ob es sinnvoll ist für den bestehenden Legacy-Code Tests zu schreiben oder ob eine vorherige Refaktorisierung vorteilhaft ist?

In einer Studie 'Why We Refactor' [STV16] werden 44 Gründe für das Refaktorisieren untersucht. In 6% aller Fälle wurde der Code refaktorisiert, um eine Methode testbar zu machen. Dabei wurde ein Teil des Codes einer Methode in eine separate Methode ausgelagert, um diese in Isolation testen zu können. Unit-Tests geben die Sicherheit, dass beim Refaktorisieren das Verhalten der Software nicht geändert wird. Bei einer Refaktorisierung ohne vorhandene Unit-Tests besteht die Gefahr, dass unentdeckt Fehler entstehen. Der ungetestete Code sollte bei eingeplanten Erweiterungen refaktorisiert werden. Bei neuen Implementierungen müssen sich die EntwicklerInnen mit dem Code auseinandersetzen und diesen verstehen. Zum Zeitpunkt der Erweiterung können EntwicklerInnen mit dem kleinsten Aufwand den Code refaktorieren und Unit-Tests schreiben. Danach kann das neue Feature implementiert werden. Im Kapitel 4.2.3 Architektur-Anpassungen wird das Thema Refaktorisierung von Legacy-Code im Zuge der AutoTestLibrary noch einmal aufgegriffen.

Usch Wildt geht in ihrem Artikel [Wil15] auf den Teufelskreis der Testautomatisierung in Legacy-Systemen ein. Für eine monolithische Software ist es sehr schwer Tests zu schreiben. Andererseits ist eine Refaktorisierung des Codes, um diesen besser testbar zu machen, mit hohem Risiko verbunden. Alt-Systeme können nur durch aufwendige GUI-Tests abgesichert werden. Sukzessiv kann nach der Erstellung von GUI-Tests der Code testbar gemacht werden. Eine Möglichkeit ist es Methoden in riesigen nicht isolierbaren Klassen mit wenig Risiko durch die Refaktorisierung 'Extract Method' von Abhängigkeiten zu befreien. Ein Stück Code, der eine bestimmte Aufgabe erfüllt, wird in eine neue Klasse ausgelagert. Externe Referenzen, wie z. B. globale Variablen, werden aus dem Code entfernt und durch Parameter ersetzt. Der Code an der ursprünglichen Stelle wird durch den Aufruf der neuen Methode ausgetauscht. In einer monolithischen Software fehlt es oft an einer klaren Trennung der Schichten. Für neue Features sollte eine Parallelwelt geschaffen werden, in der die Schichten eingehalten werden. Im Laufe der Zeit wird die alte Welt immer kleiner werden.

Im Workflow-System gab es zu Beginn eine Schichten-Struktur (Abbildung 4.3). Allerdings war der Durchgriff durch die Schichten nicht verboten. Die unerwünschte Abhängigkeit ist in der Abbildung durch den roten Pfeil dargestellt. Das UserInterface kennt die Datenstrukturen der Persistenzschicht (JDFLib). Änderungen der Persistenzschicht ziehen damit auch Änderungen im UserInterface mit sich.

Ein Blick auf die Abhängigkeiten des UserInterface-Projekts im heutigen Stand zeigt, dass zu viele Abhängigkeiten vorhanden sind. Für neue Features sollten die Zugriffe auf die Schichten eingehalten werden. Das UserInterface wird fachlich in die Bereiche *Auftrag* und *Admini-*

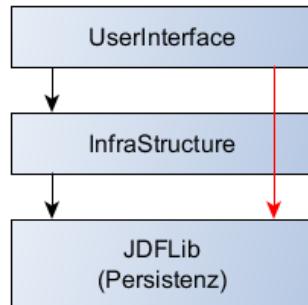


Abbildung 4.3: Anfängliches Schichtenmodell des Workflow-Systems

*stration* aufgeteilt. Für gemeinsame Funktionalitäten gibt es eine *Basis* Komponente (siehe Abbildung 4.4). Die Zugriffe auf die Anwendungslogik erhält das UserInterface über neu definierte Interfaces (siehe auch Kapitel 4.2.3 Architektur-Anpassungen). In dem InfraStructure Projekt sind zwar auch Interfaces definiert, diese enthalten allerdings unerwünschte Abhängigkeiten, die im UserInterface mit übernommen werden müssten. In der Parallelwelt werden neue Interfaces verwendet, die nur die notwendigen Methoden und erlaubten Abhängigkeiten enthalten. Die Implementierung der neuen Interfaces befinden sich in der neuen Komponente *Services* und bleiben vor der neuen Parallelwelt verborgen. Das ursprüngliche Interface leitet sich von dem Neuen ab. Dadurch besteht die Möglichkeit, dass in alten Methoden das neue Interface verwendet werden kann, sofern nur die 'sauberer' Methoden benötigt werden (siehe Abbildung 4.5). Die Klasse aus der alten Welt kann ggf. in die neue Welt verschoben werden. Durch das Aufbrechen der direkten Abhängigkeit vom UserInterface zu den InfraStructure Komponenten durch Einführen einer Interface-Schicht kann die relative zyklische Abhängigkeit von  $rACD=0,36$  auf 0,26 verbessert werden. In Abbildung 4.6 ist die Berechnung vereinfacht dargestellt.

Für die neu geschaffene Parallelwelt muss sichergestellt werden, dass keine unerwünschten Abhängigkeiten entstehen. Werden für eine neue Implementierung in einem der neuen UserInterface-Projekte Funktionalität aus den alten Infra-Projekten benötigt, könnten die EntwicklerInnen in einer Gradle-Dependencies-Datei die Abhängigkeit herstellen. Eine Methode, dies zu verhindern, ist der Einsatz weiterer Metriken-Projekte in Jenkins. Eine schnelle und wirksame Möglichkeit ist die Überprüfung der Abhängigkeiten mit JUnit-Tests. Für die Analyse der Abhängigkeiten kann das Open-Source-Tool JDepend<sup>2</sup> verwendet werden [Tam13]. Die Umsetzung einer Analyse mit JDepend folgt in Kapitel 5.4

In einigen Unit-Tests des Workflow-Systems werden statt eines Mock-Objektes konkrete Objekte instanziert. Nachteil bei diesem Vorgehen ist, dass die indirekten Abhängigkeiten des instanziierten Objektes beim Ausführen des Tests referenziert sein müssen. Werden die

---

<sup>2</sup><http://clarkware.com/software/JDepend.html>

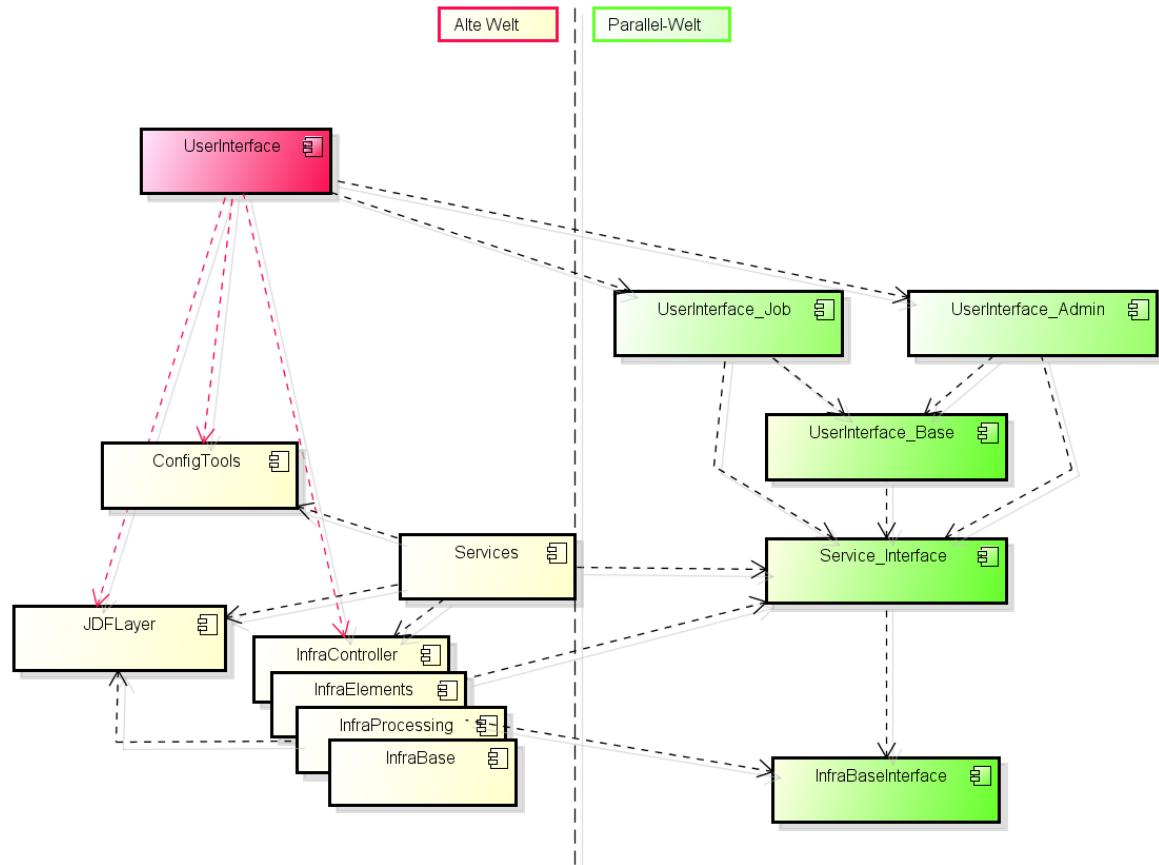


Abbildung 4.4: Neu definierte Interfaces in der Parallel-Welt

Tests aus der Entwicklungsumgebung aufgerufen, würden Tests mit einer NoClassDefFoundException scheitern. Die Abhängigkeiten müssen mit in das Projekt aufgenommen werden. Änderungen in der abhängigen Klasse müssen mit gepflegt werden. Welche Gründe gibt es, konkrete Klassen in den Unit-Tests zu verwenden? Erstens geht die Implementierung schneller, da keine Mock-Objekte erzeugt und an die Methoden angepasst werden müssen. Der wahrscheinlich häufigere Grund ist, dass zu der Klasse kein Interface existiert. Durch Refactorings kann dieser Mangel beseitigt werden, ist aber häufig mit hohem Aufwand verbunden und betrifft ggf. Projekte für die der Ersteller eines Unit-Tests nicht verantwortlich ist.

Eine Alternative bietet EasyMock ab der Version 3.0 an. Neben Interfaces können auch für konkrete Klassen Mock-Objekte erstellt werden. Auch wenn die Möglichkeit besteht konkrete Objekte zu mocken, sollte das Erzeugen von Stellvertreter-Objekten mit Interfaces der Normalfall sein. Bei Erstellung von Tests für Legacy-Code könnte es dennoch Vorteile bringen.

Wird ein Interface nachträglich implementiert, so werden nur die Methoden mit erwünschten

## 4 Konzepte zur Verbesserungen

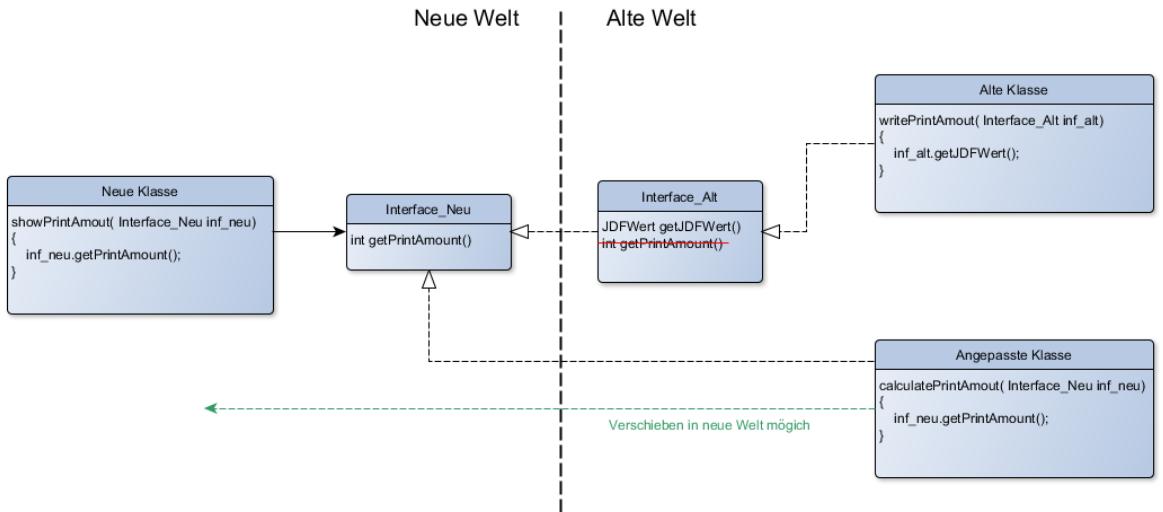


Abbildung 4.5: Komponenten-Anhängigkeiten der alten und neuen Parallel-Welt

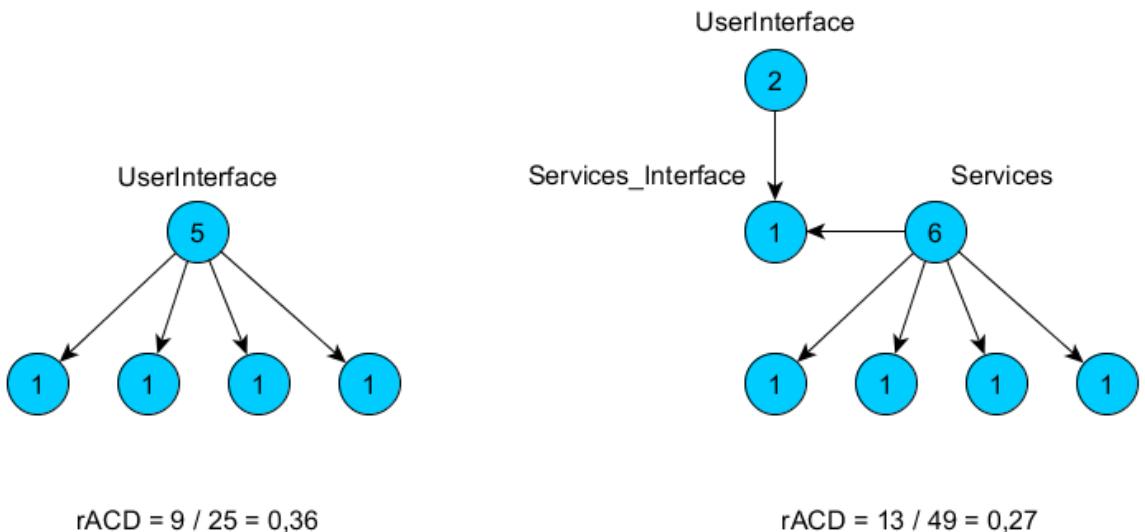


Abbildung 4.6: Verbesserung rACD durch Einführung von Services

Abhängigkeiten definiert. Die restlichen Methoden sind nur am konkreten Objekt verfügbar. Bild 4.7 zeigt ein exemplarisches Beispiel für die Erweiterung der Klasse *Paper*. Das Interface *IPaper* kennt nur die beiden Methoden *getHeight* und *getWidth*, die einen primitiven Datentyp *double* als Rückgabewert haben. Die Methode *getElement* soll dem Interface und damit die Abhängigkeit zu der Klasse *Element* verborgen bleiben. Wird in einer Methode das Interface

*IPaper* als Parameter übergeben, um einen Wert aus dem konkreten Objekt zu bekommen, ist ein Typecast notwendig. Wird *IPaper* für einen Unit-Test mit einer der Mock-Bibliotheken erstellt, so würde der Code hinter dem Typecast nicht durchlaufen werden (siehe Listing 4.1). Die Verwendung von `instanceof`, um Abhängigkeiten zu beseitigen, ist keine gute Vorgehensweise. Vielmehr sollte sich für ein Refactoring die Zeit genommen werden, damit Interfaces ausgereift sind und kein Code für den Übergang entsteht. Miller H. Borges Medeiros [BM14] schreibt in seinem Blog einen Beitrag *Type check is a code smell*. Typecasting macht Code schwieriger zu lesen und zu verstehen. Der Code kann nicht so leicht wiederverwendet werden. In 99% der Fälle können Typecasts durch aufteilen von Methoden verhindert werden.

Es gibt Fälle, in denen es legitim ist, `instanceof` zu verwenden, beispielsweise beim Überschreiben von der `equals`-Methode eines Objekts. `instanceof` wird aber häufig bei einer schlechten Klassen-Architektur eingesetzt. Durch Änderung der Klassenstrukturen kann das Verwenden von `instanceof` vermieden werden [JP04].

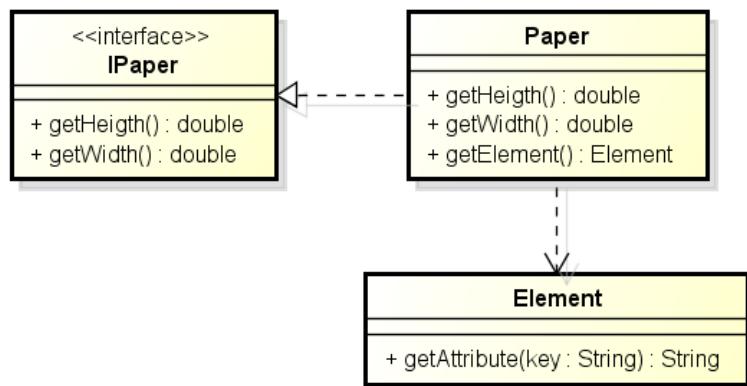


Abbildung 4.7: Nachträgliche Einführung eines Interfaces

Listing 4.1: Unit-Test mit `instanceof`

```

public double getPaperWeight(IPaper ipaper)
{
    double paper_weight = -1.0;
    if(ipaper instanceof Paper)
    {
        Element element = ((Paper)ipaper).getElement();
        String attribut = element.getAttribute("WEIGHT");
        paper_weight = Double.valueOf(attribut);
    }
    return paper_weight;
}
  
```

## 4.2 Verbesserungen des Testautomaten

### 4.2.1 Log-Datei-Analyse

Bei der Untersuchung von gescheiterten automatischen Tests muss häufig die Log-Datei des Script-Servers überprüft werden, um die Ursache eines Fehlers zu erkunden. Treten bei der Ausführung eines automatischen Tests Fehler auf, werden diese i.d.R. in eine Log-Datei geschrieben. Diese Datei enthält chronologisch alle Log-Einträge. Um eine Analyse zu vereinfachen (**Problem IV**), sollte, neben der gesamten Logdatei, für jeden automatischen Test eine eigene Log-Datei existieren. Andernfalls muss bei der Analyse der Ablaufzeitraum eines Tests beim Lesen der Logdatei berücksichtigt werden.

Weiterhin führen auftretende Fehler im Log nicht zwangsläufig zu einem gescheiterten Test. Treten die erwarteten Ergebnisse ein, wird der Test als erfolgreich angesehen. Um die Testtiefe weiter zu erhöhen, könnten die zu einem Test gehörenden Log-Einträge überprüft werden. Befindet sich in den Log-Einträgen ein Fehler, so wird der Test als gescheitert gemeldet. Vor Umsetzung dieser Erweiterung müssten auftretende Fehler in der Log-Datei beseitigt werden, da die fehlschlagenden Tests sonst stark ansteigen würden. Daher wird diese Erweiterung erst mal nicht umgesetzt.

Die Auswertung der Log-Einträge ist wesentlich einfacher, wenn für jeden Test eine eigene Datei existiert. Beim parallelen Ausführen von automatischen Tests würde es nicht zu Überschneidungen kommen. Die Log-Datei wird mit Log4j<sup>3</sup> geschrieben. Mit dem Logging-Framework ist es möglich, parallel in eine weitere Datei zu schreiben (siehe Kapitel 5.5).

### 4.2.2 Vereinfachte Erstellung neuer Tests

Da die Erstellung von automatischen Tests sehr aufwendig ist (**Problem V**), werden diese bisher nur auf Anforderung erstellt. Neue automatische Tests zu implementieren sollte so einfach sein, dass neu entwickelte Features generell durch automatische Tests abgesichert werden. Einer der Gründe für den hohen Aufwand, ist die Verwendung der Programmiersprache Python zur Erstellung von Autotest-Skripten. Viele EntwicklerInnen arbeiten mit Java und besitzen keine Python-Kenntnisse. Erschwerend kommt hinzu, dass die Integration der Entwicklungsumgebung nicht so komfortabel wie bei Java ist. Weiterhin ist das Debuggen des Python-Codes im Script-Server nicht möglich. Das Erstellen von automatischen Tests nimmt einen großen Teil der Entwicklungszeit in Anspruch und wird daher meist unterlassen. Ein weiterer Nachteil der mit Python geschriebenen Autotests ist, dass Refactorings erheblich aufwendiger sind. Der Code der AutoTest-Library wird direkt in Python aufgerufen. Ändern sich Schnittstellen, müssen alle Python-Scripte ohne Hilfe einer Refactoring-Funktion der Entwicklungsumgebung angepasst werden.

---

<sup>3</sup><http://logging.apache.org/log4j/1.2/index.html>

Oft stehen den Software-EntwicklerInnen nicht viel Zeit für die Implementierung von Tests zur Verfügung. Ein Grund ist häufig die mangelnde Unterstützung aus dem Management, auf der anderen Seite handelt es sich aber auch um ein 'hausgemachtes' Problem, da viele EntwicklerInnen allzu optimistisch den Aufwand für die Tests wieder und wieder unterschätzen. Läuft ein Projekt nicht optimal, wird der Rotstift bei den Tests angesetzt [Hof13]. Boris Brodski [Bro15] nennt in seinem Artikel *Automatisierte Tests, die Spaß machen* 3 Gründe, warum EntwicklerInnen ungern automatische Tests schreiben. Zum einen ist die Testentwicklung langweilig und auf langsame Tests zu warten, ist demotivierend. Weiterhin ist es mühselig, kaputte Tests zu reparieren. Es entsteht ein Teufelskreis. Tests sind nur noch 'quick and dirty' programmiert. Das führt zu schlecht designten und langsamen Tests. Sie gehen schnell kaputt und Tests zu reparieren, kostet Zeit. In der Folge wird dies unterlassen. Der Nutzen der Tests sowie die Motivation, weitere zu schreiben, sinkt. EntwicklerInnen erleben Tests zu schreiben, zunächst oft als Ballast, da sie die Entwicklung verzögern. Für neue Features gibt es Anerkennung und Tests halten dabei auf [Wil15]. Viele EntwicklerInnen empfinden es als Last sich mit der unbekannten Programmiersprache Python zu beschäftigen, um automatische Tests zu erstellen. Um das Problem einer nicht bekannten Sprache zu lösen, könnte der Script-Server erweitert werden JUnit-Tests auszuführen. Kent Becks Grundidee von JUnit war es, ein möglichst einfaches Testframework zu erschaffen, um Tests in der genutzten Programmiersprache zu erstellen. Eine aufwendige Erstellung der Testumgebung sollte entfallen [Lin05].

Es existieren zahlreiche JUnit-Tests, die weit mehr als isolierte Klassen testen. Dies untermauert das Verlangen der Entwickler, Tests in Java mit JUnit schreiben zu wollen. Eine verbesserte Lesbarkeit von Unit-Tests (**Problem II**) kommt der Analyse von Fehlern in den automatischen Tests, die mit JUnit erstellt wurden, zugute und kann somit das **Problem IV** der aufwendigen Analyse mildern. Da für Continuous Delivery automatische Tests fest in die Software-Entwicklung integriert sein sollten, könnte die Testerstellung für ein neues Feature mit in die Akzeptanzkriterien einer Story aufgenommen werden. Damit ist gewährleistet, dass Tests nicht nur erstellt werden, wenn es die Zeit zulässt.

Kapitel 5.6 zeigt die Umsetzung JUnit für automatische Tests verwenden zu können.

### 4.2.3 Architektur-Anpassungen

Zur einfacheren Erstellung von automatischen Tests (**Problem V**) wurde die Architektur der Clients und des Testautomaten untersucht. Die Erstellung von Tests ist für Legacy-Code schwieriger. Mängel in der Architektur wurden durch aufwendige Implementierungen in der Autotest-Library umgangen. Nochmaliges Programmieren von Funktionen in der Autotest-Library erhöht den Aufwand der Testerstellung. Weiterhin verhält sich der duplizierte Code für die Autotest-Library unter Umständen anders als der Produktivcode. Beim Verändern des Produktivcodes entsteht zusätzlicher Aufwand den duplizierten Code mit zu ändern bzw.

## 4 Konzepte zur Verbesserungen

es wird vergessen diesen mit anzupassen. Automatische Tests können ggf. scheitern, obwohl sich in dem Produktivcode an dieser Stelle kein Fehler befindet. Die Zuverlässigkeit und die Akzeptanz des Testautomaten sinkt. Bei Testerstellung sollte die Zeit statt für Reproduktion des Produktionscodes in das Refaktorisieren des Codes investiert werden. Häufig ist eine Refaktorisierung von Legacy-Codes in großen Systemen extrem zeitaufwendig, so dass die Planungen für Features vom Zeitrahmen gesprengt werden.

Nach Martin Fowler [Fow00] ist bei der Entwicklung keine extra Zeit für Refaktorisierungen einzuplanen. Vielmehr sollten die EntwicklerInnen refaktorisieren, wenn dies hilft, das Design zu verbessern und den Code verständlicher zu machen. Sinnvoll ist es zu Refaktorisieren und dann neue Features zu implementieren. Während man refaktorisiert, wird die Implementierung verstanden. Es ist wie eine Auseinandersetzung mit dem Code. Auch wenn Vorgesetzte die Refaktorisierung für nicht notwendig halten und die Vorteile nicht kennen oder einsehen sollten, ist den SoftwareentwicklerInnen zu raten Refaktorisierungen durchzuführen. Die SoftwareentwicklerInnen sind Profis in ihrem Fachgebiet, deren Aufgabe es ist, wirkungsvolle Software so schnell wie möglich zu erschaffen. Vorgesetzte erwarten eine schnelle Umsetzung der Aufgaben. Wie dies erledigt wird, ist die Sache der SoftwareentwicklerInnen.

In dem Code des Clients befindet sich Anwendungslogik, die von dem Testautomaten nicht angesprochen werden kann. Der Code der Anwendungslogik ist aus den Client- vor allem aus den GUI-Klassen herauszulösen und in fachlichen Interfaces ansprechbar zu machen. Die Entkopplung von Client-Klassen und Anwendungslogik kann über eine Service-Registry gelöst werden. Die Implementierungen der Interfaces werden beim Starten der Applikation in der Service-Registry eingetragen. Die Service-Consumer, wie die Client-Klassen, erhalten aus der Service-Registry nur die fachlichen Interfaces. Die Implementierungen mit den technischen Details bleibt verborgen. Die Services können ausgetauscht werden ohne die Servicekonsumenten zu ändern. In automatischen Tests kann nun die API verwendet werden, wie sie dem Client zur Verfügung steht. Abbildung 4.8 zeigt den gemeinsamen Zugriff des Clients und des Script-Servers auf die Services. Die Python-Scripte sowie die Java-Autotest-Library des Script-Servers sollen keine Anwendungslogik mehr beinhalten. In den Test-Scripten sollen nur die Bedienung und die Eingaben eines Benutzers simuliert werden. Wird die Anwendungslogik strikt von dem Client-Code getrennt, ist das Erstellen von automatischen Tests einfacher, da die API-Aufrufe schon vorhanden sind und diese nur mit zu definierenden Parametern aufgerufen werden müssen. Die strikte Verwendung von Interfaces führt ebenfalls zu einer Entkopplung des Clients und der Anwenderlogik. Es wird erreicht, dass Änderungen in einer Schicht nicht notwendigerweise andere Schichten betreffen [MT15].

Services sind eine Paradeanwendung für Unit-Tests. Sie sollten in sich abgeschlossen sein und damit ist die Implementierung von Unit-Tests ebenfalls ziemlich autark [Jos08]. Für Tests in Isolation ist es ein großes Hindernis, wenn die Klasse ihre Kollaboratoren selbst erzeugt. Die Abhängigkeiten können nicht durch Stellvertreter-Objekte ausgetauscht werden. Eine Lösung

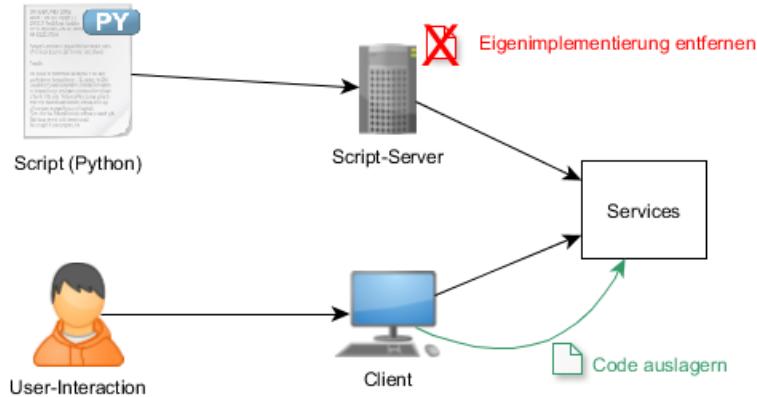


Abbildung 4.8: Gemeinsame Codebasis Client und Script-Server

ist die Injektion der Kollaboratoren durch Dritte, die sich unter dem Namen *Dependency Injection* oder *Inversion of Control* großer Beliebtheit erfreut [Sei12]. Die Client-Klassen instanziiieren keine Services, sondern bekommen die fachlichen Service-Interfaces aus der Service-Registry. Die Implementierung der Service-Registry funktioniert nach dem von Martin Fowler [Fow04] vorgestellten Pattern des Service-Locators. Dependency Injection ist nicht der einzige Weg, die Abhängigkeiten einer Klasse aufzubrechen. Bei der Idee des Service Locators existiert ein Objekt, das alle benötigten Services für eine Applikation beinhaltet. Ein Service Locator kann wie eine Singleton Registry umgesetzt werden. In vielen Beiträgen wird das Service-Locator-Pattern als 'Anti-Pattern' verurteilt, weil dieses die Testbarkeit erschwert. Nach Martin Fowler trifft dies nur zu, wenn das Pattern durch ein schlechtes Design umgesetzt wurde. Der Service Locator ist ein simpler Datencontainer, der einfach für Tests erstellt werden kann. Wird der Service Locator in eine Klasse injiziert, so entfällt der statische Zugriff auf das Service-Locator-Objekt. Das injizierte Interface ist für Tests einfach mockbar. Durch *Dependency Injection* wird die Abhängigkeit Teil der API. Der zu testenden Klasse, wird die Abhängigkeit im Konstruktor oder als Parameter im Methodenaufruf übergeben.

