



# Diplomarbeit

## Testgetriebene Entwicklung von Web-Serveranwendungen auf der Basis von Ruby on Rails

**Autor:** Wienert, Stefan

**Studiengruppe:** 07/041/02

**Betreuender Professor:** Prof. Dr. Fritzsche

**Datum:** 6. Oktober 2011



# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	2
1.1.1. Die pludoni GmbH . . . . .	2
1.1.2. Arbeitsablauf in der pludoni GmbH . . . . .	4
1.2. Projektbeschreibung und Projektziele . . . . .	5
1.2.1. Anwendungsfälle . . . . .	5
1.2.2. Nichtfunktionale Anforderungen . . . . .	7
1.3. Aufbau der Arbeit . . . . .	7
<b>2. Automatisierte dynamische Softwaretests</b>	<b>9</b>
2.1. Motivation zum Testen . . . . .	9
2.2. Arten von Tests . . . . .	10
2.3. Unittest . . . . .	12
2.4. Test Doubles – Mocks und Stubs . . . . .	13
2.5. System- und Akzeptanztests . . . . .	15
<b>3. Testgetriebene Entwicklung</b>	<b>16</b>
3.1. Motivation . . . . .	16
3.2. Ablauf . . . . .	17
3.3. Sonderfälle . . . . .	20
3.4. Studien zu den Auswirkungen von TDD . . . . .	21
3.5. Prinzip des Emergent Design – Inkrementelles Softwaredesign . . . . .	22
3.6. Varianten . . . . .	24
3.6.1. ATDD – Acceptance TDD . . . . .	24
3.6.2. Behavior Driven Development . . . . .	24
3.6.3. Design Driven Testing . . . . .	25
<b>4. Die Programmiersprache Ruby</b>	<b>26</b>
4.1. Einführung in Ruby . . . . .	26
4.2. Diskussion . . . . .	31
4.3. Ruby on Rails . . . . .	33
4.3.1. Konzepte von Rails . . . . .	33
4.3.2. Diskussion . . . . .	36
4.4. Testframeworks für Ruby . . . . .	39
4.4.1. Test/Unit . . . . .	39
4.4.2. Cucumber . . . . .	41

<b>5. Code-Metriken</b>	<b>44</b>
5.1. Überblick über Code-Metriken und Skalen . . . . .	44
5.2. Code-Metriken für Tests . . . . .	46
5.2.1. Verhältnis von Lines of Test zu Lines of Code . . . . .	46
5.2.2. Testausführungsabdeckung . . . . .	47
5.2.3. Mutations- / Perturbationstests – Defect insertion . . . . .	48
5.3. Notwendigkeit von Code Metriken . . . . .	49
<b>6. Entwicklungsmethodik und -Werkzeuge</b>	<b>50</b>
6.1. Definition eines Entwicklungsmodells für die Bedürfnisse der pludoni GmbH	50
6.1.1. Einteilung der Features in Kategorien . . . . .	51
6.1.2. Praktiken . . . . .	51
6.2. Auswahl der Entwicklungswerkzeuge . . . . .	53
6.3. Diskussion . . . . .	54
<b>7. Anwendung der Testgetriebenen Entwicklung</b>	<b>55</b>
7.1. Implementierung von Unit-Tests (Modelltests) . . . . .	55
7.2. Implementierung von Controller-Tests (functional tests) . . . . .	62
7.3. Testen von externen Abhängigkeiten . . . . .	67
7.4. System/Akzeptanztests . . . . .	74
7.5. Testen von JavaScript . . . . .	80
<b>8. Auswertung</b>	<b>82</b>
8.1. Entwicklungsstand IT-Jobs . . . . .	82
8.2. Code-Quality-Benchmark mit anderen Ruby-Projekten . . . . .	86
8.2.1. Vergleich von IT-Jobs mit eigenen Projekten . . . . .	86
8.2.2. Vergleich von IT-Jobs mit anderen Rails-Projekten . . . . .	87
8.2.3. Auswertung und Visualisierung . . . . .	88
<b>9. Fazit</b>	<b>93</b>
9.1. Zusammenfassung . . . . .	93
9.2. Ausblick . . . . .	94
<b>A. Eigenschaften erfolgreicher Tests</b>	<b>96</b>
<b>B. Nutzung von Cucumber in Verbindung mit Selenium für Firefox und Guard ohne X-Server</b>	<b>97</b>
<b>Glossar</b>	<b>101</b>
<b>Abbildungsverzeichnis</b>	<b>104</b>
<b>Quellcode-Listings</b>	<b>104</b>
<b>Stichwortverzeichnis</b>	<b>107</b>
<b>Literaturverzeichnis</b>	<b>108</b>

# Danksagung

Besonderen Dank gilt meinem Betreuer, Prof. Fritzsche, der sich regelmäßig und ausführlich mit meiner Arbeit beschäftigte und mir wertvolle Hinweise erteilte.

Mein Dank gilt auch der Fakultät Informatik und Prof. Wiedemann für das zur Verfügung gestellte Büro, der Bibliothek der HTW-Dresden und Prof. Nestler für das Bestellen von tagesaktueller Literatur.

Ich danke auch meiner Frau, die mich während des Schreibens unterstützt hat und bei der Erstellung der Grafiken Tipps gab. Für die orthografische und expressive Optimierung sei meinen Freunden Daniel Schäufele, Jörg Sachse, Martin Schneider, Akos Toth und Stefan Koch gedankt.

Diese Arbeit entstand in Zusammenarbeit mit der Firma pludoni GmbH und unter Aufsicht des Geschäftsführers Dr. Jörg Klukas. Ich danke ihm für die Möglichkeit, frei zu arbeiten und viele neue Dinge auszuprobieren und für das Vertrauen, das er dabei in mich gesetzt hat.



# Kapitel 1.

## Einleitung

Das Hauptthema dieser Diplomarbeit, die Testgetriebene Entwicklung, ist vielen Programmierern vom Namen her bekannt. Ungeachtet dessen, dass viele Studien über die (positiven) Auswirkungen dieses Entwicklungsansatzes existieren, so wird sie trotzdem relativ selten eingesetzt und es existieren große Vorbehalte, wie z.B. dass es Zeitverschwendung sei oder dass Programmierer fälschlicherweise annehmen, sie könnten die Komplexität überschauen<sup>1</sup>.

Testgetriebene  
Entwicklung...

Diese, teilweise mythenumrankte, Technik hat innerhalb des Entwicklerteams der pludoni GmbH ein großes Interesse geweckt und wird aus diesem Grund nun näher untersucht.

Die pludoni GmbH benötigt für ihre Services gut funktionierende und effizient entwickelte Webserveranwendungen. Das Framework Ruby on Rails scheint dafür wie geschaffen und erste kleinere Erfahrungen waren durchaus positiv. Durch eine vielbestätigte effektive Entwicklung ist Rails ideal für den Einsatz in kleinen Teams, wie das Entwicklerteam der pludoni GmbH eines ist, geeignet.

...von  
Webserveran-  
wendungen...

Dieses Framework ist das zweite große Thema dieser Diplomarbeit. Rails und Testen passen anscheinend sehr gut zusammen, da das Testen im Allgemeinen in der Ruby-Community einen sehr hohen Stellenwert hat. Außerdem existiert eine Vielzahl von Testwerkzeugen und die meisten bekannten Bibliotheken verfügen über ausgiebige Tests. Damit passen die Testgetriebene Entwicklung und Ruby (on Rails) perfekt zusammen.

...auf Basis von  
Ruby on Rails

Um die Auswirkungen der Testgetriebenen Entwicklung besser nachvollziehen zu können, benötigt man ein Messinstrument. Dafür sind in diesem Kontext der Webentwicklung die Code-Metriken geeignet. Sie können einen groben Überblick über den Zustand des Quellcodes geben. Daher sind sie ideal, insbesondere in der Lernphase der Testgetriebenen Entwicklung, Feedback zu geben.

Wie das Zusammenspiel von Ruby (on Rails), der Testgetriebenen Entwicklung und dem Einsatz von Code-Metriken praktisch erfolgen kann, wird in dieser Arbeit im Detail betrachtet und mit eigenen praktischen Erfahrungen belegt.

---

<sup>1</sup><http://tamasgyorfi.wordpress.com/tag/excuses-to-tdd/>

## 1.1. Motivation

Kurz nach der Firmengründung der pludoni GmbH 2009 absolvierte der Autor dort sein Pflichtpraktikum und war danach weiterhin als Werkstudent tätig. Während dieser Zeit nahmen Programmierer verschiedener Erfahrungsstufen und auch Praktikanten an der Neu- und Weiterentwicklung der Webserversoftware teil. Dies hat zur Folge, dass die Komplexität der Software inzwischen ein Level erreicht hat, bei dem das Maß an Regressionsfehlern<sup>2</sup> stark anstieg.

Da zum großen Teil keine automatisierten Softwaretests geschrieben wurden, lassen sich diese Fehler nur schwer auffinden. Ein Versuch, nachträglich Softwaretests hinzuzufügen, wurde evaluiert und als zu aufwändig befunden, da der Code in seinem jetzigen Zustand nur äußerst schwer zu testen ist. Gründe dafür sind suboptimale Codestrukturen (Spagheticode, Gott-Methode), die schwer bis unmöglich zu testen sind. Hier müsste zuerst refaktoriert werden, aber da keine Tests vorhanden sind, ist dies aufgrund der Regressionsfehler riskant.

Für ein neues Projekt, it-jobs-und-stellen.de, soll dies nun mit einem anderen Ansatz verlaufen.

Neben der Umstellung auf ein modernes Web-Framework, sollen nun Tests im Einklang zum Code erstellt werden, um auf Knopfdruck umfassende Informationen über den Systemzustand zu erhalten, wie sie ein manueller Test in der Gründlichkeit und Schnelligkeit niemals erreichen kann. Außerdem besteht die Hoffnung auf eine nachhaltige Verbesserung des Designs und der <sup>↑</sup>Code-Qualität, um in Zukunft flexibel auf Änderungen reagieren zu können.

### 1.1.1. Die pludoni GmbH



pludoni  
GmbH

Die pludoni GmbH ist ein junges dynamisches Dresdner Unternehmen, welches sich zum Ziel gesetzt hat, lokale Communitys zur gegenseitigen Fachkräfteempfehlung aufzubauen und zu betreuen sowie Tools für die Vereinfachung der Personalarbeit mittelständischer Unternehmen zu entwickeln. Einige Beispiele für diese Communitys sind ITsax.de, ITmitte.de und MINTsax.de<sup>3</sup>.

**Funktionsweise der Communitys** Die Communitys bestehen jeweils aus mittelständischer Unternehmen einer Branche, Bildungseinrichtungen sowie Vertretern von Städten und Vereinen. Für ITsax.de ist das die IT-Branche. Neben diesem Branchenfokus sammeln sich nur Unternehmen einer spezifischen Region. Bei ITsax.de sind dies Mittel- und Ostsachsen, bei ITmitte.de z.B. Mitteldeutschland, d.h. Thüringen, Sachsen-Anhalt und Westsachsen.

---

<sup>2</sup>fehlerauslösender Quelltextänderungen

<sup>3</sup><http://www.itsax.de/>, <http://www.itmitte.de>, <http://www.mintsax.de/>



Tabelle 1.1.: Übersicht über pludoni Communitys. Stand September 2011

Community	Branche	Region	Mitglieder
ITsax.de	IT, Software	Sachsen (Fokus Süden und Osten)	63
ITmitte.de	IT, Software	Mitteldeutschland (Thüringen, Sachsen-Anhalt, Westsachsen)	57
MINTsax.de	Maschinenbau, Elektrotechnik	Sachsen	29
OFFICESax.de	Vertrieb, Bürotätigkeiten	Sachsen	in Vorb.
OFFICEmitte.de	Vertrieb, Bürotätigkeiten	Mitteldeutschland	in Vorb.

Diese Unternehmen, die einen jährlichen Mitgliedsbeitrag für das Community-Management<sup>4</sup> und die Weiterentwicklung der Portale an die pludoni GmbH zahlen, dürfen ihre für die Region relevanten Jobangebote auf dem jeweiligen Portal einstellen. Was die Communitys von pludoni von der dem bisherigen Online-Jobbörsen unterscheidet, ist das sogenannte **Empfehlungssystem**.



Abbildung 1.1.: Funktionsweise des Empfehlungscode

Viele der Personalmitarbeiter der beteiligten Firmen haben dieselbe Erfahrung gemacht, dass sie sehr guten Bewerbern absagen mussten, weil z.B. die Stelle schon vergeben wurde, die Fähigkeiten des Bewerbers nicht den Bedürfnissen des Unternehmens entsprachen oder andere äußere Widrigkeiten eine Einstellung verhinderten. Hier setzt pludoni mit seinen Communitys an und stellt eine Infrastruktur zur gegenseitigen Empfehlung dieser guten

<sup>4</sup>Einen Überblick über die Aufgaben eines Community-Managers finden Sie unter <http://www.pludoni.de/leistungen>

Bewerber bereit. Ausgezeichnete Bewerber erhalten neben der Absage einen Empfehlungscode, mit dem sie sich auf dem Online-Jobportal bei einer der anderen Mitgliedsfirmen bewerben können (vgl. Abbildung 1.1). Die Software löst intern den Empfehlungscode auf und bestätigt dieser Firma nun, dass der Bewerber von einem anderen Unternehmen der Community empfohlen wurde.

Dieses Empfehlungssystem überzeugt die beteiligten Unternehmen. Aktuell wurden im letzten Jahr, z.B. auf ITmitte.de, über 800 Bewerbungen über das Portal versendet, von denen mehr als die Hälfte (440) mit Empfehlungscode versehen waren [Klu11b]. Dies motivierte mittlerweile über 150 Organisationen, bei den drei pludoni Communitys teilzunehmen [Klu11a].

### 1.1.2. Arbeitsablauf in der pludoni GmbH

Die pludoni GmbH stellt sich dem Trend der Dezentralisierung von Arbeit, um einerseits Kundenwünsche zur individuellen Betreuung und andererseits Mitarbeiterwünsche nach flexiblen und familienfreundlichen Arbeitsplätzen gerecht zu werden. Ein Großteil der Arbeit findet somit vor Ort beim Kunden oder in Heim- / Telearbeit statt. Zur persönlichen Abstimmung findet aber mindestens einmal pro Woche ein Meeting statt, in welchem sich 2-4 der Mitarbeiter treffen, um alte Aufgaben abzunehmen und neue zu diskutieren. Die Abnahme erfolgt dabei durch den Teamleiter des Projekts oder dem Geschäftsführer Jörg Klukas.

Zentrales Kommunikationsmittel der pludoni GmbH ist, neben der E-Mail, die Online-Aufgaben- und Fehlerverwaltung Redmine<sup>5</sup>. Dort werden alle Aufgaben und Fehler erfasst und an die zuständigen Personen verteilt. Neben den technischen Aufgaben der Entwickler, werden auch nicht-technische Aufgaben der anderen Mitarbeiter verwaltet, wie z.B. die Gewinnung neuer Partner (Akquise), Betreuung der Partner oder administrative Aufgaben.

Trotz dieses Tools und Vorgehensweise ist die dezentrale Kollaboration aber immer mit Nachteilen in der Kommunikation behaftet. Dies hat bei den Programmierern teils gravierende Auswirkungen auf die Produktivität. Einerseits, da gleiche Funktionalität doppelt implementiert wird, weil der Überblick fehlt und somit unnötigerweise neue Fehlerquellen eröffnet werden. Andererseits, weil aufgrund der zeitlich asynchronen Arbeitsleistungen Rückmeldungen der anderen Programmierer oder ein Code-Audit nicht immer gewährleistet werden und Regressionsfehler nicht abgeschätzt werden können, da nicht alle Module bekannt sind.

---

<sup>5</sup><http://www.redmine.org> – ein Web-basiertes Projektmanagement-Tool auf der Basis von Ruby on Rails. Redmine kann für Benutzer- und Projektverwaltung, Diskussionsforen, Wikis, zur Ticketverwaltung oder Dokumentenablage genutzt werden, *Wikipedia*

Für das kommende Projekt soll mithilfe der Testgetriebenen Entwicklung ein feinmaschiges Netz an Tests erstellt werden, um eine stets aktuelle Dokumentation des Quelltextes zu erhalten und das Risiko für Regressionsfehler zu minimieren.

## 1.2. Projektbeschreibung und Projektziele

Für den praktischen Teil dieser Arbeit soll anhand der Entwicklung einer Ruby-on-Rails-Anwendung die Testgetriebene Entwicklung erprobt und angewendet werden.

Als Ergänzung zu den lokalspezifischen Communitys mit strenger Mitgliederauswahl soll nun ein neues, allgemeines IT-Jobportal entwickelt werden. Der voraussichtliche Name wird IT-Jobs-Und-Stellen.de<sup>6</sup> sein.

Ziel soll es sein, den regionalen Organisationen auch eine Alternative zu den Community-Mitgliedschaften anzubieten, um kurzfristigen Personalbedarf zu decken, analog zu den bekannten Online-Stellenbörsen stepstone.de, monster.de und jobscout24<sup>7</sup>. Parallel zu dieser Arbeit wurden die Anforderungen analysiert und die in Abbildung 1.2 dargestellten Anwendungsfälle bestimmt.

### 1.2.1. Anwendungsfälle

Es gibt zwei verschiedene Nutzertypen. Zum einen der *Bewerber*, der das Ziel hat, einen Job zu suchen, zu finden und sich darauf zu bewerben. Zum anderen der Mitarbeiter eines Kundenunternehmens (*Arbeitgeber*), der Stellenanzeigen auf der Webseite schalten möchte. Verbal formuliert sind die Anwendungsfälle die Folgenden:

- Ein Bewerber kann die Webseite nach sichtbaren Stellenangeboten durchsuchen.
- Ein Bewerber kann die Detailanzeige einer Stelle betrachten.
- Ein Bewerber kann sich auf eine Stelle über das System bewerben. Dabei kann er auch eine Verbindung zu seinem Facebook- oder LinkedIn-Account herstellen, falls er das möchte, um so automatisch einen Lebenslauf generieren zu lassen und Stammdaten eintragen zu lassen.
- Ein Kunde kann sich am Portal registrieren, dann anmelden und wird so zum *Arbeitgeber*.
- Ein Arbeitgeber kann eine Stellenanzeige für ein Vielfaches von 30 Tagen schalten. Damit wird ein Bezahlvorgang über ein Drittanbieterportal (z.B. PayPal) ausgelöst und bei Bestätigung die Stelle sichtbar geschaltet. Er kann dazu zwischen 5 verschiedenen Designs wählen sowie Logo und Bild hinzufügen.

---

<sup>6</sup><http://www.it-jobs-und-stellen.de/>

<sup>7</sup><http://www.stepstone.de> - <http://www.monster.de> - <http://www.jobscout24.de>

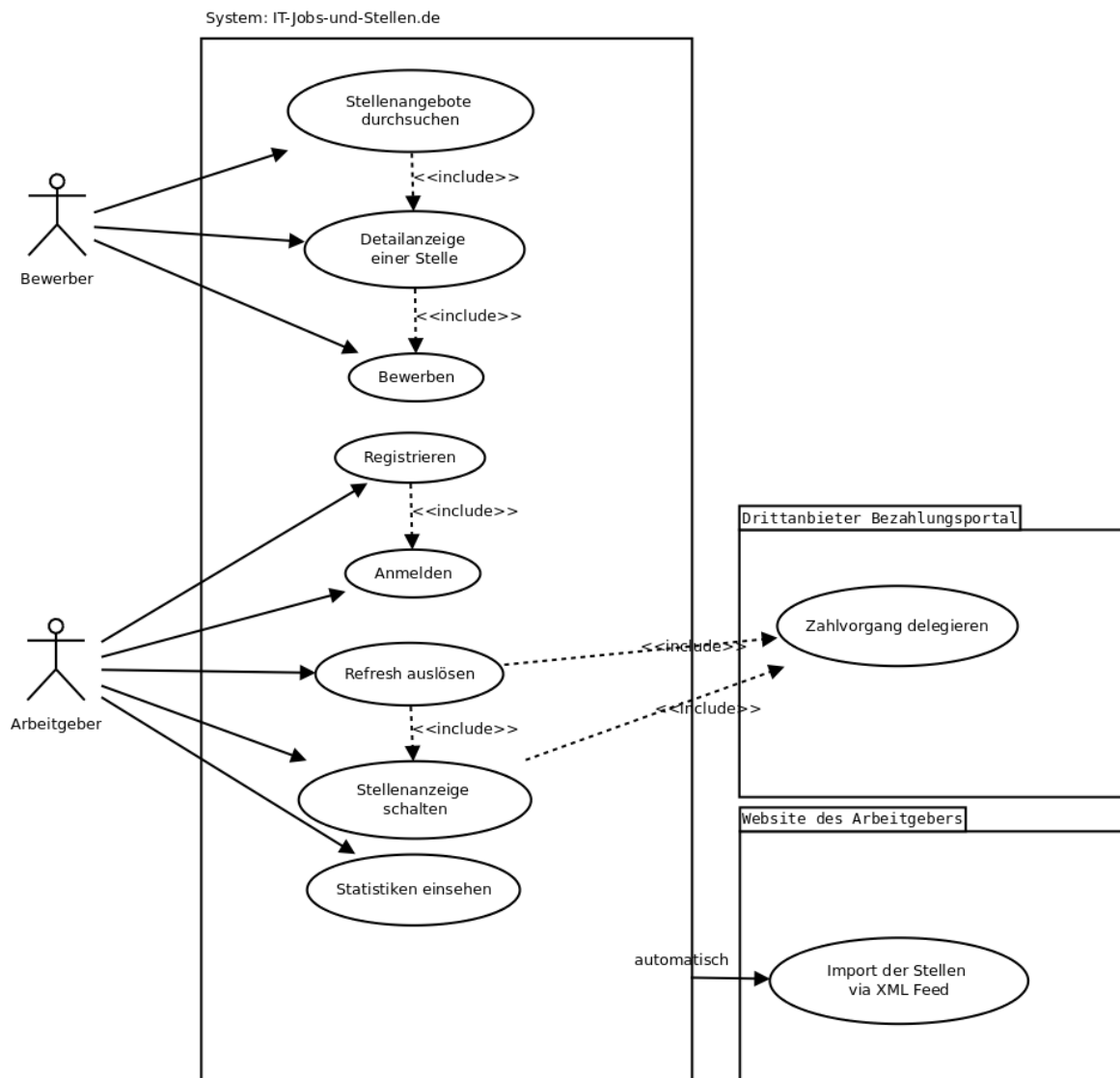


Abbildung 1.2.: Anwendungsfälle

- Ein Arbeitgeber kann eine Stelle verlängern oder kopieren, um als sie als Vorlage für ein neues Stellenangebot zu nutzen.
- Ein Arbeitgeber kann eine laufende Stelle erneuern (refresh), um so seine Platzierung verbessern. Dies ist möglich, weil die Aktualität einer Stelle Auswirkungen auf die Platzierung in den Suchergebnissen hat. Ein Refresh kostet Geld und löst somit wieder einen Bezahlvorgang aus.
- Ein Arbeitgeber kann Statistiken über seine geschalteten Stellenanzeigen einsehen. Dies beinhaltet die Anzahl der Zugriffe pro Tag und Anzahl der Bewerbungen je Stelle.
- Ein Arbeitgeber kann seine Stellen mittels eines XML-Feed-Imports automatisiert einlesen. Dazu bezahlt er einen Pauschalbetrag je Kalenderjahr. Das System liest alle 6 Stunden automatisiert diesen Feed ein, der durch HTTP erreichbar sein muss und überträgt gültige Stellenanzeigen in die Datenbank. Nicht mehr in dem Feed enthaltene Stellenanzeigen des Arbeitgebers werden in der Datenbank auf „unsichtbar“ gesetzt. Das Format ist eine Modifikation von <sup>†</sup>RSS 2.0.

#### 1.2.2. Nichtfunktionale Anforderungen

Folgende Rahmenbedingungen und zusätzliche Ziele wurden vereinbart.

- Eine hohe CO-<sup>†</sup>Testabdeckung von mindestens 95% als Grundlage für weitere Arbeiten und für den Prozess der Testgetriebenen Entwicklung
- Eine hohe Erweiterbarkeit, um langfristig auch die bereits vorhandenen Community-portale durch das neue System zu ersetzen, welche gegenwärtig auf dem Framework **Drupal 5** (PHP) basieren
- Eine moderne Suchfunktion durch einen Suchdaemon, z.B. Sphinx<sup>8</sup> oder Lucene<sup>9</sup>
- Softwarestack: Ruby 1.9.2 mit Ruby on Rails 3.1, JavaScript mit JQuery 1.6
- Die Software soll eine möglichst einfache und eingängige Bedienung haben

### 1.3. Aufbau der Arbeit

Das aktuelle einleitende Kapitel beschreibt die Motivation und Hintergründe, warum die pludoni GmbH die Testgetriebene Entwicklung mit Ruby on Rails verwenden möchte und gibt eine kurze Projektbeschreibung über das Projekt IT-Jobs-und-Stellen.de (IT-Jobs), welches parallel zu dieser Arbeit entstand.

Da die Testgetriebene Software-Entwicklung auf dem dynamischen Unittest basiert, ist das Kapitel 2 – „Automatisierte dynamische Softwaretests“ dem Testen im Allgemeinen gewid-

---

<sup>8</sup><http://sphinxsearch.com/>

<sup>9</sup><http://lucene.apache.org/java/docs/index.html>

met. Danach erfolgt eine Einführung in die *Testgetriebene Entwicklung* im Kapitel 3.

Da ein weiterer Schwerpunkt dieser Arbeit das Webframework Ruby on Rails ist, enthält das Kapitel 4 – „*Die Programmiersprache Ruby*“ einen kurzen Überblick über Ruby und Abschnitt 4.3 einen Überblick über das darauf basierende Framework Rails. Ebenfalls werden einige Möglichkeiten für Tests in diesem Umfeld vorgestellt.

Das Kapitel 5 – „*Code-Metriken*“ zeigt auf, welche Möglichkeiten existieren, um den Erfolg der Testgetriebenen Entwicklung praktisch anhand von Kennziffern nachzuweisen und den Prozess zu überwachen.

Im Kapitel 6 – „*Entwicklungsmethodik und -Werkzeuge*“ wird ausgeführt, wie die vorgestellten Werkzeuge und Methoden konkret in der pludoni GmbH eingesetzt werden sollten und welche zusätzlichen Faktoren es zu beachten gilt. Im praktischen Kapitel 7 – „*Anwendung der Testgetriebenen Entwicklung*“ wird die Testgetriebene Entwicklung mit Ruby on Rails an konkreten Beispielen erklärt und einige Sonderfälle, wie das Testen externer Abhängigkeiten im Unterabschnitt 7.3, erläutert. Danach schauen wir uns konkrete Ergebnisse im Kapitel 8 – „*Auswertung*“ der Code-Metriken an und vergleichen diese mit anderen Projekten der pludoni GmbH und als Ausblick auch mit einigen bekannteren Rails-Projekten.

Zum Ende, im Abschnitt 9 – „*Fazit*“, werden Ergebnisse dieser Arbeit zusammengefasst und Vorschläge für eine weitere Forschung gegeben.

Im Appendix sind Hinweise für die Erstellung guter Tests aus der Literatur und die Anleitung, wie eine effektive Browsersimulationsumgebung eingerichtet wird, hinterlegt. Am Ende der Arbeit erfolgt eine Auflistung der verwendeten Abbildungen, Quellcodes, das Literaturverzeichnis sowie das Stichwortverzeichnis.