Listing 4.2 zeigt ein Beispiel, dass nur am 15.5.2016 funktioniert. Die Abhängigkeit zum Datum ist in der Klasse gelöst, indem auf eine statische Methode *LocalDate.now()* zugegriffen wird. Von außen kann die Abhängigkeit nicht beeinflusst werden. Durch Injektion des Datums als Parameter der Methode wird die Abhängigkeit von *LocalDate* aus der Klasse *DateProvider* entfernt (Listing 4.3). Die bisherige Methode ohne Parameter ruft die neue Methode mit dem Parameter *LocalDate.now()* auf. Der Produktionscode wurde mit der Erweiterung nicht verändert, aber die Methode ist nun testbar. *Dependency Injection* eignet sich besonders für neu entwickelten Code, bei dem die Schnittstellen noch nicht definiert sind.

## 4 Konzepte zur Verbesserungen

Listing 4.2: Abhängigkeit in der Methode

```
package de.fhlubeck.lockemar.date;

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

public class DateProviderTest
{
    @Test
    public void testGetDateAsString() throws Exception
    {
        String dateStr = DateProvider.getDateAsString();
        Assert.assertEquals("15.5.2016", dateStr);
    }
}
```

Listing 4.3: Abhängigkeit wird injiziert

```
package de.fhlubeck.lockemar.date;

import java.time.LocalDate;
import org.junit.Assert;
import org.junit.Test;

public class DateProviderTest
{
    @Test
    public void testGetDateAsString() throws Exception
    {
        LocalDate dateStub = LocalDate.of(2016, 5, 15);
        String dateStr = DateProvider.getDateAsString(dateStub);
        Assert.assertEquals("15.5.2016", dateStr);
    }
}
```

Statische Zugriffe auf die *Service Registry* sind beim parallelen Ausführen problematisch. Sollen in mehreren Tests Stellvertreter-Objekte für das gleiche Service-Interface in der *Service Registry* registriert werden, kann dies zum Scheitern des Tests führen. Das zuletzt gesetzte Stellvertreter-Objekt wird registriert und überschreibt die Objekte der anderen Tests. Dieses Problem kann nur ausgeschlossen werden, wenn jeder Unit-Test in einer eigenen Java VM ausgeführt wird. Dieser Ansatz geht stark zulasten der Performance. Weiterhin ist es sehr aufwendig, die Unit-Tests, die sich beim parallelen Ausführen gegenseitig beeinflussen, zu identifizieren. Der statische Zugriff auf die *Service Registry* soll durch Injizierung vermieden werden. Dadurch kann der Service-Locator und alle enthaltenen Services durch Stellvertreter-Objekte ersetzt werden. In Listing 4.4 wird der Unterschied zwischen statischem Zugriff und Dependency Injection verdeutlicht. Die beiden Methoden werden aus einem Unit-Test aufgerufen. Für ISomeService wird ein Stellvertreter-Objekt erzeugt, um das Verhalten des

Services vorzugeben. In der ersten Methode wird der Service aus der statischen Klasse geholt. Sollte ein weiterer Unit-Test zu der gleichen Zeit ein Stellvertreter-Objekt für ISomeService in die statische Service-Registry Instanz setzen, wird ein Test scheitern. In der zweiten Methode wurde das Problem durch das Injizieren von ISomeService ausgeschlossen. Die Abhängigkeit der Klasse TestWithService von ISomeService ist anhand der API zu sehen. Abbildung 4.9 stellt die beiden Varianten auf Ebene der Klassen gegenüber. Im oberen Beispiel ist das Beispiel ohne Dependency Injection. Der Code vom Unit-Test muss einen größeren Initialisierungsblock enthalten, um die in der Testklasse verwendeten Services zu registrieren. Das untere Beispiel mit Dependency Injection ist übersichtlicher und einfacher zu verstehen. Die Abhängigkeiten der Test-Klasse sind an der Schnittstelle erkennbar.

Listing 4.4: Service Locator im Unit-Test

```
package de.fhlubeck.lockemar.services;

public class TestWithService
{
    public void callFromTest()
    {

        ISomeService service = ServiceRegistry.getService(ISomeService.class);
        service.doSomething();

    }

    public void callFromTestWithDI(ISomeService service)
    {
        service.doSomething();
    }
}
```

Um das direkte Aufrufen der statischen Klasse *Service Registry* zu verhindern, wird ein Interface *IServiceLocator* (Listing 4.6) definiert. Die implementierende Klasse *ServiceLocator* (Listing 4.7) wrappt die statische *Service Registry*. Das Interface erhält nur die Zugriffsmethode auf die Services. Das Registrieren der Services bleibt ein isolierter Prozess beim Starten der Applikation. Listing 6.1 zeigt die mögliche Implementierung einer *Service Registry*. Es wird sichergestellt, dass nur Interfaces registriert werden können. Um Interfaces zu erkennen, die sich in der Registry befinden, müssen diese von dem Interface *IService* abgeleitet sein. Dies kann mit Java-Generics realisiert werden (siehe Listing 4.5).

Listing 4.5: Einschränkung auf IService

## 4 Konzepte zur Verbesserungen

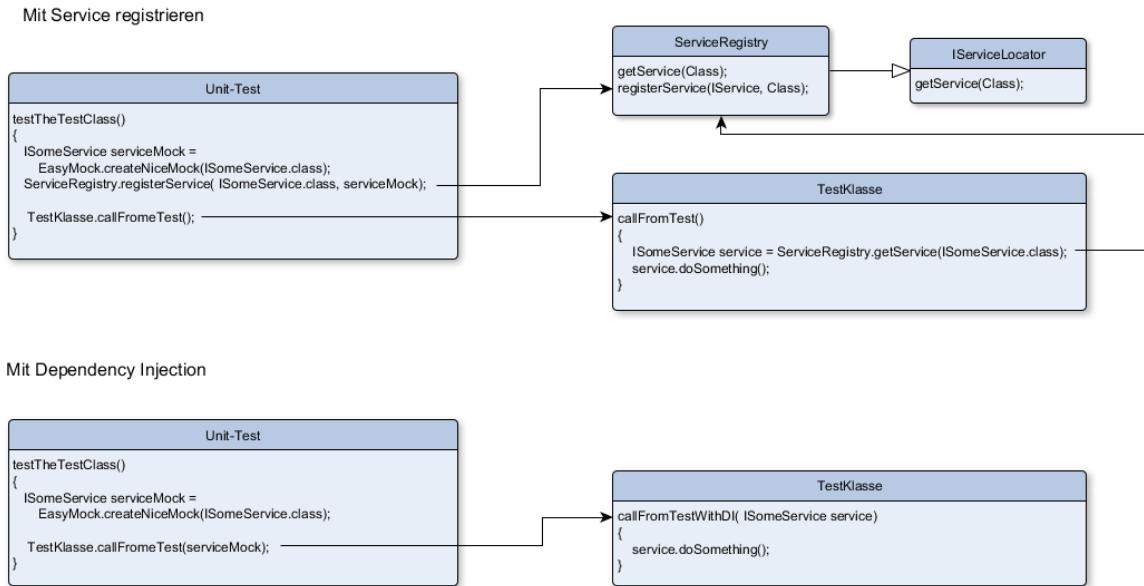


Abbildung 4.9: Unit-Tests mit und ohne Dependency Injection

```
public static <T extends IService> void registerService(final Class<T> serviceClass, final
T serviceToRegister)
{
    // Parameter serviceToRegister muss von IService abgeleitet sein
}
```

Methoden, die das Interface *IServiceLocator* statt des direkten Zugriffs auf die *Service Registry* verwenden, müssen dieses per Dependency Injection gesetzt bekommen. Jeder Test setzt nun seine eigenen Stellvertreter-Objekte, ohne andere Tests zu beeinflussen.

Listing 4.6: Interface IServiceLocator

```
package de.fhlubeck.lockemar.services;

public interface IServiceLocator
{
    <T extends IService> T getService(final Class<T> serviceClass);
}
```

Listing 4.7: Implementierung ServiceLocator

```
package de.fhlubeck.lockemar.services;

public class ServiceLocator implements IServiceLocator
{
    @Override
    public <T extends IService> T getService(Class<T> serviceClass)
    {
        return ServiceRegistry.getService(serviceClass);
    }
}
```

```

    }
}

```

Für automatische Tests könnten spezielle Implementierungen eines Services eingesetzt werden, um z. B. die Laufzeit der Tests zu verbessern. Ein Service mit persistenten Zugriffen auf die Festplatte könnte durch eine In-Memory Variante ausgetauscht werden. In Unit-Tests können die Service-Interfaces sehr einfach durch Mock-Objekte ersetzt werden. Die Autotest-Library soll nur noch Code enthalten, der die Bedienung des Userinterfaces durch Aufrufen der Service-Schnittstellen mit definierten Parametern emuliert.

Alternativ könnten die Tests mit einem installierten Client durch automatische Oberflächen-tests, wie durch einen realen Benutzer bedient, durchgeführt werden. Die Ansteuerung findet auf der höchstmöglichen Ebene der System-Architektur statt. Zusätzlich zu den automatischen Tests, die die gleichen Interfaces der Anwendungslogik aufrufen, wird die Benutzeroberfläche getestet.

GUI-Tests weisen einige Nachteile auf. Eine hohe Testabdeckung ist sehr schwer erreichbar. Weiterhin kann die Ablaufgeschwindigkeit der GUI-Tests nicht beliebig hoch sein. Im schlechtesten Fall muss die Geschwindigkeit so schnell wie eine reale Bedienung durchgeführt werden. GUI-Tests leiden unter Robustheitsproblemen durch z. B. unvorhersehbar auftretende Dialoge. Es ist nicht selten, dass Benutzerschnittstellen im Zuge einer neuen Version geändert werden. Betroffene Tests müssen überarbeitet werden. Der Oberflächentest gehört zu den wartungsintensivsten Varianten der Software-Tests [Hof13]. Nach Kaufman und Hurwitz [KH13] schlagen GUI-Tests häufig auch nach nicht sichtbaren Änderungen fehl. Um Kosten für das stetige Anpassen der Tests zu vermeiden, kann gewartet werden, bis die Änderung im Quellcode vollständig ist. Dieses Vorgehen erhöht das Risiko, dass Probleme nicht schnell genug gefunden werden und somit nicht im gewohnten Projekt-Zyklus behoben werden.

Auch das Workflow-System wird durch einige GUI-Tests abgesichert. Die Tests wurden mit Jubula<sup>4</sup> erstellt. Die Aussage über die Wartungsintensität kann bestätigt werden, da die Tests mehrfach an UI-Änderungen angepasst werden mussten und teilweise mehrere Wochen nicht mehr liefen. Weiterhin gab es häufiger Probleme, das Testframework an die neue Jubula-Version anzupassen. Die Erstellung der Tests ist sehr zeitaufwendig. Besonders für die selbst implementierten UI-Controls. Die GUI-Tests können nur unterstützend erstellt werden und ersetzen nicht die automatischen Tests des Script-Servers.

#### 4.2.4 Parallelisierung

Eine Verringerung der Ablaufzeit der automatischen Tests (**Problem VIc**) kann durch eine parallele Ausführung der Tests erreicht werden. Um die Ablaufgeschwindigkeit der ausgeführten Tests zu erhöhen, wurde vor Jahren die Möglichkeit der parallelen Testausführung

---

<sup>4</sup><http://testing.bredex.de/>

#### 4 Konzepte zur Verbesserungen

durch einen weiteren Script-Server geschaffen. Viele der Tests können allerdings nicht parallel ausgeführt werden. Grund ist eine Komponente im System, die von MIS-Systemen Dateien mit Auftragsinformationen für einen Druck in ihr Eingangs-Verzeichnis (Hotfolder) kopiert bekommt. Die Komponente nennt sich MISConnector (siehe Abbildung 4.10) und erzeugt aus diesen Eingangsinformationen eine JDF-Datei für die weitere Auftragsbearbeitung im Druckvorstufen-System. In einem produktiven System wird der MISConnector mit bestimmten Werten, je nach Arbeitsweise der Druckerei, eingestellt. Die Einstellungen sind während des Betriebs konstant und werden nicht mehr verändert. Unter diesen Voraussetzungen wurde der MISConnector designt und implementiert. Die Einstellungen werden aus einer Konfigurationsdatei ausgelesen und stehen jeder Klasse global zur Verfügung. In den automatischen Tests werden allerdings unterschiedliche Einstellungen verwendet, um bestimmte Testfälle abzudecken. Zu Beginn eines Tests werden die notwendigen Einstellungen geändert und am Ende wieder zurückgestellt. Hier zeigt sich einer der großen Nachteile von statisch globalen Variablen. Tests, die den MISConnector verwenden, müssen die benötigten Einstellungen ändern, damit diese funktionieren. Um sich nicht gegenseitig zu beeinflussen, dürfen die Tests nicht parallel ausgeführt werden.

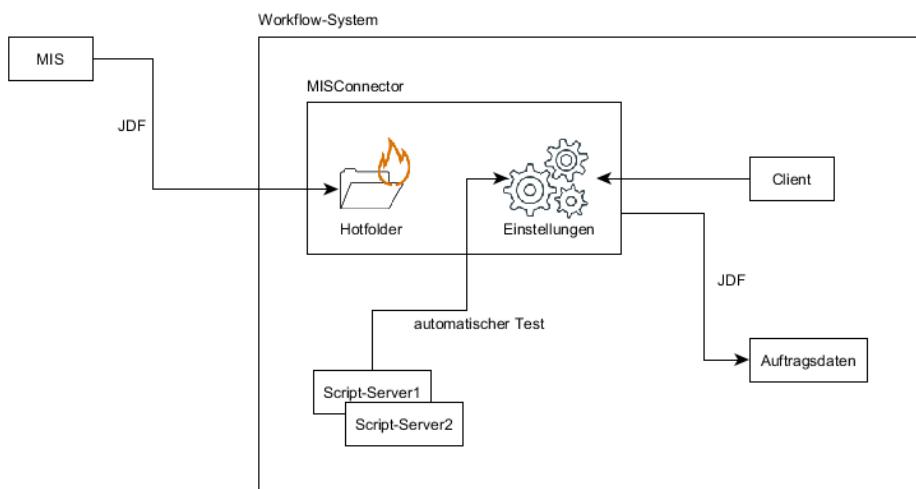


Abbildung 4.10: Verwendung der MISConnector-Komponente

Ein Lösungsansatz wäre den globalen Zustand zu beseitigen und die Konfiguration stattdessen mit *Dependency Injection* zu übergeben. Da der MISConnector nicht direkt über eine Methode, sondern über eine Datei aufgerufen wird, müsste die Konfiguration mit in die übergebene Datei geschrieben werden. Befinden sich keine Konfigurationsdaten in der Datei, wird wie bisher der Wert aus der geladenen Konfiguration verwendet. Dies trifft generell für den Ablauf im produktiven System zu. Zu Servicezwecken könnte der MISConnector die Einstellungen mit in das erzeugte Ausgabe-JDF schreiben. Im produktiven Einsatz könnten

ServicemitarbeiterInnen oder EntwicklerInnen sehen, welche MISConnector-Einstellungen verwendet wurden, ohne Zugriff auf das Kundensystem zu haben. Die Einstellungen zum Zeitpunkt der Abarbeitung des Auftrags bleiben erhalten, auch wenn nachträglich Änderungen an der Konfiguration vorgenommen wurden. Müssen die Ergebnisse eines automatischen Tests analysiert werden, stehen den EntwicklerInnen diese Informationen ebenfalls zur Verfügung. Den globalen Zustand durch Dependency Injection zu ersetzen, erfordert einen hohen Implementierungsaufwand. Die Komponente muss umstrukturiert werden. Dieser Aufwand kann für die Schaffung einer ersten Continuous Delivery Pipeline nicht geleistet werden. Wäre bei der Implementierung der Komponente die Testbarkeit mit berücksichtigt worden, wäre dieses Problem vermutlich nicht entstanden (siehe auch Kapitel 2.7 Test Driven Development).

Eine Testplanung sollte recht früh schon in der Spezifikationsphase beginnen, indem überlegt wird, wie ein System getestet werden kann. In der Entwurfs- und Designphase sind Überlegungen anzustellen, wie die Architektur getestet werden kann. Dies betrifft auch die einzelnen Komponenten. Oft bewirkt eine frühzeitige Planung der Tests eine Verbesserung der Umsetzung und der Architektur. Schlecht oder nicht testbare Umsetzungen werden vermieden [SDW<sup>+10</sup>].

Die Analyse der automatischen Tests offenbart eine alternative Optimierungsmöglichkeit. Bisher werden am Anfang eines jeden vom MISConnector abhängigen Tests die Einstellungen ausgelesen. Anschließend werden diese verändert und der Test beginnt mit seiner eigentlichen Ausführung. Ganz am Ende werden die ursprünglichen Einstellungen wieder zurückgestellt. Nach dem der MISConnector seine Arbeit beendet hat, könnte schon parallel ein anderer Test mit anderen Einstellungen starten, ohne auf den noch laufenden Test zu warten. In dem Script-Server müsste ein Mutex (mutual exclusion) implementiert werden, der nur während der aktiven Phase des MISConnectors keine weiteren vom MISConnector abhängigen Tests ausführt. Der gegenseitige Ausschluss könnte weiter optimiert werden, so dass Tests, die die zurzeit aktiven Einstellungen benötigen, auch ausgeführt werden könnten. Die erhöhte Komplexität der Implementierung wird aber in keinem Verhältnis zu dem vermutlich geringen Zeitgewinn stehen. Die Abbildung 4.11 zeigt den Ist-Zustand, wie automatische Tests von den Script-Servers abgearbeitet werden. Der 2. Script-Server führt nur parallel ausführbare Tests aus. Der 1. Test (blaues Rechteck) kann unabhängig von anderen Tests ausgeführt werden und wird vom 2. Script-Server gestartet. Der 1. Script-Server führt den 2. Test aus. Nachdem der 1. Test beendet wurde, könnte der folgende 3. Test ausgeführt werden. Da dieser nicht parallel ausführbar ist, muss auf das Ende des 2. Tests gewartet werden. Der 2. Script-Server befindet sich für diese Zeit im Leerlauf. Abbildung 4.12 zeigt den automatischen Testlauf unter Verwendung des Mutex. Der 6. Test muss nur solange warten, bis der 5. Test im MISConnector abgearbeitet wurde.

Zur Aufgabe eines Entwicklers von automatischen Tests gehört es auch, mit den EntwicklerInnen von Systemkomponenten über die Umsetzung von Maßnahmen zu diskutieren, die

## 4 Konzepte zur Verbesserungen

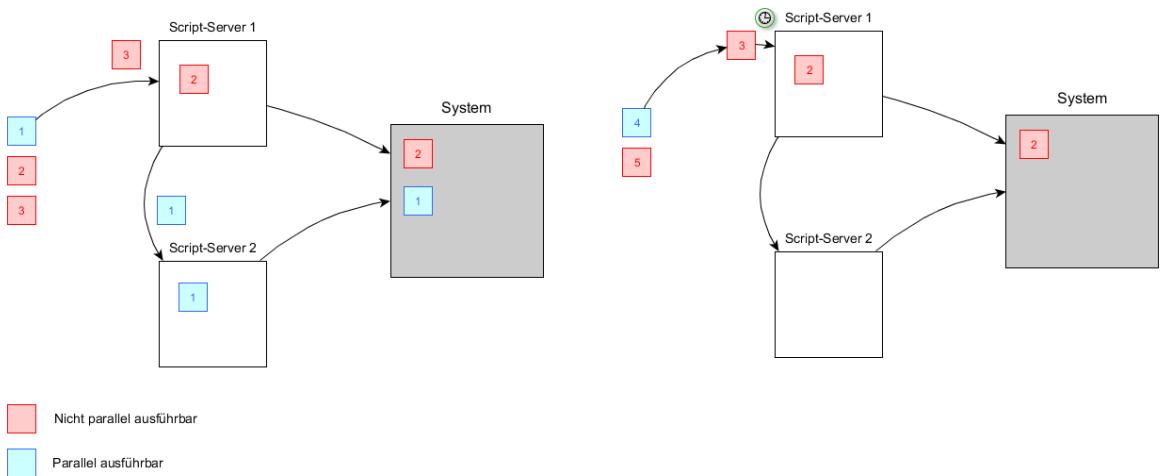


Abbildung 4.11: Bisherige Abarbeitung der automatischen Tests

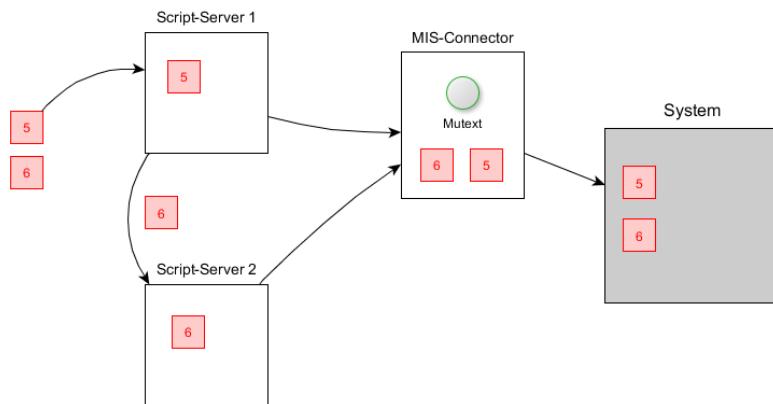


Abbildung 4.12: Abarbeitung der automatischen Tests mit Mutex

eine Komponente besser testbar macht, wie z. B. einen globalen Zustand durch Dependency Injection zu ersetzen. An den Antworten der EntwicklerInnen von Systemkomponenten ist oft der geringe Stellenwert von automatischen Tests zu erkennen. Der Fokus liegt auf dem Kundennutzen. Codeänderungen 'nur' wegen automatischer Tests durchzuführen, erfahren eine geringe Akzeptanz. Automatische Tests haben indirekt einen Kundennutzen, denn durch eine bessere Testunterstützung kann ein System eine höhere Testtiefe und damit potentiell eine niedrigere Fehlerrate erreichen. Durch Zeiteinsparung bei der Testerstellung und schnellerem Feedback nach Änderungen stehen den EntwicklerInnen mehr Zeit für die Umsetzung von Features oder zur Erstellung weiterer Tests zur Verfügung. Durch die Verwendung von Dependency Injection wird der Code testbarer, allerdings wird dieser Vorteil nicht nur positiv bewertet, sondern es wird der höhere Aufwand bemängelt. Systeme mit globalen Zuständen

sind am Anfang immer schneller zu entwickeln und führen zu einer engeren Kopplung der Komponenten. Globale Zustände verstecken die Abhängigkeiten, die nicht in der API eines Objekts deklariert sind. Die Software ist schwerer zu verstehen und zu warten. Misko Hevery hat in seinem Artikel *Singletons are Pathological Liars* [Hev08] das Problem an einem simplen Beispiel verdeutlicht. Jeder (jede Klasse) gibt seine Freunde an (Kollaborateure). Es ist bekannt, dass Joe Mary kennt, aber weder Joe noch Mary kennen Tim. Gibt man Informationen an Joe, kann angenommen werden, dass die Information zu Mary gelangt, aber unter keinen Umständen Tim erreicht. Jeder (jede Klasse) deklariert seine Freunde, aber die anderen Freunde (Kollaborateure in Form eines globalen Singletons) bleiben verborgen. Nun wundert man sich, wie die Informationen an Tim gelangt sind, die man Joe gegeben hat. Der Entwickler, der die Beziehungen der Klassen untereinander entworfen hat, kennt die wahren Abhängigkeiten. Für andere EntwicklerInnen erschließt sich ein derartig und geheimer Informationsfluss nicht.

Die durch Verwendung von globalen Zuständen gewonnene Zeit geht spätestens wieder bei der Erstellung von Unit-Tests verloren. Die Methoden der Komponente, die auf den globalen Zustand zugreifen, leiden unter dem gleichen Problem wie die automatischen Tests. Die von dem globalen Zustand abhängigen Unit-Tests sind nicht parallel ausführbar. Bei Unit-Tests ist dies im Hinblick auf die zahlreichen CPU-Kerne des Buildrechners schwer zu akzeptieren. In Kapitel 5.8 wird die Umsetzung des Mutex mit dem Performancegewinn beschrieben. In Kapitel 5.9 wird die Anzahl der parallel ausgeführten Script-Server erhöht, um von der verbesserten Parallelität weiter zu profitieren.

#### 4.2.5 Script-Server

Die Entwicklung von automatischen Tests ist durch ein eingeschränktes Debuggen und das Patchen von Systemen erschwert (**Problem V**). Wird bei der Entwicklung von automatischen Tests Quell-Code der AutoTestLib oder Produktivcode geändert, so muss ein System mit den Codeänderungen auf den aktuellen Stand gebracht werden. Dies bei jeder Änderung durchzuführen ist sehr aufwendig. Zudem können durch die Änderungen andere EntwicklerInnen oder Testabläufe gestört werden. Durch die Erstellung von automatischen Tests mittels JUnit (siehe Kapitel 4.2.2) kann eine Umgebung geschaffen werden, die das Ausführen von einem automatischen Test aus der Entwicklungsumgebung ermöglicht. Die Tests laufen mit dem aktuellen Stand des Entwicklungsrechners und zum Ausführen wird ein Script-Server auf dem Entwicklungsrechner gestartet, der sich mit einem vorhandenen System verbindet. Der Script-Server kann ohne Beeinflussung des Systems debuggt werden. Das Debuggen des Script-Servers eines Systems ist zwar möglich, erfordert aber mehr Aufwand, da mit der Entwicklungsumgebung eine remote laufende Applikation angesprochen wird. Der angehaltene Script-Server im Debug-Modus kann zu Behinderungen des Systems führen. In Abbildung 4.13 wird die neue Entwicklungsmöglichkeit der alten Vorgehensweise gegenüberge-

## 4 Konzepte zur Verbesserungen

stellt. In der Abbildung (oben) wird veranschaulicht, wie bisher nach Änderungen ein Patch des geänderten Codes erstellt und auf den Server eingespielt werden muss. Das Debuggen des Script-Servers kann nur remote durchgeführt werden. Der unterere Teil der Abbildung zeigt, dass der Entwickler nur den JUnit-Test des System-Tests ausführen muss. Es wird auf dem Entwicklungs-Rechner ein Script-Server erzeugt und mit dem System verbunden. Der System-Test wird auf dem Entwicklungs-Rechner ausgeführt und das Debuggen ist lokal möglich. Werden auf dem Server gerade andere Tests ausgeführt, wird der Test trotzdem sofort gestartet. Im Gegensatz dazu wird beim Kopieren des Tests auf den Server dieser zur Ausführung hinten eingereiht. Die Umsetzung erfolgt in Kapitel 5.7.

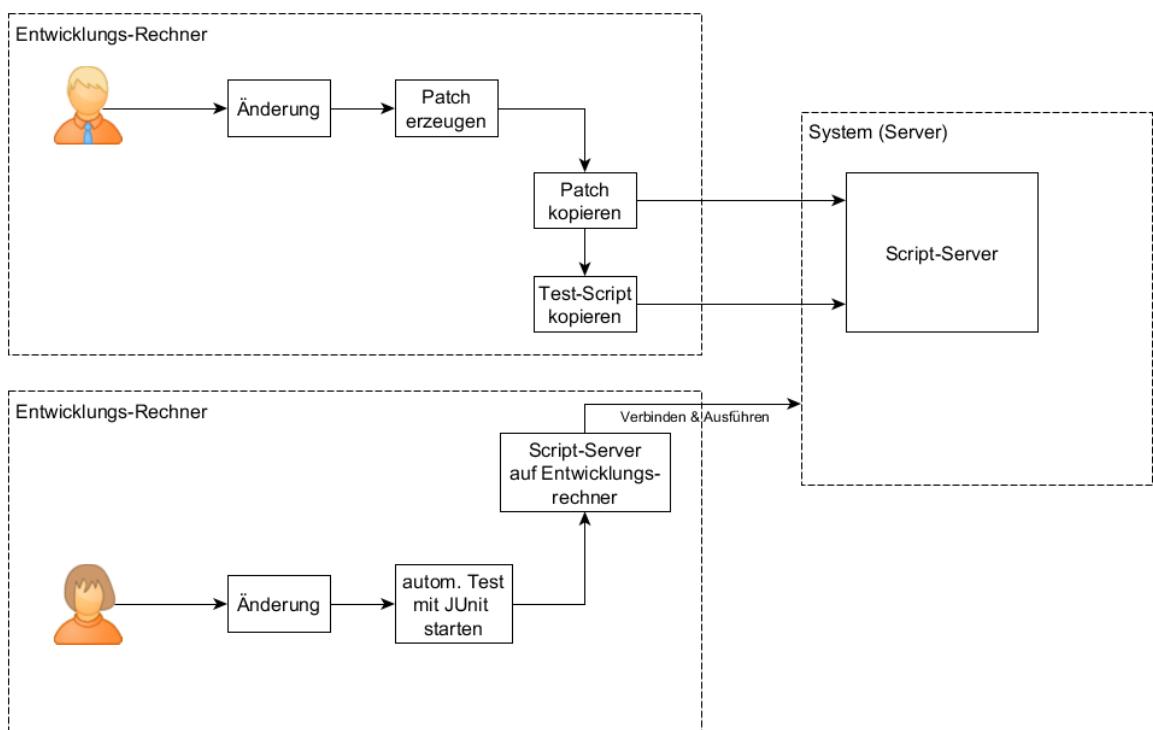


Abbildung 4.13: Ausführung vom automatischen System-Tests auf einem Entwicklungsrechner

### 4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline

Der vorhandene Continuous-Integration-Ablauf muss um die Aktualisierung eines Test-Systems mit den Build-Artefakten und die Ausführung der automatischen Tests erweitert werden. Zunächst wird für jede Änderung der Quellcode kompiliert und die Unit-Tests werden ausgeführt (Commit Stage). Danach wird nochmals ein CI-Build mit den bisher aufgelaufenen Änderungen durchgeführt. Nach diesem Durchlauf wurden im bisherigen Prozess nur noch die Metriken erzeugt und die Änderungen wurden für den nächtlichen Build freigegeben. Parallel

#### 4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline

könnten nun die Integrationstests ausgeführt und Installer gebaut werden. Nach Fertigstellung der Installer werden ein oder mehrere Systeme aktualisiert. Für die Aktualisierung der Software ist es erforderlich zunächst alle Services des Workflow-Systems zu beenden. Nach Installation der neuen Softwarekomponenten und Starten aller beendeten Services, kann mit der Abarbeitung der automatischen Tests begonnen werden. Je nach geforderten Ablaufzeiten, können die automatischen Tests auf mehrere Systeme aufgeteilt werden. Nach Ablauf der Integrations- und System-Tests melden diese ihre Ergebnisse an den Jenkins-Master zurück und werden für die betreffenden EntwicklerInnen dargestellt. Sind keine Fehler aufgetreten, könnte die Version mit Kapazitätstests und manuellen Akzeptanz-Tests weiter überprüft werden. In Abbildung 4.14 wird der geplante Aufbau der Continuous Delivery Pipeline dargestellt. Exemplarisch sind die Änderungen von zwei unterschiedlichen EntwicklerInnen an den einzelnen Stufen markiert. Für den ersten CI-Build wird pro Änderung eines Entwicklers ein CI-Build durchgeführt. Läuft dieser fehlerfrei durch, werden die Änderungen in den Developer-Branch übernommen, der für alle EntwicklerInnen sichtbar ist. Der zweite CI-Build startet mit allen Änderungen des Developer-Branches. In diesem Fall wird ein CI-Build mit den Änderungen von beiden Entwicklern durchgeführt. Wurde dieser fehlerfrei beendet, werden parallel mit den beiden Änderungen Integrations-Tests, die statische Codeanalyse und die Paketierung des Installers ausgeführt.

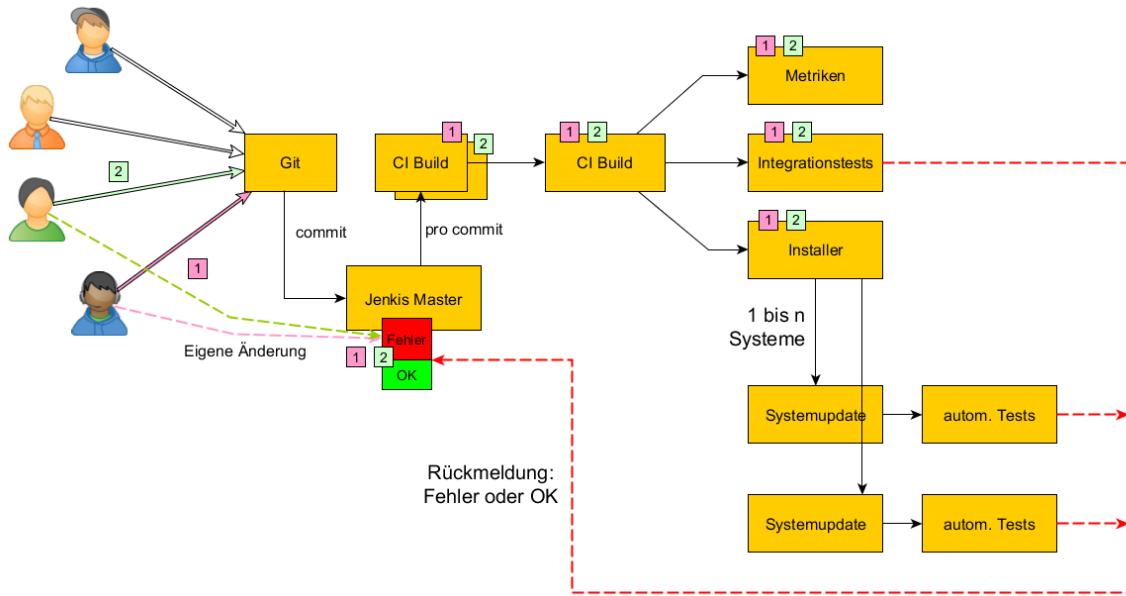


Abbildung 4.14: Geplanter Aufbau der Continuous Delivery Pipeline

Durch das Verschieben von langsamem JUnit-Tests aus dem CI-Build und die parallele Ausführung zu den System-Tests wird der CI-Build beschleunigt und das **Problem I** abgemildert. Da die Integrationstests parallel zu den langsamem System-Tests laufen, beeinflussen

## 4 Konzepte zur Verbesserungen

die Integrationstests nicht die Gesamtaufzeit eines Durchlaufs der Continuous Delivery Pipeline.

Im Idealfall sollte jede Änderung für sich durch die gesamte Continuous Delivery Pipeline laufen, damit einem Fehlschlag in der Pipeline direkt einer Änderung zugeordnet werden kann [BL16b]. Für das Workflow-System sind diese idealen Anforderungen an die Continuous Delivery Pipeline in einer ersten Umsetzung nicht praktikabel. Im Normalfall werden in einer Stunde mehrere Änderungen von den EntwicklerInnen eingecheckt. Würde jede Änderung eine Continuous Delivery Pipeline durchlaufen, müsste ein hohe Anzahl von Build- und Testservern zur Verfügung stehen. Eine sequentielle Abarbeitung der Änderungen mit nur einer Pipeline ist nicht denkbar, da der CI-Build und dessen Feedback eine hohe Wartezeit hätte. Zudem würde sich eine Warteschlange an der Pipeline bilden, die über Nacht abgearbeitet werden müsste. Um überschaubarer zu beginnen, wird die Stufe der Integrations- und System-Tests mit allen bisher fehlerfrei getesteten Änderungen aus der Commit Stage durchgeführt. Je nach Skalierung der Test-Systeme sind ein paar Durchläufe der automatischen System-Tests pro Tag möglich. Im Vergleich mit dem gegenwärtigen Zyklus der Testdurchführung ist das eine deutliche Verbesserung. Um die Forderung nach automatischen Tests am selben Tag der Änderungen umzusetzen, ist es ausreichend ab der Stufe der automatischen Tests alle bisher gemachten Änderungen gemeinsam zu testen.

Die Aktualisierung des Test-Systems für Continuous Delivery ist ein zentraler Punkt. In den folgenden Kapitel 4.3.2 bis 4.3.4 werden unterschiedliche Lösungsmöglichkeiten gesucht um das Workflow-System für einen anstehenden System-Test zu aktualisieren.

### 4.3.1 Modulare Installer

Im nächtlichen Build wird nach dem Kompilieren des Quellcodes ein Installer gebaut (siehe Kapitel 3.2), mit dem das ganze System installiert werden kann. Der Installer wird mit InstallShield 2015<sup>5</sup> erstellt.

Die Aufgaben des Installers sind:

- Firewall inkl. Ports einstellen
- Dateien kopieren
- Konfigurationsdateien anpassen
- Shares und Benutzerechte setzen
- Windows-Services einrichten

Das Bauen des Installers dauert ca. 53 Minuten. Für einen Schritt in der Continuous Delivery Pipeline muss die Zeit reduziert werden. Nach einer Änderung in der Software ist es

---

<sup>5</sup><http://www.flexerasoftware.de/producer/products/software-installation/installshield-software-installer/>

nicht praktikabel den kompletten Monolith-Installer zu erzeugen. Eine Möglichkeit ist die Verwendung der Funktionalität des Update-Installers von InstallShield. Dazu müssen die Datensätze vor der Erstellung des Installers vorhanden sein, also ein kompletter Datensatz für das installierte System und ein Datensatz, der durch den CI-Durchlauf generiert wurde. InstallShield vergleicht die Datensätze und erzeugt einen Update-Installer, der nur die geänderten Dateien enthält. Das Erstellen und die Installationsdauer des Update-Installers werden im Gegensatz zu einem vollständigen Installer reduziert.

Ziel sollte es aber eher sein, dass geänderte Komponenten separat installierbar sind. Nach Durchlauf der Pipeline entsteht ein oder entstehen mehrere kleine Installer.

Zur Umsetzung dieser Forderung wäre der Einsatz einer Installer-Software vorteilhaft, die besser im automatisierten Umfeld verwendet werden kann. Für Windows bietet sich *Windows Installer XML Toolset*<sup>6</sup> an. Dieses Projekt wurde von Microsoft ins Leben gerufen und als Open-Source veröffentlicht. Der Vorteil vom WiX Toolsets ist die Verwendung einer deklarativen Sprache. Es wird der Zustand der Zielmaschine nach verschiedenen Phasen der Installation bzw. Deinstallation spezifiziert [wixb]. InstallShield ist hingegen imperativ, d.h. es werden Befehle angegeben, die hintereinander ausgeführt zu dem gewünschten Zustand des Systems führen. Ein weiterer Vorteil des WiX Toolsets gegenüber dem proprietären Format von InstallShield sind die Definitionen in der XML-Datei. Damit können die exakten Änderungen im Versionskontrollsysteem visualisiert werden und sichergestellt werden, dass die Software und die Infrastruktur zusammen passen [Wol15a]. Das proprietäre Datenformat zieht eine unzureichende Integration in den automatisierten Buildprozess mit sich. Das WiX Toolset bietet für die Integration in den automatischen Buildprozess eine Windows-Installer-API an, mit der individuelle Lösungen möglich sind [Ker05]. Das WiX Toolset Projekt kann mit jedem beliebigen Texteditor betrachtet und verändert werden. Der Quellcode des Windows Installer XML verhält sich wie der Quellcode des restlichen Produkts. Wiederverwendbare Blöcke können für alle EntwicklerInnen in Form einer kompilierten Library (.wixlibs) zur Verfügung gestellt werden [wixa]. Der klassische Ansatz Infrastruktur bereitzustellen, wandelt sich, wie bei Windows Installer XML, in Code. Man spricht von 'Infrastructure as Code'. Der große Vorteil ist, dass die Test- der Produktionsumgebungen entsprechen. Die Aussagekraft von Tests wird somit erhöht [Wol15a]. Das WiX Toolset unterstützt inkrementelle Builds und bietet durch das WiX Toolset Plugin<sup>7</sup> eine gute Integration in Jenkins.

Mit InstallShield wird eine Paketdatei im msi-Format (Microsoft Software Installation) erzeugt. In dem WiX Toolset gibt es das Tool Dark.exe mit dem ein vorhandenes msi-Paket in ein WiX-Quelldatei (.wxs) dekompiliert werden kann. Damit wäre ein schneller Einstieg in das WiX Toolset möglich.

---

<sup>6</sup><http://wixtoolset.org/>

<sup>7</sup><https://wiki.jenkins-ci.org/display/JENKINS/WIX+Toolset+Plugin>

### 4.3.2 Remote System Update

Das Workflow-System verfügt über einen eigenständig entwickelten Update-Service. In einer zentralen Datenbank werden Updates abgelegt und können inhouse oder extern zur Installation freigegeben werden. Kunden können individuell und je nach Freigabe spezielle Software Updates erhalten und auf ihrem System installieren. Die Updates werden über einen Webservice auf dem Ziel-System manuell gestartet. Dazu wird der Update-Installer aus der zentralen Datenbank auf das Ziel-System kopiert. Der Update-Service führt alle Services des Workflow-Systems herunter und führt den Update-Installer im 'Silent-Mode' aus, da bei der Installation keine weitere Interaktion mit einem Benutzer stattfinden soll. Der Ablauf ist in Abbildung 4.15 dargestellt. In Schritt 1 wird der Update-Installer über das Netzwerk bzw. Internet auf das Zielsystem kopiert. In Schritt 2 startet ein Administrator die Installationen. Der Update-Service führt alle Workflow-Services in Schritt 3 herunter. Sind alle Services heruntergefahren, wird Schritt 4 die Installation des Updates ausführt. Nach Fertigstellung der Installation werden im Schritt 5 die Workflow-Services wieder gestartet. Das Workflow-System wurde aktualisiert und steht nun mit der aktuellen Version zur Verfügung.

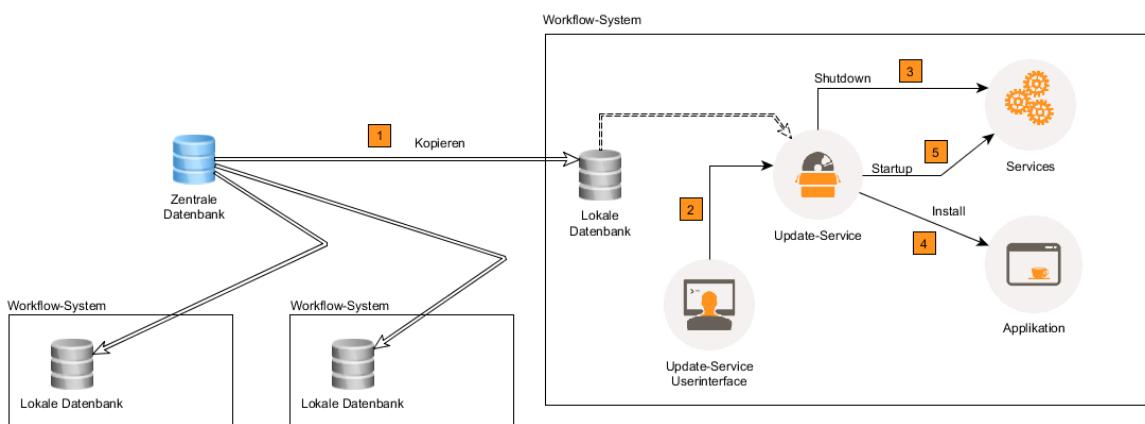


Abbildung 4.15: Remote Update System

### 4.3.3 Hotfix Installer

Der Update-Service (siehe Kapitel 4.3.2) kann neben Voll- und Update-Installern auch gezielte Änderungen in Form von Hotfixes auf ein oder mehrere Systeme installieren. Die Hotfixe bestehen aus kompilierten Artefakten (\*.exe, \*.jar, \*.war). Diese werden in Form spezieller Pakete (RUP - Remote Update Package) an den Update-Service übergeben. Das RUP-Paket enthält neben den Artefakten auch die Information, für welche Applikation das Update ausgeführt werden soll. Für jeden Hotfix, den ein Entwickler erstellt, muss dieser den Speicherort des Artefakts angeben. Die Erstellung des Hotfixes erfolgt in einem

#### *4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline*

manuellen Prozess, der das Fehlerpotential eines falsch eingetragenen Speicherorts enthalten kann. In Abbildung 4.16 ist der Ablauf der manuellen Erstellung eines Hotfixes dargestellt. Das Installieren von Hotfixes per Update-Service könnte für das automatische Updaten eines Test-Systems nach einem CI-Build verwendet werden. Für die Verwendung des Hotfix-Prozesses in Continuous Delivery müsste dieser automatisiert werden. Das selbst entwickelte Programm zur Hotfix-Erstellung kann neben der Bedienung per Bedienoberfläche auch über Kommandozeilen-Aufrufe verwendet werden. Eine Automatisierung der Hotfix-Erstellung kommt nicht nur einer Continuous Delivery Pipeline zugute, sondern kann für die Erstellung von Hotfixes verwendet werden. Bringt ein Entwickler seine Änderungen für einen Hotfix ein, folgt ein kompletter Durchlauf der Continuous Delivery Pipeline, in dem ein automatisch generierter und voll getesterter Hotfix entsteht. Der zu erstellende Ablauf ist in Abbildung 4.17 dargestellt. Für die Automatisierung der Hotfix-Erstellung muss es eine automatische Zuordnung der Artefakte zu den Dateiallageorten geben. Die automatisch generierten Hotfixe enthalten alle Komponenten, die Gradle wegen der Abhängigkeit der eingespielten Änderungen kompilieren musste. Der manuell erstellte Hotfix enthält hingegen nur genau die geänderten Klassen. Der Vorteil der manuell erstellen Hotfixe ist, dass einzelne Hotfixe einfach von einem System entfernt werden können. Dies birgt allerdings die Gefahr, dass ein System nach Änderungen von Schnittstellen nicht mehr korrekt funktioniert, da Referenzen der geänderten Klasse neu kompiliert wurden, aber nicht in dem Hotfix enthalten sind.

Für das Aktualisieren eines Test-Systems in der Continuous Delivery Pipeline ist es sinnvoll, die komplett kompilierten Komponenten zu ersetzen. Diese werden von dem System auch nicht wieder deinstalliert. Treten bei Integrations- oder automatischen System-Tests Fehler auf, müssen diese von dem Entwickler durch weiteres Einspielen von Änderungen behoben werden. Auf dem Test-System werden die neuen Software-Versionen immer nur dazu installiert. Sobald Änderungen durch den ersten Continuous-Integration-Build gelaufen sind, werden diese von allen anderen EntwicklerInnen gesehen. Deren eigene Änderungen basieren dann unter Umständen auf der neuen Software-Version, die ggf. zu Fehlern in den nachgelagerten Pipeline-Schritten, wie Integrations- und automatischen System-Tests, führen.

Ein automatisch generierter Hotfix muss von dem Update-Service ohne Bestätigung eines Administrators installiert werden. Dazu ist eine Erweiterung des Update-Services notwendig. Die Informationsdatei des RUP-Paktes müsste um einen Eintrag zur automatischen Installation erweitert werden. Die zu aktualisierenden Test-Systeme sind in der zentralen Datenbank des Remote Update Services mit aufzunehmen. Die automatisch zu aktualisierenden Systeme sollten aus Sicherheitsgründen explizit dafür freigeschaltet und auf Kunden-Systemen generell deaktiviert werden. Andernfalls stellt die Funktion ein Risiko dar, dass das System ungewollt oder mutwillig verändert wird.

Das Aktualisieren eines ausgelieferten Systems mittels gezielter Änderungen per Hotfix wird für den Fall eingesetzt, dass die Änderungen des Hotfixes zu vorher unerkannten

## 4 Konzepte zur Verbesserungen

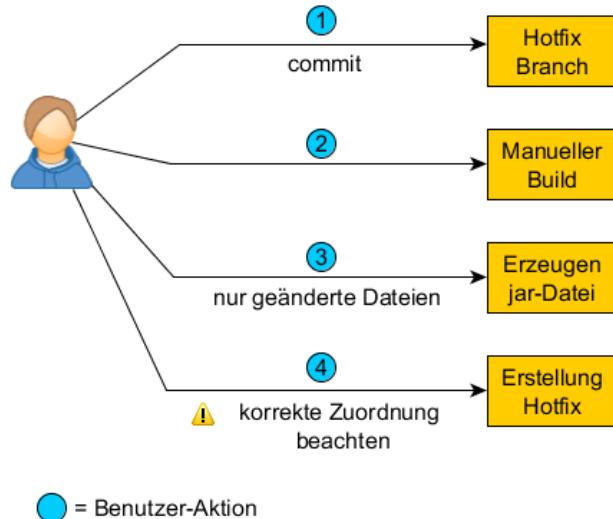


Abbildung 4.16: Manuelle Erstellung eines Hotfixes

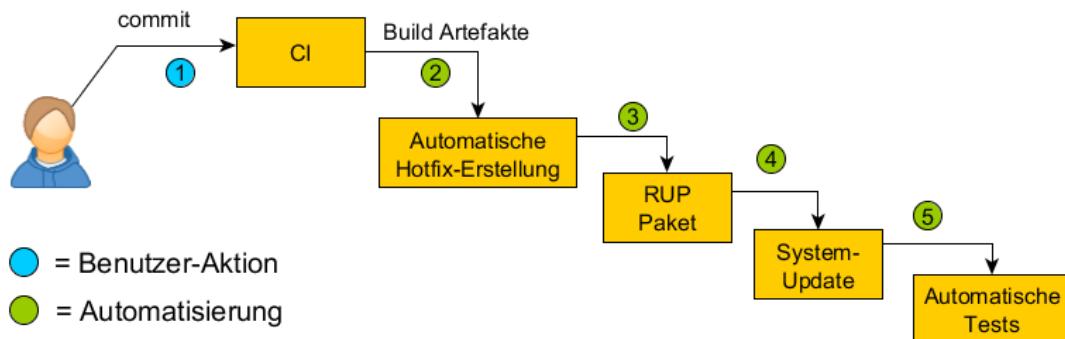


Abbildung 4.17: Automatischer Prozess zur Erstellung eines Hotfixes

Fehlern auf dem Kunden-System führen könnten. Eine Continuous Delivery Pipeline mit ausreichender Testabdeckung entschärft das Problem von fehlerhaften in die Produktion gebrachten Änderungen. Nach Birk und Lukas [BL16a] tendieren Firmen mit Continuous-Delivery-Erfahrung dazu einen aufgetretenen Fehler zu fixen, statt auf die ältere Version zurück zukehren. In der Zukunft kann davon ausgegangen werden, dass das Workflow-System statt einzelner Hotfixe auf Klassenebene komplette Softwarestände mit neu kompilierten Komponenten ausliefert. Dazu muss die Testabdeckung stetig verbessert werden.

### 4.3.4 Microservices

Ein System ist zu komplex, um es vollständig zu verstehen. Aus dem Grund wird ein System in Komponenten aufgeteilt. EntwicklerInnen können nun an der Komponente arbeiten und

#### 4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline

diese einzeln verstehen. Eine Aufteilung der Komponente geschieht aus fachlicher Sicht und ist nicht technologieabhängig. Das ganze System kann als eine Deployment-Einheit implementiert werden. Die einzelnen Komponenten werden als Packages oder Klassen implementiert. Dieser Ansatz ist technisch einfach und bietet eine hohe Performance, da die Komponenten direkt über Methodenaufrufe kommunizieren. Wäre jede Komponente eine Deployment-Einheit, wäre auch der Build-Prozess deutlich komplexer. Jede Einheit muss kompiliert und im System installiert werden. Der Vorteil ist aber, dass die Deployment-Einheiten schneller und einfacher sind. In Abbildung 4.18 sind die Deployment-Einheiten von Microservices und eines ganzen Systems gegenübergestellt. Rechts sind die Komponenten z. B. über REST entkoppelt.

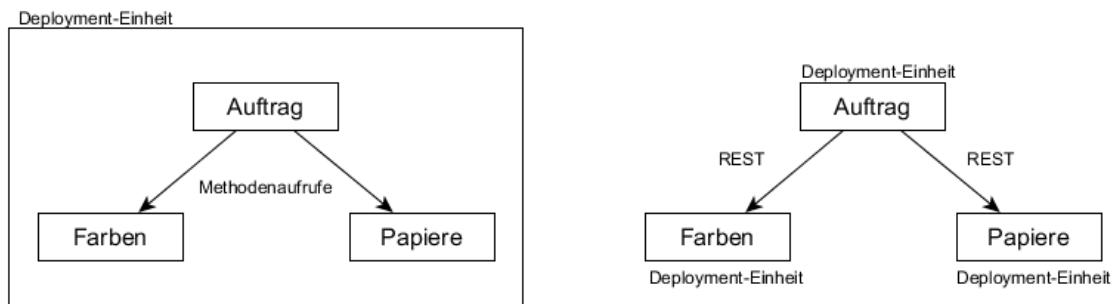


Abbildung 4.18: Gegenüberstellung Deployment-Einheiten eines ganzen Systems und Microservices

Das Workflow-System wird als Ganzes, in einem einzigen sogenannten Deployment-Monolithen, ausgeliefert (siehe Kapitel 3.2). Wurde eine Softwareversion zur Installation beim Kunden freigegeben, werden Änderungen an Kunden-Systemen nur noch durch gezielte Hotfixe einer Komponente durchgeführt (Kapitel 4.3.3). Für neue Softwareversionen existieren somit keine eigenen Deployment-Einheiten, sondern die nachträgliche Installation der Hotfixe setzt immer die monolithische Installation voraus.

Für eine Continuous Delivery ist es von Vorteil, wenn jede Komponente eine eigene Deployment-Einheit hätte. Für jede Einheit könnte eine eigene Continuous Delivery Pipeline existieren, in der nur die geänderte Komponente getestet wird. Sind die Komponenten klein, werden diese stark voneinander entkoppelt. Die Wahrscheinlichkeit, dass unbeabsichtigte Abhängigkeiten entstehen, sinkt [Wol15a]. Wie bei allen Softwareprojekten, droht sich das Workflow-System immer weiter aufzublähen und stetig unvorhersehbarer zu werden. Nach dem Conways Gesetz [Con68] schlägt sich immer die Organisation eines Projektes in der Architektur nieder.

Wenn Organisationen Systeme entwerfen, führt dies zwangsläufig zu Systemen, die Kopien dieser Kommunikationsstrukturen darstellen.

Haben Teams eigene Aufgaben wie GUI oder Backend, finden sich diese Komponenten in der Architektur wieder. Die anfängliche Struktur des Workflow-Systems spiegelt die Organisation

## 4 Konzepte zur Verbesserungen

der Gruppen in den beiden Komponenten *UserInterface* und *InfraStructure* wieder (siehe Kapitel 4.1.3 Abbildung 4.3). Die Entwicklung der Workflow-Software startete in einem nicht agilen Umfeld und war teilweise durch die Organisation geprägt. Noch heute beeinflusst die Organisation, vor allem auch der Standort der EntwicklerInnen, Architektur-Entscheidungen. Die Fachlichkeiten sind mittlerweile agilen Teams zugeordnet. In der Parallel-Welt (Kapitel 4.1.3 Abbildung 4.4) werden die Fachlichkeiten des Userinterfaces auf die Komponenten *Auftrag* und *Administration* aufgeteilt.

Der Aufbau einer Continuous Delivery Pipeline für Deployment-Monolithen ist sehr aufwendig, da dieser nur als ganzes in die Produktion gebracht werden kann. Ein Durchlauf der Pipeline dauert sehr lange. Es müssen sehr viele Tests ausgeführt werden, um Regressionen in der gesamten Funktionalität des Deployment-Monolithen zu vermeiden. Die Tests für das System sind sehr komplex und haben eine hohe Ausführungszeit. Ein wesentlicher Grund für die Nutzung von Microservices, sind die Probleme von Continuous Delivery mit Deployment-Monolithen. Häufig werden Microservices als Ergänzung eines Deployment-Monolithen eingeführt, um Continuous Delivery zu ermöglichen. Es ist allerdings risikoreich die beiden grundlegenden Dinge wie Continuous Delivery und Microservices zusammen einzuführen. Es besteht die Möglichkeit einer schrittweisen Einführung der Microservices. Ein erstellter Microservice kann ein System ergänzen und nur bestimmte Anfragen beantworten. Alle anderen Anfragen werden an das System weitergeleitet [Wol15c].

Jeder Microservice ist ein kleiner Teil der Anwendung, der einzeln deutlich einfacher zu handhaben ist. Wird ein Monolith in Microservices aufgeteilt, so verlagert sich die Komplexität des Monolithen in das korrekte Zusammenspiel der Services. Die Herausforderung liegt in der Aufteilung der Microservices. Ein guter Service sollte immer in sich fachlich geschlossen sein [Eva12]. Weiterhin erfordert das Design der Service-Schnittstellen und deren Austausch-Objekten/Strukturen ein besonderes Augenmerk. Der fachliche Zuschnitt in Microservices sollte nicht nach den Domain-Objekten wie z. B. Farben oder Papiere erfolgen. Services sollten so wenig wie möglich fachliche Berührungspunkte aufweisen, denn das führt während der Laufzeit zu weniger gegenseitigen Aufrufen. Bei Änderungen von Services sind nicht automatisch andere Services betroffen. Zur Aufteilung in Services können UseCases herangezogen werden. Wäre ein Microservice für einen UseCase verantwortlich, bräuchte dieser zur Abarbeitung des UseCases keine weiteren Services. Eine Microservice-Architektur ist nur sinnvoll, wenn die Services fachlich und unabhängig zugeschnitten werden können. Praktisch lässt sich das selten realisieren [RL16]. Durch den fachlichen Schnitt ist gewährleistet, dass nur noch ein Service gebaut und ausgeliefert werden muss. Wie in den Kapiteln 4.3.2, 4.3.1 und 4.3.3 dargestellt, ist das automatische Aktualisieren eines Workflow-Systems nach Änderungen sehr aufwendig zu implementieren. Ein geänderter unabhängiger Microservice kann hingegen relativ einfach ausgetauscht werden [TW15].

Es stellt sich die Frage, ob die Architektur des Workflow-Systems geändert und in Mi-

#### *4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline*

crosservices aufgeteilt werden sollte. Nach Martin Fowler sollte ein Monolith immer erster Ausgangspunkt für Microservices sein [Fow15]. Sam Newman argumentiert, man habe bereits mit dem schon gebauten System mehr Material, um das System Schritt für Schritt zu verteilen [New15]. Nach diesen Standpunkten ist das Workflow-System eine ideale Voraussetzung für eine Aufteilung in Microservices. Dieser These widerspricht Stefan Tilkov [Til15]. Nach seinen Erfahrungen gestaltet sich eine Partitionierung in der Praxis als schwierig, bisweilen sogar als unmöglich. Die Zerlegung eines Monolithen sollte nicht leichtfertig getroffen werden, vor allem wenn kein benennbarer Grund vorliegt oder keine genaue Kenntnis über den Anwendungsbe- reich des monolithischen Systems vorliegt. Es gibt keinen Grund, einen gut strukturierten Monolithen in Microservices zu überführen. Microservices halten EntwicklerInnen durch die Architektur davon ab, falsche oder zu enge Abhängigkeiten zwischen Komponenten herzustellen. Diese Fehler können auch bei einem Monolithen verhindert werden, erfordern allerdings das strikte Einhalten von vielen Regeln. Eine Partitionierung eines Monolithen wird komplizierter, wenn sich Monolithen-Komponenten die gleichen Domain-Objekte und das gleiche Persistenzmodell teilen. Statt einen Monolithen aufzuteilen, sollte besonderes Augenmerk auf die Subsysteme gelegt werden und diese so unabhängig wie möglich zu designen. Ein Subsystem ist dann zwar nicht das, was man als Microservice versteht, könnte aber einen eigenen Zyklus von Entwicklung, Deployment und Delivery haben und ggf. eine eigene Architektur verwenden.

Von den Abhängigkeiten der Subsysteme des Workflow-Systems wurde eine Übersicht erstellt (Abbildung 4.19). Sie zeigt eine starke Verflechtung der Abhängigkeiten. Das Workflow- System verfügt über mehrere Persistenzmodelle (Konfigurations-, Stamm-, Maschinen- und Auftragsdaten), die von vielen Komponenten genutzt werden. Der untere Teil der Abbildung zeigt einen Auszug aus der Gesamtübersicht für den Service der Konfigurationsdaten. An den eingehenden Pfeilen ist zu erkennen, dass sehr viele andere Services von dem Service der Konfigurationsdaten abhängig sind.