**Hinweis zur Notation** In dieser Diplomarbeit werden einige Begriffe verwendet, die innerhalb der Arbeit nicht weiter erläutert werden, stattdessen aber im Glossar enthalten sind. Solche Einträge sind mit einem <sup>†</sup> markiert, z.B. <sup>†</sup>Testfall.

In einigen Fällen werden konkrete Techniken oder Methoden verwendet. Diese sind **fettgedruckt** dargestellt und werden bei erstmaliger Verwendung mit einer Fußnote und Quelle gekennzeichnet.

Zitiermarken im Text und die dazugehörige Ordnungsmarke im Literaturverzeichnis bestehen aus abgekürzten Verfasserbuchstaben plus Erscheinungsjahr in eckigen Klammern, z.B. [Bec02].

## Kapitel 2.

# Automatisierte dynamische Softwaretests

Zum Prüfen der Richtigkeit seiner Arbeit führt jeder Programmierer mindestens manuelle Tests durch. Bei einer Webanwendung hieße dies konkret, den Webserver zu starten und mittels eines Browsers durch die Anwendung zu navigieren, Daten anzulegen und Ausgaben der Anwendung zu kontrollieren. Mit zunehmender Größe einer Anwendung wird es immer aufwändiger, die Software zu testen, da nach jedem Hinzufügen von Funktionalität eigentlich alle Aspekte wieder getestet werden müssten, um Regressionsfehler auszuschließen.

Stattdessen werden **automatisierte Softwaretests** genutzt, um auf Knopfdruck alle bisher programmierten Tests auszuführen und so ein Bild über den Zustand der Anwendung zu erhalten. So ist automatisiertes Testen dem manuellem Testen, das auch **Ad-Hoc-Testen** genannt wird, in kürzester Zeit zeitlich überlegen [Rap11, S.4].

**Dynamische Testverfahren**, also Tests, die Programmcode ausführen und Ergebnisse beobachten, haben gemeinsam, dass sie stichpunktartig testen. Damit können sie die Korrektheit des Programmes nicht beweisen können und führen stattdessen den Programmcode mit konkreten Eingabewerten aus [Lig90, S. 49]. Diese Stichproben werden als **Testdaten** bezeichnet, die optimalerweise repräsentativ, fehlersensitiv, redundanzarm und ökonomisch sind [Lig90, S. 51].

Neben den dynamischen Tests existieren statische Analyseverfahren, wie formale Verifikation, symbolische Testverfahren oder statische Analysen. Einige statische Analysen werden später im Abschnitt 5.2 vorgestellt. Andere statische Testverfahren sind nicht Gegenstand dieser Diplomarbeit.

### 2.1. Motivation zum Testen

Weshalb sollte überhaupt Zeit dafür aufgewendet werden, Software zu testen?

Aus funktionaler Sicht dienen Tests in erster Linie dazu, das Vorhandensein bzw. Nichtvorhandensein von Software-Fehlern zu belegen [Goo06].

Aber das Testen hat auch weitere Auswirkungen auf die Software-Entwicklung: Testen führe

zu einer Minimierung der Debugphase und mache die Software-Entwicklung für Programmierer attraktiver und für Projektleiter leichter zu planen [Ors07] und insgesamt preiswerter [Lig90, S.13].

Außerdem gilt getesteter Code im Allgemeinen als robuster, fehlerärmer und leichter zu warten [Rap11]. Im Umkehrschluss bedeutet dies, dass ungetestete Software mit sehr hoher Wahrscheinlichkeit <sup>†</sup>Software-Fehler beinhaltet [Goo06].

## 2.2. Arten von Tests

Tests können nach verschiedenen Gesichtspunkten eingeteilt werden. Einige der gängigsten Einordnungen werden hier vorgestellt.

**Einteilung nach Sichtbarkeit des Quellcodes** Tests werden in **Whitebox**- und **Blackbox**-Tests eingeteilt. Whitebox-Tests finden mit Wissen über den zugrundeliegenden Code statt. Ziel eines Whitebox-basierten Testverfahrens ist es, so viele Codeabschnitte wie möglich zu testen (wie man Testabdeckung testen kann, dazu später mehr).

Blackbox-tests dagegen ignorieren den inneren Aufbau des Codes und testen entweder nur Schnittstellen oder das Gesamtsystem und beobachten Rückmeldungen des Systems auf bestimmte Aktionen. Das Ziel eines Blackbox-basierten Tests ist es, die Korrektheit der Software gegenüber den Spezifikationen zu bestimmen.

Ein Spezialfall ist der sogenannte **Greybox**-Test, der insbesondere bei der Testgetriebenen Entwicklung auftritt. Da hier der Test zuerst entwickelt wird, ist noch kein Wissen über den Zielquellcode vorhanden, aber der Entwickler hat natürlich Einblick in den Quellcode.

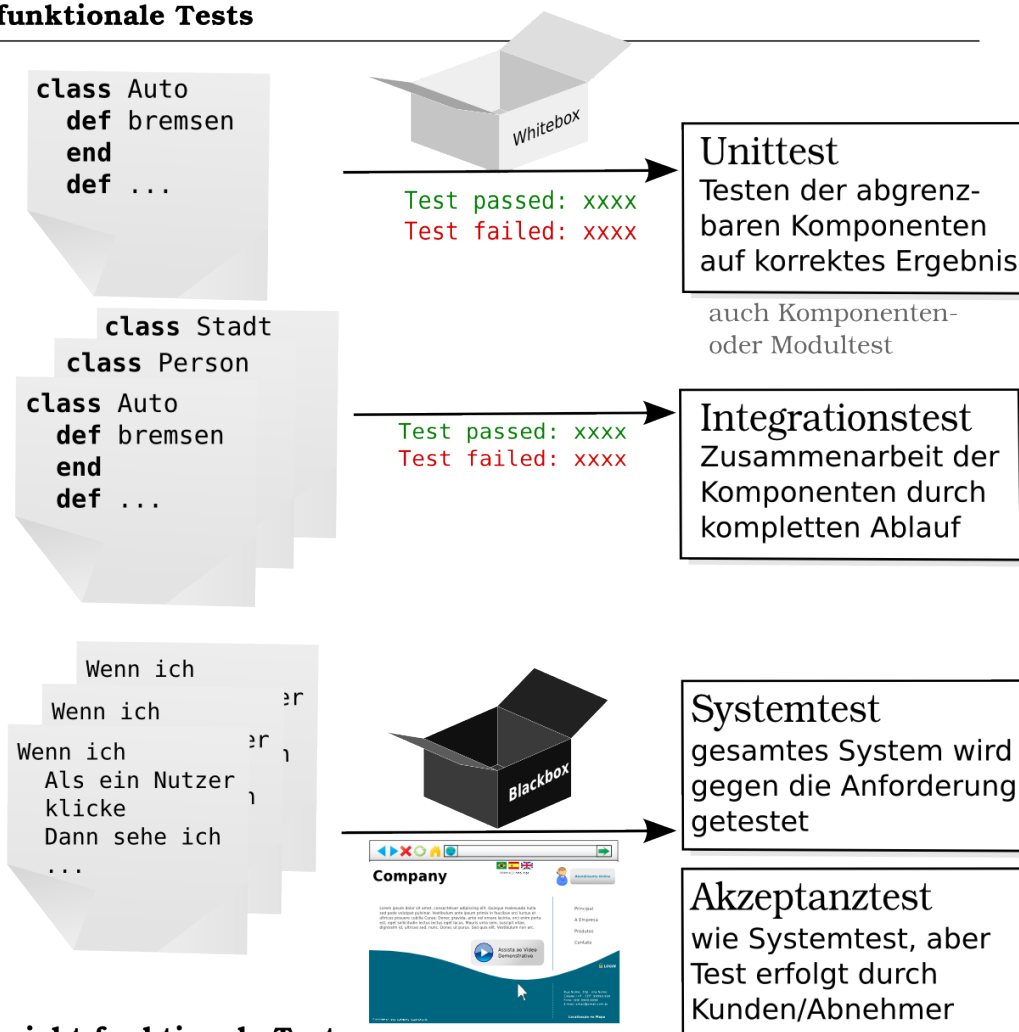
**Einteilung nach Testziel** (nach [TH99, S. 238ff])

**Unittests** Hierbei werden die Einheiten des Programmes auf ihr Verhalten getestet. Dies stellt die Basis für die meist darauffolgenden Integrationstests dar. Der Unittest wird im Detail im nächsten Abschnitt erläutert.

**Integrationstests** Das Zusammenspiel zwischen Klassen wird getestet, welche ein gemeinsames Subsystem darstellen (Modul). Hier kann eine Bottom-Up oder Top-Down-Integrationsstrategie verfolgt werden, d.h. ob man mit denjenigen Modulen beginnt, die keine Abhängigkeiten haben und im Ableitungsbaum immer weiter nach oben integriert (Bottom-Up) oder mit dem zentralen Modul anfängt und nach und nach alle abhängigen oder abgeleiteten Module testet (Top-Down) [Lig90].

**Validierung** Des Weiteren existiert der **Systemtest**, der die Korrektheit der Anwendung hinsichtlich der Anforderungsspezifikationen bestimmt. Dies ist meist ein Blackboxtest und testet das System als Ganzes (=Systemtest). Ein Spezialfall ist der **Akzeptanztest**.



**funktionale Tests****nicht-funktionale Tests**

<b>Stress/Lasttest</b> Verhalten in Extremsituationen, Zuverlässigkeit	<b>Performanztest</b> Antwortzeit, Skalierbarkeit, Nutzung von Ressourcen	...
<b>Usabilitytest</b> Verständlichkeit und Konsistenz des Nutzer-Interfaces	<b>Sicherheitstest</b> Test auf Sicherheitslücken und Programmierfehler	...

Abbildung 2.1.: Einteilung der Tests

Hierbei nimmt der Kunde eine Anforderung/ein Feature ab. Auch diesem Testtyp ist ein Unterabschnitt gewidmet.

**Ressourcennutzung, Performanz, Verhalten im Fehlerfall** Die vorherigen genannten Testarten finden i.d.R. unter idealen Bedingungen statt. Diese Testkategorie nun versucht, das Applikationsverhalten unter realen Bedingungen zu simulieren. Beim Verhalten im Fehlerfall soll getestet werden, ob der Nutzer nicht durch kryptische Fehlermeldungen verwirrt wird oder z.B. sein Fortschritt gespeichert wurde. Last-/ und Performanztests stellen sicher, dass die Anwendung eine große Zahl von Nutzern oder eine große Menge an Daten verarbeiten kann.

**Usability Testing** Diese Testmethode kann gegenüber den bisher genannten nicht automatisiert werden und benötigt immer einen (zukünftigen) Endanwender. Ziel ist es, die Benutzbarkeit und Handhabung zu testen. Dies wird durch Beobachtung von Kandidaten, meist in einer präparierten Umgebung (Usability Labor), geprüft.

**Einteilung nach Prüfverfahren** Eine Klassifikation nach Art des gewählten Prüfverfahrens ist die Einteilung in [Lig90]:

**Kontrollflussorientierter Test** Betrachtet den Quelltext und leitet daraus notwendige Unittests ab.

**Datenflussorientierter Test** Beobachtet die Variablendefinitionen, Zugriffe und Entscheidungen anhand der Variablen.

**Funktionaler Test** Leitet aus den funktionalen Spezifikationen Testfälle her und prüft, ob das Programm die Spezifikationen erfüllt.

**Diversifizierender Test** Testet verschiedene Versionen eines Programmes und vergleicht diese miteinander. Dies beinhaltet z.B. Mutationentest, der später im Abschnitt 5.2.3 *Mutations- / Perturbationstests – Defect insertion* als Möglichkeit zur Beurteilung von Tests erläutert wird.

## 2.3. Unittest

Da die Testgetriebene Entwicklung in ihrer Reinform auf dem Unittest basiert, soll diese Testgattung im Vorfeld etwas näher beleuchtet werden.

Ziel des Unittests ist es, frühzeitig <sup>†</sup>Fehler im Code zu finden. Der Unit- oder Modultest beschreibt das Testen der Einheiten eines Programms, die im Sinne der Testung nicht weiter zerlegt werden können. Dies können die Funktionen oder bei einer objektorientierten Sprache die Klassen sein. Die Objekte unter Beobachtung (*Objects under Test*) werden beim Unittest in strenger Isolation zu den restlichen Units ausgeführt. Abhängigkeiten der Module untereinander und von unterlagerten Diensten werden durch <sup>†</sup>Test-Doubles simuliert.

Dies ist notwendig, um sicherzustellen, dass gefundene Fehler von dem betreffenden Modul verursacht wurden und nicht durch äußere Einflüsse. Diese Isolierung macht das Testen auch einfacher [Goo06, Lig90].

Unittests werden fast immer durch sog. <sup>†</sup>Test-Runner automatisiert ausgeführt. Außerdem werden meist Test-Frameworks verwendet, welche häufige Aufgaben beim Testen unterstützen, z.B. Auswertung der Testergebnisse, Integration der Tests mit dem Programmcode, Anlage und Löschen von Testdaten. Eine der meist verbreitetsten sind die Frameworks auf Basis von xUnit, die in nahezu allen (objektorientierten) Sprachen Vertreter haben, so z.B. Test/Unit / MiniTest in Ruby, JUnit in Java oder NUnit in C#.

Ein Test besteht in der Regel aus 4 Teilen [Rap11] [LO11, Karte 46]:

1. Initialisierung der <sup>†</sup>Test-Umgebung und der Objekte. Man arrangiert einen Kontext, der notwendig ist, um den zu Code ausführen zu können.
2. Ausführung der zu testenden Aktion, die den Systemzustand ändert
3. Prüfen der spezifizierten Erwartungen durch Zusicherungen (Assertions): prüfen, ob das Object under Test sich wie erwartet verhalten hat.
4. Aufräumen nicht mehr benötigter Objekte, File-Pointer, Sockets, Leeren der Datenbank u.ä.

Als Faustregel für übersichtliche und nachvollziehbare Tests, gilt das **Arrange-Act-Assert**-Konzept: Dabei werden die oben genannten Phasen als Arrange (1. u. 4.), Act (2.), Assert (3.) oder auch Given-When-Then (Angenommen, Wenn, Dann) bezeichnet [LO11, Karte 46]. Nicht immer sind alle drei bzw. vier Teile notwendig.

Innerhalb von Programmiersprachen mit Objekt-Orientierung spielt die Frage, wie im Falle einer Vererbung getestet wird. Für diesen Integrationsfall gilt dann die Grundregel, erst alle Elternklassen zu testen, bevor eine konkretet Klasse getestet wird [Fri05].

Vererbung

## 2.4. Test Doubles – Mocks und Stubs

Das isolierte Testen wird durch Abhängigkeiten aller Art erschwert. Dies können z.B. Klassen sein, die noch nicht implementiert wurden, externe Ressourcen (Netzwerkzugriffe, Versenden von Mails) oder andere unterliegende Prozesse (Bezahlen in einem Onlineshop, Datenbanken) sein. In diesen Situationen ist es angebracht, auf sogenannte Test-Doubles zurückzugreifen. Diese werden u.a. in Mocks und Stubs eingeteilt.

Ein **Stub** ist eine nachahmende Funktion oder ein Objekt, welches ein schwer zu isolierende Klasse während des Testfalls ersetzt. Im Beispiel ist dies der Bezahlprozess einer Bestellung.

```

test_bestellung.rb
1 def test_report_failed_payment
2   Payment.stubs(:pay).returns(true)

```

```
3  
4  bestellung = Bestellung.new()  
5  bestellung.abschliessen()  
6  
7  assert bestellung.successful?  
8  end
```

Listing: Beispiel für den Einsatz eines Stubs in einem Bestellprozess

Mit dem oben angegeben (Pseudo-) Rubycode würde man z.B. mittels des Mock-Frameworks **mocha**<sup>1</sup> ein Stubobjekt erzeugen, welches den Bezahlprozess nachahmt und die Methode `pay` ersetzt, so dass sie immer `true` zurückgibt. So können wir simulieren, dass der extern stattfindende und möglicherweise komplexe Bezahlprozess auf unseren aktuellen Test keine Auswirkung hat. Für diesen Test interessiert uns nämlich nur, was im Falle eines erfolgreichen Bezahlens mit der weiteren Bestellung passiert.

Außerdem machen Stubs den Test meist schneller, da statt der potentiell komplexen und langsamen Operationen statische Werte geliefert werden.

Als Ergänzung dazu gibt es **Mocks**. Ähnlich wie die Stubs ersetzen sie Methoden oder Objekte, um statt komplexer Operationen fixe Werte zurückzugeben. Zusätzlich dienen Mocks selbst aber als eine Zusicherung. Ein Mock wartet darauf, ob die Methode, wie sie definiert wurde, auch tatsächlich innerhalb des Tests aufgerufen wurde.

Hier z.B. ein Mock um einen Netzwerkzugriff zu testen und abzufangen.

```
test_http.rb  
1  def test_always_fail  
2    HTTP.expects(:get).with("http://www.google.com")  
3    HTTP.get("http://www.example.com")  
4  end
```

Listing: Beispiel für den Einsatz eines Mocks zum Test eines Netzwerkzugriffes

Der gezeigte Test würde nun fehlschlagen, da von einem Mock erwartet wird, dass die nachgeahmte Funktion während des Tests genau einmal mit den genannten Parametern aufgerufen wird. Ist dies nicht der Fall, gilt der Test als nicht bestanden. Mocks fungieren somit als zusätzliche Möglichkeit, Interna des Programmflusses zu testen.

Neben den Mocks und Stubs existieren weitere Einordnungen von Test-Doubles: Spies, Fakes und Dummies. Da Mocks und Stubs als die am Häufigsten benutzten Test-Double-Klassen bereits erklärt wurden, wird auf eine tiefere Betrachtung der Test-Doubles verzichtet. Weitere Informationen sind z.B. bei Martin Fowler <http://martinfowler.com/bliki/TestDouble.html> zu entnehmen.

---

<sup>1</sup><http://mocha.rubyforge.org/>

## 2.5. System- und Akzeptanztests

Der Unittest ist als Whitebox-Test auf Wissen über den Quelltext angewiesen und sein Zweck ist es, frühzeitig Fehler in den isolierten Modulen zu finden. Der **Systemtest** dagegen testet, meist aus der Sicht eines Anwenders, das gesamte integrierte System. Er hat das Ziel, die Software gegenüber den Anforderungen zu validieren. Diese Tests finden oft unter realen Bedingungen mit realen Daten statt, meist sogar auf einer den Parametern der Produktionsumgebung nahekommende Hard- und Softwareumgebung.

Der **Akzeptanztest** ist ein Spezialfall des Systemtests. Hier führt der Auftraggeber der Software den Test selbst durch. Er nutzt den Akzeptanztest, um zu entscheiden, ob er die Software akzeptiert, woher auch der Name dieser Testkategorie rührt. Innerhalb des Kontextes der *Agilen Software-Entwicklung*, dem auch die Testgetriebene Software zuzurechnen ist, dienen Akzeptanztests dazu, den Fortschritt bei der Bearbeitung der „Geschichten“ (*user stories*) zu überwachen.

Für das Testen von Webserveranwendungen spielt die Simulation und Automatisierung von Browsern eine große Rolle. **Simulierte Browser** sind meist kleine schnelle HTTP-Clienten, die auf Skripting und Fernsteuerung optimiert sind und dabei auf viele Features von realen Browsern verzichten, z.B. Ausführung von JavaScript und Rendering des HTML-Codes.

**Automatisierte Browser** dagegen ermöglichen Tests unter möglichst realen Bedingungen. Hierbei bedient man sich meist einer Middleware, die es ermöglicht, innerhalb eines Testfalls einen Webbrowser zu starten und fernzusteuern. Eine der bekanntesten dieser Middlewares ist das Framework **Selenium**<sup>2</sup>, welches Firefox, Internet Explorer und Google Chrome fernsteuern kann. Dies ermöglicht eine sehr detaillierte Testung auf Browserinkompatibilitäten, da es unter den Browsern gewisse Unterschiede in der Ausführung von JavaScript und Darstellung von Elementen gibt. Dieses Tool wird u.a. von Google, Oracle und eBay zum Tests ihrer Anwendungen verwendet und weiterentwickelt [HQ10].

---

<sup>2</sup><http://seleniumhq.org/>

## Kapitel 3.

# Testgetriebene Entwicklung

TDD is also good for geeks who form emotional attachments to code.

[Bec02]

Extreme Programming ist eine Methodik der Agilen Software-Entwicklung<sup>1</sup>, welche das Ziel hat, die <sup>↑</sup>Code-Qualität der Software zu verbessern und flexibel auf Änderungen in den Anforderungen zu reagieren. Dazu implementiert sie einige Kernpraktiken. Eine Wesentliche, auf der viele der anderen Praktiken gründen, ist die *Test-First*-Praktik, die ab 2002 durch Kent Beck als *Test-Driven-Development* beschrieben wurde [Bec02].

Diese als Testgetriebene Entwicklung oder Test-Driven-Design (TDD) bekannte Entwicklungstechnik basiert auf der Wiederholung von sehr kurzen Entwicklungszyklen, jeder nur wenige Minuten lang. Dabei ist es das Ziel, dass sich Test schreiben, Implementation und Refaktorisierungen, d.h. das konstante Verbessern des Systemdesigns und des Quellcodes, abwechseln. Mittlerweile werden die Begriffe Test-First Development und <sup>↑</sup>Testgetriebene Entwicklung oft synonym verwendet, allerdings gibt es manchmal Unterschiede in der Herangehensweise in Punkto Design der Software: TDD findet seine Anwendung, wenn nur eine vage Idee der Funktionalität einer Klasse besteht, während TFD kein Design oder Redesign der Klassen vorsieht, sondern dieses bereits im Vorfeld der Entwicklung stattfand [SP08].

Im Folgenden wird die Entwicklungsmethode <sup>↑</sup>TDD näher beleuchtet und danach am Beispiel von Ruby/<sup>↑</sup>Ruby on Rails typische Testwerkzeuge aufzeigen.

### 3.1. Motivation

Das Erstellen einer gut abdeckenden <sup>↑</sup>Test-Suite ist für ein jedes größeres Softwareprojekt eine wichtige Voraussetzung um interne Qualitäten, wie Wartbarkeit und Zuverlässigkeit zu aktivieren. <sup>↑</sup>TDD soll nicht dazu dienen, die Software zu validieren und die Umsetzung der funktionalen Anforderungen zu belegen. Das Hauptziel ist es nämlich, den Code in Einklang

---

<sup>1</sup><http://martinfowler.com/bliki/TestDouble.html>

mit dem Test zu schreiben, so dass der Test den Code antreibt (Test drives the code), und letztendlich Design auf diese Art zum System hinzuzufügen. Die Auswirkung davon sei ein gut testbarer Code, welcher in der Regel auch ein gut wartbarer und verständlicher Code ist [Bec02]. <sup>†</sup>Code Smells, wie God-Methode und geringe Kohäsion, zu programmieren, wird alleine dadurch schon erschwert, da diese nur äußerst schwer zu testen sind.

**Psychologische Aspekte und Aspekte des Projektmanagements** Kent Beck erklärt die Hauptmotivation für TDD als das „managing fear during programming“, also dem Management von Furcht. So hat Furcht verschiedene Auswirkungen auf die Entwicklung. Sie mache zögerlich, führe zu weniger Kommunikation und damit Feedback und mache den Programmierer „mürrisch“ [Bec02, S. xi].

TDD fördert die Entwicklung in kleinen Schritten und ermöglicht durch bestandene Tests kleine „Belohnungen“ für den Programmierer. Dadurch ist es leichter, einen gewissen Arbeitsrhythmus zu erhalten, was zu dem **Flow**<sup>2</sup> führen kann und damit eine effektivere Entwicklung ermöglicht [Bro08].

Falls TDD die Dichte an Defekten<sup>3</sup> signifikant verringern würde und nur Code entstünde, der getestet wurde, so hätte dies wohl auch soziale Auswirkungen auf die Software-Entwicklung [Bec02, S. x].

1. Die Qualitätssicherung könnte von einer reaktiven auf eine proaktive Arbeit umstellen.
2. Der Projektmanager kann den Ablauf der Entwicklung besser planen, da weniger überraschende Regressionsfehler im Laufe der Entwicklung auftreten.
3. Durch eine niedrige Fehlerdichte kann die Kontinuierliche Integration (Continuous Integration) möglich gemacht werden und so der Kunde in den Entwicklungsprozess einbezogen werden

## 3.2. Ablauf

Ziel ist es, vor der Implementation eines Codes einen <sup>†</sup>Testfall (vgl. Abschnitt 2.3) zu implementieren. Davon ausgehend soll der geringstmögliche Code implementiert werden, damit der Test besteht. Als Letztes folgt die Refaktorisierung, bei TDD als Designphase genutzt. Danach beginnt der Kreislauf von Neuem.

Im Detail sind das also folgende Phasen, vgl. Abbildung 3.1:

1. Schreibe einen neuen <sup>†</sup>Testfall. Dies kann der erste eines neuen Features sein oder aber ein weiterer Test, um das aktuelle Feature weiter zu bearbeiten.

---

<sup>2</sup>Schaffen-, Tätigkeitssrausch

<sup>3</sup>Beck spricht von *defect density*, bezieht sich also auf die Anzahl pro Codemenge. Ein Defekt ist ein abnormaler Programmzusatz, der zu Programmversagen führen kann (ISO/CD 10303-226)

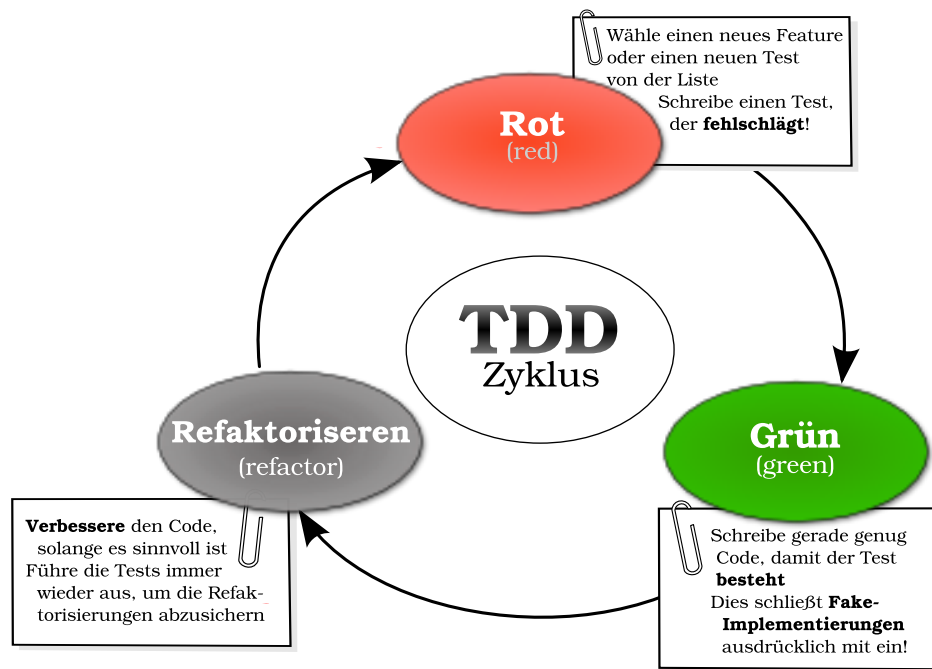


Abbildung 3.1.: Red-Green-Refactor: Der TDD Entwicklungszyklus

2. **Red**: Führe alle Tests aus, um sicherzugehen, dass der neue Test fehlschlägt. Andernfalls ist der Test überflüssig, da er keine neuen Informationen über das System gibt.
3. **Green**: Nachdem der Test fehlschlägt, implementiere nun den einfachsten Code, damit der Test besteht  
Dies kann ausdrücklich auch eine Fake-Implementierung sein, also z.B. die Rückgabe eines konstanten Wertes anstelle einer Berechnung. Wichtig ist, dass diese Phase so schnell wie möglich verlassen wird.
4. **Refactor**: Nachdem der Test bestanden wird, folgt nun die **wichtigste Phase**, die Refaktorisierungsphase.  
Da wir bereits einen Test haben, der unser gewünschtes Systemverhalten widerspiegelt, können wir gefahrlos <sup>↑</sup>Refaktorisieren, d.h. Design iterativ zum System hinzufügen und Duplikation zu eliminieren. Nun macht sich der Programmierer Gedanken, wie die vorhandenen Klassen optimal refaktorisiert werden können, um <sup>↑</sup>Code Smells zu eliminieren, und welches <sup>↑</sup>Entwurfsmuster ggf. angewendet werden kann.