Das Workflow-System in Microservices aufzuteilen, birgt ein schwer kalkulierbares Risiko. Vielmehr sollten die Komponenten in einem Redesign betrachtet werden, um eingeschlossene Abhängigkeiten zu beseitigen. Einige Eigenschaften von Microservices erleichtern die Einführung einer Continuous Delivery Pipeline. Ein wichtiger Punkt ist die hohe Robustheit ('Resilience'). Ein Service sollte ausfallen können, ohne dass andere Komponenten neu gestartet werden müssen, um die Kommunikation wieder aufzunehmen. In Kapitel 4.3.2 ist der Update-Service beschrieben. Dieser fährt die Services in einer gezielten Reihenfolge herunter und wieder hoch. Ist der Service A von Service B abhängig, so muss erst Service B hochgefahren sein, damit Service A gestartet werden kann. Die zahlreichen Abhängigkeiten zu anderen Services und deren Zustand machen das Workflow-System sehr komplex. Für eine Continuous Delivery Pipeline wäre es einfacher bei Änderungen vom Service B diesen zu bauen und auf dem Test-System durch die neue Version auszutauschen, ohne die anderen

## 4 Konzepte zur Verbesserungen

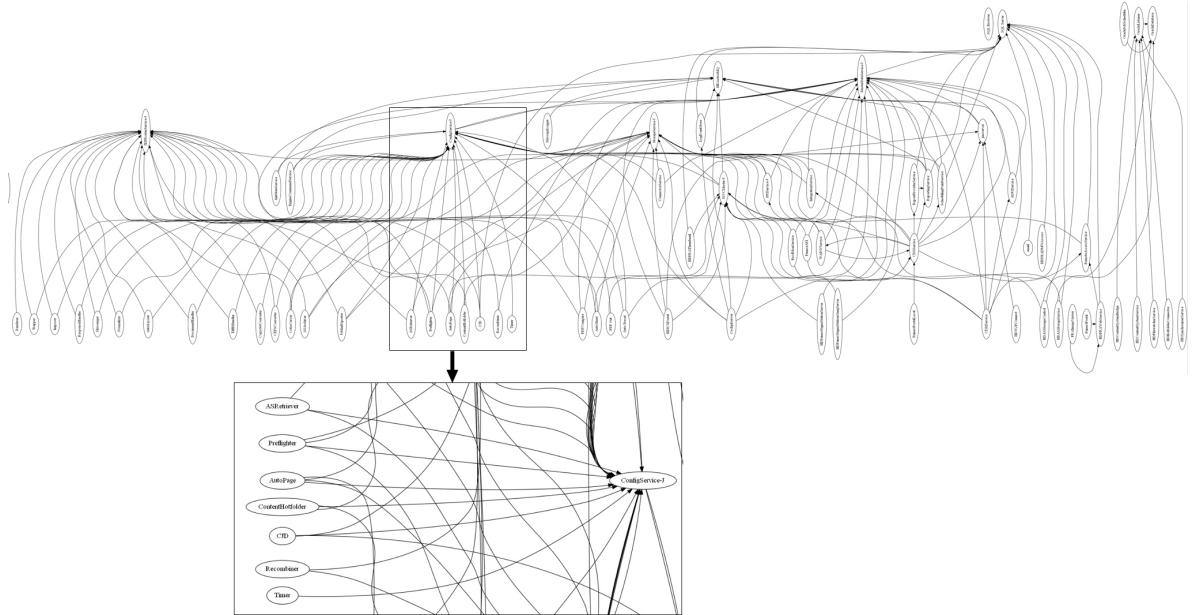


Abbildung 4.19: Abhängigkeiten der Services des Workflow-Systems

Services neu starten zu müssen.

Im Zusammenspiel mit Microservices ist der Einsatz von Container-Technologien von großem Vorteil. Durch die Entkopplung der Microservices könnte jeder Service in einem eigenen Container laufen. Wurde eine Komponente geändert, kann auf das Update des Systems verzichtet werden. Der Container mit der bisherigen Komponente wird entfernt und ein Container mit der neuen Komponente wird erzeugt. Danach können automatische System-Tests mit dem geänderten Service ausgeführt werden. Bei einer großen Anzahl von Microservices kann mit leichtgewichtigen Virtualisierungsanwendungen wie Docker der Ressourcenverbrauch deutlich gesenkt werden.

Da jeder Microservice in seiner eigenen Umgebung läuft, ist das Kontrollieren aller Microservices aufwändiger. Ein schneller Blick in die Logdatei ist nicht mehr möglich. Die Logdateien müssen zentral mit verfügbaren Tools verwaltet werden. Weiterhin unterliegen die Services keiner einheitlichen Versionskontrolle und können in unterschiedlichen Programmiersprachen entwickelt werden. Dadurch werden Refactorings deutlich erschwert [TW15]. Im Aufbau des Workflow-Systems wurde berücksichtigt, dass Code durch allgemeine Bibliotheken wiederverwendet werden kann. Die Microservices kehren sich von der Wiederverwendung ab, damit diese unabhängig sind. Der Vorteil von ausgelagertem Code ist, dass Sicherheitslücken und Fehler nur einmal behoben werden müssen [Wol15b]. Wird der wiederverwendete Code geändert, führt das zu Änderungen an allen abhängigen Microservices, so dass deren Unabhängigkeit nicht gewährleistet ist. Nach Eberhard Wolff [Wol15b] verlieren die Microservices den Qualitätsfaktor 'Collective Code Ownership'. Das Konzept stammt aus der agilen

#### 4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline

Entwicklung. Durch die Beschränkung der *Code Owner* auf das Team des Microservices und ggf. durch unterschiedlich verwendete Programmiersprachen, wird der Code von weniger EntwicklerInnen nachvollzogen und kritisch hinterfragt. Durch Makro-Architekturen können Entscheidungen für das gesamte System getroffen werden. Diese Entscheidungen müssen von allen Microservices umgesetzt werden. Die Verwendung des Kommunikationsprotokolls ist eine notwendige Vorgabe, damit die Microservices untereinander kommunizieren können. Es macht ebenfalls Sinn, Werkzeuge für Logging und Monitoring vorzugeben, damit kein 'Technologie-Zoo' entsteht. Die Programmiersprache kann vorgegeben werden, damit MitarbeiterInnen einfacher die Teams wechseln können. Gibt die Makro-Architektur zu viel vor, können die Vorteile der Microservices nicht ausgenutzt werden. Da die technische Komplexität eines Systems durch Microservices erhöht wird, ist der Einsatz nicht zu empfehlen, wenn die Vorteile der Microservices nicht zur Geltung kommen. Weiterhin muss die Kultur eines Unternehmens zu der Microservice-Architektur passen und die Teams selbst bestimmen lassen. Nicht alle MitarbeiterInnen wollen diese Freiheitsgrade. In der Regel verbessert sich bei Teams mit größerer Verantwortung das Arbeitsklima und die Produktivität.

##### 4.3.5 Aktualisierung des Test-Systems

Das Bauen des Installers und die Installation der Workflow-Software (**Problem VIa,b**) dauern für eine Continuous Delivery Pipeline zu lange. Um die Probleme zu lösen wurden in den Kapiteln 4.3.1 Modulare Installer, 4.3.3 Hotfix Installer und 4.3.4 Microservices einige Lösungsmöglichkeiten erarbeitet.

Von einer aufwendigen Implementierung des System-Updates via Hotfix-Installer wird abgesehen, da die Erstellung von Hotfixes nach Etablierung einer Continuous Delivery Pipeline vermutlich nicht mehr lange existent sein wird. Die Erweiterung einer automatischen Aktualisierung würde die Komplexität des Updates-Services unnötig erhöhen.

Im Kapitel 4.3.4 Microservices wurde schon auf das Risiko der gleichzeitigen Einführung von Continuous Delivery und Microservices eingegangen. Ein Umbau der Workflow-Software würde ein paar Jahre in Anspruch nehmen und kann für die Einführungsphase von Continuous Delivery nur unterstützend eingesetzt werden. Ein mögliches Einsatzszenario für den MISConnector wird in Kapitel 6 beschrieben.

Ein zu erreichendes Ziel für die Workflow-Software wird die Zerschlagung des monolithischen Installers sein. Jede Subkomponente sollte eigenständig deployt werden können. Die Umstellung wird nicht kurzfristig möglich sein. Aus dem Grund wird für die Aktualisierung des Test-Systems ein Prototyp implementiert, der die erstellten Build-Artefakte auf das System kopiert. Die dazu benötigten Informationen der Dateipfade sind in der ISM-Datei von InstallShield enthalten. Das Dateiformat ist ein proprietäres Binärformat, in dem die Installationsbeschreibungen gespeichert werden. Diese Information kann nicht außerhalb des Installationsprozesses für ein Kopieren per Script verwendet werden. Eine schnelle und

effiziente Lösungsmöglichkeit ist der Prototyp, der auf dem Zielsystem die vorhandenen Dateien sucht und ersetzt. Der Sinn von Continuous Delivery ist es allerdings so häufig wie möglich Releases in Produktion zu bringen [Wol15a]. Dazu muss das Update des Systems den Mechanismus verwenden, der auch beim Update eines Produktiv-Systems verwendet wird. Für eine erste Version von Continuous Delivery können die Ziele auch mit einem Prototypen erreicht werden. Die Umsetzung des Prototypen erfolgt in Kapitel 5.10.2.

### 4.3.6 Erster Schritt für Continuous Delivery

Der Durchlauf der Continuous Delivery Pipeline soll zur Übersicht, wie Continuous Integration bisher, in Jenkins visualisiert werden, damit die EntwicklerInnen und Verantwortlichen den aktuellen Stand in der Pipeline verfolgen können (**Problem VIId**). Jenkins wurde ursprünglich als Werkzeug zum Übersetzen des Quellcodes und zum Ausführen und Darstellen der Unit-Tests entwickelt. Durch die gute Erweiterbarkeit von Jenkins mittels Plugin-Schnittstellen, wurden Erweiterungen zur Konfiguration und Darstellung einer Continuous Delivery Pipeline entwickelt. Die beiden Plugins Build-Pipeline<sup>8</sup> und Delivery-Pipeline<sup>9</sup> haben sich etabliert. In der Version 2.0 wurde Jenkins um das Konzept einer Continuous Delivery Pipeline erweitert und einige Neuerungen sind vielversprechend. Für die prototypische Umsetzung der Continuous Delivery Pipeline für das Workflow-System wird daher die neuste Jenkins Version 2.7.2 LTS (Long Term Support) verwendet. Die Pipeline ist flexibel und erweiterbar. Die Definition erfolgt in einer DSL (Domain specific language), die in Groovy geschrieben ist. Damit stehen viele Befehle zur Verfügung und eigene Anpassungen oder Erweiterungen sind einfach möglich. Aus dem Pipeline Code können auch andere Plugins angesprochen werden, z. B. kann das Docker-Plugin gesteuert werden, um Container zu erstellen und zu starten. Der Code der Pipeline wird in einem Jenkinsfile gespeichert und kann mit in die Versionskontrolle aufgenommen werden. Es wird von *Pipeline as Code* gesprochen. Im Gegensatz zu den Jenkins-Konfigurationsdaten, kann der Pipeline-Code gut gelesen werden. Änderungen der Pipeline sind durch die Versionskontrolle besser nachzuvollziehen [Sel16]. Jenkins kann unterschiedliche Pipelines für verschiedene Git-Banches verwalten. Die neue Jenkins Version verfügt über notwendige Steuerungen wie Fork, Loop, Join und Parallel, um eine komplexe Continuous Delivery Pipeline zu erstellen. Jenkins kann nach geplanten und ungeplanten Neustarts die Abarbeitung der Pipeline fortführen. Mit dem Befehl *input* wird die Pipeline angehalten bis sie durch einen Benutzer fortgeführt oder abgebrochen wird. Die Version 2.x wurde kompatibel zur Version 1.x gehalten, so dass alle Jobs und Plugins weiter funktionieren sollten. Die Pipeline wird mit dem Befehl *Stage 'name'* in die einzelnen Schritte aufgeteilt. Durch den Befehl *node* wird der ausführende Code einem bestimmten Jenkins-Build-Server zugewiesen. Durch Angabe eines Labels kann die Ausführung einer

---

<sup>8</sup><https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin>

<sup>9</sup><https://wiki.jenkins-ci.org/display/JENKINS/Delivery+Pipeline+Plugin>

#### 4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline

bestimmten Gruppe von Jenkins-Slaves übergeben werden. Je nach Auslastung wählt der Jenkins-Master einen verfügbaren Slave.

Ein weiterer wichtiger Befehl ist *parallel*. Damit können Aufgaben parallel, meistens auf unterschiedlichen Slaves, ausgeführt werden. In der StagesView wird der Fortschritt, die Dauer der Ausführung pro Schritt und deren Erfolg dargestellt. Zur Übersicht wird der Kommentar des Commits mit angezeigt. Parallel ablaufende Jobs werden nicht explizit dargestellt. Im Build-Pipeline-Plugin sind diese besser visualisiert (siehe Abbildung 4.20). Da die Continuous Delivery Erweiterung von Jenkins sehr neu ist, sind dort noch Verbesserungen in der Ansicht zu erwarten.

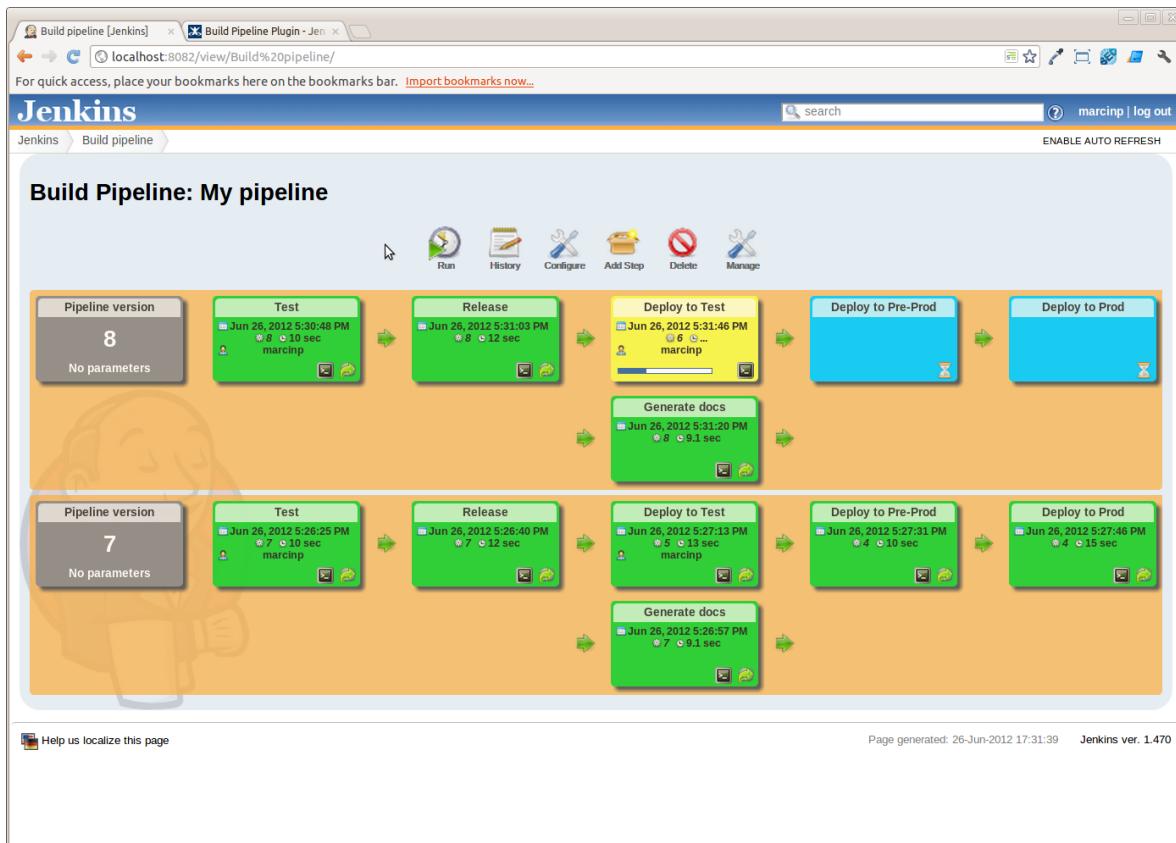


Abbildung 4.20: Pipeline View des Build Pipeline Plugins

<https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin>

Eine prototypische Umsetzung der Pipeline erfolgt in Kapitel 5.10.

##### 4.3.7 Skalierung der Jenkins-Slaves mit Docker

Für Continuous Integration sind die 4 Windows Jenkins Slaves zum Kompilieren und Ausführen der Unit-Tests als eine Gruppe zusammen gefasst. Je nach abzuarbeitenden Aufträgen verteilt der Jenkins Master diese an die Slaves. Diese Konfiguration ist nicht dynamisch

## 4 Konzepte zur Verbesserungen

skalierbar. Werden mehr Slaves zur parallelen Abarbeitung von Aufträgen benötigt, müssen weitere virtuelle Maschinen zur Verfügung gestellt werden und im Master als Slave-Knoten eingerichtet werden. Jede virtuelle Maschine benötigt einige GB Flash- bzw. Festplattenspeicher im Storage Area Network. Ein Ressourcen schonendes Skalieren der Jenkins Slaves kann mit der Container-Technologie Docker umgesetzt werden. Die Container werden automatisch als Jenkins-Slave-Knoten für das Builden erzeugt und nach Fertigstellung wieder gelöscht. Das bringt den Vorteil, wenn viele Builds parallel ausgeführt werden sollen und so um die begrenzte Anzahl von Jenkins Knoten konkurrieren [McA15]. Werden die Jenkins-Slaves für die Continuous Delivery Pipeline in Docker zur Verfügung gestellt, werden die Systemressourcen auf einem Server deutlich effizienter ausgenutzt und die Parallelisierung der Build-Ausführung erhöht [Stä16].

In der Pipeline kann das Ausführen einer Stage von einer Jenkins-Linux-Slave Gruppe ausgeführt werden. Dazu wird ein Docker-Image definiert, auf dem ein neuer Container zum Ausführen des Auftrags erzeugt wird [McD16]. In Listing 4.8 wird auf einem zur Gruppe 'ALL\_JAVA\_GIT\_LINUX\_SLAVES' gehörendem Slave ein Docker-Container aus dem Image 'mycontainer-image' erzeugt. Im Container wird der Quellecode ausgecheckt und kompiliert.

Listing 4.8: Ausführung eines Jenkins Auftrags im Docker Container

```
def nodeLabel = 'ALL_JAVA_GIT_LINUX_SLAVES'

node(nodeLabel)
{
    stage "Container Creation"
    // etwas im Container ausführen
    docker.image('mycontainer-image').inside {
        stage 'Checkout'
        git url: 'https://git@lockerma/lockerma.git'
        stage 'Build'
        // Shell-Aufruf zum Kompilieren
        sh 'gradlew build'
    }
}
```

Tobias Getrost berichtet in seinem Konferenz Journal [Get16] über praktische Erfahrungen und Lessons-Learned auf dem Weg zu Continuous Delivery. Positiv ist das Zusammenspiel von Jenkins und Docker. Bei Einrichtung der Jenkins Build Slaves wird ein vorkonfiguriertes Docker-Image verwendet und auf die Installation von Java, Maven etc. kann verzichtet werden. Werden für die Builds unterschiedliche Umgebungen benötigt, ist es mit dem Einsatz von Docker-Images nicht mehr notwendig, alle Tools auf den Slaves zu installieren. Jeder Slave kann nun alle Aufträge ausführen. Die speziellen Umgebungen befinden sich in dem ausgeführten Docker-Image. Die Installation und Wartung aller Tools auf den Slaves entfällt[Hau16]. Ein weiterer Vorteil ist, dass der Build jedes Mal in einer neuen Umgebung abläuft und somit Seiteneffekte durch Artefakte von vorherigen Builds eliminiert werden.

#### *4.3 Konzept zur Erweiterung einer Continuous Delivery Pipeline*

Die Skalierbarkeit der Slaves ist für die Continuous Delivery Pipeline des Workflow-Systems nicht dringlich, da die einzelnen Stufen der Pipeline zunächst nicht parallel ablaufen (siehe Kapitel 5.10). Sollte jedoch jede Änderung am Code alleine eine Pipeline durchlaufen, wäre das dynamische Erzeugen von Slaves sinnvoll, um eine Warteschlange bei den einzelnen Stufen zu vermeiden.



# 5 Umsetzung der Konzepte

## 5.1 Beschleunigung Unit-Tests

Um die Tests zu beschleunigen (Lösung **Problem I**), wurde der Wert für *forkEvery* erhöht. Es wird nun nicht mehr für jeden Unit-Test eine eigene JVM gestartet. Bei dem Testlauf wurden Unit-Tests identifiziert, die parallel in derselben JVM zusammen mit anderen Unit-Tests zu sporadischen Ausfällen führten. Das Buildscript wurde entsprechend modifiziert, so dass diese Tests in einer separaten JVM ausgeführt werden. Dazu wird das Feature von Gradle (seit der Version 1.10) verwendet, um Tests mit einem bestimmten Namens-Schema auszuführen. Alle problematischen Tests werden umbenannt. Der Test muss auf `OwnForkTest.java` enden. Alle Tests mit diesem Namens-Schema werden nach der Abarbeitung der parallel gestarteten Tests mit der Property `forkEvery=1` ausgeführt.

Vor der Ausführung der Unit-Tests wurden alle vorherigen Build-Ergebnisse mit *clean* gelöscht und die für den Test notwendigen Quelletexte mit *compileTest* übersetzt. Somit konnte ausschließlich die Ausführungszeit der Unit-Tests gemessen werden. Die Durchführung der Tests erfolgte auf einem Entwickler-PC. Störende Nebeneffekte eines parallelen Builds oder Leistungsschwankungen durch Resourcen-Ausnutzung in den VM-Systemen konnten ausgeschlossen werden. Der Test-PC verfügte über folgende Konfiguration:

- intel i7 CPU 8 Kerne mit 3.80 GHz
- 16 GB Ram
- 256 SSD von Samsung (Schreiben 540MB/Sek. — Lesen 520MB/Sek.)
- Windows 10 64Bit
- Gradle 2.14
- JUnit 4.11

Die Ergebnisse der Unit-Test Durchläufe mit unterschiedlichen *forkEvery*-Werten ist in Tabelle 5.1 aufgeführt und in Abbildung 5.1 als Graph dargestellt. Ab *forkEvery*=25 treten keine sichtbaren Verbesserungen mehr auf, so dass der Wert 25 in dem Build-Script für den CI-Server übernommen wird. Tests mit 8 statt 6 parallelen Unit-Tests ergaben keine Verbesserung. Schon bei *maxParallelForks*=6 war die CPU dauerhaft mit 100% ausgelastet.

## 5 Umsetzung der Konzepte

Die Projekte werden soweit es die Abhängigkeiten zulassen, von Gradle parallel gebaut. Somit ist eine volle Auslastung der CPUs schnell erreicht.

ForkEvery	Ausführungszeit
1	16 min 5 sek
5	10 min 22 sek
10	9 min 1 sek
15	8 min 10 sek
25	7 min 22 sek
50	7 min 3 sek
$\infty$	7 min

Tabelle 5.1: Einfluss von forkEvery auf die Ausführungszeit

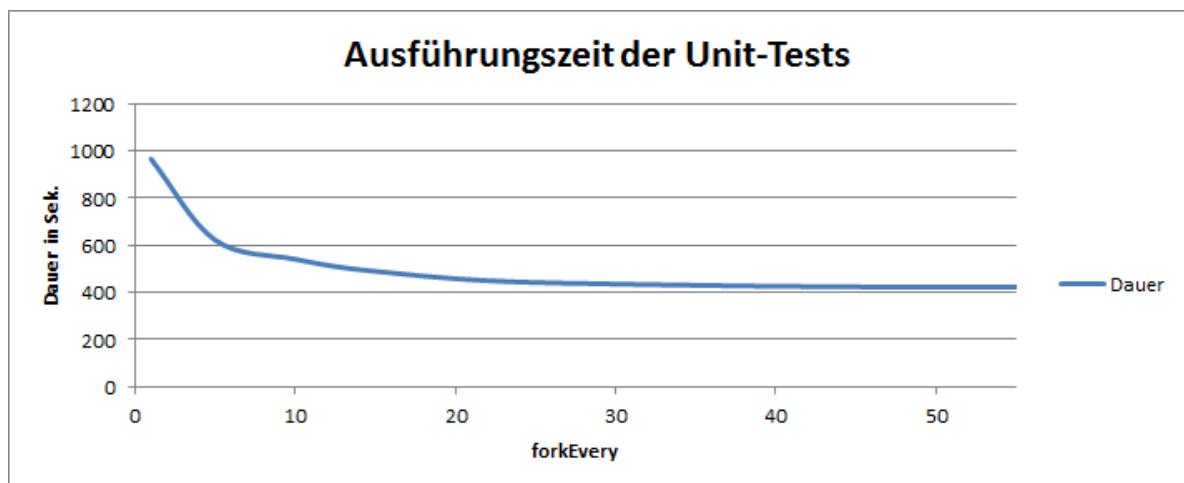


Abbildung 5.1: Ausführungszeit der Unit-Tests in Abhängigkeit von der forkEvery-Property

Die Gesamtaufzeit des CI-Build konnte mit der Optimierung von 30 auf 20 Minuten gesenkt werden.

## 5.2 Beschleunigung durch Hardware

Wie in Kapitel 3.5.1 beschrieben, kann das Builden des Source-Codes ggf. durch den Einsatz einer besseren Hardware beschleunigt werden (**Problem I**). Die Auslastung der virtuellen Maschinen der Jenkins Slaves wurde mit vCenter untersucht. Per Webclient kann auf die Analysedaten des vCenters zugegriffen werden. Es können Kennwerte wie CPU, Arbeitsspeicher, Festplattenspeicher und deren Latenzzeiten als Kennlinien angezeigt werden. Die Werte sind pro Cluster (Gruppe von Servern) oder auch von einzelnen virtuellen Maschinen möglich. Die Auswertung hat ergeben, dass alle ESX-Server des Clusters eine starke CPU

Auslastung aufweisen. D.h. auf den physikalischen ESX-Servern werden zu viele virtuelle Maschinen betrieben. Weiterhin verfügen die ESX-Server über langsame Festplatten mit 7200U/min und einer SAN-Anbindung von 4GB/s. Zur Steigerung der Performance wurden die Server auf neu angeschaffte Server verschoben. Da auf den neuen ESX-Servern nur der Jenkins-Master, die Jenkins-Slaves und die Server für das Anforderungssystem Jira laufen, ist die CPU-Auslastung sehr gering. Die Festplatten Zugriffe sind durch schnellere Festplatten (10.000U/min) und eine schnellere SAN-Anbindung 16GB/s deutlich performanter. Die Buildzeiten von 20 Minuten konnten mit der neuen Hardware deutlich reduziert werden. Die CPU-Auslastung auf dem neuen Server beträgt während des Builds 100%, was eine ausreichende Performance der Festplatten bedeutet. Die neuen Jenkins Slaves werden mit 8 CPUs ausgestattet. Für einen Slave wurden die CPUs von 8 auf 16 verdoppelt. In der Tabelle 5.2 ist der Performance-Gewinn festgehalten:

Server	CPUs	Buildzeit
Alt	8	20 min
Neu	8	13 min
Neu	16	7 min

Tabelle 5.2: Buildzeiten mit neuer Hardware

Die Gesamtaufzeit der Unit-Tests ist im Laufe der Zeit, durch schlechte Server-Performance, von 2h (siehe Kapitel 3.1.2) auf 4h eingebrochen. Die Ausführungszeit auf dem neuen Server mit 16 CPUs liegt bei 1h 20min.

### 5.3 Integrations-Tests separieren

Eine weitere Möglichkeit zur Beschleunigung der Unit-Tests ist das separate Ausführen von langsamen Tests (Kapitel 4.1.1). Durch die beiden Maßnahmen zur Beschleunigung der Unit-Tests (Kapitel 5.1 und 5.2) ist deren Ausführungszeit und damit der Continuous Integration-Build ausreichend schnell geworden. Die langsamten Unit-Tests sollten daher in dem Continuous Integration Schritt verbleiben, damit Probleme so schnell wie möglich in der Pipeline auftreten. In der ersten Stufe des CI-Builds werden andere EntwicklerInnen nicht durch den fehlerverursachenden Code beeinträchtigt. Tritt ein Fehler weiter hinten in der Pipeline auf, dann sind alle EntwicklerInnen der Komponente betroffen. Je weiter hinten ein Fehler in der Pipeline auftritt, desto länger dauert dessen Behebung.

### 5.4 Testbarkeit des Codes sicherstellen

Um sicherzustellen, dass ein neu implementierter Code einfach zu testen ist (**Problem III**), wurde eine neue Parallelwelt (siehe Kapitel 4.1.3) geschaffen. Das Konzept der ServiceRegistry

## 5 Umsetzung der Konzepte

aus Kapitel 4.2.3 wurde umgesetzt. Der Client und die automatischen Tests verwenden die selben fachlichen Interfaces. Die Implementierung bleibt Projekten aus der neuen Parallelwelt verborgen.

Es wurde ein prototypischer JUnit-Test erstellt, um zu verhindern, dass unerwünschten Abhängigkeiten entstehen und den Code schwieriger testbar macht. Im Code des Unit-Tests (Listing 5.1) werden erlaubte Abhängigkeiten definiert und mittels einer JDDepend-Analyse für alle Klassen überprüft. Die UserInterface- und InfraStructure-Klassen dürfen nur auf die Service-Schnittstelle zugreifen. Sollte eine ungewünschte Abhängigkeit im Code eingebaut worden sein, wird der Unit-Test scheitern und die Änderung der EntwicklerInnen gelangt nicht in den Developer-Branch. Abbildung 5.2 zeigt die in dem Unit-Test definierten erlaubten Abhängigkeiten. Greift das UserInterface auf die InfraStructure oder anders herum zu, wird der Unit-Test scheitern. Mit JDDepend kann der Code auch auf das Auftreten von Zyklen überprüft werden. Ein Beispiel ist in der zweiten Test-Methode *testCycles()* aufgeführt.

Listing 5.1: Verwendung von JDDepend in Unit-Tests

```
package de.fhlubeck.lockemar.dependencies;

import static org.junit.Assert.*;
import java.io.IOException;
import java.util.Collection;

import jdepend.framework.DependencyConstraint;
import jdepend.framework.JDDepend;
import jdepend.framework.JavaPackage;
import jdepend.framework.PackageFilter;

import org.junit.BeforeClass;
import org.junit.Test;

public class DependenciesTest
{
    private JDDepend m_jdepend;

    @BeforeClass
    private void setUp() throws IOException
    {
        m_jdepend = new JDDepend();
        m_jdepend.addDirectory("bin/classes");
    }

    @Test
    public void testDependencies()
    {
        DependencyConstraint constraint = new DependencyConstraint();

        JavaPackage gui = constraint.addPackage("de.fhlubeck.lockerma.gui");
        JavaPackage is = constraint.addPackage("de.fhlubeck.lockerma.infrastructure");
```

```

JavaPackage serviceinterface = constraint.addPackage("de.fhuebeck.lockerma.
    serviceinterface");

gui.dependsUpon(serviceinterface);
is.dependsUpon(serviceinterface);

m_jdepend.analyze();

assertEquals("Dependency mismatch", true, m_jdepend.dependencyMatch(constraint));
}

public void testCycles()
{
    Collection packages = m_jdepend.analyze();
    assertEquals("Cycles exist", false, m_jdepend.containsCycles());
}
}

```

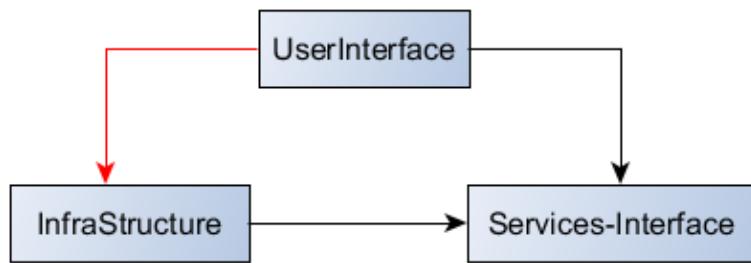


Abbildung 5.2: Zulässige Abhängigkeiten

## 5.5 Logdateien pro automatischem System-Test

Eine Umsetzung für die Vereinfachung der Testanalyse (**Problem IV**) ist das Erstellen von separaten Log-Dateien für jeden automatischen Test. Zur Lösung wurde das Feature von Log4J verwendet, die Log-Einträge parallel in mehrere Dateien zu schreiben. Dem *RootLogger* könnten alternative *FileAppender* hinzugefügt und wieder entfernt werden. Vor Ausführung eines Autotest-Scripts erstellt der Script-Server den FileAppender (Listing 6.2) und fügt diesem dem Logging-Frame-Work hinzu (siehe Abbildung 5.3). Der Script-Server loggt die Einträge nun in die zusätzliche Log-Datei, die parallel zu der Datei mit den Autotest-Ergebnissen liegt. Die Namensgebung ist identisch bis auf den Suffix. Am Ende des Dateinamens wird noch die Nummer des Script-Servers angehängt, damit ersichtlich ist, welcher Script-Server den Test ausgeführt hat ohne die Log-Datei zu öffnen.

## 5 Umsetzung der Konzepte

Dateiname der Auswertung: *Testdateiname\_HH\_mm\_ss.html*

Dateiname des Logs: *Testdateiname\_HH\_mm\_ss\_ScriptServer\_XX.dtv*

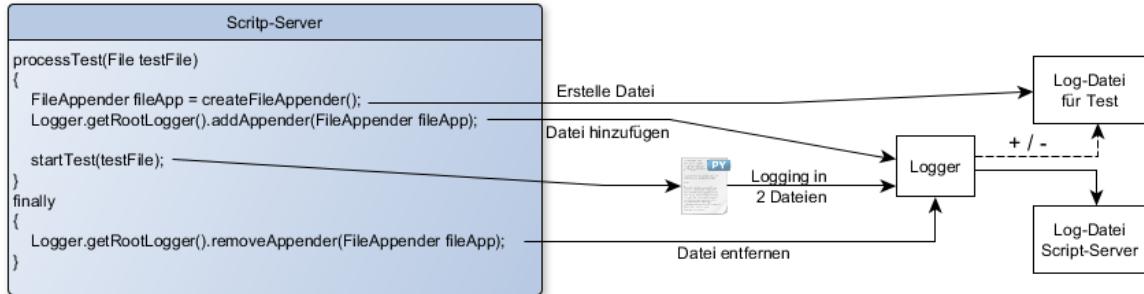


Abbildung 5.3: Erweiterung des Logging pro Autotest

## 5.6 Verwendung von JUnit für automatische Tests

Durch die Integration von JUnit im Testautomaten, soll erreicht werden, dass die Entwicklung der System-Tests vereinfacht wird (**Problem V**). Um Unit-Tests als automatische System-Tests aufzurufen, wird JUnit (junit.jar) verwendet. Die Test-Klassen der Unit-Tests können auch direkt aus Java mit der Klasse *JUnitCore* ausgeführt werden. An *JUnitCore* kann ein *RunListener* angemeldet werden, dem das Starten und Beenden von Tests mitgeteilt wird. Nach Beenden eines Tests können die fehlgeschlagenen Test-Methoden abgefragt werden (siehe Listing 6.5-UnitTestInvoker). In Abbildung 5.4 ist die Erweiterung zum Ausführen von automatischen System-Tests mittels JUnit dargestellt. Die Klasse *UnitTestInvoker* verwendet *JUnitCore* zum Ausführen von Testklassen. Die Klasse *UnitRunListener* meldet sich bei *JUnitCore* an und schreibt die Ergebnisse der Test-Methoden in den HTML-Report. In der Script-Datei eines automatischen Tests werden nur noch Informationsdaten über den Testfall definiert (z. B. Testersteller, Anforderungsnummer...). Der einzige Aufruf, der dann folgt, ist *StartTest('de.fh-luebeck.lockermar.MeinTest')* der Python-Klasse *JUnitRunnerWithLog*. Diese übergibt den Klassennamen inkl. Package (fully qualified name), den HTML-Report und das File-Objekt des Reports an die Java Klasse *UnitTestStarter* (Listing 6.6). Die Java Klasse *UnitTestStarter* ruft den *JUnitTestInvoker* auf. Die Testklasse wird ausgeführt und der *UnitRunListener* schreibt die Ergebnisse in die HTML-Report-Datei.

Zur Ausführung wird eine abgeleitete JUnit-Klasse von *BlockJUnit4ClassRunner* verwendet (siehe Listing 6.5). In JUnit werden die Test-Methoden von Testklassen standardmäßig sequentiell vom *BlockJUnit4ClassRunner* ausgeführt. Seit JUnit 4.7 besteht die Möglichkeit in der Vaterklasse *ParentRunner* einen alternativen Scheduler zu setzen (siehe Abbildung 5.5).

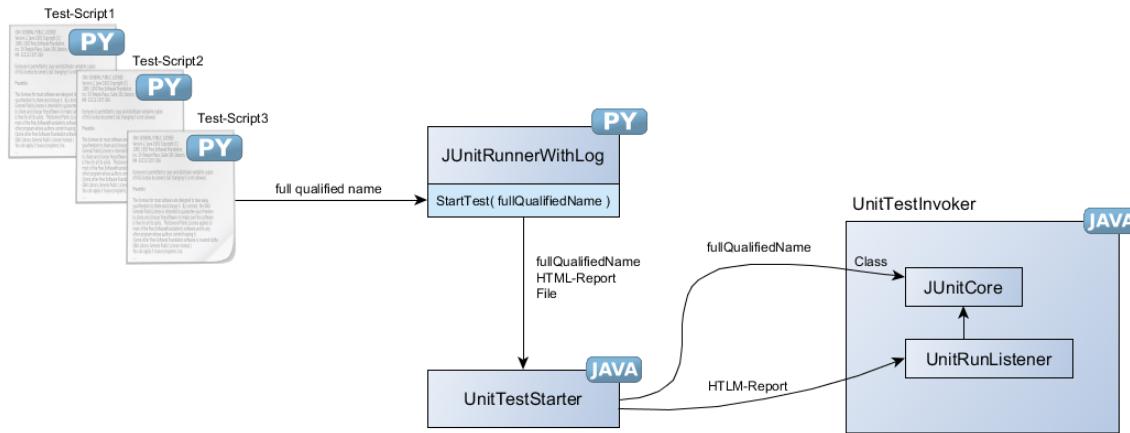


Abbildung 5.4: Erweiterung Testautomat zum Ausführen von JUnit-Tests

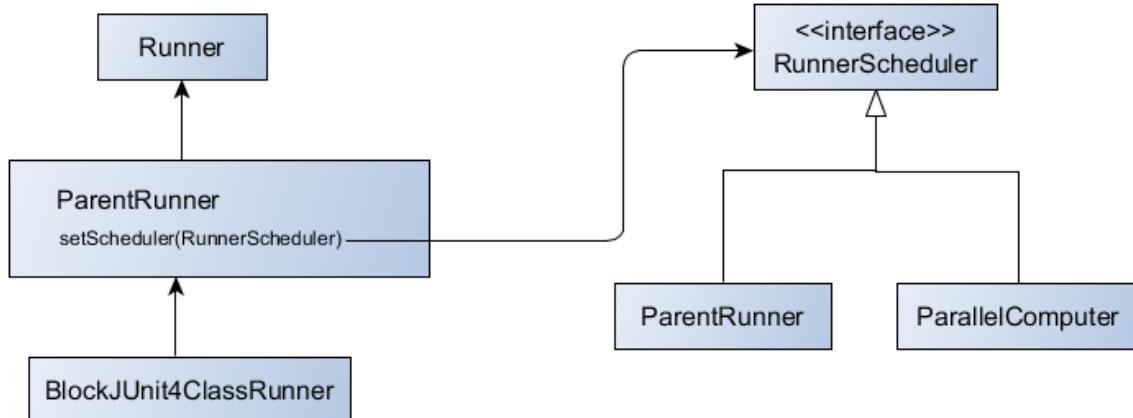


Abbildung 5.5: Klassen-Hierarchie von JUnit zum Ausführen von Testklassen

Sollen die Test-Methoden des automatischen Tests parallel ausgeführt werden, so kann die Klasse *ParallelComputer* gesetzt werden. In der anfänglichen Umsetzung, der Autotest-Erweiterung mit JUnit, werden die Methoden der Reihe nach ausgeführt.

Zur Erstellung von automatischen System-Tests, die JUnit verwenden, wird ein Unit-Test in einem dafür angelegten Projekt (AutoUnitTests) implementiert. Das Projekt besitzt die gleichen Abhängigkeiten wie die neu entstandene Parallel-Welt (siehe Kap. 4.1.3 Abbildung 4.4). Damit wird erreicht, dass der Testcode die selben APIs wie der Client verwendet und sichergestellt ist, dass die Funktionen so dicht wie möglich einer tatsächlichen Bedienung des Clients entsprechen. Abschließend müssen die EntwicklerInnen eine Python-Script-Datei erstellen, die u. a. nur die Information über den Autotest enthält (Ersteller, Anforderungsnummer, wahrscheinliche Ausführungszeitdauer...) und den Aufruf *StartTest* mit Angabe des

## 5 Umsetzung der Konzepte

*Full Qualified Classname.* Wird diese Script-Datei in den Hotfolder des Script-Servers kopiert, so wird der Unit-Test ausgeführt und das Ergebnis in einer HTML-Datei abgelegt (Abbildung 5.6). Zusätzlich wird ein Eintrag in der Gesamtübersicht aller automatischen Tests eingefügt (Abbildung 5.7). Die ersten Implementierungen wurden mit der neuen Möglichkeit System-Tests zu entwickeln getestet.

### Results of script CxFImport

TestStep		Status	Message			Time			
Script-Info	Success	Starting test for class		autotest.JavaDailyChecks.CxFDataImport		05:41:13			
Start JUnit	Success	Starting test with class		autotest.JavaDailyChecks.CxFDataImport		05:41:13			
testRunStarted	Success	null				05:41:13			
init	Success	environment variable TESTLOC: null				05:41:13			
DataHomePath=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData				05:41:13			
Path is=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles				05:41:13			
testStarted	Success	importCxFData				05:41:13			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\CXF Export.cxf				05:41:13			
Parse Result	Success	1 CxFColor found				05:41:13			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\G7_GRACol_P1_BB_M0.cxf				05:41:13			
Parse Result	Success	10 CxFColor found				05:41:13			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\G7_GRACol_P1_WB_M0.cxf				05:41:13			
Parse Result	Success	10 CxFColor found				05:41:13			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\G7_SWOP_3_WB_M0.cxf				05:41:13			
Parse Result	Success	10 CxFColor found				05:41:13			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\G7_SWOP_5_WB_M0.cxf				05:41:13			
Parse Result	Success	10 CxFColor found				05:41:14			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\JCS2011_Cert_Ctd_BB_M0.cxf				05:41:14			
Parse Result	Success	7 CxFColor found				05:41:14			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\PANTONE_GoeGuide Coated.cxf				05:41:14			
Parse Result	Success	6204 CxFColor found				05:41:25			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\PANTONE_GoeGuide Uncoated.cxf				05:41:25			
Parse Result	Success	6204 CxFColor found				05:41:35			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\PANTONE_Solid Coated.cxf				05:41:35			
Parse Result	Success	501 CxFColor found				05:41:36			
File found=	Success	\Kie-wf06prdy\TestData\Autotests\ContentData\CxFFiles\PANTONE_Solid Uncoated.cxf				05:41:36			
Parse Result	Success	5031 CxFColor found				05:41:43			
testFinished	Success	importCxFData				05:41:43			
testRunFinished	Success	RunCount: 1 RunTime: 29856 ms				05:41:43			
passed at 05:41:43 (elapsed time: 00:00:30)	Success	05:41:43: ok:				05:41:43			
Name	Status	Testarea	PRID	Creator	Description	JobCode	estimated time	elapsed time	end time
CxFImport	Success	Autotests	none	Martin Locker	parse and import a lot of CxF-Files	-	00:00:01	00:00:30	05:41:43: ok:

Abbildung 5.6: Details der mit JUnit automatisch ausgeführten System-Tests

ComplexVersioning	Success	MIS, Versioning	PD-54386		Complex Versioning - Workflow from MIS to Press	231472.rS1.1	00:10:00	00:05:24	05:39:23: ok:
CreateJobFromTemplate	Success	Create Job From Template	RA Ordup06b		Create Job From Template, introduce Content Data and check Results	Vorlage-Job 3	00:02:00	00:01:49	05:41:12: ok:
CxFImport	Success	Autotests	none	Martin Locker	parse and import a lot of CxF-Files	-	00:00:01	00:00:30	05:41:43: ok:
D9_ATS_Prepar	Success		none		Simulates HxTest device for AnalyzePoint	-	00:01:00	00:00:03	05:41:47: ok:

Abbildung 5.7: Integration in der Gesamtübersicht

## 5.7 Ausführung von automatischen System-Tests

Durch die Verwendung von JUnit für automatische System-Tests (Kapitel 5.6) kann die Entwicklung von Tests weiter vereinfacht und somit **Problem V** weiter optimiert werden. Zudem wird die Analyse der automatischen Tests (**Problem IV**) durch vereinfachtes Debuggen verbessert. Für EntwicklerInnen wird die Möglichkeit geschaffen, den Script-Server auf dem eigenen Rechner zu starten. Zum einen stört die Änderung am Code nicht andere EntwicklerInnen oder laufende System-Tests, zum anderen werden sie selbst auch nicht durch die laufenden Tests auf dem System gestört. Damit bei Ausführung eines Tests ein Script-Server erzeugt wird, muss in dem JUnit-Test eine statische Methode *connectIfNotServer()* aufgerufen werden. Statische Methoden können mit der @BeforeClass-Annotation versehen werden (siehe Listing 5.2). Damit wird die Methode für die Testklasse nur einmal vor den Testmethoden ausgeführt [Tam13]. Die Methode *connectIfNotServer* überprüft, ob der Test in einer Entwicklungsumgebung läuft. Nur in dem Fall muss ein Script-Server instanziert werden, der sich mit einem vorher konfigurierten System verbindet. Wird die Script-Server-Applikation auf dem Server gestartet, wird dieser beim Starten eine Java-Property gesetzt. Ist die Java-Property vorhanden wird die Methode zum Instanziieren des Script-Servers übersprungen.

Listing 5.2: Script-Server eines System-Test mit JUnit

```
package de.fhlubeck.lockemar.junitautotest;

import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.Test;

public class JUnitTestConnectToServer
{
    @BeforeClass
    public static void connectToServer()
    {
        EclipseEnv.connectIfNotServer();
    }

    @Test
    public void test()
    {
        // Code des Tests
    }
}
```

Mit dem erzeugten JUnit-Framework zur Testerstellung wurden bisher 17 neue System-Tests erschaffen. Nach Rücksprache mit den EntwicklerInnen ist das Implementieren einfacher

geworden. Die Möglichkeit während der Implementierung eines Tests diesen schnell ausführen zu können, ist eine große Hilfe bei der Entwicklung. Die Motivation neue Tests zu erstellen ist höher als vorher.

## 5.8 Parallelisierung der automatischen Tests

Der Script-Server wurde mit einem Verfahren des wechselseitigen Ausschlusses (Mutex) für die Zugriffe auf den MISConnector erweitert, um die automatischen Tests zu beschleunigen (**Problem VIc**). Es waren 37 Tests betroffen, die nun parallel von dem System abgearbeitet werden können. Die Gesamtlaufzeit dieser Tests betrug vorher 1h 50min und konnte auf 1h 20min verbessert werden. Das entspricht einer Beschleunigung von 27,3 %.

Statt des parallelen Ausführens der automatischen Tests wäre auch eine Beschleunigung über die Skalierung der Test-Systeme möglich. Der Vorteil wäre, dass dazu der Script-Server nicht erweitert werden muss, um Probleme bei der parallelen Ausführung zu beheben. Auf der anderen Seite erfordert es mehr Aufwand, das Gesamtergebnis von allen Test-Systemen zusammen zufassen und mehrfach das große System zu installieren. Während die automatischen Tests ausgeführt werden, wird die CPU-Last überwacht. Die durchschnittliche CPU-Auslastung liegt mit zwei Script-Servern bei 20%. Für die Überwachung werden die *Performance Counter*<sup>1</sup> verwendet, die Windows in Echtzeit für Performance-Analysen zur Verfügung stellt.

In Abbildung 5.8 ist die Implementierung des Mutex in dem Script-Server dargestellt. Um den automatischen Test exklusiven Zugriff auf den MISConnector zu gewähren, wurde die Klasse *JDFConnectorAccess* implementiert (Listing 6.3). Die Klasse erzeugt in dem TempDir des Script-Servers eine Datei *JDFConnector.mutex*. Wird die Methode *getConfigAccess()* aufgerufen, wird der FileLock der Datei gesetzt. Der FileLock wurde mit der Java-Klasse *FileChannel* implementiert (Listing 6.4). Die Datei wird mittels Aufrufs der Methode *open* für den Schreibzugriff geöffnet. Auf dem dabei erzeugten *FileChannel* kann ein exklusiver FileLock mit der Methode *lock* gesetzt werden. Hat ein anderer Thread oder eine andere Applikation einen exklusiven FileLock, so wird der aufrufende Thread geblockt. Dieser läuft erst weiter, wenn der Besitzer des FileLocks ihn wieder freigibt. Ein automatischer Test wartet beim Versuch den FileLock zu erhalten mit der weiteren Abarbeitung des Scripts, bis der FileLock wieder freigegeben wurde. Damit gewährleistet ist, dass der FileLock auch wieder freigegeben wird, erfolgt neben der Freigabe des Locks im *AutotestLibaryController.py* auch eine Freigabe des Locks ganz am Ende der Script-Ausführung im finally-Block des Script-Servers. Wird ein Script von dem Script-Server ausgeführt, dann wird die Methode *SetConnectorProperty* des *AutotestLibaryController.py* aufgerufen. Diese Methode muss erst den Zugriff via *JDFConnectorAccess* erhalten, bevor die Einstellungen geändert werden

---

<sup>1</sup>[https://msdn.microsoft.com/en-us/library/windows/desktop/aa373083\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx)

können. Vor Änderung der Einstellungen wird von diesen ein Backup angelegt. Die Einstellungen des Backups werden vor dem Beenden des exklusiven Zugriffs wiederhergestellt. Damit verändert ein automatischer Test die Einstellungen nur temporär für seine eigenen Zwecke. Bisher wurden in den Python-Scripten die vorherigen Einstellungen ausgelesen und am Ende des Tests wieder zurückgesetzt. Mit der neuen Implementierung können diese Zeilen aus dem Script gelöscht werden, was die Test-Scripte in der Übersichtlichkeit verbessert. Da die Schnittstelle für den Zugriff geändert wurde, mussten alle Scripte einzeln angepasst werden, da wie in Kapitel 4.2.2 beschrieben, kein Refactoring mit der Entwicklungsumgebung (Eclipse) möglich ist.

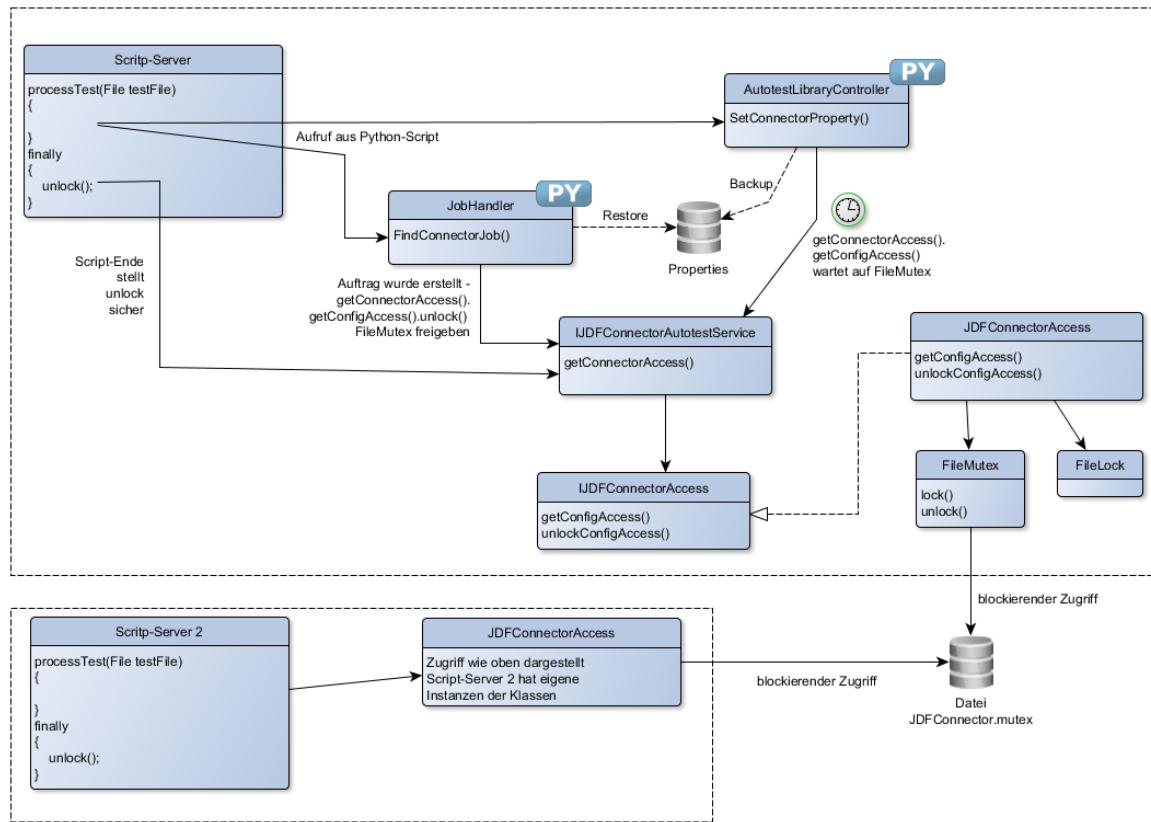


Abbildung 5.8: Erweiterung des FileMutex

Nachdem die Einstellungen durchgeführt wurden und der automatische Test das JDF in den Hotfolder des MISConnectors kopiert hat, wartet der automatische Test auf den vom MISConnector erzeugten Auftrag. Ist der Auftrag erzeugt, kann der Mutex mit dem Aufruf `IJDFConnectorAccess.unlockConfigAccess()` wieder frei gegeben werden. Der automatische Test kann mit dem erzeugten Auftrag fortgeführt werden und ein weiterer Test kann den MISConnector zur Erstellung eines Auftrags verwenden.

## 5.9 Erhöhung parallel ausgeführter Script-Server

Nachdem durch die Implementierung des FileMutex (Kapitel 5.8) das parallele Ausführen von automatischen Tests verbessert bzw. ermöglicht wurde, wird die Anzahl gleichzeitig laufender Script-Server von 2 auf 5 Instanzen erhöht. In Tabelle 5.3 sind die Ergebnisse aufgeführt. Verglichen mit dem Ausgangszustand konnte die Ausführungszeit um 35% reduziert werden.

Verbesserung	Ausführungszeit	Performance-Gewinn	CPU-Auslastung
ohne	5h 16min	-	18,8%
mit Mutex	4h 37min	12%	19,2%
mit Mutex 5 Script-Server	3h 25min	35%	24,0%

Tabelle 5.3: Steigerung der Ausführungszeit mit Mutex und Erhöhung der Script-Server

## 5.10 Erweiterung zur Continuous Delivery Pipeline

### 5.10.1 Pipeline-Prototyp mit Jenkins 2.x

Um das neue Feature von Jenkins 2.x verwenden zu können, wird ein Server mit der neueren Version benötigt. Dazu wurde der aktuelle Jenkins Master in der Version 1.x, der in einer Linux-VM läuft, geklont. Danach kann Jenkins auf die neuste Version aktualisiert werden. Auf dem Clone sind alle Jobs, Einstellungen und Plugins weiterhin vorhanden. Es können alle Funktionen getestet werden, ohne den Entwicklungsbetrieb zu stören. Beim ersten Test wurde festgestellt, dass die Jobs alle ohne Probleme mit der neuen Version funktionierten, so dass der geklonte Server als neuer Master eingesetzt wurde. Für eine kleine, überschaubare Komponente wurde eine prototypische Pipeline erstellt (siehe Abbildung 5.9).

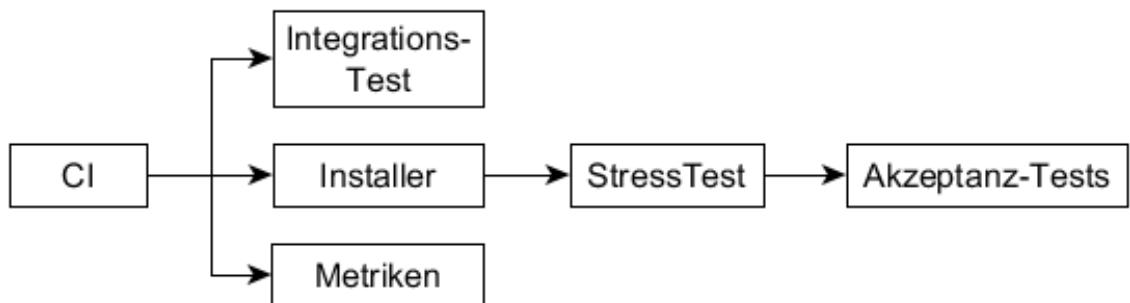


Abbildung 5.9: Prototypische Pipeline mit Jenkins 2.x

Eine weitere Möglichkeit, die neusten Jenkins Versionen zu testen, ist der Einsatz von Docker.

CloudBees<sup>2</sup> stellt alle Jenkins-Versionen als Docker-Image in einem öffentlichen Repository<sup>3</sup> zur Verfügung. Das Image einer bestimmten Version kann durch einen Befehl von dem DockerHub geladen werden. Folgender Aufruf lädt das Image mit der zurzeit aktuellsten Version von Jenkins herunter:

```
docker pull jenkinsci/jenkins:latest
```

Die Darstellung der Pipeline als Stage View in Jenkins zeigt Abbildung 5.10. Der letzte Schritt muss bestätigt werden, da die Akzeptanz-Tests von Qualitäts-TesterInnen manuell durchgeführt werden. In der untersten Zeile wurde der Akzeptanz-Test abgebrochen. In der Zeile darüber ist die Stage als bestanden markiert. In der obersten Zeile wird das Auswahl-Fenster der beiden Entscheidungsmöglichkeiten gezeigt, das beim Klicken mit der Maus geöffnet wird. Der manuelle Test soll nicht bei jedem Commit erfolgen, sondern verantwortliche TesterInnen entscheiden, wann die Version für weitere Tests freigegeben wird [Spa16]. Da nur Verantwortliche über die Freigabe entscheiden dürfen, kann der Kreis der Jenkins BenutzerInnen mit dem Befehl *submitter='benutzername'* eingeschränkt werden. Der Code der Pipeline ist in Listing 6.7 aufgeführt. In der Konsolenausgabe (siehe Abbildung 5.11) ist zu sehen, dass die drei Aufträge der Integrations-, Installer- und Metrics-Stage parallel auf drei unterschiedlichen Jenkins-Slaves ausgeführt werden. Die Stage der automatischen Tests sowie der Akzeptanz-Tests soll nicht mehrfach gleichzeitig durchlaufen werden, da die Ressourcen in Form von Test-Systemen und TesterInnen nicht verfügbar sind. In der Pipeline kann die Anzahl parallel ausgeführter Stages mit dem Befehl *concurrency=n* beschränkt werden. Für die beiden Stages wurde der Wert auf 1 gesetzt. Sollte ein Build in der Pipeline die Akzeptanz-Tests erreichen, wird der vorherige Durchlauf automatisch abgebrochen, falls die Akzeptanz-Tests dort noch zur Bestätigung anstehen sollten.