Ein genau spezifizierter Ablauf ist in Abbildung 3.2 zu finden. Auch dort ist zu sehen, dass die Testerstellungs- und Refaktorisierungsphase strikt getrennt sind. Innerhalb ersterer sollte nur möglichst schnell ein funktionierender Test erstellt und zum Bestehen gebracht werden. Die eigentliche Arbeit findet dann innerhalb der Refaktorisierungsphase statt, in der die wahrscheinlich suboptimale Implementierung verbessert wird, indem iterativ Design hinzugefügt wird.



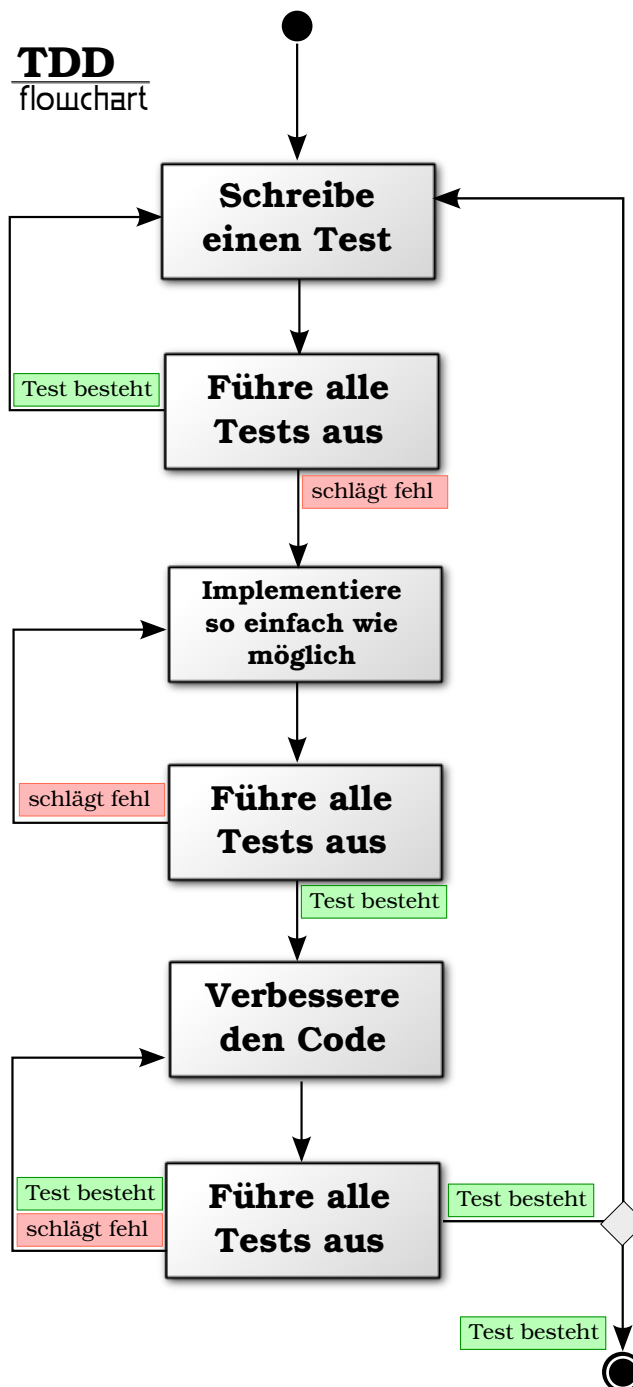


Abbildung 3.2.: Flussdiagramm für TDD

Jeder Unittest soll prinzipiell nur eine Eigenschaft testen, die Entwicklung erfolgt also in sehr kleinen Schritten. Dies hat direkte Auswirkungen auf die zu entwickelnden Objekte und Methoden, die dadurch ebenfalls übersichtlich werden sollen und somit dem Objektbegriff, eine Klasse für eine Aufgabe, gerecht wird.

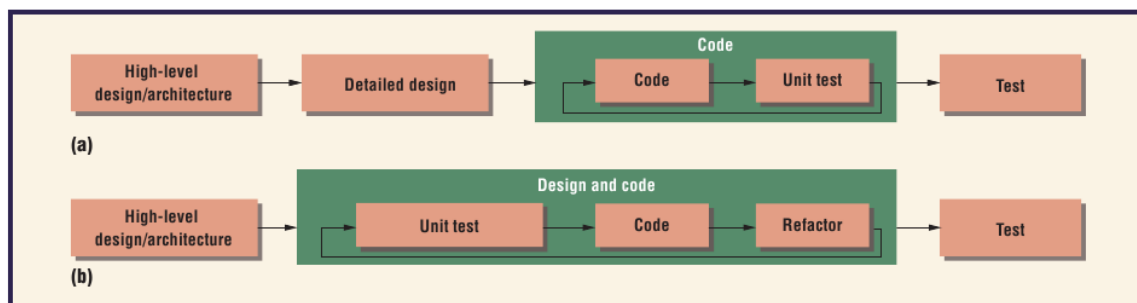


Abbildung 3.3.: Entwicklungsablauf

(a) Traditionelle Entwicklung, b) Testgetriebene Entwicklung

Quelle: [JS08]

TDD ist nur eine Entwicklungsmethodik. Der äußere Software-Entwicklungsprozess kann variieren und z.B. ein iterativer oder Agiler Entwicklungsprozess sein. TDD lässt sich in einen solchen leicht einbetten, insofern der Prozess es erlaubt, das Design während der Entwicklung ändern zu lassen und nicht schon vor den Iterationen festgelegt wird. Somit schreibt TDD nicht vor, wie eine Analyse oder ein vorhergegangener Grobentwurf, bzw. Architekturentscheidungen zustande gekommen sind.

Innerhalb einer Iteration oder eines Durchlaufes des gewählten Vorgehensmodells wird TDD an der Stelle des Design und der Entwicklung eingebettet, wie in Abbildung 3.3 dargestellt. Scheinbar wird so die (Fein-)Design-Phase aus dem Ablauf entfernt. Stattdessen findet sie sich aber in Form der Refaktorisierung wieder.

It's about the design, not the tests.

Kent Beck

### 3.3. Sonderfälle

Der oben gezeigte Ablauf ist für den Normalfall, dem Entwickeln eines neuen Features, gedacht. Für einige gesonderte Problemstellungen im Programmieralltag existieren ebenso gewisse Abläufe.

**Fehlerbehebung** Falls trotz der Verwendung von <sup>†</sup>TDD Fehler in der Software gefunden werden, so erfolgt:

1. Schreibe einen Test, der den Fehler auslöst bzw. nachbildet.
2. Behebe den Fehler im Programmcode.
3. Führe alle Tests aus.

Somit wird sichergestellt, dass jeder bisher gefundene Fehler durch einen Test abgesichert wird und jeder Programmfehler nur einmal auftritt.

**Spikes oder Spike Solution** In einigen Fällen ist es nicht ratsam, sofort mit einer Testgetriebenen Entwicklung zu beginnen. Gerade wenn Prototypisierung, d.h. schnelle, erforschende, explorative Entwicklung mit dem ausschließlichen Ziel schnell ein lauffähiges Ergebnis zu erhalten, gewünscht ist, dann kann auf Tests verzichtet werden. Eine solches, isoliert entwickeltes Experiment wird im <sup>†</sup>TDD-Jargon **Spike** (zu Deutsch: Spitze, Nadel) genannt. Die Idee dahinter ist es mehr über das Problem oder die zu erforschende Bibliothek zu lernen. Der produzierte Code sollte danach aber immer gelöscht werden und in originärer TDD-Manier neu entwickelt werden. Dies soll auch die gewählte Metapher einer Nadel aufzeigen: Schnell eine Nadel durch ein Brett bringen, als provisorische Lösung, die später durch eine besser designte ersetzt wird [Sho07]. Ausführliche Informationen über das Wann und Wie eines Einsatzes von Spikes sind in „The Art of Agile Development“ [Sho07]<sup>4</sup> zu finden.

**Testen von privaten Methoden/Attributen** Da die objektorientierte Modellierung das Konzept des Information Hiding und der Kapselung vorsieht, soll der interne Aufbau einer Klasse nach außen nicht sichtbar sein. Dies erreicht man mit den privaten Methoden und Attributen, die nur von klasseneigenen Methoden benutzt werden dürfen. Da Tests aber von außen auf eine Klasse zugreifen, stellt dies scheinbar ein Problem dar. In einigen Sprachen löst man dieses Problem mittels Reflections, um über Umwege auf private Felder zuzugreifen.

Dies spielt allerdings nur für das nachträgliche Testen von Legacy-Anwendungen eine Rolle. TDD in der Reinform betrieben sieht keine dedizierten Tests von privaten Methoden oder Attributen vor, da diese ausschließlich durch Refaktorisierung entstanden sein könnten [Car08]. Die Tests der privaten Methoden erfolgt also nur implizit über das Testen der öffentlich sichtbaren Methoden.

## 3.4. Studien zu den Auswirkungen von TDD

Viele Studien belegen, dass <sup>†</sup>TDD positive Auswirkungen auf die Software-Entwicklung hat.

So zeigt eine Fallstudie, dass TDD zu größerer Produktqualität bei gleichzeitig hoher Flexibilität führt, was ebenso zu einer höheren Zufriedenheit bei den Programmierern führt [WG07].

In dem Artikel des IEEE-Software-Journals stellen Janzen und Saiedian eine Studie vor, die

<sup>4</sup>Der Autor hat sogar das betreffende Kapitel online verfügbar gemacht [http://jamesshore.com/Agile-Book/spike\\_solutions.html](http://jamesshore.com/Agile-Book/spike_solutions.html)

TDD führt zu übersichtlicheren Klassen

akademische und industrielle Java-Projekte, die testgetrieben durchgeführt wurden (*test first*), mit denen, bei denen hinterher getestet wurde (*test last*), verglichen. Demnach zeigen die Ergebnisse an, dass Programmierer, die einen test-first Ansatz verfolgen, tendenziell Software in kleineren Einheiten, die weniger komplex sind, schreiben als solche, die erst nach der Entwicklung testen [JS08].

Auswirkungen  
auf die Struktur

Einer anderen Studie nach führt Testen im Allgemeinen dazu, dass weniger Methoden verwendet werden und Klassen geringer gekoppelt sind [Mü06]. Der Autor stellt auch eine potenzielle Metrik vor, um statistisch signifikant Projekte, die nach TDD betrieben wurden, von traditionellen Projekten zu unterscheiden: Assignment Controllability<sup>5</sup> [Mü06]. Allerdings rät der Autor zu weiteren Untersuchungen und setzt auch keinen Grenzwert an, ab welchem Grad der Controllability ein Projekt als TDD-Projekt bezeichnet werden kann.

Auswirkung auf  
die  
Code-Qualität

-Einer empirischen Studie zufolge, sei TDD schwierig zu erlernen und in einigen Metriken (Klassen pro Methode, Development Speed, Anteil der bestandenen Akzeptanztests) nicht signifikant besser als traditionelle Test-Last-Methoden. Allerdings hatten die beteiligten TDD-geführten Projekte eine signifikant bessere Testabdeckung und geringere Kopplung unter den Klassen. Test-First sei letztendlich eine mächtige aber kontraintuitive Technik [Mad09].

Verständnis von  
TDD in der  
Industrie

Eine Umfrage unter 25 IT- und Entwicklungsleitern ergab, dass diese zwar die positiven Effekte befürworten, aber nur 16% TDD in der Praxis einsetzen und nur 21% Testvollständigkeit messen. Auch versteht anscheinend etwa die Hälfte der Befragten den Begriff TDD falsch, nämlich als die Praxis, Tests für alle denkbaren Problemfälle zu schreiben [Inc07].

Dass unter den Entwicklungsleitern der Fortune-500-Firmen, die von sich selbst behaupten, TDD zu betreiben, einige von Fehlannahmen ausgehen, wird in dem oben genannten Artikel beschrieben [JS08]. So setzen diese TDD mit automatisierten Tests gleich oder behaupten sogar, TDD sei das Schreiben *aller* Testfälle vor der Implementationsphase anstelle der eigentlich gedachten kurzen Entwicklungs-Iterationen [JS08].

Auswirkungen  
auf die  
Produktivität

Eine Studie von Microsoft ergab, dass TDD-entwickelnde Teams eine 60% – 90% geringere Fehlerdichte, aber eine 15% – 35% längere Entwicklungszeit hätten als nicht-TDD-Entwickelnde [NMBW08].

## 3.5. Prinzip des Emergent Design – Inkrementelles Software-Design

Ein Konzept, das TDD effektiv macht, ist das sich iterativen Design (Emergent Design). Gegenüber traditionellen Entwicklungsansätzen erfolgt der Entwurf (Design) hierbei nicht als eigenständige Phase im Prozess, sondern ist streng in die Implementation integriert.

---

<sup>5</sup>Dies ist ein Maß, inwieweit der lokale Zustand einer Klasse/Methode von außen durch Parameter beeinflusst werden kann

Immer wenn ein Zyklus beim Refaktorisieren angelangt ist, findet nämlich effektiv Design statt. Eine Entwicklung nach TDD sucht den minimalsten Code, der die Anforderungen (Tests) erfüllt. Analog dazu, will ein Emergentes Design die kleinste Menge an benötigtem Design suchen, die die Anwendung benötigt. Durch die vielen Iterationen und die daraus folgenden zahlreichen Refaktorisierungen tritt nach und nach das Design hervor, welches optimal für das System ist.

Einige Software-Architekten ([For10, Ree92, Van10]) stellen fest, dass praktische Software-Entwicklung kein Produktionsprozess ist, der sich nach herkömmlichen Methoden planen lässt. Dort sind Ingenieure für die Planung und das Design verantwortlich, welches in der Regel ausschließlich im Vorfeld der Implementierung stattfindet. Das klassische Software-Engineering empfindet diesen Prozess nach und erstellt ein Design im Vorfeld der Entwicklung. Diese Autoren äußern nun, dass dies für die meisten Softwareprojekte nicht ideal sei, da sich die Businessanforderungen fast immer im Laufe einer Entwicklung ändern. Traditionelles Softwaredesign abstrahiere zu früh und spekuliere, ohne die letztendlichen Fakten zu kennen. Die ingenieurswissenschaftlichen Disziplinen hätten außerdem die Beschränkung, dass ein Build-Prozess äußerst teuer ist (man denke den Bau einer Brücke oder die Entwicklung von Flugzeugen), wohingegen Software-Entwicklung diese Beschränkung nicht besitzt, da ein Build meist (fast) nichts koste. Dadurch kann die Software-Entwicklung ein iteratives Vorgehen nutzen [For10, Van10, Ree92]. Statt eines großen Designs am Anfang<sup>6</sup> soll das Design durch Entdeckung und Extrahieren aus dem Quelltext gewonnen werden.

Gründe für eine Minimierung des Designs

Beim inkrementellen Design wird jedes Design-Artefakt, ob nun Methode, Klasse, Namespace oder sogar Architektur, nur deshalb erstellt, um ein bestimmtes, aktuelles Problem zu lösen. Dieses Design tendiert dazu, sich in Schüben zu entwickeln: Meist ändert man nur kleine Dinge, bis man durch eine Idee einen Durchbruch (Breakthrough) erzielt und eine größere Abstraktion durchführt [Sho07, Eva03]. Dies kann in allen Abstraktionsleveln (Methode, Klasse oder Architektur) stattfinden.

Wie findet Abstraktion statt?

Falls man **Paar-Programmierung** anwendet, so ist es die Aufgabe des passiven Teilnehmers, über das aktuelle Design zu reflektieren und Refaktorisierungen zu überdenken. Auf diese Weise können Durchbrüche auf Methoden-Level mehrmals pro Stunde stattfinden.

Während der Entwicklung sollte der passive Teilnehmer der Paarprogrammierung ebenfalls darüber nachdenken, ob Ähnlichkeiten anderer Klassen zu der aktuell Entwickelten existieren. Falls er Redundanzen sieht, so sollte er diese in einer passenden Refaktorisierungsphase zur Sprache bringen. Beide Teilnehmer diskutieren dann mögliche Abstraktionen. Durchbrüche dieser Art können mehrmals pro Tag stattfinden (abhängig vom bereits vorhanden Design).

Ähnliches gilt dann im Architekturlevel, allerdings finden hier Durchbrüche nur alle paar Monate statt und werden im gesamten Entwickler-Team diskutiert. Als Grundregel soll-

---

<sup>6</sup>(BDUF – Big Design Upfront)

te man aber beim Einführen neuer Architekturelemente so konservativ wie möglich sein [Sho07].

Ein Emergent Design kann auch ohne TDD in iterativen Entwicklungsprozessen verwendet werden. Allerdings ist die Umsetzung ohne das Vorhandensein einer guten Test-Suite ein risikoreiches Unterfangen. TDD sieht bereits eine Refaktorisierungsphase vor und zusammen mit einer Paarprogrammierung bietet es damit perfekte Voraussetzungen für den Einsatz eines inkrementellen Designs.

### 3.6. Varianten

#### 3.6.1. ATDD – Acceptance TDD

Oftmals ist eine Dokumentation und automatisierte Prüfung der Anforderungen durch Akzeptanztests gewünscht. Hier findet die Akzeptanztest-getriebene Entwicklung als Modifikation von <sup>↑</sup>TDD ihre Anwendung. Statt der Unittests stehen hier die Akzeptanztests im Vordergrund (vgl. 2.5 zum Thema Akzeptanztests).

Die Entwicklung von Unittests nach TDD lässt sich einbetten, wodurch man nun zwei Testebenen erhält [CAH<sup>+</sup>10, S. 285]:

1. Schreibe einen Akzeptanztest/Systemtest für ein Szenario des Feature
2. Prüfe, ob der Akzeptanztest fehlschlägt
3. Implementiere das Feature. Da die Features teilweise komplex sind, verfähre nun nach dem TDD-Schema, um benötigte Klassen zu entwickeln, die notwendig sind, um das Feature zu realisieren:
  - a) Schreibe einen Unittest
  - b) Prüfe, ob der Test fehlschlägt, andernfalls zurück zu a)
  - c) Implementiere mit so wenig wie Code möglich, so dass der Test besteht
  - d) Refaktorisiere
4. Nachdem der Akzeptanztest besteht, prüfe etwaige (klassenübergreifende) Refaktorisierungen

Somit werden 2 Testebenen erstellt, die der Akzeptanz- und der Unittests. Das Ganze wird auch als Outside-In-Entwicklung bezeichnet [CAH<sup>+</sup>10].

#### 3.6.2. Behavior Driven Development

Bei der Behaviour-Driven-Development (BDD) geht es um die Implementierung einer Anwendung durch Beschreibung ihres Verhaltens **aus Sicht des Kunden** [CAH<sup>+</sup>10, S. 138]. Im

Gründe entstehen neue Programmfunktionen auch hier nur durch vorhergegangene Tests. Allerdings bezieht sich das Vokabular weniger auf die Domäne „Test“, sondern sucht ein gemeinsames Vokabular für Kunden und Programmierer.

Die Begriffe aus der Testgetriebenen Entwicklung werden innerhalb von BDD wie folgt verwendet [CAH<sup>+</sup>10, S. 151]:

- Zusicherung (assertion) → Erwartung (expectation), oft auch ausgedrückt meist durch das Modalverb „sollte“ (should)
- Test-Methode → Code Beispiel (code example)
- Test-Fall/Test → Beispielgruppe (example group)
- Test → Spezifikation (spec)

BDD ist demnach eine Umformulierung von TDD, um sie besser in moderne agile Prozesse einzubetten, da diese ein ähnliches Vokabular nutzen.

#### 3.6.3. Design Driven Testing

Design Driven Testing soll eine Umkehrung von Testgetriebener Software sein und wird als Alternative dazu vorgeschlagen [SR10]. Die Autoren kritisieren, dass TDD in Reinform betrieben, lediglich Unittests, aber keine Dokumentation oder Tests höherer Ebenen produziert (Integrations- oder Akzeptanztests). Weiterhin monieren sie, dass TDD zu schwierig und aufwändig sei. Sie schlagen vor, stattdessen die Tests durch das Software-Design steuern zu lassen und sich auf komplexe Code-Abschnitte zu konzentrieren, anstatt wirklich jeden Code durch einen vorausgegangen Test entstehen zu lassen. Sie empfehlen die Nutzung von Akzeptanz- anstelle der Unittests. Code-Qualität soll durch ein gründliches vorheriges Design anstelle nachträglicher massiver Refaktorisierungen bewerkstelligt werden. DDT eignet sich für größere Teams, da Wert auf manuelle Tests gelegt wird und z.B. ein QA-Team eingebunden wird. Da das Entwicklerteam der pludoni GmbH ein sehr kleines ist, wird auf diesen Entwicklungsprozess nicht näher eingegangen.

## Kapitel 4.

# Die Programmiersprache Ruby

Sometimes people jot down pseudo-code on paper. If that pseudo-code runs directly on their computers, it's best, isn't it? Ruby tries to be like that, like pseudo-code that runs.

Yukihiro Matsumoto

Ruby ist eine Programmiersprache, die von 1993 von Yukihiro Matsumoto bis heute entwickelt wurde. Dabei ließ er sich von seinen Lieblingsprogrammiersprachen Perl, Smalltalk, Eiffel, Ada und Lisp inspirieren, um eine neue Programmiersprache zu entwickeln, die sowohl funktionale, objektorientierte als auch prozedurale Programmierung ermöglicht [Rub11].

Eine vollständige Einführung in Ruby zu geben würde den Rahmen dieser Diplomarbeit sprengen. Stattdessen legen wir einen Querschnitt durch die Sprache an und stellen die Hauptmerkmale und Unterschiede zu anderen Sprachen heraus. Auch werden Auswirkungen auf das Testen diskutiert und mögliche Testwerkzeuge vorgestellt.



### 4.1. Einführung in Ruby

Ruby, zu dt.  
Rubin

Ruby ist eine interpretierte Sprache, auch Skriptsprache genannt. Dies bedeutet, dass der Programmcode zur Laufzeit analysiert und ausgeführt wird. Ruby ist auch eine Multiparadigma-Sprache, die Objektorientierung, prozedurale und funktionale Programmierung unterstützt.

**Prozedural** Funktionen und Variablen können außerhalb von Klassen definiert werden, in dem sogenannten `main`-Objekt.

**Objektorientierung** Alle Datentypen sind Objekte. Alle Variablen beinhalten Referenzen auf ein Objekt. Dies betrifft auch die primitiven Datentypen wie Integer und String.

**Funktional** Anonyme Funktionen und Closures sind Sprachbestandteil. Alle Anweisungen haben einen Rückgabewert. Für eine Funktion ist dies immer ihr letztes Statement, falls kein expliziter Rücksprungpunkt gesetzt wurde.



Das Ziel von Ruby ist es, nicht (nur) maschinenlesbar zu sein, sondern vor allem die Lesbarkeit und Nutzbarkeit durch Menschen zu verbessern. Dies drückt sich durch eine Syntax aus, die oft laut als englische Sprache gelesen werden kann und an vielen Stellen auf den Einsatz von Sonderzeichen verzichtet. So ist die Klammerung von Funktionsaufrufen optional und kann weggelassen werden, solange die Zuordnung der Parameter eindeutig ist. Auch hält Ruby eine Vielzahl von redundanten Keywords bereit, um dem Programmierer mehrere Wege zur Lösung seines Problems zu eröffnen (‘syntaktischer Zucker’).

Im Nachfolgenden sind einige Beispiele für die Verwendung von Ruby dargestellt, insbesondere die „Alles ist ein Objekt“-Philosophie. Dazu kommt die **Interaktive Ruby Shell (IRB)** zum Einsatz, welche im Dialogverfahren Ruby-Code auswertet. Eingaben sind mit `>>` gekennzeichnet, die Ergebnisse darunter mit `=>`.

Ruby is simple  
in appearance,  
but is very  
complex  
inside, just  
like our  
human body

Yukihiro  
Matsumoto

```

----- Interaktive Ruby Sitzung (IRB) -----
1  >> 2.even?
2  => true
3  >> "hallo".upcase
4  => "HALLO"
5  >> Date.today + 2
6  => #<Date: 2011-06-30>
7  >> a = 4 + Math.sqrt(9)
8  => 7.0
9  >> if (0..10).include? a
10 >>   puts "a liegt zwischen 0 und 10"
11 >> end
12 => a liegt zwischen 0 und 10

```

Listing: Ruby Beispiele

In den ersten beiden Beispielen sieht man, dass Zahlen und Strings Objekte sind und Instanzmethoden besitzen. Im ersten Beispiel wird geprüft, ob die Zahl gerade ist. Dabei existiert eine Konvention, dass Methoden mit boolschen Rückgabewerten mit einem Fragezeichen am Ende notiert werden. Im dritten Beispiel wird eine Klassenmethode `today` auf die Klasse `Date` ausgeführt, welche ein Datumsobjekt, das den heutigen Tage beinhaltet, konstruiert und zurückliefert. Da auch die Nutzung von Operatoren letztendlich nur syntaktischer Zucker für Methodenaufrufe sind, wird die Instanz-Methode `.+()` auf dieses Datumsobjekt ausgeführt und liefert ein neues Datumsobjekt, welches zwei Tage in der Zukunft liegt.

Im vierten Beispiel wird der Einsatz von Variablen demonstriert. Das letzte Beispiel zeigt den Einsatz von Kontrollstrukturen. Als Besonderheit sei hier auf den Ausdruck vom Typ `Range(0..10)` hingewiesen, der ein Intervall für den Integerzahlenbereich von 0 bis einschließlich 10 liefert. Die Methode `.include?(a)` testet nun, ob die Variable `a` in diesem Intervall liegt. Bei Eindeutigkeit können, wie oben bereits angesprochen, die Klammern eines Methodenaufrufes weggelassen werden.

Weiterhin erlaubt Ruby die Arbeit mit Lambdas, also anonymen Funktionen. Eine beliebte Verwendungsmöglichkeit ist die Bearbeitung von Arrays und listenähnlichen Strukturen.

```
1 >> adder = lambda { |a,b| a + b }
2 >> adder.call(1,2)
3 => 3
4
5 # Sortiere nach Standardvergleichsoperator
6 >> [4,5,7,3].sort()
7 => [3, 4, 5, 7]
8
9 # Es kann auch eine benutzerdefinierte Sortierfunktion
10 # angegeben werden
11 >> [ "string", "rails", "ruby" ].sort_by{ |item| item.length }
12 => ["ruby", "rails", "string"]
13
14 # Die ersten Zahlen der 2er Multiplikationsreihe
15 >> (1..5).map{ |element| element * 2 }
16 => [2, 4, 6, 8, 10]
```

Listing: Ruby Beispiel: Blöcke

Das erste Beispiel zeigt, wie Funktionsausdrücke in Variablen gespeichert werden können, um später aufgerufen zu werden. Hier wird eine Addierfunktion definiert und aufgerufen. Das zweite Beispiel zeigt, wie Arrays verwendet werden und durch eine bereits eingebaute Methode `sort` sortiert werden können. Falls die enthaltenen Objekte nicht trivial verglichen werden können, ermöglicht die Methode `sort_by` des dritten Beispiels die Angabe einer benutzerdefinierten Sortierfunktion, hier z.B. die Sortierung nach der Länge eines Strings.