### 5.10.2 Erweiterung von Continuous Integration

Die Pipeline wurde wie in Abbildung 5.12 umgesetzt, in dem das aktuell verwendete Jenkins Projekt für Continuous Integration um die Stage 3 erweitert wurde. Die Umsetzung erfolgt zunächst nicht mit dem neuen Pipeline-Feature von Jenkins. Der Fokus liegt auf der Umsetzung der Stage 3. Diese kann mit Umstellung auf eine Pipeline wieder verwendet werden. In der Stage 1 erfolgt der Build und die Unit-Tests eines Commits. Nach dessen Erfolg wird der geänderte Code auf dem Developer-Branch für alle EntwicklerInnen sichtbar eingeccheckt. Die Dauer der Stage 1 ist abhängig von den geänderten Komponenten. Wird eine UserInterface-Komponente geändert, von der keine weiteren Komponenten abhängig sind, dauert der Build und die Unit-Tests nur 1 Minute und 20 Sekunden. Werden hingegen viele Komponenten oder eine Basis-Komponente geändert, von der viele andere Komponenten abhängig sind, so werden alle Komponenten neu gebaut und deren Unit-Tests ausgeführt. Die maximale

---

<sup>2</sup><https://www.cloudbees.com/>

<sup>3</sup><https://hub.docker.com/r/jenkinsci/jenkins/tags/>

## 5 Umsetzung der Konzepte

### Stage View

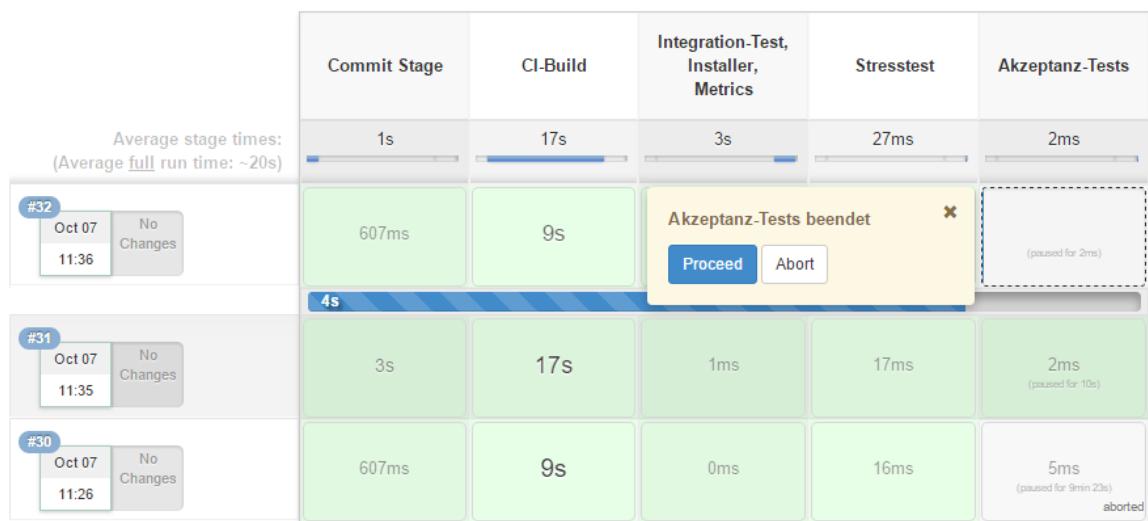


Abbildung 5.10: Anzeige der Pipeline in Jenkins

BUILD SUCCESSFUL

```
Total time: 1 mins 28.803 secs
Build finished at: 2016-08-24 12:00:57

[Pipeline] }
[Pipeline] // node
[Pipeline] stage (Integration-Test, Installer, Metrics)
Entering stage Integration-Test, Installer, Metrics
Proceeding
[Pipeline] parallel
[Pipeline] [Intregration-Test] { (Branch: Intregration-Test)
[Pipeline] [Installer] { (Branch: Installer)
[Pipeline] [Metrics] { (Branch: Metrics)
[Pipeline] [Intregration-Test] node
[Intregration-Test] Running on KIE-JENKINSBLD1 in c:\jenkins2\workspace\PipelineTest
[Pipeline] [Installer] node
[Installer] Running on KIE-JENKINSBLD3 in c:\jenkins2\workspace\PipelineTest
[Pipeline] [Metrics] node
[Metrics] Running on KIE-JENKINSBLD4 in c:\jenkins2\workspace\PipelineTest
[Pipeline] [Intregration-Test] {
[Pipeline] [Installer] {
[Pipeline] [Metrics] {
[Pipeline] [Intregration-Test] echo
```

Abbildung 5.11: Konsolenausgabe der Pipeline

Ausführungszeit liegt bei 7 Minuten. Die Ausführungszeit von ursprünglich 30 Minuten (siehe Kapitel 3.1.2) konnte durch die Maßnahmen deutlich verbessert werden. Stage 2 stellt nur sicher, dass ein kompilierbarer Softwarestand auf dem Master-Branch eingecheckt wird. Aus diesem Grund wurde in der Umsetzung der Pipeline auf das Ausführen der Unit-Tests verzichtet. Sobald nach einem Durchlauf der Pipeline installierbare Versionen entstehen ist der nächtliche Build und somit die Stage 2 überflüssig.

Für Stage 3 wurde eine virtuelle Maschine für das Test-System auf der neuen Hardware (siehe Kapitel 5.2) mit 16 CPUs erstellt. Mit Stage 3 werden die erzeugten Artefakte auf das Test-System kopiert und die automatischen Tests ausgeführt. Die Ausführungszeit der Stage 3 setzt sich wie folgt zusammen (Tabelle 5.4): Die Zeit für das Herunterfahren der Services dauert mit 7 Minuten noch zu lange, fällt aber bei der langen Ausführungszeit (230 Minuten mit 5 Script-Servern) der automatischen Tests nicht ins Gewicht. Die Ausführungszeiten der automatischen Tests sind auf dem Test-Server langsamer als auf dem physikalischen Server (siehe Tabelle 5.3).

Aktion	Ausführungszeit
Kopieren der Artefakte Jenkins	vernachlässigbar
Herunterfahren Services	7 min 55 sek
Kopieren der Artefakte Workflow-System	vernachlässigbar
Hochfahren der Services	3 min 10 sek
Ausführung automatischer Tests	230 min
Gesamlaufzeit	241 min

Tabelle 5.4: Ausführungszeit von Stage 3

Jeder der drei Stages in der Pipeline kann nur einmal zurzeit ausgeführt werden und nicht parallel ablaufen. In einem Jenkins-Projekt kann dies durch die Einstellung '*Parallele Builds ausführen, wenn notwendig*' definiert werden. Jede Stage startet mit allen zuvor erfolgreich durchlaufenden Ergebnissen der vorherigen Stage. Aufgrund der schnellen Ausführungszeit von Stage 2 wird diese i.d.R. direkt nach Beenden von Stage 1 starten. Wird von einer mittleren Commit- und Ausführungszeit der Stage 1 von 10 Minuten ausgegangen, sind bei einem Durchlauf der Stage 3 von 4 Stunden in der Zwischenzeit 24 Änderungen durchgeführt worden. Stage 3 wird im Mittel also mit 24 Änderungen ausgeführt.

In der prototypischen Pipeline wird das Test-System nach dem Kompilieren des Source-codes auf den aktuellen Software-Stand gebracht. Auf dem Test-System wurde manuell ein Workflow-System installiert, das noch durch manuelle Arbeitsschritte für das Ausführen von automatischen Tests vorbereitet werden musste. Eine dynamische Skalierung der Test-Systeme setzt eine automatische Konfiguration voraus. Ziel sollte es sein, dass die automatischen Tests keine speziell eingestellten Konfigurationen benötigen. Die Tests sollten die benötigten Zustände vor der Ausführung selber herstellen. Alternativ könnten die Grundeinstellun-

## 5 Umsetzung der Konzepte

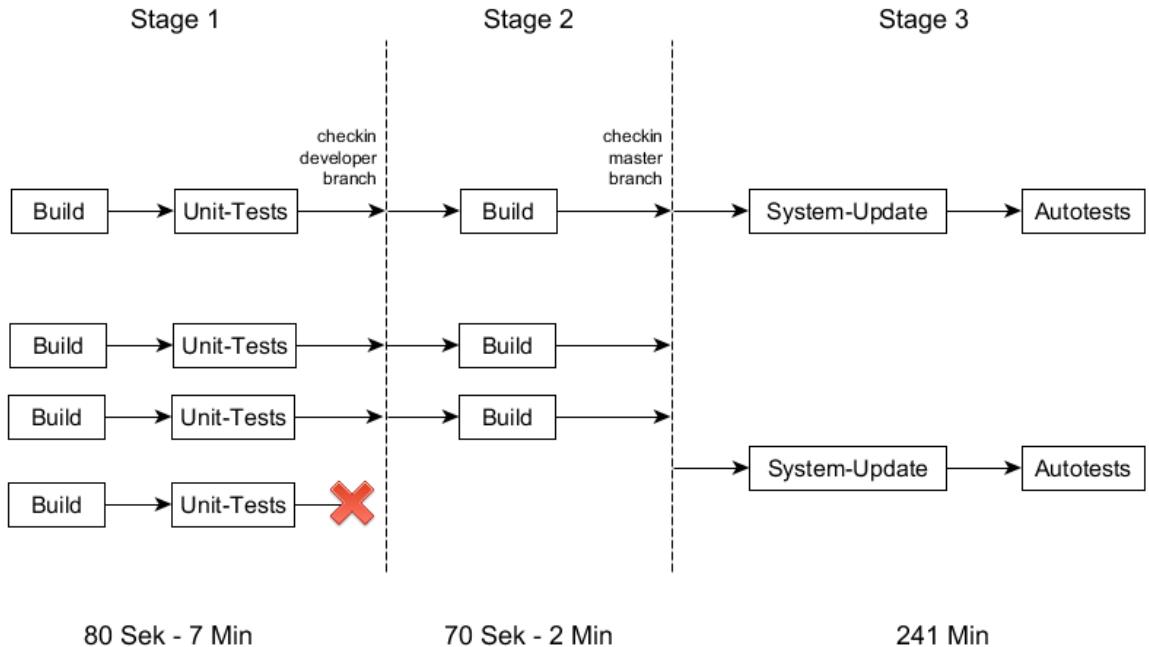


Abbildung 5.12: Umgesetzte Pipeline und deren Ausführungszeiten

gen auch durch einen initial ausgeführten Test eingestellt werden. Die Aktualisierung des Test-Systems ist in Abbildung 5.13 dargestellt. Auf dem Test-System wird ein Jenkins-Slave installiert. Dadurch wird das Kopieren der Build-Artefakte auf das Test-System ermöglicht. Zum Kopieren der Build-Artefakte und für den Zugriff auf den Update-Service wurde ein Programm (ArtifactDeployer Listing 6.8-6.10) implementiert. Das Programm bekommt den Pfad der Build-Artefakte und den Installations-Pfad des Workflow-Systems übergeben. Rekursiv wird im Installationsverzeichnis nach den zu ersetzenen Dateien gesucht, die auch mehrfach vorkommen können. Aus dem Grund wird in dem Programm für jede Build-Artefakt-Datei eine Liste mit Zielpfaden ermittelt (siehe Abbildung 5.14). Vor dem Kopieren der Dateien kommuniziert das Programm mit dem Update-Service, um die Services des Workflow-Systems in geordneter Reihenfolge zu beenden. Ist das Kopieren beendet, werden die Services wieder mittels Update-Service gestartet.

Das Programm ArtifactDeployer ist der prototypische Ersatz für einen Installer, der nach dem CI-Build erzeugt werden müsste. Durch die Suche nach den Dateien ist es nicht notwendig, den Zielpfad in dem installierten Workflow-System zu kennen. Die Pfade sind nur in den InstallShield-Dateien bekannt (siehe Kapitel 4.3.1). Für die erste Version der Pipeline ist der Prototyp ausreichend, bis eine endgültige Lösung für die Zerlegung des Deployment-Monolithen umgesetzt wurde.

Zum Kopieren der Artefakte wird ein neues Projekt in Jenkins angelegt. In der Konfiguration

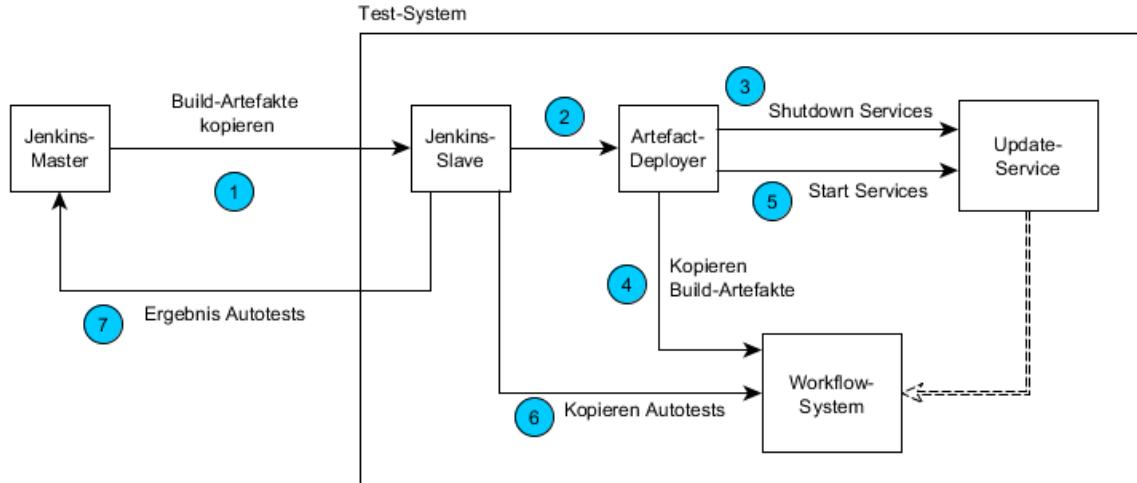


Abbildung 5.13: Aktualisierung eines Test-Systems in der Pipeline

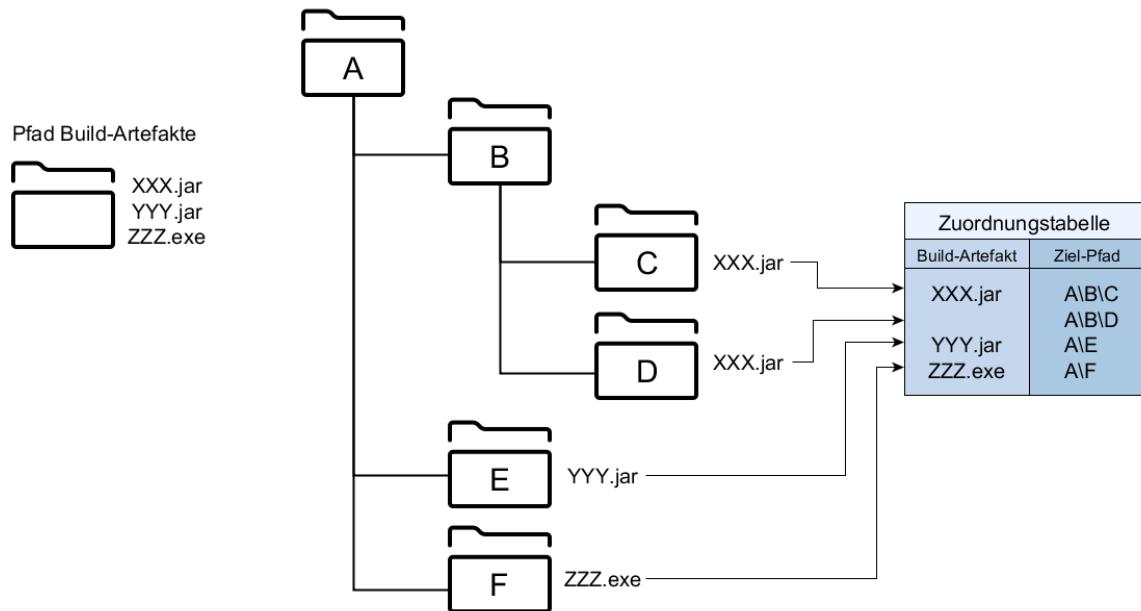


Abbildung 5.14: Aktualisierung eines Test-Systems in der Pipeline

wird angegeben, dass das Projekt nur auf dem Jenkins-Slave des Test-Systems ausgeführt werden darf. Als Build-Auslöser wird das Projekt für den CI-Build (PRT\_current\_CI) definiert (Abbildung 5.15). D.h. nach dem der CI-Build erfolgreich durchgelaufen ist, wird der Build-Job zum Kopieren auf dem Test-System ausgeführt. In der Einstellung Buildverfahren können Artefakte aus anderen Projekten kopiert werden. Für das neue Jenkins-Projekt, werden die

## 5 Umsetzung der Konzepte

Artefakte aus dem letzten erfolgreichen CI-Build verwendet und in das Verzeichnis *libs* kopiert. Dazu muss zunächst in der Postbuild-Aktion *Artefakte archivieren*, des CI-Jenkins-Jobs, die kompilierten jar- und war-Dateien archiviert werden, damit diese anschließend auf den Jenkins-Slave des Test-Systems kopiert werden können. Eine Archivierung erfolgt nur, wenn der Build erfolgreich durchgelaufen ist, da andernfalls kein anschließender System-Test durchgeführt wird. Abbildung 5.15 zeigt das Groovy-Script, das in dem neuen Projekt ausgeführt wird, um die Dateien in das Verzeichnis C:\Artefacts zu kopieren und die Versionsnummer am Ende des Artefakt-Dateinamens zu entfernen. Die mit Gradle erstellten jar- und war-Dateien enthalten am Ende die Version in dem Dateinamen. Im installierten Produkt kommen die Versionbezeichnungen nicht vor. Nach Ausführung des Groovy-Skripts wird das Java-Programm *ArtefactDeployer* zum Austauschen der Artefakte des Workflow-Systems mit einem Windows Batch-Datei Kommando gestartet. Für das Kopieren der Autotest-Skripte in den Hotfolder des Script-Servers existiert eine Batch-Datei, die ebenfalls per Windows-Batch in dem Jenkins-Projekt ausgeführt wird.

Am Ende fehlt noch die Auswertung, ob die automatischen Tests fehlerfrei durchgelaufen sind. Die Umsetzung erfolgt mit einer kleinen Java Applikation *TestAnalyser* (Listing 6.11), die von Jenkins als externer Jobs aufgerufen wird. Beim externen Job führt Jenkins diesen nicht selber aktiv aus, sondern ist nur Buchhalter. Über den Exit-Code des aufgerufenen Programms kann Jenkins mitgeteilt werden, ob der Job erfolgreich war. Der Exit-Code 0 wird als Erfolg und der Exit-Code ungleich 0 als Fehlschlag gewertet [Wie11]. Um das Ende der Tests festzustellen, überwacht das Programm das Verzeichnis, in dem die Script-Server die Ergebnisse der automatischen Tests schreiben. Der letzte System-Test wertet die automatischen Tests aus und schreibt bei Fertigstellung eine Datei *Zz-DailyAutoTestResult.finished* in das Verzeichnis mit den Ergebnissen. Das Warten auf die Datei wird mit dem nio-package von Java 7 implementiert. Damit ist eine Lösung ohne ständiges Pollen des Verzeichnisses möglich [Rob13].

Für einen Durchlauf der automatischen Tests existiert ein Test-Script, das die erfolgreichen und fehlgeschlagenen Tests auswertet und in eine Datei schreibt. Auf diese Datei muss von dem extern gestarteten Programm gewartet werden. Durch das Auslesen der Datei kann festgestellt werden, ob ein Test mit einer Fehlermeldung beendet wurde. In dem Fall wird das Programm mit einem Exit-Code ungleich 0 beendet.

Die Installation eines Jenkins-Slaves auf dem Test-System benötigt einige Arbeitsschritte. Als erstes muss ein neuer Jenkins-Slave-Knoten in dem Master eingerichtet werden. Dazu kann ein vorhandener Knoten als Vorlage zum Kopieren verwendet werden. Das Label differiert von den bisher vorhandenen Knoten (siehe Abbildung 5.16). Der neue Jenkins-Slave soll nur für das Ausführen von automatischen Tests und nicht für das Builden von Sourcecode verwendet werden. Dieser darf nicht mit in die Gruppe der Build-Slave aufgenommen werden, sondern erhält ein eigenes Label 'ALL\_CD\_SLAVES'.

## 5.10 Erweiterung zur Continuous Delivery Pipeline

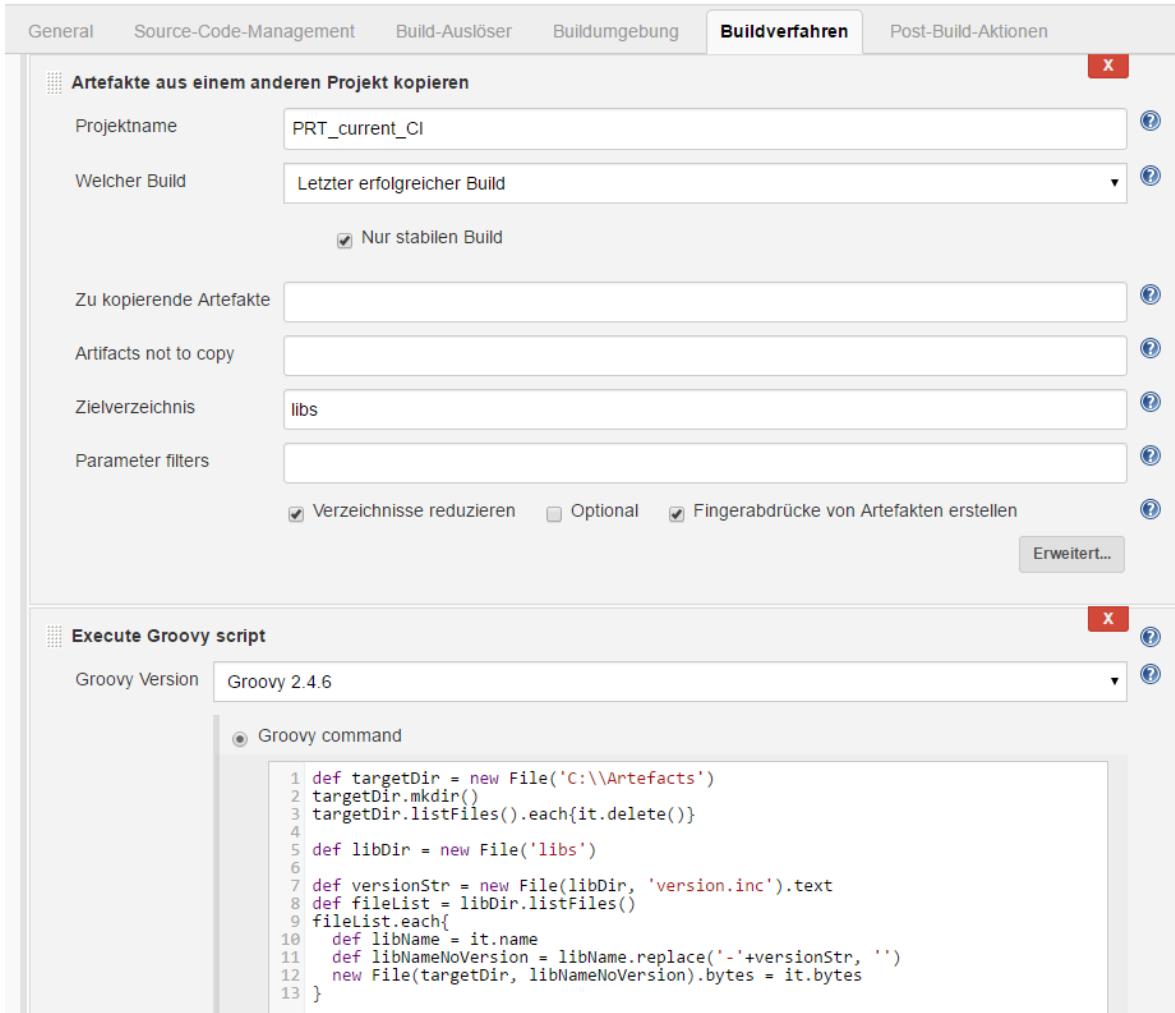


Abbildung 5.15: Jenkins Projekt zum Kopieren der Artefakte

Name	CONTINUOUS-DELIVERY-TEST
Beschreibung	a windows cd test slave
Anzahl der Build-Prozessoren	1
Stammverzeichnis in entferntem Dateisystem	c:\\jenkins2
Labels	ALL_CD_SLAVES

Abbildung 5.16: Einrichtung eines neuen Slave-Knotens

Für Jenkins existiert ein User-Account, der auf dem Test-System in die Rechte-Gruppe der Administratoren aufgenommen werden muss. Von einem Jenkins-Slave wird nun das

## 5 Umsetzung der Konzepte

Verzeichnis C:\Jenkins2 kopiert. In dem Verzeichnis (siehe Abbildung 5.17) befindet sich die Datei *jenkins-slave.xml*, in der ggf. der Pfad auf Java an den Installationspfad des Test-System angepasst werden muss, sofern dieser von der Installation des Jenkins-Slaves abweicht. Beim Anlegen des Jenkins-Slave-Knotens wird in dem Master eine URL angezeigt (Abbildung 5.18), diese muss in der XML-Datei eingetragen werden (Abbildung 5.19). Zur Installation des Jenkins-Slaves als Windows-Service wird eine Command-Shell mit Administrator-Rechten geöffnet und der Befehl *jenkins-slave.exe install* aufgerufen. Der neu erzeugte Service muss mit dem Account von Jenkins ausgeführt werden.

Name	Date modified	Type	Size
tools	13.10.2016 14:38	File folder	
workspace	14.10.2016 09:14	File folder	
jenkins-slave.err.log	13.10.2016 11:12	Text Document	1 KB
jenkins-slave.exe	22.08.2016 07:53	Application	58 KB
jenkins-slave.exe.config	22.08.2016 07:53	XML Configuratio...	1 KB
jenkins-slave.out.log	22.08.2016 08:07	Text Document	33 KB
jenkins-slave.wrapper.log	13.10.2016 12:04	Text Document	5 KB
jenkins-slave.xml	13.10.2016 12:04	XML File	3 KB
slave.jar	15.12.2015 12:20	Executable Jar File	495 KB

Abbildung 5.17: Verzeichnis eines Jenkins-Slaves

The screenshot shows the Jenkins master interface with the following details:

- Agent CONTINUOUS\_DELIEVERY\_TEST (a windows cd test slave)**: The name of the slave node.
- Knoten temporär abschalten**: A button to temporarily disable the node.
- Connect agent to Jenkins one of these ways:**
  - Launch**: A button to launch the agent from a browser.
  - Run from agent command line:** A link to a command-line script:

```
java -Xmx1024M -jar slave.jar -jnlpUrl http://kie-jenkinslts.b998271b605539b869060cdbd77fb59a65af45b8275544b05230274dbe243373 com:8080/computer/CONTINUOUS_DELIEVERY_TEST/slave-agent.jnlp -secret
```
- Labels**: The label assigned to this slave node.
- ALL\_WORKFLOW\_CD\_SLAVES**: The label group to which this slave belongs.

Abbildung 5.18: Verbindung vom Jenkins-Master zum Slave

```
<!--
Windows service definition for Jenkins slave

To uninstall, run "jenkins-slave.exe stop" to stop the service, then "jenkins-slave.exe uninstall" to uninstall the service.
Both commands don't produce any output if the execution is successful.
-->
<service>
  <id>jenkinsslave-c_jenklns2</id>
  <name>jenkinsslave-c_jenklns2</name>
  <description>This service runs a slave for Jenkins continuous integration system.</description>
  <!--
    If you'd like to run Jenkins with a specific version of Java, specify a full path to java.exe.
    The following value assumes that you have java in your PATH.
  -->
  <executable>C:\Program Files\Java\jre1.8.0_101\bin\java.exe</executable>
  <arguments>-Xrs -Xmx1G -jar "%BASE%\slave.jar" -jnlpUrl http://kie-jenkinsls...com:8080/computer/KIE-JENKINS-CD/slave-agent.jnlp -secret cd38391b3cf565ad091e99133e9c8c6a
  <!--
    interactive flag causes the empty black Java window to be displayed.
    I'm still debugging this.
  -->
  <logmode>rotate</logmode>
  <onfailure action="restart" />
</service>
```

Abbildung 5.19: Jenkins-Slave Konfigurationdatei



# 6 Zusammenfassung und Ausblick

In diesem Kapitel werden die zu lösenden Probleme einzeln zusammengefasst und in einem Ausblick weitere mögliche Schritte erwähnt. Zunächst erfolgt ein Überblick der messbaren Verbesserung in den Tabellen 6.1 und 6.2.

Maßnahmen	Kapitel	Alter Wert	Neuer Wert	Zeitgewinn
Optimierung der JVM-Erzeugung	5.1	30 min	20 min	33%
Optimierung der Hardware	5.2	20 min	7 min	65%
Beide Maßnahmen zusammen		30 min	7 min	<b>77%</b>

Tabelle 6.1: Verbesserte Ausführungszeit des CI-Builds (Problem I)

Maßnahmen	Kapitel	Alter Wert	Neuer Wert	Zeitgewinn
Einführung Mutex zur Verbesserung der Parallelität	5.8	5h 16min	4h 37min	12%
Erhöhung Anzahl der Script-Server	5.9	5h 16min	3h 25min	<b>35%</b>

Tabelle 6.2: Verbesserte Ausführungszeit der System-Tests (Problem VIc)

Es folgen die Zusammenfassungen und Ausblicke der einzelnen Probleme:

## Problem I - Ausführungszeit CI-Build

Ergebnis: Die Ausführungszeit der Unit-Tests und des CI-Builds konnten auf ein akzeptables Niveau gesenkt werden. In Abhängigkeit von den geänderten Komponenten dauert der CI-Build jetzt zwischen 80 Sekunden und 7 Minuten.

Ausblick: Wie in Kapitel 4.3 vorgesehen, könnten die langsamten Unit-Tests separat in der Pipeline ausgeführt werden. Der Code der Unit-Tests, die in einer eigenen JVM ausgeführt werden, sollten von den statischen Zuständen befreit werden. Dadurch könnten alle Unit-Tests in der selben JVM ablaufen und die Sonderbehandlung mit der Namensgebung (OwnForkTest Kapitel 5.1) wäre nicht mehr notwendig.

## Problem II - Lesbarkeit der Unit-Tests

Ergebnis: Zur Verbesserung der Lesbarkeit von Unit-Tests und Verwendung der Hamcrest-Matcher im Produktionscode, wurde die Hamcrest-Library mit in das Aritfactory der Workflow-Software aufgenommen. Die EntwicklerInnen werden in einem Meeting über die

## 6 Zusammenfassung und Ausblick

Möglichkeiten von Hamcrest informiert. Weitere Maßnahmen zur Sicherstellung der Qualität von Unit-Tests wurden nicht getroffen. Das Problem wurde mit aufgeführt, da es für eine Continuous Delivery Pipeline wichtig ist, viele automatische Tests mit guter Qualität zu besitzen. Die Coderichtlinien für den Produktionscode sollten in gleicher Weise für die Unit-Tests gelten. Für Continuous Delivery sind die Implementierungen alle beteiligten Teams stets mit Unit-Tests abzusichern. Nach dem Grundgedanken der agilen Vorgehensweise *Scrum* sind die Teams dafür allein zuständig. Gleiches würde für Teams eines Microservices gelten.

### Problem III - Testbarkeit des Legacy-Codes

Ergebnis: Für eine bessere Testbarkeit des Codes wurden für die Userinterface Komponente neue fachlich, getrennte Komponenten erstellt, die keine Abhängigkeiten mehr zu den Implementierungen der Anwendungslogik und in die Persistenzschicht haben (Kapitel 4.1.3). Die Anwendungslogik soll nicht mehr im Userinterface implementiert werden, damit diese in automatischen System-Tests überprüft werden kann. Durch die Einführung der ServiceRegistry (Kapitel 4.2.3), bleibt die Implementierung des Codes generell durch die Interfaces den ServiceRegistry-Konsumenten verborgen. Der Code wird dadurch einfacher testbar, da alle Services aus der ServiceRegistry durch Stellvertreter-Objekte ersetzt werden können. Damit sich in der neuen Parallelwelt keine ungewünschten Abhängigkeiten einschleichen wurde ein Unit-Test erstellt (Kapitel 5.4), mit dem die Abhängigkeiten von Packages kontrolliert werden können. Es soll erreicht werden, dass der Code verständlich und testbar bleibt.