Im letzten Beispiel wird demonstriert, wie die Funktion `map` verwendet wird, die eine beliebige Funktion auf alle Elemente der Liste ausführt. Hier quadrieren wir alle Elemente unserer Liste und erhalten die quadrierte Liste als Rückgabewert von `map` (Die originale Liste bleibt dabei unverändert).

**Typ- und Objektsystem** Wie schon erwähnt, sind bei Ruby alle Datentypen ein Objekt. Dies schließt insbesondere Klassen und primitive Datentypen mit ein, wie wir im Folgenden Beispiel sehen können:

```
1 >> 2.class
2 => Fixnum
3 >> Fixnum.class
4 => Class
5 >> Class.class
6 => Class
7
8 >> Fixnum.superclass
9 => Integer
```

```

10 >> Fixnum.ancestors
11 => [Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]

```

Listing: Klassenhierarchien

Das Literal 2 ist somit ein Objekt vom Typ `Fixnum`. Die Klasse `Fixnum` ist ihrerseits vom Typ `Class`. Da Ruby sowohl (Einfach-)Ableitungen als auch sogenannte Includes bzw. Mixins unterstützt, kann eine Klasse auch eine Menge von „Vorfahren“ haben. Die gezeigte Klasse `Fixnum` verfügt somit standardmäßig sogar über 7 Oberklassen, die entweder aus einer direkten Ableitung entstammen oder aber durch das Einmischen eines Mixins.

Ruby ist **dynamisch stark** typisiert. *Dynamisch* bedeutet, dass die Prüfung des Typs einer Variable zur Laufzeit stattfindet und sich dieser Typ ausschließlich aus dem aktuell beinhaltenen Objekt ergibt. Durch die *starke* Typisierung ist es aber nicht möglich, invalide Operationen auf Typ-inkompatible Objekte auszuführen, beispielsweise eine Addition von Integer mit String.

Rubys Typsystem ist „Duck-typed“, d.h. dass die Semantiken eines Objekts nicht durch seine Klasse und Ableitungshierarchie, sondern durch seine Methoden und Attribute bestimmt wird.

Ruby verfügt über lexikalische und dynamische Bindung<sup>1</sup>, letztere wird allerdings seltener verwendet. In der Basissyntax verwendet Ruby statische Bindung. Es existiert eine im Ruby-Core enthaltene Bibliothek `Dynamic` zum dynamischen Binden, falls dies gewünscht sein sollte.

**Reflektion und Introspection** Sprachen mit Reflektion erlauben es, ihre Strukturen zur Laufzeit zu analysieren und zu verändern. So können in Ruby z.B. Objekte Informationen über ihre vorhandenen Instanzvariablen oder Methoden abfragen. Wichtig anzumerken sei noch, dass Klassen in Ruby nie geschlossen sind, sondern jederzeit erweitert werden können und vorhandene Methoden überschrieben werden können. So ist es z.B. möglich, die String-Klasse um eigene Funktionen zu erweitern.

"When I see a  
bird that  
walks like a  
duck and  
swims like a  
duck and  
quacks like a  
duck, I call  
that bird a  
duck."

James  
Whitcomb  
Riley

```

1 >> class String
2 >>   def remove_whitespace
3 >>     self.gsub(/\s+/, "")
4 >>   end
5 >> end
6
7 >> "Dies ist ein Test".remove_whitespace
8 => "DiesisteinTest"

```

Listing: Ruby Beispiel offene Klassen

<sup>1</sup> **Statische Bindung:** Variablen werden zur Compile-Zeit gebunden, ohne den aufrufenden Code zu berücksichtigen.

**Dynamische Bindung:** Variablen-Bindung kann nur im Moment der Ausführung des Codes festgestellt werden.

**Generische Programmierung und Aspekte der Metaprogrammierung** Metaprogrammierung umfasst die Analyse, Transformation und Generierung von Objektprogrammen durch Metaprogramme [Her05]. Sie ermöglicht es, Probleme effektiv zu lösen, die andernfalls nur mit erheblichem Aufwand oder gar nicht zu lösen sind.

Ein beliebtes Idiom innerhalb der Ruby-Community ist es, verwendete Methoden auf Basis des Methodennamens zur Laufzeit zu erstellen (Generierung). Dies verwendet z.B. das beliebte ORM<sup>2</sup>-Datenbankframework ActiveRecord, um einfache SQL-Statements anhand des Funktionsnamens zu erstellen (Ruby on Rails verwendet standardmäßig ActiveRecord als Schnittstelle zur Datenbank).

```
1 >> Person.find_by_first_name("Stefan")
2 # Person Load (0.2ms)  SELECT persons.* FROM persons
3 #   WHERE users.first_name = 'Stefan' LIMIT 1
```

Listing: Demonstration von generischen Funktionen bei ActiveRecord

Die Methode `find_by_first_name` existiert nicht und wird zur Laufzeit auf Basis des Namens gebaut. Dies ist möglich, da Ruby sogenannte Hooks (Callbacks) bereitstellt. Der Hook `method_missing` z.B. wird immer dann aufgerufen, wenn eine nicht-existierende Funktion aufgerufen wurde (wie in unserem Beispiel `find_by_first_name`). Hier können wir nun die neue Methode auf Basis des gewünschten Funktionsnamens zur Laufzeit erstellen oder andernfalls selbst eine Exception werfen. Ein weiterer Hook ist `method_added`, der aufgerufen wird, wenn in einer Klasse eine Methode definiert wird<sup>3</sup>. Auf diese Weise könnten z.B. die Modifikatoren `private` und `public` implementiert werden.

Ein weiteres Beispiel für generische Programmierung ist die Definition der relationalen Beziehungen zwischen den einzelnen Modellen innerhalb von <sup>†</sup>Rails (ebenfalls unter der Verwendung von ActiveRecord).

```
1 class Job < ActiveRecord::Base
2   belongs_to :user
3 end
```

Listing: Nutzung von Metaprogrammierung zur Erstellung von Objektbeziehungen

Hiermit definieren wir, dass ein Job einem User gehört, es also eine 1:N (oder 1:1)-Beziehung zwischen beiden gibt. Die dafür benötigten Getter und Setter werden mittels des Methodenaufrufs `belongs_to` in die Klasse Job geschrieben.

Diese Beispiele sollten als kurzer Einstieg in Ruby dienen und einen Querschnitt durch die Besonderheiten der Sprache aufzeigen. Für eine weitere Vertiefung sei das Buch „Program-

<sup>2†</sup>Objekt-relationales Mapping (ORM)

<sup>3</sup>Einen guten Überblick über die Callbacks, die Ruby bereitstellt, und was man damit tun kann, finden sie hier:  
<http://www.khell11.com/blog/ruby/ruby-callbacks/>

ming Ruby 1.9“ empfohlen, das im Detail auf die neueste Version der Programmiersprache eingeht [FD09].

## 4.2. Diskussion

Dynamisch typisierte Sprachen wie Ruby haben gegenüber statisch typisierten Sprachen einige Nachteile. Oft wird der Geschwindigkeitsnachteil angesprochen, den der Prozess des Interpretierens und die dynamische Typisierung verursachen. Der genaue Faktor variiert allerdings je nach Algorithmus und Implementierung stark. Ein beliebter Benchmark, „shootout.alioth“, vergleicht beliebte Algorithmen der Informatik, die in verschiedenen Sprachen implementiert sind, miteinander. So ergibt sich z.B. in der Gegenüberstellung von Ruby mit C ein 4-fach bis 300-fach langsamere Ausführungszeit. Dem gegenüber steht allerdings nur die Hälfte bis 1/7 der Menge an benötigtem Code [Gam11].

Dynamische  
Typisierung und  
Performanz

Wichtig ist auch die verwendete Laufzeitumgebung. Neben der Referenzimplementierung von Matsumotu (Ruby MRI) existieren noch JRuby, eine Implementierung auf Basis der Java virtuellen Maschine, und Rubinius. Die letzten beiden unterstützen auch eine sogenannte Just-in-time (JIT)-Kompilierung zur Verbesserung der Performanz bei längerer Ausführungszeit des Programms. Des Weiteren gibt es gerade im Bereich Laufzeitoptimierung viel Bewegung innerhalb der Ruby-Implementierungen und fast alle Ruby-Implementierungen nehmen stetig an Geschwindigkeit zu [Can10].

Allerdings bleiben Fehler, die der Compiler bereits entdeckt hätte, bis zur Ausführung oder schlimmstenfalls noch länger unentdeckt. Dazu gehören z.B. Tippfehler, bei denen der Wert einer nicht deklarierten Variable ausgelesen wird. Im Gegensatz zu z.B. PHP wirft Ruby aber dann eine Exception.

... und fehlende  
Typsicherheit

Auf das Testen hat dies eine direkte Auswirkung. Viele Autoren sind der Meinung, dass eine dynamisch typisierte Sprache mehr Tests benötigt als eine statisch typisierte [SH10].

Ein Vorteil des Interpretierens, also der Übersetzung zur Laufzeit, ist eine hohe Plattformunabhängigkeit und ein leichter Buildprozess, da das Kompilieren entfällt. Verfechter dynamischer Sprachen erklären weiterhin, dass diese sich ideal für prototypische Implementierungen eignen, da sich Anforderungen ständig ändern können. Weiterhin hätten Programme dynamischer Sprache eine potenziell hohe Wiederverwendbarkeit und eine höhere Lesbarkeit [MD05, Ous98].

Vorteile aus der  
dynamischen  
Typisierung

**Schlussfolgerung** Die Verwendung von Ruby und anderen dynamischen Sprachen birgt durchaus Risiken, die zu beachten sind. Falls man sich dieser Risiken bewusst ist und die Möglichkeiten der Sprache nutzt, um die Lesbarkeit zu verbessern, sind sie gerechtfertigt. Gerade bei der Entwicklung mit kleinen Entwicklerteams und Projekten mit engem Budget können dynamische Sprachen ihre Vorteile ausspielen, da sie eine schnellere Entwicklung ermöglichen. Im Gegensatz zu den meisten auf der Syntax von C basierenden Sprachen

(z.B. Java oder C#) ist die Syntax von Ruby äußerst leserlich, da nur wenige Sonderzeichen verwendet werden. Auch ist Ruby sehr ausdrucksstark, weil die Deklaration entfällt, es viel sogenannten syntaktischen Zucker gibt und das flexible Objektsystem viele Möglichkeiten eröffnet. Alles dies kann, richtig angewendet, der Lesbarkeit zuträglich sein.

## 4.3. Ruby on Rails

Ruby on Rails is a breakthrough in lowering the barriers of entry to programming. Powerful web applications that formerly might have taken weeks or months to develop can be produced in a matter of days.

---

Tim O'Reilly, Gründer von O'Reilly Media

Für das Projekt IT-jobs-und-stellen.de soll das Webframework Ruby-on-Rails verwendet werden. Rails wurde 2004 von der Firma 37signals<sup>4</sup> unter der Leitung von David Heinemeier Hansson entwickelt und erlangte seitdem eine wachsende Popularität. Rails inspirierte viele andere Frameworks, wie z.B. cakePHP, Symfony, Groovy on Grails und ASP.NET MVC<sup>5</sup>.



Viele professionelle Websites, die meist als Startup begannen, setzen bis heute auf Rails. Darunter Github, eine sehr beliebte Community für OpenSource-Programmierer, Groupon, das führenden Unternehmen bei Online-Gutscheinen, XING, eine deutsche Online-Community für Business-Kontakte, aber auch nicht-StartUps z.B. Yellow Pages, die Gelben Seiten der USA [37s11]. Viele bekannte Firmen nutzen Rails auch auf die eine oder andere Weise, z.B. zur Entwicklung ihrer internen Webanwendungen. Dazu gehören BBC, Cisco, IBM, Oracle, die NASA, Siemens oder Yahoo<sup>6</sup>.

Im Folgenden werden die Grundprinzipien und -konzepte von Ruby on Rails näher erläutert.

### 4.3.1. Konzepte von Rails

Rails ist ein Applikationsframework für Webanwendungen und basiert auf dem <sup>†</sup>Model-View-Controller (MVC) <sup>†</sup>Entwurfsmuster, welches eine 3-Schichten-Architektur darstellt. Jede Schicht hat fest definierte Aufgaben und sie bilden bei Rails normalerweise zusammen ein Dreigespann, **Ressource** genannt. Im Folgenden werden die Schichten kurz erläutert und am Beispiel einer Ressource „Job“ erklärt.

**Model** In Klassen dieser Schicht werden Zugriffe auf die Persistenzschicht vorgenommen. Meist geschieht dies durch Ausführung von SQL-Befehlen auf eine relationale Datenbank. Innerhalb von Rails ist das Schreiben von SQL aber meist nicht notwendig, da das <sup>†</sup>ORM-ActiveRecord häufig verwendete SQL-Befehle abstrahiert. Die Geschäftslogik soll per Definition zu großem Teil in dieser Schicht erfolgen.

Für einen Job ist das ein Modell, welches die Datenbanktabelle „jobs“ anspricht und z.B. die Attribute „titel“, „datum“ und „beschreibung“ besitzt. Dabei können auf diesem Level auch datenbankunabhängige Constraints, die **Validierungen** definiert wer-

---

<sup>4</sup><http://37signals.com/>

<sup>5</sup>[http://de.wikipedia.org/wiki/Ruby\\_On\\_Rails#Verwandte\\_Frameworks](http://de.wikipedia.org/wiki/Ruby_On_Rails#Verwandte_Frameworks)

<sup>6</sup>Weitere Firmen: <http://www.workingwithrails.com/high-profile-organisations>

den, z.B. dass ein Job nur dann gespeichert werden soll, wenn der Titel mindestens 20 Zeichen lang ist und das Datum mindestens das Heutige ist.

**Controller** Klassen dieser Schicht beinhalten Methoden, die von außen per HTTP erreichbar sind. Diese Methoden kommunizieren mit den korrespondierenden Models und bestimmen, welche View im Einzelnen ausgeliefert wird. Weitere Funktionen eines Controllers sind Authentifizierung und Autorisierung<sup>7</sup>.

Standardmäßig stellt Rails die <sup>↑</sup>Create Read Update Delete (CRUD)-Operationen bereit, welche in Form eines REST Schemas angesprochen werden (mehr zu REST später).

**View** Eine View ist in der Regel ein Stück HTML-Code, welches einem Model zugeordnet ist, das dem Clienten bei einer bestimmten Aktion ausgeliefert wird. Neben HTML ist auch JavaScript oder XML eine mögliche Auslieferungsform.

Für den Job wären Beispiele für eine View die Auflistung aller Jobs, einen Job im Detail anzeigen sowie das Formular zum Anlegen und Bearbeiten eines Jobs.

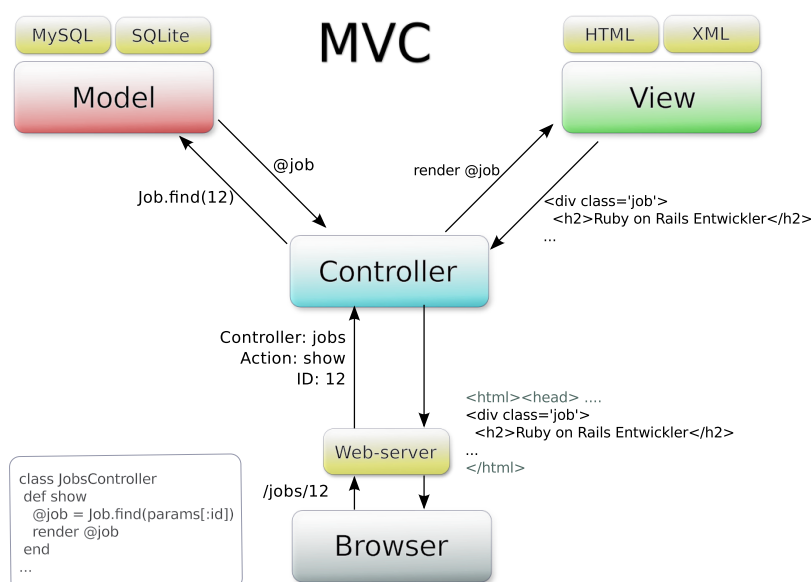


Abbildung 4.1.: MVC-Modell von Rails am Beispiel einer Anfrage `/jobs/12`

In Abbildung 4.1 ist der Ablauf einer Anfrage an den Server dargestellt. Die Anfrage des Browsers an die Website `http://localhost/jobs/12` wird über den Webserver, z.B. Apache2, an die Railsanwendung gestellt. Innerhalb von Rails wird dieser Anforderungsstring anhand der Routen, die die Anwendung anbietet, gematcht. In unserem Falle würde `/jobs/12` auf den Controller `jobs` aufgelöst werden. Innerhalb dieses Controllers wird eine Methode (Aktion) `show` erwartet. Diese Methode wird nun ihrerseits eine Anfrage an das Model `Job` stellen, den Job mit der ID 12 aus der Datenbank zu holen. Danach wird ein HTML-Template zur Detailanzeige des Jobs generiert.

<sup>7</sup>Authentifizierung: Wer ist der Nutzer?  
Autorisierung: Was darf der Nutzer?



Neben diesem architektonischen Konzept verfolgt Rails noch andere Strategien, um das Entwickeln produktiver zu gestalten.

**Convention over Configuration** Rails wurde konzipiert, um als Framework komplett out-of-the-box zu funktionieren. Außer der Datenbankeinstellung wird im Vorfeld keine Konfiguration benötigt. Diese Methodologie zieht sich auch durch das Ökosystem von Ruby. Die meisten externen Bibliotheken, bei Ruby <sup>↑</sup>Gems genannt, erschließen sich und funktionieren bereits binnen weniger Minuten. Dies macht das prototypische Entwickeln äußerst effektiv. Weiterhin ist die Struktur eines Railsprojektes fest definiert. So gibt es u. a. einen Ordner `app` mit den Model-, Controller- und View-Dateien und einen Ordner `test`, der wiederum in `unit`, `functional`, `integration` und `performance` unterteilt ist. So finden sich Railsprogrammierer auch in fremden Projekten sofort zurecht.

**Don't repeat yourself (DRY)** Hier ist das Ziel, die Duplikation soweit wie möglich zu reduzieren, um bei Änderungen nur an einer Stelle ansetzen zu müssen. Ein Beispiel ist die Definition der Attribute der Objekte durch das <sup>↑</sup>ORM. Im Gegensatz zu anderen ORM-Frameworks ist dies bei Rails nicht notwendig. Rails erstellt automatisch Getter und Setter für die in der Datenbank definierten Tabellenspalten. Hintergrund ist, dass die Definition über Name und Typ der Attribute bereits in der Datenbank vorliegt und eine Wiederholung im Quellcode dem DRY-Prinzip widerspräche.

**REST** Representational State Transfer ist eine Software-Architektur für HTTP-Web-Services. Dabei werden neben den Standard-HTTP-Methoden GET und POST auch die selten benutzten Verben DELETE und PUT verwendet, um Aktionen auf eine Ressource zu definieren. Das Ziel ist ein sehr einfaches Design der URLs.

Eine Verwendung der <sup>↑</sup>CRUD-Operationen mittels Rails würde am Beispiel einer Ressource `jobs` wie folgt aussehen:

**GET /jobs.html** Auflisten aller Jobs, Ausgabe als HTML Format.

**GET /jobs/12.xml** Job mit der ID 12 anzeigen, Formatiere als XML.

**POST /jobs** Einen Job anlegen. Alle benötigten Parameter wie Titel, Beschreibung oder Datum sollten im POST-Body der HTTP-Anfrage enthalten sein.

**PUT /jobs/12** Den Job mit der ID 12 aktualisieren. Die Eigenschaften, die aktualisiert werden, müssen als Parameter mit übergeben werden.

**DELETE /jobs/12** Lösche den Job mit der ID 12.

Rails macht das Arbeiten im Kontext dieser Architektur sehr einfach und REST gilt als die bevorzugte Methode in der Community APIs<sup>8</sup> zu erstellen.

---

<sup>8</sup>Application Programming Interface – Eine Schnittstelle, um extern mit der Anwendung zu kommunizieren

**Codegeneratoren** Rails bietet viele Codegeneratoren an, um schnell benötigte Klassen, Testfälle und Datenbanktabellen anzulegen. Wie mit Hilfe der Codegeneratoren praktisch gearbeitet wird, wird später in Kapitel 7 dargelegt werden.

**Full-Stack Webframework** Rails bringt out-of-the-box alles mit, was zur Webentwicklung benötigt wird. Im Gegensatz zu anderen Webframeworks wurde für Datenbank-anbindung, Templatesystem, JavaScript-Framework, Testframework und Webserver-API bereits eine Vorauswahl getroffen. Im aktuellen Rails 3.1 sind dies ActiveRecord (Datenbank-Framework), ERB (Templating-Sprache), JQuery (JavaScript-Framework) sowie Test/Unit und Rack (Kommunikationsinterface für Komponenten). Die meisten dieser Komponenten lassen sich zwar leicht austauschen, Rails selbst aber behauptet „opinionated“, also rechthaberisch / eigensinnig, zu sein und dem Entwickler Standards vorzugeben [Han11].

Rails will have strong defaults. They might change over time but Rails will remain opinionated.

---

D. Hansson,  
Begründer  
von Rails

Eine komplette Einführung in die Programmierung mit Rails soll nicht Bestandteil dieser Diplomarbeit sein. Für eine weitere Einarbeitung seien die folgende Quellen empfohlen:

**Rails for Zombies** Dies ist ein moderner, interaktiver Onlinekurs. Greg Pollack und das Team von RailsEnvy verpackt die Lektionen in humorige interaktive Lernerfahrungen. Jeweils eingeleitet durch ein Video, muss der Teilnehmer Aufgaben direkt im Quelltext lösen. Die Teilnahme ist kostenlos.

<http://railsforzombies.org/>

**Agile Webdevelopment with Ruby on Rails** Das quasi-Standardwerk, u. a. geschrieben vom Rails-Begründer David Hansson. Das Buch wird meist parallel mit einer neuen Rails-Version in einer neuen Auflage gedruckt, aktuell die Dritte [HT09].

**Rails Guides** Die von Ruby-on-Rails herausgegebenen „Rails Guides“ sind eine gut strukturierte, kostenlose Online-Dokumentation, die nahezu alle Aspekte von Rails beleuchtet und anhand von praktischen Beispielen erläutert.

[http://guides.rubyonrails.org/getting\\_started.html](http://guides.rubyonrails.org/getting_started.html)

### 4.3.2. Diskussion

Nach einem kurzen Überblick über Rails sollen nun die Eigenschaften des Frameworks sowie die sich daraus ergebenden Auswirkungen auf das Testen diskutiert werden.

Performanz

**Nachteile** Oft wird angeführt, dass Ruby als Skriptsprache und Rails als darauf aufbauendes Framework eine schlechte Performanz besitzt und dadurch ungeeignet für große Webanwendungen ist.

Andererseits existieren Erfahrungen, dass eine clevere Architektur und Caching für skalierende Anwendung entscheidender ist als die letztendliche Ausführungszeit [Hai10]. Dass eine Skalierung mit Rails möglich ist, zeigen z.B. Groupon, der führende Online-Coupon-

Anbieter mit mehr als 50 Mio. Abonnenten und Twitter, die jeweils Rails verwenden bzw. verwendet haben [37s11].

Da Rails ein komplettes Webframework ist, wurden bei der Auswahl der einzelnen Komponenten bereits Entscheidungen getroffen. Erst Anfang 2011 gab dazu es Kritik aus Teilen der Community, da ab der Version 3.1 „CoffeeScript“ und „SASS“ Bestandteil einer Rails-Distribution sind [Coo11]. Beide sind Zwischensprachen, die in JavaScript respektive CSS kompilieren und diese um Funktionalität erweitern. Allerdings kann jeder selbst wählen, ob er diese verwenden möchte.

Auswahl der  
Komponenten

Interaktion mit Legacy-Software ist nicht immer möglich. ActiveRecord reserviert einige Spaltennamen, wie `type` und `class`. Eine Benennung der Spalten sollte der Ruby-Namenskonvention entsprechen, also nur Buchstaben, Zahlen und Unterstriche enthalten. Ansonsten können die Spalten nur über Umwege angesprochen werden. Rails ist äußerst effektiv, wenn man den gegebenen Konventionen folgt. Im Umkehrschluss bedeutet dies, dass man deutlich ineffektiver wird, wenn man die Konventionen von Rails ignoriert, z.B. wegen äußerer Umstände und Anforderungen.

Integration in  
bestehende  
Software

Rails verzichtet auf die Verwendung von Constraints innerhalb der Datenbank. In einigen Fällen, z.B. Bankensoftware, sind diese aber unbedingt erforderlich.

Ein Nachteil aus Sicht des Projektmanagement ist der Mangel an freien Ruby-on-Rails Programmierern auf dem Arbeitsmarkt<sup>9</sup>. So kann bei einem langfristig angesetzten Projekt u. U. die Wartung nicht garantiert werden.

Mangel an  
Entwicklern

**Vorteile** Rails unterstützt den Software-Lebenszyklus, indem es von Haus aus drei Umgebungen definiert: Development, Test und Production. Diese unterscheiden sich in der Datenbank, die sie benutzen, und Konfigurationsparametern zu Caching und Performanz. Weitere Umgebungen (z.B. Staging) können jederzeit definiert werden.

Software-  
Lebenszyklus

Dank der Modularität können als Persistenzgrundlage sowohl relationale Datenbanken wie MySQL, SQLite und Oracle, aber auch andere Formen wie NoSQL-Datenbanken transparent verwendet werden. Dank einer einfach zu verstehenden Syntax ist das Schreiben von SQL in vielen Fällen überflüssig und die Sicherheitslücke einer SQL-Injection wird ausgeschlossen. Bei relationalen Datenbanken verwendet ActiveRecord standardmäßig Transaktionen, sofern die Datenbank dies unterstützt.

Flexible Daten-  
anbindung

Rails bietet eine gute Ausgangsbasis, um sichere Websoftware zu entwickeln. Das Verwenden eines Datenbankframeworks macht SQL-Injections unmöglich.

Sicherheit

Cross-Site-Request-Forgery und Session-Angriffe werden erschwert, da Session und Cookie-Variablen standardmäßig verschlüsselt werden und bei Nutzung der Formulargeneratoren ein CSRF-Token generiert wird, um Replay-Angriffe zu unterbinden.

I needed to be  
way more  
productive...

<sup>9</sup>z.B. <http://rubypays.blogspot.com/2011/04/rubyruby-on-rails-development-job.html>

Die Codegeneratoren, die Aufteilung in <sup>†</sup>MVC und die mitgelieferten Werkzeuge machen Rails äußerst produktiv und in wenigen Minuten lassen sich so bereits erste Anwendungen bauen. Der Begründer von Rails führte z.B. in einer Konferenz vor, wie man mit Rails ein Blog in 15 Minuten bauen könnte<sup>10</sup>.

**Testen mit Rails** Rails bietet ausgezeichnete Voraussetzungen zum Softwaretest. Dafür spricht, dass...

- benötigte Bibliotheken bereits mitgeliefert werden. Dies umfasst einen Test-Runner, vorkonfigurierte Test-Datenbanken (auf Basis von SQLite) und das Testframework Test/Unit,
- die Verwendung stark erleichtert wird, da Rails beim Nutzen der Codegeneratoren analoge Testdateien gleich mitgeneriert,
- neben den mitgelieferten Tools das Rails-Ökosystem eine Vielzahl von Testtools bereitstellt, u.a. Rspec (<sup>†</sup>Behavior Driven Development-Testframework), Rcov (Testabdeckung), diverse Mockbibliotheken (mocha, FlexMock, RR, Rspec Mocks), Tools zum Generieren und Bereitstellen von Testdaten (Fixtures, Factories, Faker) und <sup>†</sup>Code-Metriken (metric-fu).
- das Testen einen sehr hohen Stellenwert in der Ruby- und Rails-Community hat. Nahezu alle namhaften Ruby-Programmierer schreiben umfassende Tests [DN08]. Das Resultat ist, dass auch fast alle <sup>†</sup>Gems bei Ruby eine „solid suite of tests“ haben [DN08].

Rails unterstützt verschiedene Testarten out-of-the-box:

**Unittests** oder Modelltests (model test)

Testziel: alle (komplexeren) Methoden, die das Modell anbietet, seine Validierungen und Beziehungen zu anderen Objekten.

**Funktionale Tests** Untersuchungsgegenstand sind die Controller.

Testziel: Getestet wird meist der Arbeitsablauf innerhalb eines Controllers, also Weiterleitungen, Benachrichtigungen und welches Template gerendert wird. Damit können gleichzeitig auch View-Tests unternommen werden, also z.B. die Untersuchung, ob ein bestimmtes HTML-Element auf der Webseite zu sehen ist.

**Integrationstests** Es wird ein Browser simuliert, der von außen auf die Applikation zugreift.

Zielstellung: Testen komplexer Interaktionen zwischen verschiedenen Teilen der Software.

Beispiel: Ein User loggt sich ein und legt einen neuen Job an.

**Performanz-Tests** Eine Testart, die alle Methoden aus Unittests und funktionalen Tests beinhaltet.

---

<sup>10</sup><http://media.rubyonrails.org/video/rubyonrails.mov>

Zielstellung: Lokalisation von Performanz-Flaschenhälsen in allen Bereichen der Software.

Beispiel: Es werden 1000 Jobs generiert und geprüft, ob die Anzeige schnell genug läuft.

## 4.4. Testframeworks für Ruby

Ruby has laid the way in having a test-infected culture around the language.

Nathaniel Talbott (Entwickler von Test/Unit)

Rails liefert standardmäßig das auf dem xUnit-Schema basierende Testframework Test/Unit mit. In der Version 1.9.x bringt Ruby das Testframework Minitest in der Standardbibliothek mit, welches ebenfalls Tests nach dem xUnit-Schema unterstützt. Darüber hinaus existieren zahllose weitere Testframeworks für Ruby. Eines für den Akzeptanztest ist Cucumber, welches eine Business-readable <sup>↑</sup>Domain-Specific-Language (DSL)<sup>11</sup> für die Definition von Spezifikationen darstellt.

### 4.4.1. Test/Unit

Test/Unit basiert auf dem xUnit-, bzw. SUnit-Design von Kent Beck und ist für Nutzer von z.B. JUnit oder NUnit leicht nachvollziehbar.

Für eine zu testende Klasse wird analog eine Testklasse erstellt. Diese trägt per Definition den selben Namen wie die zu testende Klasse mit einem `Test` am Anfang. Um z.B. eine Klasse `Job` zu testen, wird eine Datei `job_test.rb` erstellt. Dort wiederum wird eine Klasse mit Namen `TestJob` definiert.

Eine solche Testklasse kann wie folgt aussehen:

Testen mit Test/Unit

```

1 require "job"
2
3 class TestJob < Test::Unit::TestCase
4   def setup
5     @job = Job.create
6   end
7
8   def teardown
9     Job.delete_all
10  end
11
12  def test_job_exists
13    @job.title = "Ruby on Rails in Entwickler"
```

<sup>11</sup>Eine domänenspezifische Sprache, die durch Kunden nachvollzogen werden kann. Siehe <http://martinfowler.com/bliki/BusinessReadableDSL.html>

```
14     @job.save
15
16     assert_equal( "Ruby on Rails Entwickler in Dresden", Job.first.title)
17 end
18 end
```

Listing: Testen mit Test/Unit in Ruby

Unsere Klasse `TestJob` erbt von der `TestUnit`-Basisklasse. Sie beinhaltet die besonderen Methoden `setup` und `teardown`, die jeweils vor bzw. nach jedem einzelnen Testfall aufgerufen werden. In der `setup`-Methode nehmen wir z.B. das Anlegen eines Jobs vor, in der `teardown`-Methode löschen wir alle Jobs in der Datenbank, um einen sauberen Test zu gewährleisten (Isolation).

Danach können nun beliebig viele Testmethoden folgen, deren Namen mit `test` beginnen müssen. Jede Testmethode besteht in der Regel aus einer Initialisierung, der Ausführung einer zu testenden Aktion und dem Prüfen der danach geltenden Eigenschaften mittels **Assertions** (vgl. ebenfalls Abschnitt 2.3. Diese Assertions, also zu Deutsch „Zusicherungen“, sind vom Testframework bereitgestellt Funktionen, die übergebene Parameter auf gewisse Eigenschaften testen und daraus einen Testerfolg oder Fehlschlag ableiten. Sollte eine Assertion innerhalb eines Tests fehlschlagen, so gilt der gesamte Testfall als fehlgeschlagen. Einige dieser Zusicherungen sind z.B.:

**`assert(statement)`** Prüft, ob der angegebene Ausdruck wahr ist (In Ruby sind alle Ausdrücke außer `false` und `nil` wahr).

**`assert_equal(expected, actually)`** Prüft, ob die beiden Statements hinsichtlich des `==`-Operators gleich sind<sup>12</sup>

**`assert_raise(exception, &block)`** Prüft, ob innerhalb des übergebenen Codeblocks eine Exception vom Typ `exception` geworfen wird.

**`assert_match(regexp, string)`** Prüft, dass der Ausdruck vom Typ `String` mit dem spezifizierten regulären Ausdruck `matcht`.

Natürlich lassen sich beliebige weitere Zusicherungen definieren. <sup>↑</sup>Rails z.B. definiert Zusicherungen um zu testen, ob ein Objekt eine gültige Instanz hinsichtlich der definierten Validierungen ist (Validierungen wurden bereits im Abschnitt 4.3.1 erläutert).

**Testdatengenerierung** Nachdem Testdaten (vgl. Abschnitt 2) einmal in zentraler Form definiert wurden, erledigt <sup>↑</sup>Rails das Management, d.h. Laden und Löschen dieser, selbstständig. Diese Art der Testdatenbereitstellung wird bei Rails **Fixtures** bezeichnet. Rails setzt selbstständig die Datenbank nach jedem einzelnen Test auf die Fixtures zurück oder kapselt die Tests innerhalb von Transaktionen, sofern die verwendete Datenbank dies unterstützt.

<sup>12</sup>Viele eingebaute Klassen prüfen auf Strukturgleichheit. Eigene Objekte werden ansonsten auf Adressengleichheit getestet. Man kann allerdings eine eigene Vergleichsoperation durch die Implementation der Instanzmethode `==()` definieren

Allerdings ist diese Form der Testdatenbereitstellung nicht unumstritten. Fixtures sind globale Daten, die in jedem Test verfügbar sind, obwohl sie nur in wenigen Testfällen benötigt werden, und machen es schwierig, Grenzfälle effektiv zu definieren. Außerdem sind die Testdaten nicht in derselben Datei wie der Test zu finden, womit man Tests nur verstehen kann, wenn man die Fixtures kennt. Eine mögliche Lösung ist es, stattdessen **Factories**<sup>13</sup> einzusetzen, die zentral Regeln definieren, wie valide Instanzen von Modellen gebaut werden. Die Tests nutzen dann die Factory, um sich z.B. einen neuen Job generieren zu lassen und für den aktuellen Testfall verändern.

Zur Generierung von größeren Mengen an zufälligen Daten einer bestimmten Domäne (z.B. für Stresstests) existieren ebenfalls Lösungen. Mittels der <sup>†</sup>Gems „populator“ und „faker“ lässt sich beispielsweise eine beliebige Menge an gültig erscheinenden Personendaten (Name, Vorname, Adresse, E-Mail-Adresse, Passwort, ...) oder Blindtexten generieren<sup>14</sup>.

#### 4.4.2. Cucumber

Cucumber ist ein relativ neues Framework (2008), um mittels einer domainspezifischen Sprache (<sup>†</sup>DSL) verständliche automatisierte Tests zu schreiben. Dabei gibt es 2 Ebenen: in der Oberen werden *Testschritte* in Englisch, Deutsch oder einer anderen der mehr als 30 unterstützten Sprachen spezifiziert. In der darunter Liegenden werden diese Schritte in echtem Testcode implementiert.

Im Folgenden sei ein Trivialbeispiel einer Anwendung, die Addieren implementiert, gezeigt.

```

features/addition.feature
1 # language: de
2 Funktionalität: Addition zweier Zahlen
3   Hier würde eine grobe Beschreibung des Businessvalues
4   und der Rahmenbedingungen stehen
5   Szenario: Addition von ganzen positiven Zahlen
6     Wenn ich "1" für a und "2" für b eingebe
7     Und auf "Addieren" klicke
8     Dann sehe ich "3"

```

Listing: Cucumber: Definition eines Additionsfeature (in Deutsch)

Wenn man nun die Datei mittels Cucumber ausführt, so wird darauf hingewiesen, dass die Testschritte noch nicht implementiert sind.

Eine Beispielimplementation (ohne Verwendung einer GUI-Anwendung) der Testschritte wäre:

```

features/step_definitions/steps.rb
1 Wenn /^ich "([^"]*)" für a und "([^"]*)" für b eingebe$/ do |arg1, arg2|
2   @addierer = Addierer.new(arg1, arg2)
3 end

```

<sup>13</sup><http://www.dan-manges.com/blog/38>

<sup>14</sup>Eine sehr gute Erklärung zur Nutzung beider Gems ist im Railscast #128 zu finden <http://railscasts.com/episodes/126-populating-a-database>

```
4
5 Wenn /^auf "([^\"]*)" klicke$/ do |arg1|
6   @result = @addierer.add()
7 end
8
9 Dann /^sehe ich "([^\"]*)"$/ do |arg1|
10  assert_equal( arg1.to_f, @result)
11 end
12
```

Listing: Cucumber: Implementierung der Additiontestschritte in Ruby

**Funktionsweise** Die Anweisungen des Akzeptanztests werden auf die definierten Testschritte gemappt, die Cucumber dann sequentiell ausführt. Jeder dieser Testschritte kann nun eine beliebige Implementierung besitzen. Meist ist es entweder eine Initialisierung, eine Aktion oder eine Erwartung, ausgedrückt durch die Schlüsselwörter Angenommen, Wenn und Dann, bzw. Given, When und Then im englischen Originaldialekt. Die Einteilung in klare Testschritte fördert die Wiederverwendbarkeit der Testschritte in anderen Szenarien.

**Vorteile für Kundenzusammenarbeit** Der Vorteil von Cucumber ist nun, dass diese Feature-Datei zusammen mit dem Kunden durchgesprochen werden kann und dass man damit an Ende eine funktionale Validierung gegenüber der Spezifikation automatisiert durchführen kann.

Die Syntax von Cucumber ist so generisch, dass damit beliebige Anwendungen getestet werden können, da keine Annahmen über die darunterliegende Implementierung der Testschritte gemacht wird. Neben Ruby wird auch die Implementierung der Testschritte in Java und C# unterstützt.

**Testen von Webanwendungen** Eine Einsatzmöglichkeit von Cucumber, besonders im Zusammenhang mit <sup>↑</sup>Rails, ist das Testen von Webanwendungen. Dabei ist es üblich, einen Browser zu automatisieren bzw. simulieren (vgl. Abschnitt 2.5), um den Test so authentisch am echten Nutzungsprozess wie möglich zu orientieren.

Um unsere Tests von der Steuerung eines konkreten Browsers unabhängig zu machen, verwenden wir verschiedene Middlewares. Wir haben uns für Capybara entschieden, welches sehr flexibel bei der Wahl der Browser-Engine ist. Der eigentliche Test, d.h. die Auswertung der Zusicherung kann durch ein beliebiges Testframework durchgeführt werden; in unserem Falle wieder mit Test/Unit. Durch Capybara haben wir nun die Möglichkeit, zwischen einem simulierten Browser (z.B. <sup>↑</sup>RackTest) und automatisierten Browsern (Firefox, Chrome, etc. durch eine weitere Middleware, Selenium) je nach Bedarf ohne große Änderungen am Testcode zu wechseln.

Der gesamte Ablauf eines Cucumber-Akzeptanztests in Verbindung mit Capybara ist in Abbildung 4.2 abgebildet. Cucumber matcht die definierten Testanweisungen auf die Testschritte und führt den darin enthaltenen Code aus. Dies kann zum Einen die Steuerung eines Webbrowsers durch Capybara sein oder zum Anderen eine Zusicherung.



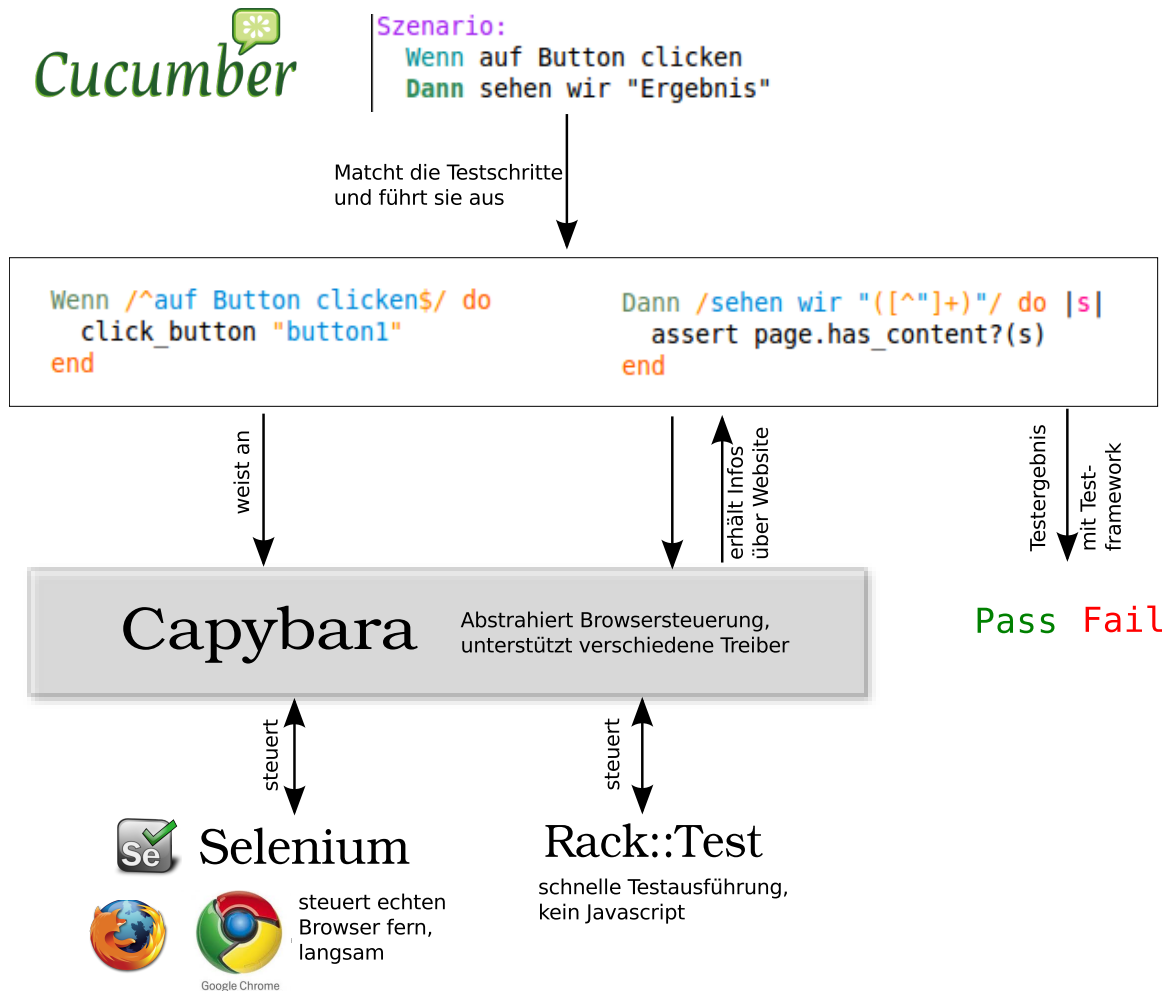


Abbildung 4.2.: Ablauf beim Akzeptanztest mit Cucumber und Capybara

# Kapitel 5.

## Code-Metriken

[A Software-metric is a] function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.

---

Definition Software-Metrik [IEE98]

Eine <sup>†</sup>Code-Metrik ist eine Maßzahl, die zum Vergleich dient und ein Qualitätsmerkmal für ein Stück Code oder ein Programm darstellt. Sie ist wird den Software-Metriken und Produkt-Metriken zugeordnet.

Dem Verwenden von Code-Metriken liegt der Wunsch zugrunde, komplexe Codeteile auf einfache Zahlen zu reduzieren, um potenziell suboptimale Codestellen schnell zu finden, welche möglicherweise in Zukunft Defekte verursachen könnten. Aus Business-Sicht stellen Code-Metriken auch eine Methode dar, Entwicklungsfortschritt zu messen und zu beurteilen.

### 5.1. Überblick über Code-Metriken und Skalen

Hier seien nun einige der geläufigsten Code-Metriken vorgestellt.

**Lines of Code (LOC)** ist eine häufig verwendete und die am leichtesten zu bestimmende Größe. Sie repräsentiert den Umfang eines Programmes. Es werden alle Zeilen einer Datei gezählt, die nicht leer und keine Kommentare sind. Kommentare wiederum können als eigene Metrik verwendet werden, um den Grad der Quelltextdokumentation zu bestimmen.

Diese Größe erhält eine größere Aussage, wenn man sie ins Verhältnis z.B. der Klasse oder eines Codefiles setzt. So kann man mit „LOC / Klasse“ schon diejenigen Klassen finden, die wahrscheinlich zu komplex sind.

**Zyklomatische Komplexität** ist ein Indikator für die Komplexität auf Basis des Kontrollflussgraphen eines Programms. Gemessen wird die Anzahl der linear unabhängigen Programmpfade. Sie ist für einen Graphen definiert durch:

$$M = E - N + 2P$$

E	Anzahl der Kanten
N	Anzahl der Knoten
P	Anzahl der verbundenen Komponenten

In einem normalen Programm ist die zyklomatische Komplexität die Anzahl der Entscheidungspunkte + 1 [McC76, S. 314].

Ein daraus abgeleitetes Testverfahren *Basis Path Testing* schlägt vor, dass die Anzahl der Tests mindestens genauso groß sein sollte wie der Grad der Komplexität [McC76, S. 318]. Dadurch erreicht man Branch-Coverage (C1) (Mehr zu Testabdeckung später im Abschnitt 5.2.2).

## Code-Smells

Ein Code-Smell ist ein oberflächliches Symptom für ein möglicherweise tiefer liegendes Designproblem. Ob ein konkreter Smell relevant ist, muss im Einzelfall entschieden werden. Für eine grobe Übersicht genügt aber z.B. auch einfach die Summe oder die Anzahl der Codesmells relativ zur Codemenge (Smells pro Tausend LOC). Welche Bad Smells für die Entwicklung entscheidend sind, hängt von der gewählten Sprache, dem damit einhergehenden Programmierparadigma und manchmal auch den verwendeten Frameworks ab. Einige für Ruby relevante Smells sind z.B. (Nach [Rut10]): Die Anzahl der Code-Smells ist eine aggregierte Metrik über Anzahl und Vorkommen dieser suboptimalen Codestellen (↑Code Smells).

**Geringe Kohäsion** ist ein Oberbegriff für verschiedene andere Smells, anzuwenden bei objektorientierten Programmen. Einer davon ist z.B. *Feature Envy* (deutsch: Neid). Eine Klasse weiß zu viel über die internen Strukturen einer anderen Klasse und implementiert Funktionalität, die eigentlich in jene Klasse gehören sollte. Im Beispiel würde die Berechnung eines Gesamtpreises z.B. in die Klasse `Checkout` gehören.

```
@checkout.total = @checkout.total_price * MWST
```

**Nichtssagender Name** gilt für alle Programmiersprachen. Falls Bezeichner weniger als 3 Zeichen lang sind oder Funktionen den Namen `do` oder `run` haben. Ausnahmen könnte man z.B. für die Schleifenvariable `i` rechtfertigen

**Gesetz von Demeter** bzw. die Verletzung desselben. Objekte sollten nur mit den Objekten in ihrer unmittelbaren Nähe kommunizieren und nicht etwa in Nachrichtenketten, wie z.B.:

```
@job.user.address.street
```

Das Gesetz von Demeter erlaubt eine solche Kette bis maximal Länge 1.

**Duplikation** und offensichtliche Ähnlichkeiten zwischen Programmstücken (oder sogar Deckungsgleichheit bei Anwendung von Copy & Paste). Fortgeschrittene Analysemethoden betrachten den abstrakten Syntaxbaum und können so strukturelle Ähnlichkeiten feststellen.

Diese und weitere Smells können mit dem Tool `reek`<sup>1</sup> festgestellt werden.

Weitere Informationen zu Smells und deren Beseitigung sind in „Refactoring“ von M. Fowler [FBB<sup>+</sup>99] zu finden.

## 5.2. Code-Metriken für Tests

Tests haben (auch) die Aufgabe, ein Programm oder Codestück auf Korrektheit<sup>2</sup> zu untersuchen. Die Tests allerdings haben ihrerseits meistens keine Tests. Um ein Mittel zu haben, Tests zumindest in ihrer Nützlichkeit zu untersuchen, existieren auch hier verschiedene Code-Metriken.

Tests sind in erster Linie natürlich auch Code und können mit den oben genannten Metriken beurteilt werden. Zudem gibt es aber einige weitere exklusive Methoden, die Qualität von Tests zu beurteilen.

### 5.2.1. Verhältnis von Lines of Test zu Lines of Code

Neben den Lines of Code kann auf dieselbe (einfache) Weise die Anzahl der Codezeilen der Testklassen ermittelt werden. Daraus ermittelt sich das Verhältnis:

---

<sup>1</sup><https://github.com/kevinrutherford/reek/wiki/Code-Smells>

<sup>2</sup>Korrektheit gegenüber den Spezifikationen, keine mathematische Korrektheit

$$R = \frac{\text{Lines of Code}}{\text{Lines of Test}}$$

$R \ll 1$	Falls es deutlich weniger Testzeile (LoT), als Codezeilen (LoC) geben sollte, so ist dies ein Indiz für zu wenige Tests
$R > 1$	Eine große Anzahl an Tests ist zwar wünschenswert, aber dies stellt keine Aussage über den Vollständigkeit oder die Qualität der Tests dar

Sollte dieser deutlich kleiner als 1 sein, so ist dies ein Symptom für zu wenige Tests. Diese Zahl ist stark von dem Testframework und dem Programmframework abhängig. Gute Projekte sollten (deutlich) mehr Testcode als Programmcode besitzen [TH99, S. 238].

### 5.2.2. Testausführungsabdeckung

Die Testabdeckung misst den Grad, inwieweit ein Programm durch die Tests berührt wurde. Dabei wird die vorhandene Test-Suite ausgeführt und währenddessen der entsprechende Quellcode beobachtet. Die Angabe erfolgt in Prozent, wobei 100% bedeuten, „das Programm wurde durch die Tests komplett ausgeführt“ und 0% „Das Programm wurde durch die Tests überhaupt nicht berührt“. Es wird festgehalten, welche Anweisungen, Zweige oder sogar Programmpfade ausgeführt wurden. Diese 3 Abstufungen sind im Detail:

- C0** (Anweisungsüberdeckung, Statement Coverage) ist die am Einfachsten zu bestimmende Abdeckung. Dabei wird geprüft, ob jede Zeile des Quellcodes während der Codeausführung mindestens einmal ausgeführt wurde.
- C1** (Zweigüberdeckung, Branch Coverage) prüft zusätzlich, ob jeder Zweig jeder Zeile ausgeführt wurde. Dies ist wichtig, falls man ternäre Ausdrücke<sup>3</sup> verwendet.
- C2** (Pfadüberdeckung, Path Coverage) prüft, ob jeder mögliche Codepfad durchlaufen wurde. Ein Codepfad ist eine einmalige Abfolge von Zweigen innerhalb einer Funktion von Eintritt bis Rücksprung [Cor96]. So werden z.B. bei 10 Bedingungen 1024 Pfade generiert, denen bei einer 100% Abdeckung auch 1024 Tests entgegenstehen müssten.

Anmerkung: In der Literatur startet in einigen Fällen die Nummerierung bei C0 [Pow08], in anderen Fällen aber bei C1 [Cor96].

Für Ruby 1.8.7 gibt es das Tool **rcov**<sup>4</sup>, für Ruby ab 1.9.1 **simple-cov**<sup>5</sup>, welche beide die C0-Testabdeckung bestimmen können. Zum aktuellen Zeitpunkt sind keine weiteren Tools bekannt, um C1- oder C2-Abdeckungen zu bestimmen.

<sup>3</sup>if-then-else in einer Zeile: `int a = (1==1) ? 5 : 3`

<sup>4</sup><http://relevance.github.com/rcov/>

<sup>5</sup><https://github.com/colszowka/simplecov>

**Wie viel Testabdeckung ist sinnvoll oder notwendig?** Beim Messen der Abdeckung stellt sich schnell die Frage, wie viel Testabdeckung sinnvoll oder gar notwendig ist. Zuerst ist die Art des Messverfahrens, also C0 bis C2, wichtig. Je komplexer das Messen erfolgte, desto geringer kann die Testabdeckung am Ende ausfallen [Pow08].

Falls dem TDD-Prozess minutiös gefolgt wurde, so müsste die C0 Testabdeckung immer 100% sein [Bec02]. Für ein Rails-Projekt ist es auch relativ leicht, 100% oder nahezu 100% zu erreichen [Rap11]. Die Zahl „100%“ ist für sich genommen nutzlos, aber sie zu erreichen, ist für den Prozess der Testgetriebenen Entwicklung nützlich [Rap11, S. 270].

Viele Autoren bringen zum Ausdruck, dass es von der Situation abhängt, wie viel Testabdeckung sinnvoll ist [Els07]. Test-Anfänger sollten sich zuerst überhaupt ans Testen gewöhnen und erfahrene Entwickler sollte wissen, dass es keine einzige einfache Antwort auf diese Frage gibt [Els07]. Zudem gibt eine hohe Abdeckung keinen Aufschluss darüber, dass die Tests „gut“ sind. Eine niedrige Testabdeckung zeigt aber deutlich auf Missstände hin.

Einem pragmatischen Ansatz von Savoia folgend, kann man aus dem Verhältnis der zyklomatischen Komplexität mit der Testabdeckung eines Codestückes suboptimale Teile finden. Je mehr Verzweigungen eine Methode hat, desto höher sollte ihre Testabdeckung sein [Sav07]. In einem Artikel empfiehlt Cornett eine Liste von Zielen, die es je nach vorhandenen Budget und Zeit zu erreichen gilt, beginnend damit, dass mindestens eine Funktion in 90% der Quelltextdateien durch die Tests aufgerufen wird bis zum finalen Schritt einer vollständigen C1-Testabdeckung [Cor96].

Zusammenfassend kann man sagen, dass es keine eindeutige Antwort gibt. Eine niedrige C0-Abdeckung von 50% oder weniger zeigt allerdings deutliche Missstände beim Testverfahren an.