Ausblick: In den Teams sollte über eine testgetriebene Entwicklung nachgedacht werden. Das bisherige Verhalten, schnell zu implementieren und nicht auf Testbarkeit und Übersichtlichkeit des Codes zu achten, lässt sich in der über Jahre gewachsenen AutoTest-Library wiederfinden. Hier enthält eine Klasse die umfangreichste Funktionalität für die Ausführung der automatischen Tests. Diese Klasse ist 5491 Zeilen lang und besitzt 343 Methoden. Daher sollte der Code von neuen Implementierungen auch immer im Hinblick auf die Testbarkeit optimiert werden. Das Ziel ist erreichbar, wenn nach der Designstrategie der testgetriebenen Entwicklung vorgegangen wird. Weiterhin wird die Erstellung von automatischen Tests, für den erfolgreichen Einsatz von Continuous Delivery, im Software-Entwicklungsprozess verankert (siehe auch Kapitel 3.5.5).

### Problem IV - Fehleranalyse automatischer System-Tests

Ergebnis: Zur Erleichterung der Analyse von fehlgeschlagenen System-Tests, wird für jeden Test eine separate Logdatei geschrieben (Kapitel 5.5). System-Tests, die mit JUnit entworfen worden sind, können direkt am Entwicklungsrechner ausgeführt und gedebuggt werden. Das aufwendige Patchen und Debuggen eines Workflow-Systems kann entfallen (Kapitel 5.7).

Ausblick: Die implementierten Umsetzungen erleichtern den EntwicklerInnen zwar die Arbeit bei der Analyse, aber ein großer Fortschritt wäre die Big-Bang-Integration der externen

Komponenten abzuschaffen (Kapitel 3.5.4). Vor der Integration jeder Komponente sollten die Änderungen mit System-Tests überprüft werden. Als Test-System kann eine virtuelle Maschine dienen, die den letzten Stand eines fehlerfrei getesteten Systems hat (siehe Abbildung 6.1). Die Forderung ist im Endeffekt, dass jede Komponente über eine eigene Continuous Delivery Pipeline verfügen sollte. Treten Fehler in den Tests auf, muss der Entwickler die Fehleranalyse machen, der die Änderung gemacht hat. Mit den Kenntnissen über die Änderung ist die Fehleranalyse einfacher. Es werden nur fehlerfreie Versionen ins Artifactory übernommen.

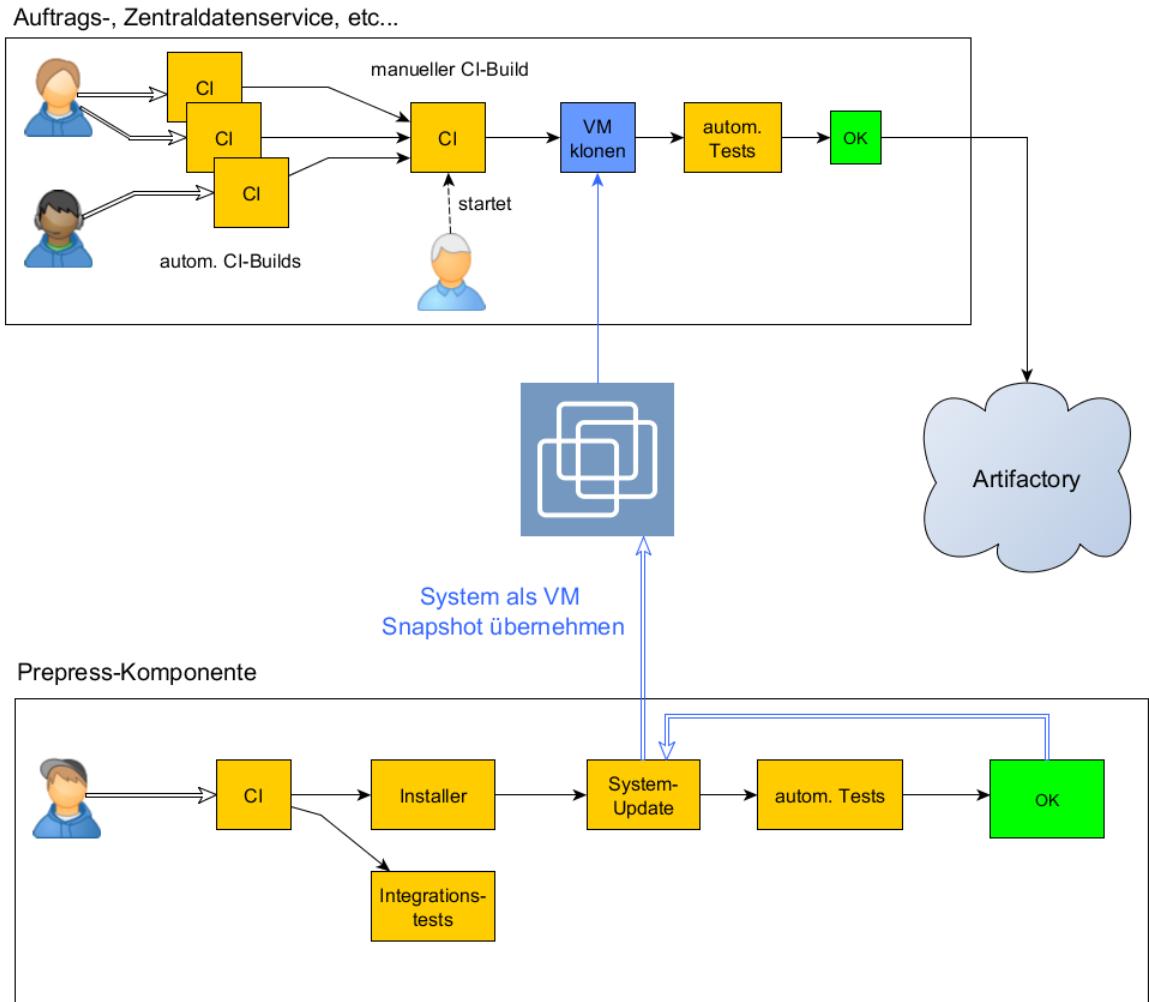


Abbildung 6.1: Automatische Tests für externe Komponenten

Das Konzept für externe Komponenten, den letzten fehlerfrei getesteten Software-Stand der Workflow-Software zur Verfügung zu stellen, kann mit der Pipeline-Funktionalität von Jenkins 2.x umgesetzt werden. Unter den verfügbaren Schritten<sup>1</sup> der Jenkins-Pipeline existie-

<sup>1</sup><https://jenkins.io/doc/pipeline/steps/>

## 6 Zusammenfassung und Ausblick

ren auch vSphere-Aktionen, wie das Klonen einer virtuellen Maschine. Nach dem fehlerfreien Durchlauf der Continuous Delivery Pipeline kann am Ende mit dem Clone-Befehl ein Test-System als Vorlage für die externen Komponenten angelegt werden.

### Problem V - Vereinfachung der Test-Erstellung

Ergbnis: Damit automatische System-Tests von allen Java-EntwicklerInnen ohne langwierige Einarbeitung erstellt werden können, wurde der Script-Server erweitert JUnit-Tests ausführen zu können (Kapitel 5.6). Eine weitere Erleichterung ist, dass die System-Tests wie ein Unit-Test mit der Entwicklungsumgebung ausgeführt werden können (Kapitel 5.7). Innerhalb von drei Monaten wurden 17 neue Tests mit der neuen Implementierungsmöglichkeit entwickelt. Dies ist eine deutliche Steigerung gegenüber der Entwicklung vergangener Software-Releases.

### Problem VIa,b - Dauer des Installations-Prozesses

Ergbnis: Die Paketierung und die Installation (Problem VIa,b) werden von einem Prototypen ersetzt (Kapitel 5.10.2). Die Ausführung des Prototypen ist sehr schnell und fällt bei der Aktualisierung des Test-Systems nicht ins Gewicht.

Ausblick: Das Deployment des Workflow-Systems erfolgt durch einen monolithischen Installer. Dieser enthält mehrere Subinstaller, die Funktionalitäten wie Stamm- oder Auftragsdaten verwalten. Die Ausrichtung ist eher eine organisatorische, anstatt -wie für Microservices gefordert- eine fachliche. Das Workflow-System fachlich in Microservices aufzuteilen, sollte vorher analysiert und gut durchdacht werden. Für Continuous Delivery ist es, statt der Fachlichkeit, viel wichtiger, dass die Services unabhängig installiert werden können. Andere Services sollten mit dem kurzen Ausfall anderer Services zurecht kommen. Wird ein Service zur Aktualisierung beendet, werden von dem Update-Service alle direkt und indirekt abhängigen Services beendet. Zurzeit dauert das Beenden und Starten der Services über 11 Minuten. Wird eine Ausführungszeit der System-Tests von 60 Minuten angepeilt, würde die Aktualisierung einen zu hohen Anteil einnehmen. Ist ein Service unabhängig, kann dieser ohne andere Services zu beeinflussen ausgetauscht werden. Dazu ein grober Entwurf, den MISConnector als unabhängigen deploybaren Service zu entwickeln (siehe Abbildung 6.2). Werden alle Abhängigkeiten von dem MISConnector mit einer REST-Schnittstelle entkoppelt, kann dieser in einem Docker-Container ausgeführt werden. Wird der Code des MISConnectors geändert, entsteht durch die Continuous Delivery Pipeline ein Installer. Dieser wird verwendet um ein Docker-Image zu erzeugen, das in der Docker-Registry abgelegt wird. Auf den Test-Systemen wird nur der ausgeführte Container mit der aktuellen Version getauscht. Wird das System auf das Vorhandensein von mehreren MISConnectoren erweitert, bestünde die Möglichkeit, dass jeder automatische Test seinen eigenen MISConnector in einem Docker-Container ausführt. Die Probleme beim parallelen Ausführen von System-Tests in Kapitel 4.2.4 wären damit ohne

Mutex gelöst.

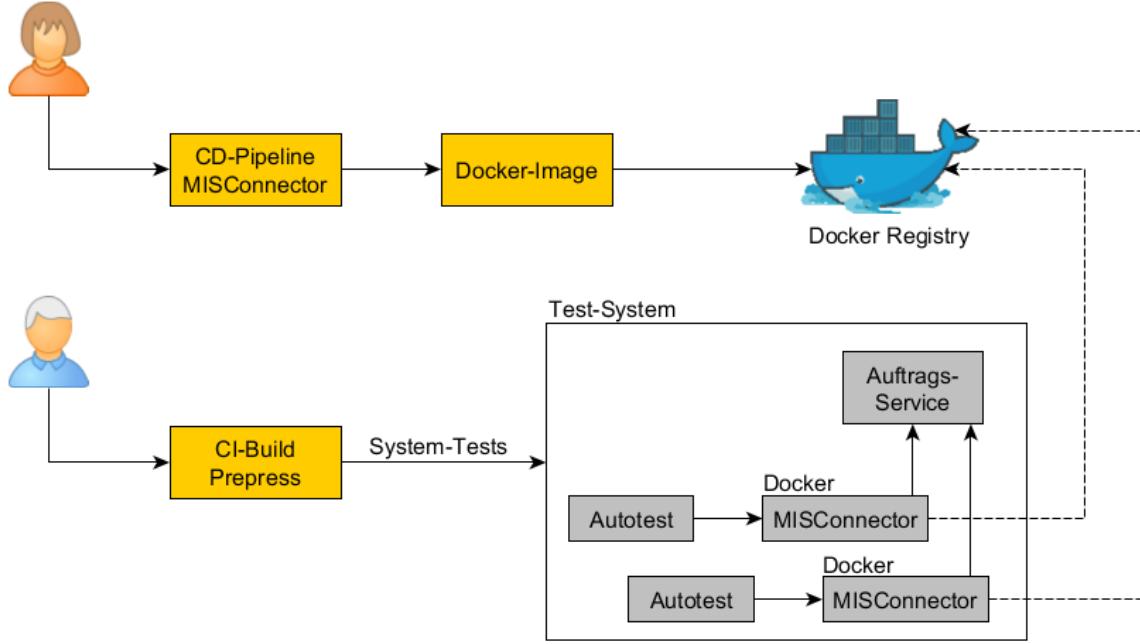


Abbildung 6.2: MISConnector als unabhängiger Service

### Problem VIc - Ablaufgeschwindigkeit der automatischen Tests

Ergebnis: Die automatischen Tests konnten beschleunigt werden indem die Parallelität der Testausführung erhöht wurde. Durch die Integration eines Mutex in dem Script-Server konnte die Wartezeit von einigen Tests gesenkt werden (Kapitel 5.8). Weitere Verbesserungen konnten durch die Erhöhung gleichzeitig ausgeführter Script-Server erreicht werden (Kapitel 5.9).

Ausblick: Die Laufzeit der automatischen Tests ist immer noch sehr hoch. Da die System-Tests von vielen unterschiedlichen Teams erstellt wurden, sollte untersucht werden, ob es in den System-Tests zu Überschneidungen bei der Funktionsprüfung kommt. Durch eine Konsolidierung könnten Tests zusammengelegt oder gekürzt werden. Da die CPU bei Testausführung mit erhöhter Anzahl an Script-Servern noch nicht ausgelastet ist (24%), könnte die Anzahl der Instanzen für eine Beschleunigung weiter erhöht werden. In der Zukunft könnten weitere Verbesserungspotentiale, wie die Skalierung der Test-Systeme oder bessere Hardware, zur Senkung der Zeit genutzt werden.

Die automatischen System-Tests des Workflow-Systems sind über die Jahre im Kontext einer sequentiellen und alphabetischen Abarbeitung entwickelt worden. Durch eine Skalierung der automatischen System-Tests auf unterschiedliche Test-Systeme bzw. Script-Server kann es vorkommen, dass Tests scheitern, da die Änderungen am System eines vorher abgelaufenen Tests nicht mehr vorhanden sind. Um dies zu verhindern, müssen die automatischen System-

## 6 Zusammenfassung und Ausblick

Tests angepasst werden, damit diese unabhängig von anderen Tests korrekte Ergebnisse liefern können. Jeder Test muss seinen benötigten Zustand selber herstellen. Nach Birk und Lukas [BL16a] ist eine Voraussetzung für Continuous Delivery, dass alle automatischen Tests unabhängig voneinander sind und die Laufzeit angemessen bleibt.

Konkurrierende Zugriffe auf Ressourcen eines Systems können bei parallel ausgeführten Tests immer auftreten. In dem Fall müssen die Zugriffe über einen wechselseitigen Ausschluss erfolgen. Werden viele Tests parallel ausgeführt, fällt es nicht ins Gewicht, wenn ein Test auf den Zugriff auf eine exklusive Ressource warten muss. Bei einer verstärkten parallelen Ausführung sind auch die höheren Wartezeiten zu berücksichtigen. Zum Beispiel erhält der Service zum Belichten der Druckplatten von verschiedenen Tests mehrere Aufträge zur Abarbeitung. In den Tests ist die höhere Wartezeit zu berücksichtigen, damit der Test nicht mit einem Timeout abbricht.

### Problem VIId - Einführung Continuous Delivery

Ergebnis: Für eine kleine Komponente des Systems wurde mit dem neuen Pipeline-Feature von Jenkins 2.x eine prototypische Pipeline erstellt (Kapitel 5.10.1). Für die prototypische Umsetzung von Continuous Delivery für das Workflow-System wurde ein Jenkins-Job erstellt, der nach Continuous Integration ausgeführt wird. Der Job aktualisiert ein Test-System und führt dann die System-Tests aus. Die Testergebnisse werden Jenkins über den Exit-Code des aufgerufenen Programms mitgeteilt (Kapitel 5.10.2).

Ausblick: Als nächsten Schritt sind die jetzigen Jenkins-Jobs in eine Pipeline zu überführen. Der erstellte Prototyp zur Aktualisierung des Test-Systems kann für die weitere Umsetzung genutzt werden.

Der Script-Server schreibt die Ergebnisse der automatischen Tests in einen Ordner, der das aktuelle Datum enthält 'Autotest-yyyy-MM-DD'. Bisher sind die Tests nur einmal am Tag gelaufen. Durch Continuous Delivery wird es mehrere Durchläufe pro Tag geben. Der Ordner für die Test-Ergebnisse sollte von dem Script-Server ohne die Datumsangabe angelegt werden, da die Implementierung auf das Warten der Ende-Datei einfacher ist. Am Ende der Pipeline sollten die Ergebnisse der Tests von Jenkins mit archiviert werden und auf dem Test-System gelöscht werden. Eine explizite Angabe des Datums in dem Ordnernamen ist damit überflüssig.

Gegenwärtig wird der Installer des Workflow-Systems noch in einem nächtlichen Build erstellt. In Zukunft sollte nach jedem Durchlauf der Continuous Delivery Pipeline eine installierbare Software entstehen. Dazu ist es erforderlich eine Alternative für den monolithischen Workflow-Installer zu schaffen. Eine Lösungsmöglichkeit ist, dass der Installer eine Hülle für die zahlreichen modularen Installer ist. Der Workflow-Installer wird aus der letzten Version der einzelnen Installer gebaut, die im Artifactory abgelegt sind. Das Erstellen des Installers sollte keine große Zeit in Anspruch nehmen. In Abbildung 6.3 ist ein möglicher Ablauf skizziert,

wie der Workflow-Installer nach Änderung des MISConnectors erstellt werden könnte.

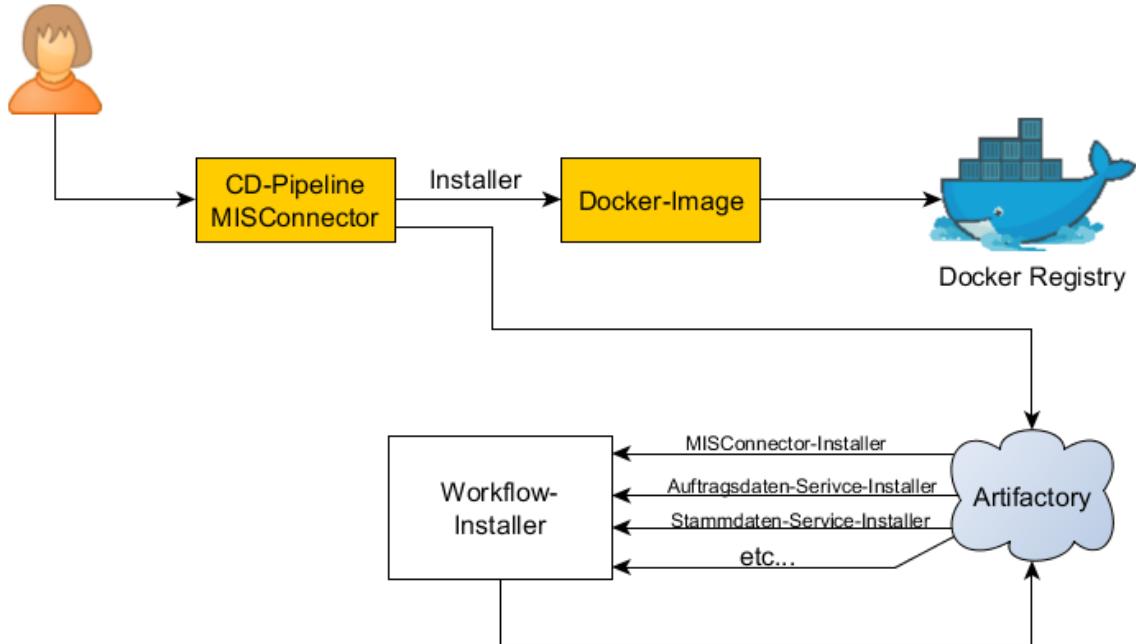


Abbildung 6.3: Erstellen des Workflow-Installers nach Änderung vom MISConnector

Die Einführung von Continuous Delivery ist für große und gewachsene Systeme eine langwierige Aufgabe. Besonders die Aktualisierung der Software kann für das Workflow-System nicht ohne hohen Aufwand geändert werden. Für die vollständige Einführung von Continuous Delivery ist ein stetiges weiterentwickeln einiger angesprochener Probleme notwendig. Der Installer des Workflow-Systems wird einer der nächsten Schritte sein.



# Appendix A

## 1 Service-Registry

Listing 6.1: Implementierung einer Service-Registry

```
package de.fhlubeck.lockemar.services;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ServiceRegistry
{
    private static Map<Class<? extends IService>, Object> sm_services = new ConcurrentHashMap<>();

    public static <T extends IService> void registerService(final Class<T> serviceClass,
        final T service)
    {
        if(serviceClass.isInterface())
        {
            sm_services.put(serviceClass, service);
        }
        else
        {
            throw new UnsupportedOperationException("Class " + serviceClass + " ist kein Interface");
        }
    }

    public static <T extends IService> T getService(final Class<T> serviceClass)
    {
        return (T) sm_services.get(serviceClass);
    }
}
```

## 2 Log-Datei Erweiterung

Listing 6.2: Erstellung eines Logfiles pro Autotest

```
private static FileAppender createFileAppender(final File file)
{
    final FileAppender fileApp = new FileAppender();
```

## Appendix A

```
final Path logPath = getLogPath();
final LocalDateTime now = LocalDateTime.now();
final DateTimeFormatter formatter = DateTimeFormatter.ofPattern(TIME_FORMAT_PATTERN);
final String formattedDateTime = now.format(formatter);
final File logFile = new File(logPath.toFile(), getTestName(file) + formattedDateTime
    + "_" + getScriptServerName() + LOGFILE_SUFFIX );

try
{
    fileApp.setFile( logFile.getCanonicalPath() );
    sm_Log.info( "Set Path for LogFile: " + logFile.getCanonicalPath() );
}
catch( final IOException e )
{
    sm_Log.error("Can get getCanonicalPath. " , e);
}

fileApp.setLayout( new DebugTraceLog4jLayout() );
fileApp.activateOptions();
fileApp.setThreshold(Level.DEBUG);
fileApp.setAppend( true );

return fileApp;
}
```

## 3 FileMutex

Listing 6.3: Implementierung Zugriffsbeschränkung auf MISConnector

```
package de.fhlubeck.lockemar.mutex;

import java.io.File;
import java.nio.channels.FileLock;

import com.heidelberg.prinect.baselayer.utils.mutex.FileMutex;
import com.heidelberg.prinect.utils.CommonUtils;
import com.heidelberg.printready.autotest.util.connector.IJDFConnectorAccess;

/**
 * @author Martin Locker
 *
 */
class JDFConnectorAccess implements IJDFConnectorAccess
{
    private static final String MUTEXT_FILENAME = "JDFConnector.mutex";

    private static FileMutex sm_fileMutex;

    private FileLock m_lock;

    static
```

```

{
    final File mutextFile = new File(CommonUtils.getTempDir(), MUTEXT_FILENAME);
    sm_fileMutex = new FileMutex( mutextFile.toPath() );
}

@Override
public void getConfigAccess( )
{
    m_lock = sm_fileMutex.lock(); // Auf Lock warten
}

@Override
public void unlockConfigAccess( )
{
    if(m_lock != null)
    {
        sm_fileMutex.unlock( m_lock );
        m_lock = null;
    }
}
}

```

Listing 6.4: Implementierung des FileMutex

```

package de.fhlubeck.lockemar.mutex;

import java.io.IOException;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 *
 * @author Martin Locker
 *
 */
public class FileMutex
{
    private static final Log sm_Log = LogFactory.getLog( FileMutex.class );

    private final Path m_pathForMutex;

    public FileMutex( final Path pathForMutex )
    {
        super();
        m_pathForMutex = pathForMutex;
    }
}

```

## Appendix A

```
/***
 *
 * @return null wenn es Probleme beim Zugriff auf das File gibt
 */
public FileLock lock()
{
    FileLock fileLock = null;
    final boolean check = checkAndCreateFile(m_pathForMutex);
    if(check)
    {
        final FileChannel fileChannel;

        try
        {
            sm_Log.info("Try to get mutex for " + m_pathForMutex.toAbsolutePath());
            fileChannel = FileChannel.open(m_pathForMutex, StandardOpenOption.WRITE);
            fileLock = fileChannel.lock(); // Wenn Lock von anderem Programm, dann
                                         // bleibt Thread hier stehen
            sm_Log.info("Get mutex for " + m_pathForMutex.toAbsolutePath());
        }
        catch (final IOException e)
        {
            sm_Log.error("Error at lock File '" + m_pathForMutex.toAbsolutePath() + "'.
                         ", e);
        }
    }
    return fileLock;
}

public void unlock(final FileLock lock)
{
    if(lock != null)
    {
        try
        {
            sm_Log.info("Release lock for File '" + m_pathForMutex.toAbsolutePath() + "
                        . ");
            lock.release();
        }
        catch( final IOException e )
        {
            sm_Log.error( "Error at release the lock ", e );
        }
    }
}

private boolean checkAndCreateFile(final Path path)
{
    boolean isOK = false;

    if(path != null)
```

```

    {
        final boolean exists = Files.exists( path );
        if(exists)
        {
            isOK = true;
        }
        else
        {
            try
            {
                final Path createdFile = Files.createFile( path );
                if(createdFile != null)
                {
                    sm_Log.info( "Mutex file '" + createdFile.toAbsolutePath() + "'"
                                + " created" );
                    isOK = true;
                }
            }
            catch( final Throwable t )
            {
                sm_Log.error( "Can't create File " + path.toAbsolutePath(), t );
            }
        }
        return isOK;
    }
}

```

## 4 Verwendung JUnit in automatischen Tests

Listing 6.5: Ausführung von Event-Handling einer Testklasse

```

package de.fhuebeck.lockemar.unittest;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import org.apache.log4j.Logger;
import org.junit.runner.Description;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.Runner;
import org.junit.runner.notification.Failure;
import org.junit.runner.notification.RunListener;
import org.junit.runners.model.InitializationError;

import com.heidelberg.prinect.common.status.IPrinectResult;
import com.heidelberg.prinect.common.status.PrinectResult;
import com.heidelberg.printready.autotest.util.logging.JUnitLogger;

```

## Appendix A

```
import com.heidelberg.printready.error.PrinectMessage;

/**
 *
 * @author Martin Locker
 *
 */
public final class UnitTestInvoker
{
    private static final Logger sm_Log = Logger.getLogger(UnitTestInvoker.class);

    private static Collection<String> sm_Failures = new ArrayList<String>();

    private UnitTestInvoker() {}

    public static void runUnitTest(final String fullQualifiedClassname) throws
        PrinectResult
    {
        if(fullQualifiedClassname != null)
        {
            try
            {
                final Class<?> unitTestClass = Class.forName(fullQualifiedClassname);
                runUnitTest(unitTestClass);
            }
            catch (final ClassNotFoundException | InitializationError e)
            {
                if (e instanceof ClassNotFoundException)
                {
                    sm_Log.warn("Class " + fullQualifiedClassname + " not found. ", e);
                }
                else
                {
                    sm_Log.warn("Class " + fullQualifiedClassname + " can not be
                        initialized. ", e);
                }
                JUnitLogger.logWarn("Start JUnit", "Starting test "+fullQualifiedClassname+
                    " failed: "+e.getMessage());
            }
        }
    }

    private static void runUnitTest(final Class<?> unitTestClass)
        throws InitializationError, Result
    {
        sm_Failures.clear();
        final JUnitCore junitCore = new JUnitCore();
        junitCore.addListener(new UnitRunListener());
        final Runner runner = new JUnit4EnhancedTimeoutClassRunner(unitTestClass);
        junitCore.run(runner);
        if(!sm_Failures.isEmpty())
        {
```

```

final String failures = Arrays.toString(sm_Failures.toArray(new String [
    sm_Failures.size())));
throw new Result(new Message(failures), IResult.ERROR);
}

}

public static class UnitRunListener extends RunListener
{

    @Override
    public void testFinished(final Description description) throws Exception
    {
        JUnitLogger.logInfo("testFinished", description.getMethodName());
        super.testFinished(description);
    }

    @Override
    public void testIgnored(final Description description) throws Exception
    {
        JUnitLogger.logInfo("testIgnored", description.getMethodName());
        super.testIgnored(description);
    }

    @Override
    public void testRunFinished(final Result result) throws Exception
    {
        if(result.getFailureCount() > 0)
        {
            final List<Failure> failures = result.getFailures();
            for (final Failure failure : failures)
            {
                final String message = failure.getDescription().getMethodName() + " " +
                    failure.getMessage();
                final Throwable exception = failure.getException();
                if( exception != null )
                {
                    sm_Log.error( "testRunFinished(): exception during test ",exception
                    );
                }
                sm_Failures.add(message);
                JUnitLogger.logError("test Failed", message );
            }
        }
        JUnitLogger.logInfo("testRunFinished", "RunCount: " + result.getRunCount() + " "
            + RunTime: " + result.getRuntime() + " ms");
        super.testRunFinished(result);
    }

    @Override
    public void testRunStarted(final Description description) throws Exception
    {
        JUnitLogger.logInfo("testRunStarted", description.getMethodName());
    }
}

```

## Appendix A

```
        super.testRunStarted(description);
    }

    @Override
    public void testStarted(final Description description) throws Exception
    {
        JUnitLogger.logInfo("testStarted", description.getMethodName());
        super.testStarted(description);
    }

}
```

Listing 6.6: Klasse zum Starten eines Tests aus dem Python-Script

```
import org.apache.log4j.Logger;
import com.heidelberg.printready.autotest.util.logging.HTMLReport;
import com.heidelberg.printready.autotest.util.logging.JUnitLogger;

public final class UnitTestStarter
{
    private UnitTestStarter() {}

    private static final Logger sm_Log = Logger.getLogger(UnitTestStarter.class);

    public static void startTest(final String fullQualifiedClassname, final HTMLReport
        htmlReport, final String fileName)
    {
        JUnitLogger.init(htmlReport, fileName);
        JUnitLogger.logInfo("Start JUnit", "Starting test with class " +
            fullQualifiedClassname);
        try
        {
            UnitTestInvoker.runUnitTest(fullQualifiedClassname);
        }
        catch (final Throwable e)
        {
            sm_Log.error("startTest(): Exception caught!", e);
            JUnitLogger.logWarn("Start JUnit", "Starting test failed: "+e.getMessage());
        }
    }
}
```

## 5 Continuous Delivery Pipeline

Listing 6.7: Code der Continuous Delivery Pipeline

```
def nodeLabel = 'ALL_JAVA_GIT_KIE_SLAVES'
def repoURL = 'ssh://git@kie-gitsrv01.xxx.com/INF/INF-xxx.git'
def parallelNodeLabel = 'ALL_JAVA_GIT_KIE_SLAVES'

node(nodeLabel)
```

```
{
    stage 'Commit Stage'

    // Checkout code
    git branch: 'development', credentialsId: '67ac36e5-dae4-4dcd-a6341-94514092d536', url:
        repoURL

    stage 'CI-Build'

    def gradleCall = 'gradlew build jenkinsTest --refresh-dependencies --configure-on-
        demand --parallel'
    if(isUnix()){
        sh gradleCall
    }
    else{
        bat gradleCall
    }

}

stage name: 'Integration-Test, Installer, Metrics', concurrency: 1
def parallelStages = [:]

parallelStages["Intregation-Test"] = {
//    node('KIE-JENKINSBLD-LX') {
    node(nodeLabel) {
        // Call IntrTest
        echo 'Call Integration-Test'
        def gradleCall = 'gradlew integTest'
        git branch: 'development', credentialsId: '67ac36e5-dae4-4dcd-a6341-94514092
            d536', url: repoURL
        if(isUnix()){
            sh gradleCall
        }
        else{
            bat gradleCall
        }

        junit '**/build/test-results/*.xml'
    }
}

parallelStages["Installer"] = {
    node(parallelNodeLabel) {
        // Call Installer or Copy-Job
        echo 'Call Installer'
        bat 'timeout /t 20'
    }
}

parallelStages["Metrics"] = {
    node(parallelNodeLabel) {