### 5.2.3. Mutations- / Perturbationstests – Defect insertion

Eine weitere Methode, um die Qualität von Testcode zu bestimmen, ist der Mutationstest. Dies ist ein diversifizierendes, fehlerbasiertes Testverfahren [Lig90]. Hierbei werden (automatisiert oder manuell) nacheinander Anweisungen des Programmcodes geändert und geprüft, ob danach ein Test fehlschlägt [Bec02]. Sollte nämlich kein Test fehlgeschlagen sein, dann sind die Tests zu oberflächlich.

Für Ruby gibt es ein Werkzeug, **Heckle**<sup>6</sup>, welches dieses Verfahren implementiert. Im Detail werden Bedingungen negiert, konstante Zahlen und Funktionsaufrufe verändert, Zuweisungen verändert usw. [CD10]. Dabei wird immer eine Änderung (Mutation) vorgenommen und dann alle Tests ausgeführt. Sollte dennoch in einer Mutation kein Test fehlschlagen so waren die Tests für diese Codezeile zu oberflächlich.

---

<sup>6</sup><http://ruby.sadi.st/Heckle.html>

Um hieraus eine Metrik zu gewinnen, kann die Anzahl der Mutationen gemessen werden, bei denen der Test nicht fehlschlug. Im Verhältnis zu den Klassen gesetzt, lassen sich auf diese Weise diejenigen Klassen finden, die zu oberflächlich getestet wurden.

## 5.3. Notwendigkeit von Code Metriken

Code-Metriken geben dem Programmierer automatisiert und schnell ein Feedback über die Qualität seiner Arbeit. Sie helfen dabei, Probleme frühzeitig zu erkennen und die Wartbarkeit durch gezielte Refaktorisierungen nachhaltig zu verbessern. Auch psychologische Auswirkungen dürfen nicht unterschätzt werden. Alleine der Fakt, dass Code-Metriken in einem Unternehmen regelmäßig verwendet werden, motiviert den Programmierer, keinen sogenannten „Big Ball of Mud“<sup>7</sup> zu schreiben. Insbesondere in kleinen Projektteams, die keine dedizierte Qualitätssicherung haben, sind Code-Metriken als kostengünstiges Kontrollinstrument unerlässlich. Studien zeigen, dass der konsequente Einsatz von Code-Metriken und Analysebenchmarks die Fehlerdichte und Entwicklungskosten stark verringern kann [BCSV11, S.10f].

Für die Testgetriebene Software-Entwicklung dient insbesondere in der Anfangsphase die Testabdeckung als Kontrollinstrument um zu prüfen, ob der TDD-Prozess korrekt umgesetzt wird [NMBW08, S. 300]. Außerdem sollte der zeitliche Verlauf der Metriken beobachtet werden, um Trends abzuschätzen und frühzeitig gegensteuern zu können. Für in <sup>↑</sup>TDD erfahrene Programmierer mag die Beobachtung der Testabdeckung nicht notwendig sein, für Einsteiger ist es allerdings eine effektive Kontrollmöglichkeit.

Nach Erfahrungen in der pludoni GmbH sind Code-Metriken ein wichtiges Feedbackinstrument und unterstützen damit das Schreiben sauberen Codes. Wichtig ist, dass die Metriken regelmäßig berechnet werden, entweder als Cronjob oder nach jedem Einchecken in den Hauptentwicklungszweig der Versionsverwaltung, und in regelmäßigen Abständen von den Programmierern und Teamleitern gelesen und besprochen werden. Allerdings besteht bei einer zu hohen Beachtung der Metriken die Gefahr eines Hawthorne-Effektes, d.h. dass die unter Beobachtung stehenden Programmierer ihr Verhalten den Code-Metriken anpassen, um optimale Ergebnisse zu erhalten [LO11, 52. Karte] und so nur eine scheinbare Verbesserung erzielen.

---

<sup>7</sup>Ein Antipattern, in dem ein System keinerlei offensichtliche Architektur zu haben scheint.

# Kapitel 6.

## Entwicklungsmethodik und -Werkzeuge

### 6.1. Definition eines Entwicklungsmodells für die Bedürfnisse der pludoni GmbH

Viele der gängigen Vorgehensmodelle wie das *V-Modell*<sup>1</sup> oder der *Rational Unified Process*<sup>2</sup> finden ihre Anwendung in großen Projektteams. Für mittelgroße und kleine Projektteams gibt es seit ca. 10 Jahren die *agilen Vorgehensmodelle*. Sie haben einen eher pragmatischen Ansatz mit dem Ziel, gemeinsam mit dem Kunden eine funktionierende Software zu bauen. Sie machen sich dabei kurze Releasezyklen zu eigen, welche ein regelmäßiges Feedback erlauben. Damit wird der klassische GAU am Ende des Projektes, wenn die Wünsche des Kunden mit den tatsächlichen Umsetzungen doch nicht konform gehen, vermieden. Viele dieser Methoden wie z.B. *SCRUM*<sup>3</sup> erfordern jedoch eine Schulung für das gesamte Team, die nicht immer finanzierbar ist.

Für die Arbeit von sehr kleinen Teams mit weniger als 4 Mitgliedern wird nun ein Konzept auf Basis der Testgetriebenen Entwicklung mit Ruby on Rails vorgestellt, das auf die Bedürfnisse der pludoni GmbH zugeschnitten ist.

Diese Bedürfnisse umfassen:

- kurze Feedbackzyklen von einer Woche
- Arbeit meist aus der Ferne ohne direkte Kommunikation mit den anderen Teammitgliedern. Daraus folgt ein äußerst selbstständiger Arbeitsstil.
- möglichst fehlerfreie Software
- Kontinuierliche Integration
- pragmatisches Testen, 100% <sup>†</sup>Testabdeckung ist nicht erforderlich. Offensichtliche <sup>†</sup>CRUD-Operationen müssen nicht getestet werden. Wichtige Systemlogiken wie Bezahlvorgang und Suche müssen dagegen getestet werden.

---

<sup>1</sup><http://v-modell.iabg.de/>

<sup>2</sup><http://www.ibm.com/software/awdtools/rup/>

<sup>3</sup>[http://www.scrumalliance.org/pages/what\\_is\\_scrum](http://www.scrumalliance.org/pages/what_is_scrum)



### 6.1.1. Einteilung der Features in Kategorien

Grundsätzlich teilt die pludoni GmbH Features in zwei Kategorien ein:

- A. Features, die in der Ansicht für Kunden und Besucher der Website sichtbar sind → Detailansichten, Listen, Bezahlvorgänge, ...
- B. Features, die nur für den Admin sichtbar sind oder im Backend ausgeführt werden → Reporting, Statistiken, Indizierung der Datenbank, Cron-Skripte, Caching, ...

Features der **Kategorie A** sollen in Zukunft Akzeptanztest-getrieben entwickelt werden. Die Entwicklung verläuft nach dem Schema, das in Abschnitt 3.6.1 vorgestellt wurde. Die Akzeptanztests sollen in Cucumber geschrieben werden und mittels Capybara auf simulierten Browsern ausgeführt werden (vgl. Abschnitt 4.4.2). Ziel ist es, das bei Webanwendungen übliche wiederholte manuelle Ausprobieren mit dem Browser auf ein Minimum zu reduzieren. Jeglicher Vorgang, den der Kunde am Browser testet, lässt sich auch als ein Akzeptanztest formulieren. Ein automatisierter Test hat zudem den Vorteil, zu einem späteren Zeitpunkt leicht wiederholbar zu sein.

Der Vorteil dieser Outside-In-Entwicklung ist, dass sie auf den Kunden ausgerichtet ist. Die Verwendung der <sup>†</sup>DSL Cucumber fördert zudem die Implementierung von Business-relevanten Features gemeinsam mit dem Kunden. Das gesamte Vokabular orientiert sich an der Anforderungsanalyse und an Businessprozessen, die auch der Kunde, der meist keinen technischen Hintergrund besitzt, verstehen kann.

Für die von außen nicht sichtbaren Features der **Kategorie B** sollen aus Kostengründen normale Unittests, entwickelt nach der klassischen Testgetriebenen Entwicklung, genügen. Die zusätzliche Abstraktionsebene der Akzeptanztests ist nicht notwendig.

### 6.1.2. Praktiken

Der oben genannte Ablauf bezieht sich in erster Linie auf die Erfüllung der funktionalen Anforderungen. Weitere Praktiken, die für den Programmieralltag wichtig sind, umfassen:

**Kontinuierliche Integration** Die Verwendung einer Versionsverwaltung, z.B. **git**<sup>4</sup>, ist allgemein für Software-Projekte obligatorisch. Bei der kontinuierlichen Integration wollen wir sicherstellen, dass der Hauptzweig unserer Entwicklung stets ein lauffähiges Produkt enthält und dass laufende Änderungen so oft wie möglich integriert werden. Das Vorhandensein einer großen Test-Suite ermöglicht es, diese beim Einchecken in den Hauptzweig immer komplett auszuführen, um so sicherzustellen, dass dort stets eine lauffähige Version zu finden ist.

---

<sup>4</sup><http://git-scm.com/>

In großen Projekten ist es üblich, komplexe Testpläne zu erstellen. Anscheinend sind aber automatisierte Tests, die bei jedem Einchecken durchgeführt werden, effektiver als rein formale Testpläne [TH99, S. 238].

**Code-Metriken** Ein tägliches Messen des Code-Zustandes mittels Code-Metriken ermöglicht es den Programmierern, sich selbst und gegenseitig auf die Finger zu schauen. Sollte ein Programmierer nämlich schlechten Code im Hinblick auf Komplexität und <sup>↑</sup>Code Smells abliefern, so macht die Code-Analyse dies sichtbar. Dies dient nicht in erster Linie dazu, den Programmierer zu maßregeln, sondern ihm dabei zu helfen, den <sup>↑</sup>TDD-Prozess zu lernen und seinen Programmierstil ständig zu verbessern. Die Erfahrungen zeigen, dass die Programmierer meist selbst unzufrieden mit schlechtem Code sind, den sie geschrieben haben. Code-Metriken können dabei helfen, dem Programmierer so schnell ein Feedback zu seinem Code zu geben, wie es ein Code-Audit durch Andere in dieser Geschwindigkeit und Effizienz nie könnte.

**Regelmäßige Paarprogrammierung** Die Paarprogrammierung (Pair Programming) ist eine Maßnahme aus dem Katalog von Extreme Programming, bei der zwei Programmierer zusammen an einem Computer arbeiten. Sie wechseln sich beim Programmieren ab. Die Wichtigkeit der Paarprogrammierung für ein inkrementelles Design wurde in Abschnitt 3.5 diskutiert.

Aber auch beim Lösen schwieriger Aufgaben und beim Anlernen neuer Teammitglieder ist die Paarprogrammierung eine effektive Methode [HA05, S. 9]. Erfahrungsgemäß führt sie zu besser dokumentiertem Code, kann die Anzahl der Fehler verringern und zu einer höheren Arbeitseffektivität führen [HA05]. Allerdings erschwert die in Abschnitt 1.1.2 angesprochene Dezentralisierung der Zusammenarbeit eine regelmäßige Paarprogrammierung. Nichtsdestotrotz sollten in regelmäßigen Abständen Features zu zweit in wechselnder Zusammensetzung entwickelt werden.

Außerdem werden auch Werkzeuge wie Videokonferenzen und Screen-Sharing eine Rolle spielen, um eine entfernte Paarprogrammierung zu ermöglichen.

**Einsatz eines iterativen Designs** Das im Abschnitt 3.5 vorgestellte Emergent Design ergänzt sich ideal mit der bereits verwendeten Paarprogrammierung und dem Einsatz der Testgetriebenen Entwicklung. Außerdem wurde die Erfahrung gemacht, dass sich Anforderungen und Spezifikationen im Laufe der Entwicklung immer ändern, um flexibel auf neue Gegebenheiten reagieren zu können.

Falls ein Projekt in Ruby-on-Rails entwickelt wird, so gibt das Framework die Architektur vor, und ein Feindesign kann sich während der Entwicklung von selbst herausbilden. Dadurch ist es möglich, die Entwurfsphase auf ein Minimum zu reduzieren, z.B. ein grobes

ERM<sup>5</sup> und Mock-Ups<sup>6</sup> zu erstellen.

Da allerdings Paarprogrammierung nicht immer möglich ist, sind regelmäßige Abstimmungen unter den Programmierern erforderlich, um neue Designideen zu diskutieren. Dafür sollten feste Termine anberaumt werden.

## 6.2. Auswahl der Entwicklungswerkzeuge

Für die zukünftige Entwicklung von zumeist Webanwendungen werden folgende Werkzeuge berücksichtigt.

**Werkzeuge für Tests** Die Spezifikationssprache **Cucumber** dient als Schnittstelle zwischen den Anforderungen des Kunden und den vom Programmierer entwickelten Testschritten. Diese könnten in einem von vielen Testframeworks geschrieben werden. Die Entscheidung fiel hierbei auf **Test/Unit**, da die Syntax und Prädikate denen von JUnit und NUnit sehr ähneln und so den Übergang zu Ruby leichter machen. Da es auch das Standard-Testframework von Ruby on Rails ist, ist eine gute Unterstützung durch gängige Werkzeuge garantiert.

Für die Simulation eines Browsers gibt es ebenfalls verschiedene Ansätze. Als Basis fungiert dabei **Capybara**, welches unterschiedliche Browsersimulationen abstrahiert. Damit lässt sich z.B. **Selenium** ansteuern, welches wiederum Mozilla Firefox, Internet Explorer oder Google Chrome fernsteuern kann. Dies ist allerdings sehr langsam, da ein kompletter Browser gestartet und ferngesteuert wird. Daher ist die Nutzung von Selenium nur für das Testen von möglicherweise problematischen Interaktionen und das Testen von JavaScript notwendig. Für alle anderen Fälle kann man auf <sup>↑</sup>**RackTest** zurückgreifen, welches extrem schnell eine Rack-Anwendung startet und testet. Falls in Zukunft mehr Geschwindigkeit in der Testausführung, insbesondere bei den Selenium-Tests, gewünscht wird, so kann man auf Parallelisierung auf mehreren Computern zurückgreifen.

Für ein unmittelbares Feedback sind auch automatische Test Runner erwünscht. Hierbei existieren **autotest**<sup>7</sup> und **guard**<sup>8</sup>. Diese Programme beobachten den Projektbaum und führen bei Änderung der Dateien automatisch die relevanten Tests aus. Um die Geschwindigkeit und damit den Feedbackzyklus zu verbessern, können diese Programme so gesteuert werden, dass sie nur den Testfall ausführen, an dem gerade gearbeitet wird.

---

<sup>5</sup>Entity-Relationship-Model

<sup>6</sup>Wegwerfprototypen der Oberfläche

<sup>7</sup><http://zentest.rubyforge.org/ZenTest/>

<sup>8</sup><https://github.com/guard/guard>

Durch **spork**<sup>9</sup> lässt sich eine <sup>↑</sup>Railsanwendung starten und im Hintergrund halten, so dass eine erneute Testausführung deutlich schneller von statten geht, als wenn die komplette Anwendung neu geladen werden müsste.

**Werkzeuge für Code-Metriken** Für die Generierung von <sup>↑</sup>Code-Metriken dient das <sup>↑</sup>**Gemetric-fu**, welches seinerseits über verschiedene Code-Metriken Zusammenfassungen bildet und diese auch in der zeitlichen Entwicklung darstellt. Darunter fallen z.B. die zyklomatische Komplexität, den Grad der Duplikationen, verschiedene Code Smells, Nutzung der Versionsverwaltung und <sup>↑</sup>Testabdeckung.

Die Testabdeckung wird durch **simple-cov** berechnet, welches eine CO-Anweisungsüberdeckung bestimmt.

Einige Code-Metriken wurden im Kapitel 5 – „Code-Metriken“ vorgestellt.

### 6.3. Diskussion

Viele dieser Maßnahmen dienen dazu, den Feedbackzyklus so kurz wie möglich zu halten. Für eine Testgetriebene Entwicklung ist es unerlässlich, dass die Testausführung schnell abläuft, d.h. dass der Programmierer innerhalb weniger Sekunden ein Feedback erhält. Andernfalls, so die Erfahrung, führt eine lange Testausführung zu einer verminderten Ausführungsrate und ist damit hinderlich für die iterative Entwicklung. Dazu ist es notwendig, dass die Tests modular und unabhängig sind, um einzeln ausgeführt werden zu können und dass bei zunehmender Größe des Projekts eine vollständige Testausführung, insbesondere der relativ langsamen Akzeptanztests, nur durch den Integrationsserver (kontinuierliche Integration) vorgenommen wird.

Um sich mit der Testgetriebenen Entwicklung vertraut zu machen, dienen die Paarprogrammierung als Lernmethode und die Code-Metriken als Kontrollinstrument. Die Erfahrung wird zeigen, wie lange eine regelmäßige Kontrolle und Nachbesserung erforderlich ist, bis die Testgetriebene Entwicklung vollständig angenommen wird. Ab diesem Zeitpunkt kann z.B. der Einsatz der Code-Metriken reduziert werden.

---

<sup>9</sup><https://github.com/timcharper/spork>

# Kapitel 7.

## Anwendung der Testgetriebenen Entwicklung

In den nachfolgenden Abschnitten wird exemplarisch an dem Objekt „Job“, also der internen Repräsentation einer Stellenanzeige, die Testgetriebene Entwicklung mit praktischen Beispielen näher erläutert. Besonderes Augenmerk soll dabei auf den Entwicklungsfluss von <sup>†</sup>TDD gelegt werden. Zu dessen Verdeutlichung ist am Dokumentenrand die jeweilige Phase innerhalb des TDD-Zyklus zu finden (Red, Green, Refactor), der die im Text gezeigten Codeabschnitte zuzuordnen sind.

Ziel dieses Kapitels wird es sein, einen Überblick über die Art und Weise zu erhalten, mit der die verschiedenen Teilbereiche einer Webanwendung testgetrieben entwickelt werden können. Die ersten beiden Abschnitte richten sich an zwei der Grundbausteine einer <sup>†</sup>MVC-Webanwendung (vgl. Abschnitt 4.3.1), die Modell- und Controllertests. Im dritten Abschnitt sehen wir, wie Test Doubles verwendet werden können, um Zugriffe auf externe Datenlieferanten zu simulieren. Danach betrachten wir die Anwendung aus Anwendersicht und widmen wir uns der Implementierung von Akzeptanz- und Systemtests (vgl. Abschnitt 2.5). Zum Schluss gibt es einen Ausblick auf das Testen von JavaScript-Ereignissen.

### 7.1. Implementierung von Unit-Tests (Modelltests)

Ein Rails-Modell, wie in 4.3.1 auf S. 33 beschrieben, repräsentiert die Daten der Anwendung und die Regeln, wie diese zu verändern sind. Bei Rails werden sie hauptsächlich dazu verwendet, um mit der zugrundeliegenden Datenbanktabelle zu interagieren. Per Konvention von Rails findet hier die Hauptarbeit, also die Business-Logik, statt.

Fast jeder Unittest bei Rails beinhaltet das Testen auf Validierungskriterien seines korrespondierenden Modells, d.h. wann eine Instanz dieses Modells gültig ist und damit gespeichert werden darf (man denke z.B. an Pflichtfelder für ein Modell „Nutzer“, oder die Validierung des Formates seiner E-Mail-Adresse).

Diese Validierungen werden durch das <sup>↑</sup>ORM-Framework ActiveRecord, welches Rails standardmäßig nutzt, bereitgestellt.

Job
+title: String +description: Text +link: String +is_visible: bool +user_id: ref +company_site_id: ref -start_date -end_date +...

Abbildung 7.1.: Attribute des Modells „Job“

**1. Der Anfang** Während der Analyse wurden die benötigten Attribute eines Jobs bestimmt. In Abbildung 7.1 ist ein Fragment des Grobdesigns dargestellt, welches die Basisattribute der Jobs-Tabelle aufzeigt. Neben den einfachen Attributen, wie Title, Description und Link existieren auch Referenzen auf andere Objekte (d.h. dies stellen Fremdschlüssel zu anderen Tabellen dar), wie z.B. Schlagwörter (Tags), ein Besitzer einer Stellenanzeige (User) und so weiter.

Einer der häufigsten Wege, ein Modell und dessen Datenbankschema zu generieren, ist die Nutzung des mitgelieferten Codegenerators. Mittels des Kommandos:

```
rails generate model MODELNAME spalte1:datentyp1 spalte2:datentyp2 ...
```

generieren wir ein Modell mit dem angegebenen Modellnamen. Dazu geben wir paarweise die gewünschten Spaltennamen und deren Datentypen an (string, text, datetime, references, integer, boolean, decimal, ...).

```
~/it-jobs$ rails generate model job title:string link:string \  
description:text user:references visible:boolean  
  
invoke active_record  
create db/migrate/20110828160636_create_jobs.rb  
create app/models/job.rb  
invoke test_unit  
create test/unit/job_test.rb  
create test/fixtures/jobs.yml
```

Mit der Anweisung uns ein Modell `job` mit den nachfolgenden Attributen zu generieren, hat Rails uns nun schon ein Stück Arbeit abgenommen. Dabei delegiert der Codegenerator `model` nun die Arbeit an den Codegenerator für ein ActiveRecord-Modell (erkennbar an

dem `invoke active_record`). Dieser wiederum generiert 2 Dateien und ruft seinerseits einen Codegenerator zum Generieren der Tests auf (`invoke test_unit`)

Es wurden erstellt:

- Eine **Migration** (`db/migrate/2011xxxxxx\_create\_jobs.rb`). Dies stellt eine datenbankunabhängige Repräsentation einer Änderung an der Struktur unserer Datenbank dar. In diesem ist es die Erstellung einer Tabelle `jobs` (beachte: Plural!), mit den Spalten `Titel`, `Link` als String, `Description` als Textfeld, eine `User_ID` als Referenz auf ein anderes Modell und die Sichtbarkeit der Stelle (`visible`),
- Die **Modelklasse** (`app/models/job.rb`). Trotz unserer Definition der Spalten und deren Typen über die Kommandozeile, ist diese Klasse leer. Da wir ActiveRecord verwenden, definieren wir die Attribute, die unser Modell hat, nicht in der Modellklasse, sondern ausschließ in der Datenbank. Die Migration erspart uns die manuelle Arbeit, selbst in unserer Datenbank Spalten anzulegen. Bei Initialisierung eines Modells lädt ActiveRecord die Spalteninformationen aus der Datenbank und generiert dafür Getter- und Setter-Methoden (vgl. auch Abschnitt 4.3.1),
- die dazugehörige **Testklasse** (`app/unit/job\_test.rb`) von Test/Unit (vgl. Abschnitt 4.4.1)
- und **Fixtures**-Datei (`test/fixtures/jobs.yml`), zur Definition von Testdaten (vgl. ebenfalls Abschnitt 4.4.1).

Die Migration liegt nun zwar vor, aber es existiert noch keine Datenbank und demnach auch noch keine Tabelle mit dem Namen `jobs`. Wir wollen Rails nun mitteilen, dass es die Migration anwenden soll, um damit die Datenbank und Tabelle zu erstellen

Rails stellt uns mittels des Kommandozeilenwerkzeugs <sup>↑</sup>**Rake** eine Schnittstelle zu unserer Anwendung bereit, mit der wir meist Wartungsaufgabe ausführen können. Rake erwartet die Angabe eines Tasks und optional die Angabe einer Umgebungsvariable für die Rails-Umgebung, in der der Test ausgeführt werden soll<sup>1</sup>.

```
rake TASK [RAILS_ENV=production]
```

Dazu weisen wir nun Rails an, alle offenen Migrationen auszuführen. Standardmäßig erstellt Rails dann selbstständig eine SQLite Datenbank unter `db/development.sqlite3`.

Listing 7.1: Shell

```
$ rake db:migrate

== CreateJobs: migrating =====
-- create_table(:jobs)
```

<sup>1</sup>Jede Umgebung verfügt normalerweise über eine eigene Datenbank. standardmäßig befinden wir uns in der „Development“-Umgebung

```
-> 0.0020s
== CreateJobs: migrated (0.0021s) =====
```

Danach können wir die Rails-Test-Suite mithilfe eines weiteren Rake-Tasks auch schon ausführen. Dieser Rake-Task legt selbständig eine Testdatenbank an, führt offene Migrationen darauf aus und führt dann alle vorhandenen Tests aus.

Listing 7.2: Shell

```
$ rake test
(in /home/zealot64/TEST)
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.8.7/lib/rake/rake_test_loader
Started
.
Finished in 0.043818 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Es wurde also schon ein Testfall erfolgreich ausgeführt, nämlich ein Dummytestfall von Rails:

```
test/units/job_test.rb
1 #test/unit/job_test.rb
2 require 'test_helper'
3
4 class JobTest < ActiveSupport::TestCase
5   # Replace this with your real tests.
6   test "the truth" do
7     assert true
8   end
9 end
```

Listing: Standardtest generiert durch Rails

**2. Testen auf Validierung** Ein Feature von Rails umfassen die sogenannten Validierungen. Diese stellen sicher, dass eine Instanz eines Modells nur dann gespeichert wird, wenn es gewissen Kriterien entspricht. Viele der Validations sind vergleichbar mit den Datenbank-Constraints einiger Datenbanken. Rails nutzt diese standardmäßig nicht, da es auch andere Persistenzsysteme unterstützt, wie z.B. Key-Value-Store oder sogenannte NoSQL Datenbanken. So stellt Rails die Konsistenz und referenzielle Integrität innerhalb der Applikationsschicht sicher.

Nun möchten wir sicherstellen, dass eine Stellenanzeige nur dann gespeichert wird, wenn sie einen Titel beinhaltet. Der Test dazu würde wie folgt lauten:

```
test/units/job_test.rb
1 require 'test_helper'
2
3 class JobTest < ActiveSupport::TestCase
4   test "ein Job muss einen Titel haben" do
```

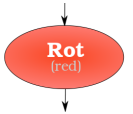


```

5  job = Job.new
6  job.title = nil
7  assert !job.save
8  end
9  end

```

Listing: Test auf Vorhandensein eines Titels



Zuerst instanziiert man einen Job und gibt ihm explizit einen leeren Titel, um das Testziel nochmal herauszustellen. Danach ruft man die `save`-Methode auf, die prüft, ob alle Validierungskriterien erfüllt sind und speichert das Objekt persistent in der Datenbank im Erfolgsfall. Dann gibt `save` ein `true` zurück, andernfalls, d.h. wenn die Validierung fehlschlug, `false`. Der Ablauf ist in der Abbildung 7.2 noch einmal erläutert.

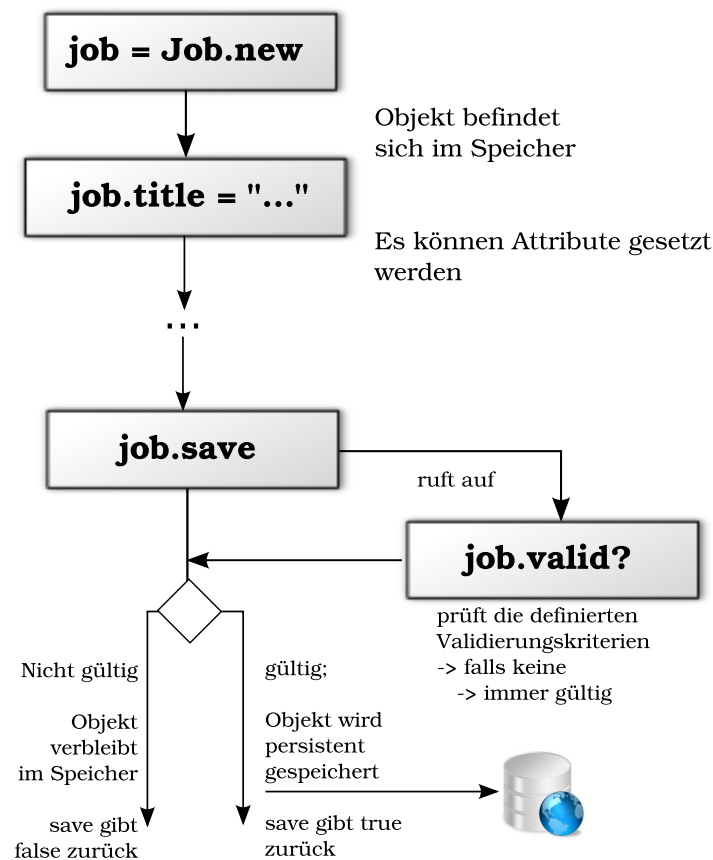


Abbildung 7.2.: Funktionsweise von save bei ActiveRecord Objekten

Da wir noch keine Validierungskriterien implementiert haben, gibt `save` `true` zurück und die Zusicherung schlägt fehl.

Unser nächstes Ziel ist es nun, mit so wenig Code wie möglich den Test bestehen zu lassen. Das können wir mittels der eingebauten wie schon erwähnten Validierungen:

```

app/models/job.rb
1 class Job < ActiveRecord::Base
2   validates :title, :presence => true
3 end

```

Listing: Implementierung der Validierung in die Klasse Job

`validates` ist eine Funktion aus der ActiveRecord Bibliothek, die zwei Parameter entgegennimmt: Der erste ist die Spalte, auf der sich die Validierung bezieht, als Zweites folgt eine Liste an Validierungskriterien. Hier ist das Kriterium `presence`, also das Vorhandensein eines nicht-leeren Wertes für diese Spalte. Weitere Kriterien sind z.B. Format, Länge, Minimum, Maximum oder auch selbst definierte Kriterien.

Nach erneuter Ausführung der Testsuite besteht der Test nun. Jetzt folgt die Refaktorisierungsphase. Der Programmcode lässt sich nicht weiter vereinfachen. Aber der Testcode ist ausdrücklich nicht von Refaktorisierungen befreit und eine Refaktorisierung wäre z.B.:

```
test/unit/job_test.rb
1 test "ein Job muss einen Titel haben" do
2   job = Job.new :title => nil
3   assert !job.save
4 end
```

Listing: refaktorisierter Test

Nun wollen wir dasselbe für das Feld E-Mail tun, hierbei aber nicht nur das Vorhandensein prüfen, sondern auch das Format.

```
test/unit/job_test.rb
1 test "ein Job muss eine gültige E-Mail haben" do
2   job = Job.new :email => "invalid_email"
3   assert !job.save
4 end
```

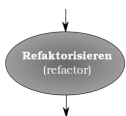
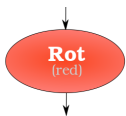
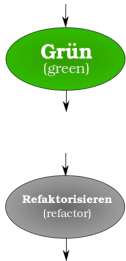
Die Implementierung wäre dann:

```
app/models/job.rb
1 class Job < ActiveRecord::Base
2   validates :email, :format => /^[^\w\d_-]+@[^\w\d_-]\. [\w\d]{2,3}$/
3   ...
4 end
```

Eine Refaktorisierung ist aufgrund der Einfachheit des Beispiels hier nur gering möglich. Man könnte z.B. den regulären Ausdruck, der das Format der E-Mail Adresse beschreibt in eine neue Klasse oder zumindest eine Konstante auslagern. Wir wählen eine Konstante, die beim Laden von Rails bereitgestellt wird.

```
config/initializers/regex.rb und app/models/job.rb
1 # config/initializers/regex.rb
2 REGEX_EMAIL_FORMAT = /^[^\w\d_-]+@[^\w\d_-]\. [\w\d]{2,3}$/
3
4 # app/models/job.rb
5 class Job < ActiveRecord::Base
6   validates :email, :format => REGEX_EMAIL_FORMAT
7   ...
8 end
```

Listing: Auslagerung des Regulären Ausdrucks in einen Initialisierer



Ein erneutes Ausführen der Tests betätigt den Erfolg der Refaktorisierung.

**3. Refaktorisierungen der Testklasse** Nun fehlt aber noch die Definition eines Positiv-Beispiel für einen gültigen Job.

```
test/unit/job_test.rb
1 ...
2 test "ein vollstaendiger Job muss gueltig sein" do
3   job = Job.new :title => "Rails Entwickler", :email => "stefan@mail.com"
4   assert_valid job
5 end
```

Grün  
(green)

Dieser Test besteht sofort, macht also genau genommen keine weitere Aussage über unser System. Nach der „reinen“ Testgetriebenen Lehre sollte dieser entfernt werden. Es ist allerdings eine gute Strategie, bei Validierungen mindestens ein Beispiel zu präsentieren, das den Kriterien entspricht und angenommen wird. Nichtsdestotrotz können wir nun refaktorisieren. Insbesondere unsere Testfunktionen enthalten unnötige Redundanzen:

```
test/unit/job_test.rb
1 test "ein Job muss einen Titel haben" do
2   job = Job.new :title => nil
3   assert !job.save
4 end
5 test "ein Job muss eine gültige E-Mail haben" do
6   job = Job.new :email => "invalid_email"
7   assert !job.save
8 end
9 test "ein vollstaendiger Job muss gueltig seinn" do
10  job = Job.new :title => "Rails Entwickler", :email => "stefan@mail.com"
11  assert_valid job
12 end
```

Refaktorisieren  
(refactor)

Listing: Alle bisherigen Testmethoden in der Klasse JobTest

In allen drei Methoden wird ein Job instanziiert und lediglich verschiedene Attribute überprüft. Auch haben unsere ersten beiden Tests keine gültige Aussage mehr, da der jeweilige Job sowieso nicht gültig ist, da jeweils das andere Attribut fehlt<sup>2</sup>. Es ist also höchste Zeit, die Tests zu refaktorisieren. Dies geschieht am besten durch die Verwendung einer Testdaten-Generation, z.B. den eingebauten Fixtures, die Rails uns bei der Codegeneration schon mit generiert hatte. Dabei definieren wir zentralisiert unsere (gültigen) Testdaten, die von Rails vor jedem einzelnen Test in der Datenbank bereitgestellt werden (Mehr zu Fixtures in Abschnitt 4.4.1):

```
test/fixtures/jobs.yml
1 valid_job:
2   title: Rails Entwickler
3   email: stefan@mail.com
```

<sup>2</sup>Im ersten Test ist nicht nur der Titel nicht gesetzt, sondern auch die E-Mail entspricht nicht dem Format

```
4 link: "http://www.example.com/jobs"
5 visible: true
6 ...
7 invisible_job:
8   title: Rails Entwickler
9   visible: false
10 ...
```

Listing: Fixtures Testdaten für zwei Jobs

Nun können wir diese Fixtures in unseren Tests verwenden und das ganze in einer `setup`-Methode, die vor jedem Testfall aufgerufen wird, laden:

```
1 class JobTest < ActiveSupport::TestCase
2   setup do
3     @job = jobs :valid_job
4     # Dies lädt den Job mit dem Schlüssel "valid_job" und schreibt ihn
5     # in die Instanzvariable @job der Testklasse
6   end
7   test "stelle sicher, dass die Fixtures valide sind" do
8     assert_valid @job
9   end
10  test "ein Job muss einen Titel haben" do
11    @job.title = nil
12    assert !@job.save
13  end
14  test "ein Job muss eine gültige E-Mail haben" do
15    @job.email = "invalid_email"
16    assert !@job.save
17  end
18 end
```

Listing: Finale Job-Test Klasse nach Refaktorisierung

Am Ende dieser Refaktorisierungen ist es notwendig, die Tests noch einmal auszuführen. Danach würde die Implementierung einer nächsten Teilanforderung sein.

In diesem Abschnitt war zu sehen, dass die Testgetriebene Entwicklung das Arbeiten und Testen in kleinen Schritten favorisiert.

## 7.2. Implementierung von Controller-Tests (functional tests)

Skinny Controller, Fat Model [...] Try to keep your controller actions and views as slim as possible.

Jamie Buck, Programmierer bei 37signals

Neben den Unittests stellt Ruby on Rails eine weitere Testart nativ bereit. Technisch gesehen handelt es sich bei diesen *functional tests* aber auch um Unittests, da deren Testobjekt eine Klasse, der Controller, ist. Ein Controller, wie in Abschnitt 4.3.1 vorgestellt, hat bei Ruby on Rails die Aufgabe, Anfragen für bestimmte Routen, also Web-Adressen, anzunehmen, die Arbeit an eine Modelklasse auszulagern und eine View aufzurufen, die letztendlich HTML-Code generiert.

Im ersten Beispiel wollen wir testen, dass ein Gast-Nutzer, also z.B. ein Bewerber, eine sichtbare Stellenanzeige aufrufen darf (`visible = true`). Hierbei verwenden wir wieder unser oben definiertes Fixture für einen gültigen Job.

```
test/functional/jobs_controller_test.rb
1 require 'test_helper'
2
3 class JobsControllerTest < ActionController::TestCase
4   test "Gast Nutzer kann Stellen betrachten" do
5     session[:user_id] = nil
6     job = jobs(:valid_job)
7
8     get :show, :id => job.id
9
10    assert_response :success
11    assert_equal job, assigns(:job)
12  end
13 end
```

Listing: JobsController-Test: Gast Nutzer darf Stellen betrachten

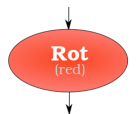
Zuerst loggen wir jeglichen Nutzer aus, der eventuell eingeloggt war, dann laden wir das Fixture und führen einen simulierten HTTP-Request auf die Detailansicht der Stellenanzeige `http://localhost/jobs/12` aus (Die Aktion `show` mit der ID des Jobs). Nun erwarten wir, dass wir einen HTTP-Status-Code 200 (Erfolg) erhalten (Zeile 10) und dass der Controller eine Variable `@jobs` bereitstellt, die mit unserem Fixture identisch ist (Zeile 11).

Die Implementation dieser Anforderung könnte wie folgt umgesetzt werden:

```
app/controllers/jobs_controller.rb
1 class JobsController < ApplicationController
2   ...
3   def show
4     @job = Job.first
5   end
6   ...
7 end
```

Listing: JobsController: Erfüllung des Tests durch eine Fake-Implementierung

Das Laden des ersten Jobs aus unserer Datenbank, anstelle des in der URL angegebenen, genügt zum Erfüllen der Anforderungen und ist ein schneller Weg, den Test bestehen zu lassen. Allerdings handelt es sich hierbei um eine **Fake-Implementierung**, da zwar unser Test erfüllt wird, aber die Anwendung nicht das macht, was man sich erhofft hat. Solche



Zwischenschritte sind aber ausdrücklich vorgesehen, da das Ziel ist, so schnell wie möglich einen funktionierenden Test zu erhalten mit dem man arbeiten kann.

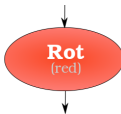
Wenn wir nun weitere Tests schreiben, so wird es immer schwieriger, die Fake-Implementierung beizubehalten und früher oder später wird eine korrekte Implementierung folgen. Aber wir können auch die nun folgende Refaktorisierungsphase nutzen, um diesen Makel zu beseitigen:



```
1 def show
2   @job = Job.find(params[:id])
3 end
```

Listing: JobsController: Ersetzung der Fake-Implementierung


Nun wollen wir testen, ob ein Gast von einer nicht-sichtbaren Stellenanzeige weitergeleitet wird und einen Hinweis erhält.



```
1 test "Gast Nutzer kann nicht-sichtbare Stellen nicht betrachten" do
2   session[:user_id] = nil
3   job = jobs(:invisible_job)
4
5   get :show, :id => job.id
6
7   assert_response :redirect
8   assert flash[:notice].present?
9 end
```


Listing: JobsController-Test: Gast Nutzer dürfen nicht-sichtbare Stellen nicht betrachten

Wir laden unser zweites definiertes Fixture, dass eine unsichtbaren Stellenanzeige. Nun erwarten wir einen HTTP-Status-Code 301 (Redirect/Weiterleitung) und dass unser Controller eine Hinweismessage generiert. Uns reicht hier nur das Vorhandensein einer solchen Nachricht, nicht aber deren Inhalt zu prüfen. Wichtig für wartbare Tests ist es, dass diese flexibel gegenüber wahrscheinlichen Änderungen sind, ansonsten besteht die Gefahr, dass die Tests zu „brüchig“ (*brittle*) werden.



```
1 def show
2   @job = Job.find(params[:id])
3   if not @job.visible?
4     redirect_to root_path, :notice => "Diese Stelle ist nicht sichtbar"
5   end
6 end
```

Listing: JobsController: Weiterleitung, falls ein Job nicht sichtbar ist



Falls der aktuelle Job nicht sichtbar ist, dann erfolgt eine Weiterleitung auf die Startseite und die Bereitstellung des Hinweistextes. Da auch hier der Quelltext wieder sehr kurz ist, ist ein Refaktorisieren nicht notwendig.

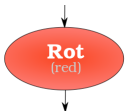
Nun möchten wir, dass ein Kunde dieser Anwendung, also ein Arbeitgeber seine Stellenanzeige betrachten kann, auch wenn diese unsichtbar ist, sei es aus Gründen der Archivierung als auch der Vorbereitung für eine Veröffentlichung.

```

test/functional/job_controller_test.rb
1 test "Ein Kunde darf aber seine unsichtbaren Jobs betrachten" do
2   job = jobs(:invisible_job)
3   session[:user_id] = job.user_id
4
5   get :show, :id => job.id
6
7   assert_response :success
8 end

```

Listing: JobController-Test: Neuer Testfall



Über die globale Session Variable simulieren wir das Einloggen durch Setzen der User-ID in dieses Array. Die genaue Implementation hängt natürlich davon ab, wie man die Authentifizierung implementiert hat oder welche Bibliothek man verwendet. In diesem Beispiel sei darauf hingewiesen, dass die Definition, ob ein Nutzer eingeloggt ist oder nicht, davon abhängt, ob in seiner Session-Variable eine User-ID enthalten ist.

```

app/controllers/job_controller.rb
1 def show
2   @job = Job.find(params[:id])
3   if !@job.visible? and @job.user != User.find(session[:user_id])
4     redirect_to root_path, :notice => "Diese Stelle ist nicht sichtbar"
5   end
6 end

```

Listing: JobsController: Implementierung der Weiterleitung, falls Stelle unsichtbar



Wir lösen diesen Test damit, dass wir in der Weiterleitungsbedingung prüfen, ob der betrachtende Nutzer und der Eigentümer des Jobs gleich sind.

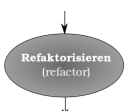
Nun können wir refaktorisieren. Was auffällt, ist z.B. dass unser Controller und die Klasse Job nicht lose gekoppelt sind, da die Bedingung zweimal auf Attribute des Jobs zurückgreift. Eine Lösung wäre die Auslagerung in die Modelklasse von Job:

```

app/models/job.rb
1 class Job < ActiveRecord::Base
2   ...
3   def visible_for_user?(user)
4     self.visible or self.user == user
5   end
6 end

```

Listing: Job: Einführung einer Methode zur Bestimmung der Sichtbarkeit eines Jobs im Job-Modell



```

app/controller/jobs_controller.rb
1 def show
2   @job = Job.find(params[:id])
3   unless @job.visible_for_user?(User.find(session[:user_id]))

```

```

4   redirect_to root_path, :notice => "Diese Stelle ist nicht sichtbar"
5   end
6 end

```

Listing: JobsController: Anwendung der Refaktorisierung

Ebenfalls wurde das syntaktische Element `unless` verwendet, welches ein Alias für `if not` ist. Weiterhin könnte die Suche nach dem aktuell eingeloggten Nutzer in eine für alle Controller sichtbare Funktion ausgegliedert werden:

app/controllers/application\_controller.rb

```

1 ...
2 def current_user
3   User.find(session[:user_id])
4 end

```

Listing: ApplicationController: Einführung einer controller-globalen Methode `current_user`

app/controllers/jobs\_controller.rb

```

1
2 # app/controllers/jobs_controller.rb
3 def show
4   @job = Job.find(params[:id])
5   unless @job.visible_for_user? current_user
6     redirect_to root_path, :notice => "Diese Stelle ist nicht sichtbar"
7   end
8 end

```

Listing: JobsController: Nutzung der neuen Methode `current_user`

Nun haben wir einen Großteil der Logik aus dem Controller in andere Klassen ausgegliedert und haben damit einen übersichtlichen Controller erhalten.

**Zusammenfassung** Funktionale Tests und deren Controller-Implementierungen sind häufig nicht länger als ein paar Zeilen. Qua Konvention des <sup>↑</sup>MVC-Patterns und von Rails sollen komplexe Abläufe in den Modellen oder auch in Bibliotheken stattfinden. Die Aspekte, die üblicherweise bei Controllern getestet werden, sind:

- HTTP-Status-Codes und Weiterleitungen,
- das Vorhandensein von Statusmeldungen, genannt „Flash“-Messages
- dass eine bestimmte View geladen und gerendert wird
- dass Instanzvariablen gesetzt werden, die die View später darstellen sollen
- falls man Viewtests mit einschließt, dann wird u.U. auch auf das Vorhandensein von bestimmten HTML-Elementen in der am Ende generierten View getestet. Z.B. möchte man wissen, ob das Überschriftenelement `h1` dem Job-Titel entspricht, wenn die Detailansicht eines Jobs aufgerufen wird. Hierbei besteht allerdings die Gefahr, dass



die Tests brüchig (brittle) werden, da sich Views normalerweise häufiger angepasst werden, weshalb man diese mit Bedacht testen sollte.

Intern nutzen Controller-Tests viele Stubs, um HTTP-Anfragen und Antworten durch eigene Test-Klassen zu ersetzen. Sie sind damit sehr schnell, testen aber nicht alle Aspekte einer HTTP-Anfrage an die Anwendung (z.B. Cookie oder Routing). Auch ist das Testen von mehreren Controllern, um z.B. einen Ablauf wie Einloggen → Bestellen nachzubilden, in einem Functional-Test nicht vorgesehen.

## 7.3. Testen von externen Abhängigkeiten

Fast alle Webapplikationen sind auf Kommunikation mit anderen Servern angewiesen. Als Beispiel seien die diversen APIs der sozialen Netzwerke genannt oder Webservices. Für die vorliegende Jobanwendung war gewünscht, ein Feedimport-Feature zu implementieren, sodass bestimmte Kunden ihre Stellenanzeigen automatisiert einlesen lassen könnten.

Die genannten Partner stellen einen XML-Feed nach dem <sup>†</sup>RSS 2.0 Format<sup>3</sup> bereit, der ein häufig verwendetes Format zum Austausch von Informationen ist und durch eine Vielzahl von Werkzeugen und <sup>†</sup>Content Management Systemen unterstützt wird. Dabei wird der Inhalt des Haupttextfeldes <description> um weitere Informationen in einem Subdialekt angereichert.

Im Nachfolgenden sei z.B. eine Stellenanzeige in dem Format beschrieben:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <rss version="2.0">
3    <channel>
4      <title>RSS Feed für Jobangebote </title>
5      <language>de</language>
6      <item>
7        <title>Softwareentwickler Java/JEE (m/w) </title>
8        <description>
9          <![CDATA[
10             <!--
11             <nummer>example_job_01</nummer>
12             <tags>Java,Webentwickler,Softwareentwickler</tags>
13             <ort>Dresden</ort>
14             <kontakt>Max Mustermann bewerbung@example.com</kontakt>
15             <link>http://www.example.com/jobs/512.html</link>
16             -->
17             Zur Verstärkung unseres Teams suchen wir zum nächstmöglichen
18             Zeitpunkt einen Softwareentwickler Java/JEE (m/w) zur Festanstellung.<br />
19             Ihre Aufgaben: ...
20             ]]>
21        </description>
22        <link>http://www.example.com/jobs/512.html</link>
23        <pubDate>Wed, 25 Mar 2011 13:30:00 +0100</pubDate>
24        <guid>example_job_01</guid>

```

<sup>3</sup>Spezifikation des RSS 2.0 Formats: <http://cyber.law.harvard.edu/rss/rss.html>

```
25     </item>
26   </channel>
27 </rss>
```

Listing: Feedimport Beispiel-XML Datei mit einem Job

Der RSS-Feed in dem oben genannten Beispiel enthält eine Stellenanzeige (`<item>`). Die `description` beinhaltet einen HTML-Kommentar, in dem `nummer`, `tags`, `ort`, `kontakt` und `link` für die Stellenanzeige definiert werden. Das ganze wurde mit einem Kommentar und nicht mit einer Erweiterung der Syntax durch eine DDT oder XSD, realisiert, da sich eine Eingliederung der Syntaxelemente mittels DDTs und XSDs in einige der Systeme der Kunden als problematisch herausgestellt hat. Da alle Spezialtags innerhalb eines HTML-Kommentars und damit bei normaler Betrachtung des Feeds nicht sichtbar sind, kann dieser Feed von dem Arbeitgeber gleichzeitig seinen Bewerbern angeboten werden, um sich über neue Jobs dieses Unternehmens auf dem Laufenden zu bleiben.

Diese Art des Feedimports ist bereits in den Community-Job-Portalen in Funktion. Allerdings besitzt dieser, in PHP geschriebene Code, keinerlei automatisierte Tests und war in der Vergangenheit schon oft die Ursache von Fehlern. So ist es notwendig, den Feedimport nun in Ruby als Bibliothek im Rahmen von IT-Jobs neu zu schreiben und für die bereits laufenden Portale schnellstmöglich einzubauen. Diese Bibliothek soll also unabhängig von Rails funktionieren.

Ziel diesen Abschnittes ist es, zu zeigen, wie das Einlesen eines externen XML-Feeds getestet werden kann.

**1. Initialer Test** Bevor wir anfangen zu implementieren, ist es sinnvoll sich Gedanken darum zu machen, was von den zu implementierenden Objekten erwartet wird. Da wir letztendlich eine gewisse Menge von RSS-Feeds einlesen wollen, ist es angebracht, ein entsprechendes Objekt, z.B. „ImportedFeed“ einzuführen.

Auch wenn wir noch nicht genau wissen, wie ein Feed funktioniert, so können wir doch zumindest annehmen, dass ein HTTP-Zugriff auf eine URL erfolgt, um den Feed vom Kunden abzuholen.

Da wir unsere Tests nicht davon abhängig machen wollen, ob ein solcher Feed bereitsteht und sich stets im selben Zustand befindet, müssen wir diesen HTTP-Zugriff simulieren.

```
test/test_imported_feed.rb
1 require "test_helper" # Hilfsfunktionen und eigene Assertions
2 require "imported_feed" # Object under Test
3
4 class TestImportedFeed < ActiveSupport::TestCase
5
6   test "get an feed through httparty" do
7     HTTParty.expects(:get).with("http://www.example.com/feed.xml")
```

```

8   ImportedFeed.new("http://www.example.com/feed.xml")
9   end

```

Listing: Feed Test I.

Hier definieren wir einen ersten Test für den `ImportedFeed`. Für die HTTP-Zugriffe wollen wir die Bibliothek `HTTParty`<sup>4</sup> benutzen. In der ersten Zeile des Tests nutzen wir das Mock-Framework wie folgt: Wir legen eine Erwartung fest, dass innerhalb dieses Tests die Klassenmethode `get` der Klasse `HTTParty` aufgerufen wird, mit einem Parameter der die URL angibt.

Dann rufen wir unsere (noch nicht existente) `ImportedFeed`-Klasse mit dem einzulesenden Feed.

Nach der Ausführung des Tests erhalten wir einen Fehler.

```
NameError: uninitialized constant TestImportedFeed::ImportedFeed
```

In der reinen TDD-Lehre würde nun als erstes eine Behebung aller Fehler stattfinden, d.h. eine Implementierung der leeren Klasse `ImportedFeed`. Danach würden wir noch einen Fehler erhalten, da unser Konstrukt noch keinen Parameter entgegennimmt. Erst dann würde man sich den Testfehlschlägen widmen. Aus Platzgründen sind diese Schritte hier nicht explizit ausgeführt. Der Fehlschlag lautet dann:

```

Failure
  not all expectations were satisfied
  unsatisfied expectations:
    - expected exactly once, not yet invoked: HTTParty.get('http://www.example.com/feed.xml')

```

Das Mockobjekt hat unseren Test fehlschlagen lassen, ohne dass wir selbst eine Assertion festgelegt haben. Da wir bisher noch keine Implementation eines Netzwerkzugriffes durch `HTTParty` implementiert haben, schlägt der Test fehl.

Die Implementierung wäre:

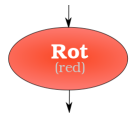
```

lib/imported_feed.rb
1  require "httparty"
2
3  class ImportedFeed
4    def initialize(url)
5      HTTParty.get(url)
6    end
7  end

```

Listing: Feed Implementation I.

<sup>4</sup><https://github.com/jnunemaker/httparty>



Die Funktion **initialize** stellt innerhalb von Ruby den Konstruktor dar. Dort führen wir unseren Netzwerkzugriff durch, der allerdings durch unser definiertes Mock-Objekt abgefangen wird. Dies stellt die definierte Erwartung zufrieden und der Test besteht.

**2. Komplexe Objekte durch Mocks zurückgeben** Wir haben zwar den Netzwerkzugriff abgefangen, geben aber nun keinerlei Antwort, d.h. ein XML-Dokument zurück. Für den nächsten Test müssen wir unsere Mockanweisung also modifizieren.

```
test/test_imported_feed.rb
1 test "really get content from a feed" do
2   fake_response = OpenStruct.new
3   fake_response.code = 200 # HTTP OK!
4   fake_response.body = "<?xml version='1.0'?><Hallo/>"
5
6   HTTParty.expects(:get).with("http://www.example.com/feed.xml") .
7     returns(fake_response)
8
9   import = ImportedFeed.new(@url)
10  assert_match "Hallo", import.body
11 end
```

Listing: Feed Test II

Wir bilden das Antwortobjekt, das HTTParty normalerweise generieren würde, beschränken uns hierbei aber nur auf die für uns notwendigen Attribute von „body“ und „code“ (Dem HTTP-Status Code). Da Ruby Duck-Typed ist, reicht es, wenn ein Objekt die benötigten Methoden und Attribute implementiert, es muss nicht von einem gewissen Interface erben um Semantiken zu erhalten. Darum nutzen wir die Klasse OpenStruct, die sehr einfach Getter und Setter für das Objekt beim Benutzen erstellt. Unserem Mock können wir dann anweisen, diese Antwort zurückzugeben.

Bei Ausführung des Tests stellen wir fest, dass zwar die Erwartung erfüllt wurde, aber unser ImportedFeed noch kein Attribut „body“ besitzt (Fehler) und dass dieser keine String beinhaltet.

```
app/models/job.rb
1 class ImportedFeed
2   attr_reader :body
3   def initialize(url)
4     response = HTTParty.get(url)
5     @body = response.body
6   end
7 end
```

Listing: Feed Implementation II

Mithilfe des Makros **attr\_reader** generieren wir ein Attribut body und gleichzeitig einen Getter für den Zugriff von außen. Innerhalb unseres Konstruktors speichern wir den Body der HTTP-Antwort in diesem Attribut.