```

## Appendix A

```
// Call
echo 'Call Metrics'
}

}

parallel parallelStages

stage 'Stresstest'
node(nodeLabel)
{
    echo 'Stresstest'
}

stage name: 'Akzeptanz-Tests', concurrency: 1
input message: 'Akzeptanz-Tests beendet', submitter:'lockerma'
```

## 6 Artefact-Deployer

Listing 6.8: Programm zum Kopieren der Build-Artefakte

```
package de.fhluebeck.lockemar.autotest.environment.deployment;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.Vector;

/**
 * @author Martin Locker
 */
public final class ArtefactDeployer
{
    private static final Log sm_Log = LogFactory.getLog(ArtefactDeployer.class);

    private final Map<String, Path> m_fileNamePathMap = new HashMap<>();

    private ArtefactDeployer(final String directoryFilesToCopy, final String
        targetDirectory)
    {
        final Vector<String> componentNames = getComponentNames();
        try
        {
            UpdateService.shutdownInstances(componentNames);
        }
        catch( final Exception e )
```

```

{
    e.printStackTrace();
}

final Path artifactPath = Paths.get(directoryFilesToCopy);
final List<Path> filesToCopy = fileListUtil.fileList(artifactPath);
fillFileNameMap(filesToCopy);

final Path startDirectory = Paths.get(targetDirectory);
final boolean hasWritePermission = fileListUtil.hasWritePermission(startDirectory);
if(!hasWritePermission)
{
    System.out.println("No write permission to " + startDirectory.toString());
}

final Map<String, List<Path>> fileReplaceMap = FileIndexer.createIndex(
    m_fileNamePathMap.keySet(), startDirectory);

copyFilesToTarget(fileReplaceMap);

try
{
    UpdateService.startInstances(componentNames);
}
catch( final Exception e )
{
    e.printStackTrace();
}
}

private static Vector<String> getComponentNames()
{
    final Vector<String> names = new Vector<String>();
    Map<ComponentInfo, String> componentsAndStatus = null;
    try
    {
        componentsAndStatus = SuperVisorAccess.getComponentsAndStatus();
        final Set<ComponentInfo> components = componentsAndStatus.keySet();
        for( final ComponentInfo component : components )
        {
            names.add(component.getName());
        }
    }
    catch( InterruptedException | TimeoutException e )
    {
        e.printStackTrace();
    }

    return names;
}
}

```

## Appendix A

```
private void copyFilesToTarget(final Map<String, List<Path>> fileReplaceMap)
{
    final Set<Entry<String, List<Path>>> entrySet = fileReplaceMap.entrySet();
    for( final Entry<String, List<Path>> entry : entrySet )
    {
        final String fileName = entry.getKey();
        final Path sourcePath = m_fileNamePathMap.get(fileName);
        copy(sourcePath, entry.getValue());
    }
}

private static void copy(final Path srcPath, final List<Path> targets)
{
    for( final Path targetPath : targets )
    {
        try
        {
            Files.copy(srcPath, targetPath, StandardCopyOption.REPLACE_EXISTING);
        }
        catch( final IOException e )
        {
            sm_Log.error("Error at copy file ", e);
        }
    }
}

private void fillFileNameMap(final List<Path> filesToCopy)
{
    if(filesToCopy != null)
    {
        for( final Path path : filesToCopy )
        {
            final String fileName = path.getFileName().toString();
            m_fileNamePathMap.put(fileName, path);
        }
    }
}

public static void main(final String[] args)
{
    new ArtefactDeployer(args[0], args[1]);
}
```

}

Listing 6.9: Code zum Suchen der zu ersetzenen Dateien

```
package de.fhlubeck.lockemar.autotest.environment.deployment;

import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
```

```

import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * @author Martin Locker
 */
final class FileIndexer
{
    private FileIndexer()
    {
        super();
        // Utility class
    }

    /**
     * @return Enthält für jedes zur ersetzenes File ein oder mehr mögliche Ziele zum
     * Kopieren
     */
    static Map<String, List<Path>> createIndex(final Collection<String> fileNamesToReplace,
                                                final Path startIndexDirectory)
    {
        final Map<String, List<Path>> replacingFileToPaths = new HashMap<>();

        final Set<String> fileNames = new HashSet<>(fileNamesToReplace);
        final List<Path> fileList = FileListUtil.fileList(startIndexDirectory);

        for( final Path filePathInReplaceDir : fileList )
        {
            final String fileName = filePathInReplaceDir.getFileName().toString();
            if(fileNames.contains(fileName)) // Ist es ein zu ersetzenes File
            {
                List<Path> list = replacingFileToPaths.get(fileName);
                if(list == null)
                {
                    list = new ArrayList<Path>();
                    replacingFileToPaths.put(fileName, list);
                }
                list.add(filePathInReplaceDir);
            }
        }

        return replacingFileToPaths;
    }
}

```

Listing 6.10: Datei Hilfs-Methoden für das Kopieren der Artefakte

```
package de.fhlubeck.lockemar.autotest.environment.deployment;
```

## Appendix A

```
import java.io.FilePermission;
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.security.AccessController;
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 *
 * @author Martin Locker
 *
 */
public final class FileListUtil
{
    private static final Log sm_Log = LogFactory.getLog(FileListUtil.class);

    private FileListUtil()
    {
        super();
        // Utility class
    }

    public static List<Path> fileList(final Path directory)
    {
        final List<Path> files = new ArrayList<>();
        try (DirectoryStream<Path> directoryStream = Files.newDirectoryStream(directory))
        {
            for (final Path path : directoryStream)
            {
                if(Files.isDirectory(path))
                {
                    final List<Path> fileList = fileList(path);
                    if(!fileList.isEmpty())
                    {
                        files.addAll(fileList);
                    }
                }
                else
                {
                    files.add(path);
                }
            }
        }
        catch (final IOException ex)
        {
            sm_Log.error("IOException occurred ", ex);
        }
    }
}
```

```

        return files;
    }

    static boolean hasWritePermission(final Path path)
    {
        boolean hasPermission = true;
        try
        {
            AccessController.checkPermission(new FilePermission(path.toString(), "read,
                write"));
            // Has permission
        }
        catch (final SecurityException e)
        {
            hasPermission = false;
        }
        return hasPermission;
    }
}

```

## 7 Testauswertung

Listing 6.11: Programm zum Auswerten der Testergebnisse

```

package de.fhlubeck.lockemar.autotest.environment.testanalyser;

import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardWatchEventKinds;
import java.nio.file.WatchEvent;
import java.nio.file.WatchKey;
import java.nio.file.WatchService;
import java.util.List;

/**
 * @author Martin Locker
 */
public class TestAnalyser
{
    private static final String FINISHED_SUFFIX = ".finished";

    public TestAnalyser(String autoTestDirectory, String waitFileName)
    {
        final Path path = Paths.get(autoTestDirectory);
        try
        {
            WatchService watchService = path.getFileSystem().newWatchService();
            WatchKey watchKey = path.register(watchService, StandardWatchEventKinds.ENTRY_CREATE)
            ;
            boolean endFileFound = false;

```

## Appendix A

```
while (!endFileFound)
{
    try
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }

    List<WatchEvent<?>> pollEvents = watchKey.pollEvents();
    if(pollEvents != null)
    {
        for (WatchEvent<?> watchEvent : pollEvents)
        {
            Path createdPath = (Path)watchEvent.context();
            String fileName = createdPath.getFileName().toString();
            endFileFound = isFinishedFileCreated(waitFileName, fileName);
        }
    }
}
catch (IOException e)
{
    e.printStackTrace();
}

int exitCode = checkIfAllTestsOK(autoTestDirectory) ? 0 : -1;
System.exit(exitCode); // Wenn alle Tests ok mit ReturnValue 0 das Programm beenden
}

private boolean isFinishedFileCreated(String waitFileName, String newCreatedName)
{
    // Nur Start und Ende berücksichtigen. Datumsangabe in der Mitte ausblenden.
    if(newCreatedName.startsWith(waitFileName) || newCreatedName.endsWith(FINISHED_SUFFIX))
    {
        return true;
    }
    return false;
}

private boolean checkIfAllTestsOK(String autoTestDirectory)
{
    return new TestReportReader().checkReports(autoTestDirectory);
}

public static void main(final String[] args)
{
    new TestAnalyser(args[0], args[1]);
}
```

# Abbildungsverzeichnis

1.1	Zusammenhang zwischen Phasen der Fehlerverursachung und deren Fehlerkosten [SP10] . . . . .	1
2.1	Ablauf in einer Druckerei . . . . .	3
2.2	Ideale Test-Pyramide [Sco12] . . . . .	6
2.3	Ablauf eines Unit-Tests . . . . .	7
2.4	Berechnung des ACD-Werts mit und ohne Zyklen . . . . .	11
2.5	Beispiel für den schlechtesten rACD-Wert . . . . .	12
2.6	Typischer CI-Zyklus . . . . .	15
2.7	Erweiterung Continuous Integration . . . . .	17
3.1	Identifizierte Probleme bei Einführung einer Continuous Delivery Pipeline . . . . .	24
3.2	Ist-Zustand Continuous Integration . . . . .	25
3.3	Durchschnittliche Dauer der CI-Builds . . . . .	26
3.4	Ergebnis einer Testabdeckungsanalyse . . . . .	27
3.5	Erstellung des Installers im nächtlichen Build . . . . .	28
4.1	Fehlermeldung des IsCloseTo-Matchers . . . . .	37
4.2	Gegenüberstellung der Ausgaben von JUnit und Hamcrest . . . . .	38
4.3	Anfängliches Schichtenmodell des Workflow-Systems . . . . .	40
4.4	Neu definierte Interfaces in der Parallel-Welt . . . . .	41
4.5	Komponenten-Anhängigkeiten der alten und neuen Parallel-Welt . . . . .	42
4.6	Verbesserung rACD durch Einführung von Services . . . . .	42
4.7	Nachträgliche Einführung eines Interfaces . . . . .	43
4.8	Gemeinsame Codebasis Client und Script-Server . . . . .	47
4.9	Unit-Tests mit und ohne Dependency Injection . . . . .	50
4.10	Verwendung der MISConnector-Komponente . . . . .	52
4.11	Bisherige Abarbeitung der automatischen Tests . . . . .	54
4.12	Abarbeitung der automatischen Tests mit Mutex . . . . .	54
4.13	Ausführung vom automatischen System-Tests auf einem Entwicklungsrechner	56
4.14	Geplanter Aufbau der Continuous Delivery Pipeline . . . . .	57
4.15	Remote Update System . . . . .	60
4.16	Manuelle Erstellung eines Hotfixes . . . . .	62

## Abbildungsverzeichnis

4.17 Automatischer Prozess zur Erstellung eines Hotfixes . . . . .	62
4.18 Gegenüberstellung Deployment-Einheiten eines ganzen Systems und Microservices . . . . .	63
4.19 Abhängigkeiten der Services des Workflow-Systems . . . . .	66
4.20 Pipeline View des Build Pipeline Plugins . . . . .	69
5.1 Ausführungszeit der Unit-Tests in Abhängigkeit von der forkEvery-Property .	74
5.2 Zulässige Abhängigkeiten . . . . .	77
5.3 Erweiterung des Logging pro Autotest . . . . .	78
5.4 Erweiterung Testautomat zum Ausführen von JUnit-Tests . . . . .	79
5.5 Klassen-Hierarchie von JUnit zum Ausführen von Testklassen . . . . .	79
5.6 Details der mit JUnit automatisch ausgeführten System-Tests . . . . .	80
5.7 Integration in der Gesamtübersicht . . . . .	80
5.8 Erweiterung des FileMutex . . . . .	83
5.9 Prototypische Pipeline mit Jenkins 2.x . . . . .	84
5.10 Anzeige der Pipeline in Jenkins . . . . .	86
5.11 Konsolenausgabe der Pipeline . . . . .	86
5.12 Umgesetzte Pipeline und deren Ausführungszeiten . . . . .	88
5.13 Aktualisierung eines Test-Systems in der Pipeline . . . . .	89
5.14 Aktualisierung eines Test-Systems in der Pipeline . . . . .	89
5.15 Jenkins Projekt zum Kopieren der Artefakte . . . . .	91
5.16 Einrichtung eines neuen Slave-Knotens . . . . .	91
5.17 Verzeichnis eines Jenkins-Slaves . . . . .	92
5.18 Verbindung vom Jenkins-Master zum Slave . . . . .	92
5.19 Jenkins-Slave Konfigurationdatei . . . . .	93
6.1 Automatische Tests für externe Komponenten . . . . .	97
6.2 MISConnector als unabhängiger Service . . . . .	99
6.3 Erstellen des Workflow-Installers nach Änderung vom MISConnector . . . . .	101

# Tabellenverzeichnis

3.1	Verbesserungspotentiale für Continuous Integration und Testautomatisierung	23
5.1	Einfluss von forkEvery auf die Ausführungszeit . . . . .	74
5.2	Buildzeiten mit neuer Hardware . . . . .	75
5.3	Steigerung der Ausführungszeit mit Mutex und Erhöhung der Script-Server .	84
5.4	Ausführungszeit von Stage 3 . . . . .	87
6.1	Verbesserte Ausführungszeit des CI-Builds (Problem I) . . . . .	95
6.2	Verbesserte Ausführungszeit der System-Tests (Problem VIc) . . . . .	95



# Literaturverzeichnis

- [BBKS08] BENGEL, GÜNTHER, CHRISTIAN BAUN, MARCEL KUNZE und KARL-UWE STUCKY: *Virtualisierungstechniken*. Masterkurs Parallele und Verteilte Systeme: Grundlagen und Programmierung von Multicoreprozessoren, Multiprozessoren, Cluster und Grid, Seiten 395–414, 2008.
- [BK13] BROY, MANFRED und MARCO KUHRMANN: *Projektorganisation und Management im Software Engineering*. Xpert.press. Springer Berlin Heidelberg, 2013.
- [BKL09] BAUN, CHRISTIAN, MARCEL KUNZE und THOMAS LUDWIG: *Servervirtualisierung*. Informatik-Spektrum, 32(3):197–205, 2009.
- [BL16a] BIRK, ALEXANDER und CHRISTOPH LUKAS: *Eine Einführung in Continuous Delivery - Teil1: Grundlagen*. iX Developer - Effektiver Entwickeln, Seiten 36–39, 2 2016.
- [BL16b] BIRK, ALEXANDER und CHRISTOPH LUKAS: *Eine Einführung in Continuous Delivery - Teil2: Commit Stage*. iX Developer - Effektiver Entwickeln, Seiten 40–44, 2 2016.
- [BL16c] BIRK, ALEXANDER und CHRISTOPH LUKAS: *Eine Einführung in Continuous Delivery - Teil4: Bereitstellen der Infrastruktur*. iX Developer - Effektiver Entwickeln, Seiten 50–53, 2 2016.
- [BM14] BORGES MEDEIROS, MILLER H.: *Type check is a code smell*. <http://blog.millermedeiros.com/type-check-is-a-code-smell/>, 8 2014. Accessed: 2016-04-02.
- [Boe75] BOEHM, BARRY W: *The high cost of software*. Practical Strategies for Developing Large Software Systems, Seiten 3–15, 1975.
- [Bro15] BRODSKI, BORIS: *Automatisierte Tests, die Spaß machen*. Eclipse Magazin, Seiten 84–90, 3 2015.
- [CL07] CULLMANN, XAVIER-N. und KLAUS LAMBERTZ: *Komplexität und Qualität von Software*. MSCoder, 1(1):36–43, 2007.

## Literaturverzeichnis

- [Con68] CONWAY, MELVIN E: *How do committees invent.* Datamation, 14(4):28–31, 1968.
- [DM11a] DRAEGER, JOACHIM und KIARESCH MUSSAWISADE: *Clean Code Teil 2.* Java Magazin, Seiten 28–33, 1 2011.
- [DM11b] DRAEGER, JOACHIM und KIARESCH MUSSAWISADE: *Clean Code Teil 3 - Der Kreis schließt sich.* Java Magazin, Seiten 45–52, 2 2011.
- [Emm10] EMMANUEL, TORSTEN: *Planguage-Spezifikation nichtfunktionaler Anforderungen.* Informatik-Spektrum, 33(3):292–295, 2010.
- [End03] ENDRES, ALBERT: *Softwarequalität aus Nutzersicht und ihre wirtschaftliche Bewertung.* Informatik-Spektrum, 26(1):20–25, 2003.
- [ER04] ENDRES, ALBERT und H-D. ROMBACH: *Handbook on Software Engineering.* Seite 131, 2004.
- [Eva12] EVANS, ERIC: *Strategic Design.*  
<http://dddcommunity.org/strategic-design/>, 9 2012. Accessed: 2016-09-07.
- [Fow00] FOWLER, MARTIN: *Refactoring.* Addison-Wesley, 2000.
- [Fow04] FOWLER, MARTIN: *Inversion of Control Containers and the Dependency Injection pattern.*  
<http://www.martinfowler.com/articles/injection.html>, 1 2004. Accessed: 2016-04-16.
- [Fow15] FOWLER, MARTIN: *MonolithFirst.*  
<http://martinfowler.com/bliki/MonolithFirst.html>, 6 2015. Accessed: 2016-09-08.
- [Fra03] FRAST, DENIS: *Software-Qualitätssicherung.*  
<http://qse.ifs.tuwien.ac.at/courses/QS/download/QSVU-Skriptum-Teil2-030411.pdf>, 4 2003. Accessed: 2016-08-07.
- [Gär16] GÄRTNER, MARKUS: *Zuverlässige Tests in der agilen Softwareentwicklung.* iX Developer - Effektiver Entwickeln, Seiten 12–15, 2 2016.
- [Get16] GETROST, TOBIAS: *Continuous Delivery - eine Reise durch die Evolution unserer Build Pipeline.* [https://getrost.org/talk\\_sqd2016.html](https://getrost.org/talk_sqd2016.html), 1 2016. Accessed: 2016-10-16.

- [Göp15] GÖPEL, R.: *Praxishandbuch VMware vSphere 6: Leitfaden für Installation, Konfiguration und Optimierung*. O'Reilly, 2015.
- [Hal77] HALSTEAD, MAURICE HOWARD: *Elements of software science*, Band 7. Elsevier New York, 1977.
- [Hau16] HAUER, PHILIPP: *Running the Build within a Docker Container*.  
[https://blog.philippauer.de/  
improving-continuous-integration-setup-docker-gitlab-ci/](https://blog.philippauer.de/improving-continuous-integration-setup-docker-gitlab-ci/), 10 2016. Accessed: 2016-10-16.
- [Hev08] HEVERY, MISKO: *Singletons are Pathological Liars*.  
[http://misko.hevery.com/2008/08/17/  
singletons-are-pathological-liars/](http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/), 1 2008. Accessed: 2016-05-08.
- [HG12] HARTMANN, ANDREAS und HALIL-CEM GÜRSOY: *Gradle: neue Konkurrenz für Ant und Maven*. [http://www.heise.de/developer/artikel/  
Gradle-neue-Konkurrenz-fuer-Ant-und-Maven-1545620.html/](http://www.heise.de/developer/artikel/Gradle-neue-Konkurrenz-fuer-Ant-und-Maven-1545620.html), 4 2012. Accessed: 2016-07-07.
- [Hof08] HOFMANN, DOMINIK: *Cobertura*. Java Magazin, Seiten 114–118, 10 2008.
- [Hof13] HOFFMANN, D.W.: *Software-Qualität*. eXamen.press. Springer Berlin Heidelberg, 2013.
- [Jac09] JACKSON, RECARDO: *Testmanagement: Professionelles Testen*. Informatik-Spektrum, 32(1):37–41, 2009.
- [Joo15] JOOS, THOMAS: *Docker und Windows Server 2016 Das müssen Profis wissen*. [http://www.zdnet.de/88246989/  
docker-und-windows-server-2016-das-muessen-profis-wissen/](http://www.zdnet.de/88246989/docker-und-windows-server-2016-das-muessen-profis-wissen/), 9 2015. Accessed: 2016-10-25.
- [Jos08] JOSUTTIS, NICOLAI: *SOA in der Praxis: System-Design für verteilte Geschäftsprozesse*. dpunkt-Verlag, 2008.
- [JP04] JUHL-PETERSEN, RUNE: *Instanceof Code Smell*. <http://pmu.blogspot.de/2004/09/instanceof-code-smell.html>, 9 2004. Accessed: 2016-04-02.
- [Kap13] KAPELONIS, KOSTIS: *The correct way to use integration tests in your build process*. [http://zeroturnaround.com/rebellabs/  
the-correct-way-to-use-integration-tests-in-your-build-process/](http://zeroturnaround.com/rebellabs/the-correct-way-to-use-integration-tests-in-your-build-process/), 2 2013. Accessed: 2016-04-19.

## Literaturverzeichnis

- [Ker05] KERL, ANDREAS: *Extended Installer Language*. Dotnetpro, (10):90–97, 2005.
- [KH13] KAUFMAN, MARCIA und JUDITH HURWITZ: *Service Virtualization for Dummies*. <https://www-01.ibm.com/software/rational/servicevirtualization/>, 2013. Accessed: 2016-05-02.
- [KK16] KANE, SEAN P. und MATTHIAS KARL: *Docker Praxiseinstieg: Deployment, Testen und Debugging von Containern in Produktivumgebungen*. mitp Professional. mitp-Verlag, 2016.
- [Kre14] KREMS, BURKHARDT: *10er-Regel der Fehlerkosten*. <http://www.olev.de/0/10er-regel.htm>, 2 2014. Accessed: 2016-05-18.
- [KvdABV08] KOOMEN, T, L VAN DER ALST, B BROEKMAN und M VROON: *TMap NextEin praktischer Leitfaden für ergebnisorientetes Softwaretesten*. dpunkt, 2008.
- [KWT<sup>+</sup>06] KALENBORN, AXEL, THOMAS WILL, ROUEN THIMM, JANA RAAB und RONNY FREGIN: *Java-basiertes automatisiertes Test-Framework*. Wirtschaftsinformatik, 48(6):437–445, 2006.
- [Lak96] LAKOS, JOHN: *Large-scale C++ software design*. Addison-Wesley, 1996.
- [Lin05] LINK, JOHANNES: *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*. dpunkt-Verlag, 2005.
- [Man14] MANDL, PETER: *Betriebssystemvirtualisierung*. In: *Grundkurs Betriebssysteme*, Seiten 297–318. Springer, 2014.
- [Mar13] MARTIN, R.C.: *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. mitp Professional. mitp/bhv, 2013.
- [McA15] MCALLISTER, JONATHAN: *Mastering Jenkins*. Packt Publishing, 2015.
- [McD16] McDONOUGH, RYAN J.: *Creating containerized build environments with the Jenkins Pipeline plugin and Docker*. <https://damnhandy.com/author/damnhandy/>, 3 2016. Accessed: 2016-10-01.
- [MT15] MARQUIS, SAMUEL und DANIEL TAKAI: *Automatische Unit und Systemtests für Integrationsaufgaben*. Java Magazin, Seiten 62–67, 3 2015.
- [MZN10] MAIRIZA, DEWI, DIDAR ZOWGHI und NURIE NURMULIANI: *An investigation into the notion of non-functional requirements*. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*, Seiten 311–317. ACM, 2010.

- [New15] NEWMAN, SAM: *Microservices For Greenfield?* <http://samnewman.io/blog/2015/04/07/microservices-for-greenfield/>, 4 2015.  
Accessed: 2016-09-08.
- [PB11] PARISEAU, BETH und RALPH BEUTH: *Typ 1 versus Typ 2: KVMs und andere Hypervisoren.* <http://www.datacenter-insider.de/typ-1-versus-typ-2-kvms-und-andere-hypervisoren-a-318394/>, 6 2011. Accessed: 2016-10-03.
- [Pet16] PETERSON, NEIL: *Hyper-V-Container.* [https://msdn.microsoft.com/de-de/virtualization/windowscontainers/management/hyperv\\_container](https://msdn.microsoft.com/de-de/virtualization/windowscontainers/management/hyperv_container), 9 2016.  
Accessed: 2016-10-25.
- [PG74] POPEK, GERALD J und ROBERT P GOLDBERG: *Formal requirements for virtualizable third generation architectures.* Communications of the ACM, 17(7):412–421, 1974.
- [PTvV02] POL, MARTIN, RUUD TEUNISSEN und ERIK VAN VEENENDAAL: *Software Testing: A Guide to the TMap Approach.* A Pearson Education book. Addison-Wesley, 2002.
- [RL16] RÖWEKAMP, LARS und ARNE LIMBURG: *Der perfekte Microservice.* iX Developer - Effektiver Entwickeln, Seiten 106–113, 2 2016.
- [Roß14] ROSSBACH, PETER: *Docker mit boot2docker starten.* <http://www.infrabricks.de/blog/2014/06/30/docker-mit-boot2docker-starten/>, 6 2014. Accessed: 2016-10-25.
- [Rob13] ROBERT, CHRISTIAN: *java.nio.file: Zeitgemäßes Arbeiten mit Dateien.* <https://jaxenter.de/java-nio-file-zeitgemases-arbeiten-mit-dateien-2581>, 8 2013. Accessed: 2016-10-21.
- [Rös10] RÖSLER, PETER: *Agile Inspektionen.* Embedded Software Engineering Kongress, Sindelfingen, 12 2010.
- [Sco12] SCOTT, ALISTER: *Introducing the software testing ice-cream cone (anti-pattern).* <https://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/>, 1 2012.  
Accessed: 2016-05-01.

## Literaturverzeichnis

- [SDW<sup>+</sup>10] SCHATTEN, ALEXANDER, MARKUS DEMOLSKY, DIETMAR WINKLER, STEFAN BIFFL, ERIK GOSTISCHA-FRANTA und THOMAS ÖSTREICHER: *Best Practice Software-Engineering*. Spektrum Akademischer Verlag, 2010.
- [Sei12] SEITZ, JOHANNES: *Eine Tour durch die Testpyramide*. <https://blog.namics.com/2012/06/tour-durch-die-testpyramide.html/>, 6 2012. Accessed: 2016-04-16.
- [Sel16] SELLMAYR, FLORIAN: *Delivery Pipeline as Code*. <https://www.informatik-aktuell.de/entwicklung/methoden/delivery-pipelines-as-code-ein-ueberblick.html>, 9 2016. Accessed: 2016-10-05.
- [SHT04] SNEED, HARRY M, MARTIN HASITSCHKA und MARIA-THERESE TEICHMANN: *Software-Produktmanagement*. In: *Software Management*, Seiten 143–144, 2004.
- [SJ11] SNEED, HARRY M und STEFAN JUNGMAYR: *Mehr Testwirtschaftlichkeit durch Value-Driven-Testing*. Informatik-Spektrum, 34(2):192–209, 2011.
- [SP10] SCHMITT, ROBERT und TILO PFEIFER: *Qualitätsmanagement: Strategien–Methoden–Techniken*. Carl Hanser Verlag GmbH Co KG, 2010.
- [Spa16] SPANNEBERG, BASTIAN: *Continuous Integration und Continuous Delivery mit Jenkins*. iX Developer - Effektiver Entwickeln, Seiten 54–59, 2 2016.
- [Spi08] SPILLNER, ANDREAS: *Systematisches Testen von Software*. dpunkt-Verlag, 2008.
- [Stä16] STÄHLER, MICHAEL: *Auf dem Weg von Continuous Integration zu Continuous Delivery*. <https://www.informatik-aktuell.de/entwicklung/methoden/devops-in-der-praxis-von-continuous-integration-zu-continuous-delivery.html>, 5 2016. Accessed: 2016-10-05.
- [STV16] SILVA, DANILO, NIKOLAOS TSANTALIS und MARCO TULIO VALENTE: *Why We Refactor? Confessions of GitHub Contributors*. arXiv preprint arXiv:1607.02459, 2016.
- [Tam13] TAMM, M.: *JUnit-Profiwissen: Effizientes Arbeiten mit der Standardbibliothek für automatische Tests in Java*. Dpunkt.Verlag GmbH, 2013.
- [Tan09] TANENBAUM, ANDREW S.: *Moderne Betriebssysteme*. Pearson Deutschland GmbH, 2009.

- [Til15] TILKOV, STEFAN: *Dont start with a monolith.* <http://martinfowler.com/articles/dont-start-monolith.html>, 6 2015. Accessed: 2016-09-08.
- [TW15] TILKOV, STEFAN und EBERHARD WOLFF: *Unabhängigkeitserklärung. Microservices: Modularisierung ohne Middleware.* iX Magazin für professionelle Informationstechnik, Seiten 108–111, 4 2015.
- [Vog09] VOGEL, RONNY: *Begriffe erklärt: Lasttest, Stresstest, .* <https://blog.xceptance.com/2009/09/16/begriffe-erklart-lasttest-stresstest/>, 9 2009. Accessed: 2016-10-03.
- [Wie11] WIEST, SIMON: *Continuous Integration mit Hudson: Grundlagen und Praxiswissen für Einsteiger und Umsteiger.* dpunkt-Verlag, 2011.
- [Wil15] WILDT, USCH: *Testautomatisierung in Legacy-Systemen.* <http://www.heise.de/developer/artikel/Testautomatisierung-in-Legacy-Systemen-2670410.html>, 6 2015. Accessed: 2016-06-12.
- [wixa] *Why WiX.* <https://www.firegiant.com/why-wix/>. Accessed: 2016-06-25.
- [wixb] *WiX Tutorial.* <https://www.firegiant.com/wix/tutorial/>. Accessed: 2016-06-25.
- [Wol15a] WOLFF, E: *Continuous Delivery: Der pragmatische Einstieg.* dpunkt. verlag. Heidelberg, 2015.
- [Wol15b] WOLFF, E.: *Microservices: Grundlagen flexibler Softwarearchitekturen.* dpunkt.verlag, 2015.
- [Wol15c] WOLFF, EBERHARD: *Microservices und Continuous Delivery: Nur zusammen möglich?* <https://www.innoq.com/de/blog/microservices-und-continuous-delivery/>, 8 2015. Accessed: 2016-08-26.
- [Zit07] ZITZEWITZ, ALEXANDER VON: *Erosionen vorbeugen.* Java Magazin, Seiten 28–33, 2 2007.