Da unser Testfall nun ziemlich lang geworden ist und beide Testfälle ein Mock initialisieren, ist dies eine gute Gelegenheit, den Mock zentral zu definieren. Dazu nutzen wir z.B. eine Datei `test_helper.rb`, in der wir Anweisungen schreiben, die alle Testfälle nutzen können:



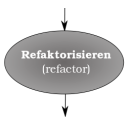
```

test/test_helper.rb
1 class ActiveSupport::TestCase
2   def mock_feed(opts={})
3     options = {
4       :url => "http://example.com/feed.xml",
5       :code => 200,
6       :body => '<?xml version="1.0" encoding="UTF-8"?><Hallo>Hallo</Hallo>'
7     }.merge(opts)
8     response = OpenStruct.new
9     response.code = options[:code]
10    response.body = options[:body]
11    HTTParty.expects(:get).with(options[:url]).returns(response)
12  end
13 end

```

Listing: Zentrale Implementierung des Mocks in der Test Helper

Da Ruby offene Klassen unterstützt öffnen wir die Basisklasse der Testfälle und definieren eine neue Methode. Diese erhält einen Hash als Parameter, den wir mit unseren Standardwerten zusammenführen (merge). Diese Art der Parameterübergabe ist ein sehr gebräuchliches Idiom innerhalb der Ruby-Community.



Der Aufruf unsere neuen Hilfsfunktion erfolgt dann mittels:

```

test/test_imported_feed.rb
1 def setup
2   @url = "http://example.com/feed.xml"
3 end
4
5 test "get an feed through httparty" do
6   mock_feed :url => @url
7   ImportedFeed.new(@url)
8 end
9
10 test "really get content from an feed" do
11   mock_feed :url => @url, :body => "Hallo"
12   import = ImportedFeed.new(@url)
13   assert_match "Hallo", import.body
14 end

```

Listing: Feed Test IIb nach Refaktorisierung

Als zusätzliche Maßnahmen haben wir die Definition der URL in eine gemeinsame Initialisierungsmethode gesetzt.

Rot  
(red)

**3. Validität** Unser Feed soll später feststellen können, ob er Fehler beinhaltet oder nicht, um dann ggf. eine E-Mail an den Verantwortlichen zu schreiben.

```
test/test_imported_job.rb
1 test "have a valid method" do
2   mock_feed
3   import = ImportedFeed.new(@url)
4   assert import.respond_to?(:valid?)
5 end
```

Listing: Feed Test III

Nun testen wir lediglich darauf, ob das ImportedFeed Objekt eine Methode oder ein Attribut mit dem Namen **valid?** besitzt.

Um den Test zu bestehen, reicht es, eine leere Methode zu definieren:

```
lib/imported_job.rb
1 ...
2 def valid?
3 end
```

Listing: Feed Implementation III - Fake Implementierung

Nun werden wir etwas konkreter und erwarten, dass falls der kontaktierte Server einen Status-Code 404 (Dokument nicht gefunden – Ein wahrscheinlicher Fehlerfall, falls das XML-Dokument verschoben wurde) erhalten.

```
test/test_imported_job.rb
1 test "not validate if the user server reports a problem" do
2   mock_feed :code => 404
3   feed = ImportedFeed.new(@url)
4   assert !feed.valid?
5 end
```

Listing: Feed Test IV – Test auf nicht-erfolgreichen HTTP-Status Code

Implementieren können wir das so:

```
lib/imported_job.rb
1 class ImportedFeed
2   attr_reader :body, :status_code
3   def initialize(url)
4     response = HTTParty.get(url)
5     @status_code = response.code
6     @body = response.body
7   end
8
9   def valid?
10    @status_code == 200,
11  end
12 end
```

Listing: Feed Implementation IV - Implementation der Validierung

Grün  
(green)

Rot  
(red)

Grün  
(green)

Zu bemerken ist, dass wir den Funktionsrückgabewert nicht explizit mit **return** kennzeichnen müssen. Bei Ruby hat jeder Ausdruck einen Rückgabewert. Innerhalb einer Funktion ist dies das letzte Statement, falls nicht mit **return** spezifiziert.

**4. Testen auf Exceptions** Zum Abschluss dieses Kapitels möchten wir noch sicher gehen, dass unser `ImportedFeed` robust gegenüber Exceptions von Fremdbibliotheken ist. Auch dies können wir in einer Erwartung durch unser Mock-Objekt spezifizieren.

```

test/test_imported_job.rb
1 test "should be resistant to any thrown errors from library" do
2   HTTParty.expects(:get).raises( ArgumentError)
3
4   feed = nil
5   assert_nothing_raised(ArgumentError) do
6     feed = ImportedFeed.new(@url)
7   end
8   assert !feed.valid?
9
10 end

```

Listing: Feed Test V - Testen der Robustheit gegen Exceptions

Durch die Zusicherung `assert_nothing_raised(exception, message, block)` testen wir, dass innerhalb des übergebenen Blocks keine Exception vom Typ `exception` (Hier: `ArgumentError`) geworfen wird. Wir möchten in diesem Fall auch sichergehen, dass unser Feed als nicht-valide markiert wird.

```

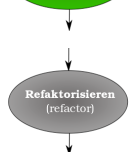
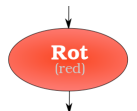
lib/imported_job.rb
1 class ImportedFeed
2   attr_reader :body, :status_code
3   def initialize(url)
4     response = HTTParty.get(url)
5     @status_code = response.code
6     @body = response.body
7     @error_thrown = false
8   rescue Exception => e
9     @error_thrown = true
10  end
11  def valid?
12    !@error_thrown and @status_code == 200
13  end
14 end

```

Listing: Feed Implementation V

Alle Exceptions die innerhalb des Konstruktors geworfen werden, werden abgefangen und das neue Attribut `@error_thrown` auf `true` gesetzt. Dies kann dann unsere `valid` Funktion verwenden.

Unser Konstruktor ist nun schon relativ lang geworden und hat inzwischen schon mehrere Aufgaben: Abruf eines Feeds und setzen der HTTP-Antwort. Jede unserer Methoden sollte



eine klar umrissene Aufgabe haben. Dies erreichen wir nun durch das Auslagern in eine neue (private) Funktion.

```
lib/imported_job.rb
1 class ImportedFeed
2   attr_reader :body, :status_code
3   def initialize(url)
4     get_feed(url)
5     @error_thrown = false
6   rescue Exception => e
7     @error_thrown = true
8   end
9   def valid?
10    !@error_thrown and @status_code == 200
11  end
12
13  private
14  def get_feed(url)
15    response = HTTParty.get(url)
16    @status_code = response.code
17    @body = response.body
18  end
19 end
```

Listing: Feed Implementation Vb - nach Refaktorisierung

Diese private Methode hat nun keine eigenen Tests, ist aber aus einer Refaktorisierung hervorgegangen und damit innerhalb von TDD ein erlaubter Schritt (Das scheinbare Dilemma des Testens privater Methoden wurde bereit auf S. 20 erläutert).

In diesem Abschnitt wurden einige Spezialfälle beim Umsetzen von TDD praktisch erläutert. Dies waren:

- Einsatz von Mocks zur Entkoppelung von externen Datenquellen und Spezifizierung von Erwartungen (Expectations)
- Nutzung von privaten Methoden bei der Refaktorisierung
- Tests mit Exceptions

## 7.4. System/Akzeptanztests

Nun wollen wir am Beispiel einer Suche nach Stellenanzeigen die Entwicklung von Akzeptanztests und damit die Akzeptanztestgetriebene Software-Entwicklung (vgl. Abschnitt 3.6.1) betrachten.

Zum Einsatz kommt dabei die für Systemtests entwickelte domainspezifische Sprache Cucumber in Verbindung mit der Browser-Steuerung Capybara, die beide bereits in Abschnitt 4.4.2 vorgestellt wurde.



Ein Cucumber-Verzeichnis ist qua Konvention immer gleich aufgebaut. In Abbildung 7.3 ist der Verzeichnisbaum, wie er in den Beispielen dieses Kapitels benutzt wird, zu sehen. Innerhalb unseres Projektverzeichnisses existiert ein Unterverzeichnis „features“ in dem alle Feature-Dateien mit der Endung „.feature“ lagern. Ebenfalls existiert ein Unterverzeichnis „step\_definitions“, das die Implementation der Testschritte beinhaltet sowie ein Verzeichnis „support“, für z.B. Standardmethoden, Aktionen die vor jedem Feature-Durchlauf ausgeführt werden oder die Definition der verwendeten Browser-Engine. Cucumber lädt automatisch alle Ruby-Dateien in den beiden Verzeichnissen.

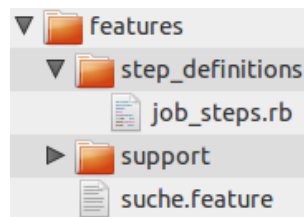


Abbildung 7.3.: Aufbau eines Cucumber- Verzeichnisses

**1. Definition des gesamten Akzeptanztest** Zusammen mit dem Kunden oder basierend auf den Anforderungen entwickeln wir zuerst eine Spezifikation für ein Feature. Da die Akzeptanztests in erster Linie dazu dienen, die Software gegenüber den Anforderungen des Kunden zu validieren und auch als Kommunikationsmittel genutzt wird, orientiert sich das Vokabular an gebräuchlichen Begriffen. So ist ein einzelner Testfall ein „Szenario“ und eine Test-Suite ein „Feature“. Statt „Assertions“ (Zusicherungen) gibt es Vor- und Nachbedingungen.

Nachfolgend sei ein erstes Szenario für eine Suche nach Stellenanzeigen gezeigt.

```

1  # language: de
2  Funktionalität: Job-Suche
3  Um Jobs zu finden
4  Als ein Gast
5  Soll es möglich sein mittels einer Suche Jobs zu finden
6  Szenario: Auffinden durch Titel
7  Angenommen wir befinden uns auf der Startseite
8  Und die folgenden Jobs sind vorhanden:
9      | title                | visible |
10     | Ruby on Rails Entwickler | true    |
11     | Java Programmierer       | true    |
12  Wenn wir "Rails" für "search" eintippen
13  Und wir auf den Button "Suchen" klicken
14  Dann sehen wir "Ruby on Rails Entwickler"

```

Listing: Definition eines Szenarios in einem Cucumber-Feature

Die ersten drei Zeilen des Features („Funktionalität“) beinhalten hier einen Kommentar, der lediglich die Testziele und Rahmenbedingungen definiert. Danach folgen die Testschritte.

Eine Besonderheit ist der zweite: „Und die folgenden Jobs sind vorhanden“. Dort ermöglicht es ein Syntaxelement von Cucumber über eine Tabelle mehrere Datensätze zu definieren. Alle anderen Elemente sollten aus dem Abschnitt 4.4.2 bereits bekannt sein.

Wenn wir dieses Feature nun ausführen, erhalten wir folgende Ausgabe:

```
$ cucumber features/suche.feature
1 scenario (1 undefined)
5 steps (5 undefined)

You can implement step definitions for undefined steps with these snippets

Angenommen /^wir befinden uns auf der Startseite$/ do
  pending # express the regexp above with the code you wish you had
end
Angenommen /^die folgenden Jobs sind vorhanden:$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end
Wenn /^wir "([^"]*)" für "([^"]*)" eintippen$/ do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end
Wenn /^wir auf den Button "([^"]*)" klicken$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
Dann /^sehen wir "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Bevor wir also beginnen können das Feature zu entwickeln, müssen wir erst die Testschritte implementieren. Cucumber hat schon ein paar Codeschnipsel generiert, mit denen wir eigene Testschritte implementieren können.

Hier zuerst die Basis-Testschritte, die für fast jedes Feature benötigt werden. Falls wir die englische Testschrittdefinition nutzen, entfällt dieser Schritt, da Capybara bereits häufig gebrauchte Testschritte mitliefert.

```
_____ features/step_definitions/job_steps.rb _____
1 Angenommen /^wir befinden uns auf der Startseite$/ do
2   visit "/"
3 end
4 Wenn /^wir "([^"]*)" für "([^"]*)" eintippen$/ do |text, input_name|
5   fill_in input_name, :with => text
6 end
7 Und /^wir auf den Button "([^"]*)" klicken$/ do |text|
8   click_button text
9 end
10
11 Dann /^sehen wir "([^"]*)"$/ do |string|
```

```

12   assert page.has_content?(string)
13 end

```

Listing: Implementierung der Testschritte zur Interaktion mit einem Browser

Der erste Testschritt gibt dem simulierten Browser die Anweisung, die Startseite zu besuchen. Der zweite Testschritt „Wenn wir <text> für <input\_name> eingeben“ besitzt zwei variable Texte, die an den Funktionsblock mit übergeben werden. Wir möchten, dass der erste Begriff in „...“ als Text in ein Formularelement mit dem Bezeichner des zweiten Begriffes eingetragen wird. Der dritte Schritt implementiert den Klick auf einen HTML-Button, der den angegebenen Text enthält.

Während die ersten 3 Schritte Aktionen ausführen, hat der 4. eine andere Funktion: Er spezifiziert eine Zusicherung, dass auf der aktuellen Webseite irgendwo ein gewisser Inhalt steht.

Ein weiterer Testschritt fehlt noch:

```

# features/step_definitions/job_steps.rb
1 # table.hashes ->
2 # [ {:title => "Ruby on Rails Entwickler", :visible => true},
3 #   {:title => "Java Programmierer", :visible => true}]
4 Angenommen /^die folgenden Jobs sind vorhanden:$/ do |table|
5   valid_job = jobs(:visible_job).attributes
6   table.hashes.each do |hash|
7     attributes = valid_job.merge(hash)
8     Job.create(attributes)
9   end
10 end

```

Listing: Testschritt zum Anlegen von Testdaten auf Basis der Cucumber Definition

Dieser Testschritt für das Anlegen der definierten Stellenanzeigen durch das Tabellen-Syntaxelement von Cucumber ist etwas umfangreicher. Wir bekommen von Cucumber ein Array von Hashes übergeben, die die im Szenario angegebene Tabelle beinhaltet. Über diese können wir nun mit dem Iterator `each` iterieren und die definierten Attribute mit denen unserer Job-Fixture<sup>5</sup> vereinigt wird, um somit einen neuen Job in der Testdatenbank anzulegen. Wir haben den Testschritt so allgemein gehalten, dass wir ihn in späteren Szenarien gut verwenden können, um weitere Jobs mit speziellen Attributen zu generieren.

Nun können wir Cucumber erneut ausführen und das Ergebnis ist ein anderes:

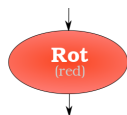
```

Wenn wir "Rails" für "search" eintippen # features/step_definitions/job_steps
.rb:5
cannot fill in, no text field, text area or password field with id, name, or
label 'search' found (Capybara::ElementNotFound)
(eval):2:in 'fill_in'

Failing Scenarios:

```

<sup>5</sup>Anmerkung: Für diese Aufgabe eignen sich Factories deutlich besser. Da mit Fixtures schon eine Testdatengenerierung eingeführt wurde, bleiben wir aber aus Konsistenzgründen dabei.



```
cucumber features/suche.feature:6 # Scenario: Auffinden durch Titel

1 scenario (1 failed)
5 steps (1 failed, 3 skipped, 1 passed)
```

Der Test schlug also fehl, da noch kein Suchfeld eingebaut wurde. Dies lösen wir, indem wir auf der View der Startseite eines einbauen:

```
app/views/layouts/application.html.erb
1 ...
2 <div id='search-field'>
3   <%= form_tag "/jobs" do %>
4     <%= text_field_tag(:search) %>
5     <%= submit_tag("Suchen") %>
6   <% end %>
7 </div>
```

Listing: Implementation eines Suchfeldes im Layout der Webseite

Wir implementieren im Layout der Applikation ein Suchfeld mit den Form-Hilfsfunktionen, die Rails anbietet. Das Layout ist eine Basisview, die für alle Webseiten generiert wird. Diese beinhaltet Elemente der Website, die überall benutzt werden, z.B. eine Navigation, Einbindung von Stylesheets oder wie hier, ein Suchfeld, das auf jeder Seite erscheinen soll. Dabei verwenden wir ERB<sup>6</sup>, eine Template-Sprache, die normalerweise HTML-Code erzeugt und es ermöglicht innerhalb der „<%= ... %>“ Ruby-Code auszuführen.

Jetzt folgt die Implementierung der Suche im Jobs-Controller:

```
app/controllers/jobs_controller
1 class JobsController
2   def index
3     @jobs = Job.where("title like ?", "%" + params[:search] + "%")
4   end
5 end
6 ...
```

Listing: JobsController am Ende der Testphase

Im Jobs-Controller implementieren wir die Bereitstellung der Jobs mittels einer Datenbankabfrage. Dabei nutzen wir die Datenbankfunktion von ActiveRecord und führen eine partielle Match-Suche der Suchparameter über die Tabellenspalte `title` aus. ActiveRecord übernimmt das sichere Escapen des übergeben Strings selbstständig in den dafür vorgesehenen Platzhalter „?“<sup>6</sup>. Das Ergebnis speichern wir in der Instanzvariable `@jobs` und stellen es so der View zur Verfügung.

Damit nun der Test besteht, muss als letztes noch eine View implementiert werden, die (zumindest) die Titel aller in der Suche gefundenen Jobs ausgibt, hier z.B. als Überschrift.

```
app/views/jobs/index.html.erb
1
2 <%= @jobs.each do |job| %>
```

<sup>6</sup>Embedded Ruby, eine Syntax für Rails Views

```

3   <div class='job'>
4     <h3><%= job.title %></h3>
5   </div>
6   <%= end %>

```

Listing: View-Implementation für eine Liste von Jobs

Mittels des Iterators `each`, geben wir für jeden gefundenen Job ein `DIV` mit einer darin enthaltenen `H3`-Überschrift des Titels aus.

Cucumber bestätigt uns nun die erfolgreiche Implementation des Features:

```

1 scenario (1 passed)
5 steps (0 failed, 0 skipped, 5 passed)

```

Auch innerhalb der Akzeptanztestgetriebenen Entwicklung ist Refaktorisieren ein fester Bestandteil innerhalb des Entwicklungszyklus. Ein guter Ansatz wäre z.B. die Suchlogik aus dem Controller in die Job-Modelklasse auszulagern.

```

----- app/models/job.rb -----
1 class Job
2   ...
3   def self.perform_search(params)
4     where("title like ?", "#{params[:search]}")
5   end

```

Listing: Auslagerung der Suchlogik in die Job-Klasse

Wir lagern die Suchmethode in eine neue Klassenmethode der Job-Klasse aus. Klassenmethoden werden definiert, indem `self.` vor den Methodennamen geschrieben wird.

```

----- app/controllers/jobs_controller.rb -----
1 class JobsController
2   def index
3     @jobs = Job.perform_search(params)
4   end
5 end

```

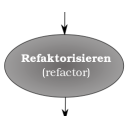
Listing: Finaler Jobcontroller nach Refaktorisierung

Nun können wir die neu implementierte Funktion in unserem Controller verwenden.

**Ausblick** Nun haben wir ein erstes Szenario mit Cucumber implementiert. Da wir schon mehrere Testschritte implementiert haben, ist die Entwicklung von weiteren Szenarien leichter.

In diesem Abschnitt wurde gezeigt:

- wie Cucumber zur Definition von Testschritten in deutscher Sprache geschrieben werden,



- wie diese Testschritte auf reale Testanweisungen übertragen werden, um einen Browser zu steuern und Erwartungen zu definieren,
- wie ein Testen des vollständigen Rails-Software-Stacks möglich ist, und wie Model, Controller und View dabei zusammen agieren.

Die Tests wurden bisher mittels **Capybara** durch den simulierten Browser RackTest ausgeführt. Da wir Capybara als Abstraktion des Browsers genutzt haben, wäre eine Umstellung auf eine reale Browserumgebung, z.B. Firefox oder Chrome, kein Problem. Deren Ausführung ist zwar deutlich langsamer als bei RackTest, dafür findet hier der Test unter realen Bedingungen statt, nämlich in einem Browser, wie ihn auch später Nutzer der Web-Anwendung besitzen.

### 7.5. Testen von JavaScript

Mit dem ständig zunehmenden Grad der Entwicklung von JavaScript-Frameworks und den in den letzten Jahren zunehmenden Ausführungsgeschwindigkeiten von JavaScript in den einzelnen Browsern ist JavaScript bei der Programmierung von modernen Webanwendungen ein immer wichtigerer Teil geworden. Das Aufkommen von Cloud Computing propagiert die Ablösung traditioneller Desktop-Software durch browserbasierte Anwendungen. So beinhaltet fast jede moderne Webanwendung JavaScript-Funktionalität.

Meist geschieht dies durch **Ajax**<sup>7</sup>, also durch asynchrone Kommunikation mit einem Server und Aktualisieren der Seite, ohne dass diese neu geladen werden muss. Durch sehr gute Frameworks wie z.B. **jQuery**, das ab Rails 3 standardmäßig Teil der Rails-Distribution ist, sind solche Aktualisierungen meist in sehr wenigen Zeilen zu implementieren. Aber auch diese Zeilen sind Teil unseres Applikationscodes und es existiert ein Bedürfnis, diesen Teil ebenfalls zu testen.

**Test im Rahmen von Systemtests** Die für uns leichteste Methode ist das Testen von JavaScript im Rahmen unserer System- / Akzeptanztests, die bereits im vorherigen Abschnitt Thema waren. Dafür müssen wir nur eine Browser-Engine wählen, die JavaScript ausführen kann. Für die Browsersimulation verwenden wir Capybara, haben also nun leicht die Möglichkeit, die Engine zu wechseln. So kann z.B. mittels der Middleware Selenium Firefox oder Chrome simuliert werden. Dies verzögert zwar unsere Testausführung, stellt aber sicher, dass auch JavaScript in einer Umgebung ausgeführt wird, die der letztendlich Realen des Benutzers sehr nahe kommt.

**Dedizierte Unittests für JavaScript** Falls in einer Anwendung nicht nur einfache Ajax-Funktionen der Art „Klicke Link – Sende Anfrage an Server – setze fertig generierten

---

<sup>7</sup>asynchrones JavaScript und XML (oder JSON)

HTML-Code der Antwort direkt in den DOM<sup>8</sup> ein“, sondern Funktionen, die klassische Desktopanwendungen nachbilden, gefordert sind, ist der Einsatz von eigenen Unittests für JavaScript angebracht. Diese Art der Anwendungsgestaltung gewinnt zunehmend an Popularität, z.B. Twitter oder Google Mail machen nach dem initialen Laden der Webseite keine weitere komplette Seitenaktualisierung mehr. Auch hier bieten sich einige JavaScript Test-Frameworks an, z.B. JsUnit<sup>9</sup>, ein Vertreter der xUnit-Reihe und Jasmine<sup>10</sup>, ein Vertreter aus dem <sup>†</sup>Behavior Driven Development-Kontext (vgl. Abschnitt 3.6.2).

Bei IT-Jobs beschränkt sich der Einsatz von JavaScript allerdings auf die erwähnten einfachen DOM-Manipulationen, wodurch JavaScript-Unittests nicht erforderlich sind.

**Diskussion** Das explizite Testen von JavaScript ist zwar nicht Thema dieser Diplomarbeit, aber im Kontext der Entwicklung einer Webanwendung sollte JavaScript nicht unerwähnt bleiben. Für die vorliegende Anwendung IT-jobs hat die Methode, JavaScript im Rahmen von Systemtests zu testen, ausgereicht. Für komplexe JavaScript-Anwendungen sind aber unbedingt dedizierte Unittests erforderlich.

---

<sup>8</sup>Document Object Model – eine Schnittstelle, die die Objekte eines XML-Dokuments und ihrer Aktionen beschreibt

<sup>9</sup><http://www.jsunit.net/>

<sup>10</sup><http://pivotal.github.com/jasmine/>

# Kapitel 8.

## Auswertung

In diesem Abschnitt wollen wir nun verschiedene Resultate des Projekts IT-Jobs im Bezug auf Code-Qualität untersuchen und vergleichen.

Zuerst werden verschiedene Metriken, die im Verlauf der Entwicklung regelmäßig festgehalten wurden, näher betrachtet. Des Weiteren vergleichen wir die erreichten Maßzahlen mit denen früherer Projekte der pludoni GmbH und einiger bekannter OpenSource Ruby-Projekte, um die Ergebnisse besser einordnen zu können.

### 8.1. Entwicklungsstand IT-Jobs

Zum Zeitpunkt der Beendigung dieser Arbeit sind folgende Teilmodule abgeschlossen:

- Modul <sup>†</sup>CMS: Blog und statische Seiten sowie die Möglichkeit variablen Inhalt auf beliebigen Seiten innerhalb der Sidebar darzustellen
- Modul Job, Basis: Implementation des Datenbankschemas für Jobs und die Möglichkeit Jobs über das Interface anzulegen.
- Modul Feedimport: Möglichkeit, Jobs mittels einer XML-Schnittstelle in das System einzuspielen und E-Mail-Benachrichtigung im Falle eines Fehlers an den Inhaber der XML-Feeds.

Noch offen für eine zukünftige Entwicklung sind:

- Modul: Bewerbung. Bewerber können sich über die Webseite bewerben. Ebenfalls können sie sich mit Facebook und LinkedIn verbinden, um automatisch ihre Stammdaten eintragen zu lassen
- Modul: Bezahlungssystem: Beim Anlegen eines Jobs wird der Bezahlvorgang über einen externen Dienstleister abgewickelt.

Während der Arbeiten an IT-Jobs wurden andere Aufgaben priorisiert und deswegen der Abgabetermin für das System nach 2012 verlegt. Nichtsdestotrotz soll IT-Jobs als Basistechnologie und Referenzprojekt für weitere Rails-Anwendungen dienen und die Entwicklung



kann insgesamt als Erfolg bewertet werden, auch wenn sie zum gegenwärtigen Zeitpunkt noch nicht abgeschlossen ist.

An der Entwicklung waren einschließlich des Autors zwei Programmierer beteiligt. Beiden Programmierern gelang es, sich mit der Testgetriebenen Entwicklung und mit Rails vertraut zu machen und wertvolle Erfahrungen zu sammeln.

## Code-Qualität und Testabdeckung

Während der Entwicklung wurden in regelmäßigen Abständen Code-Metriken festgehalten und können nun so über den Verlauf der Entwicklung dargestellt werden.

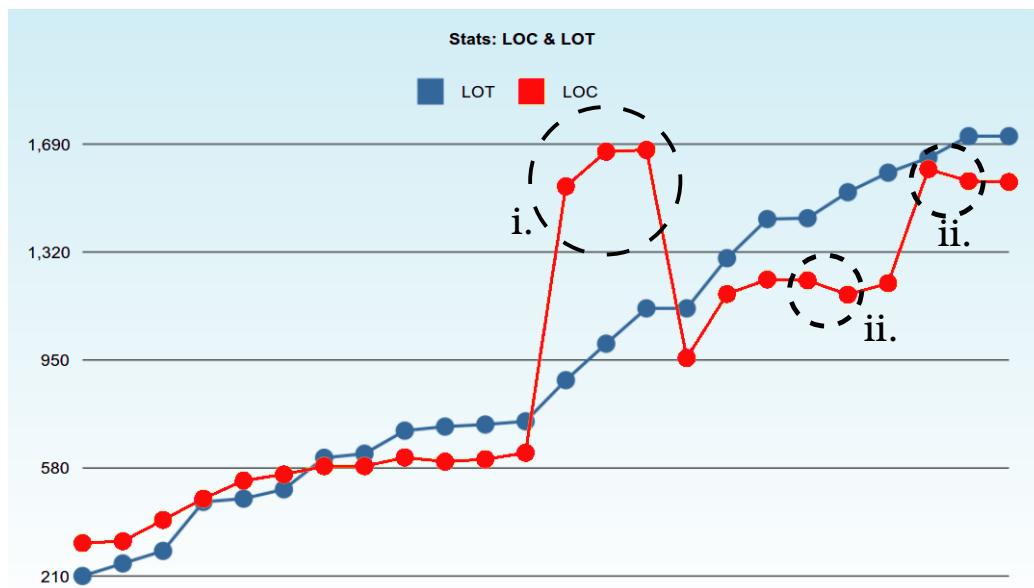


Abbildung 8.1.: Entwicklung des Umfangs des Test- und Programmcodes

In Abbildung 8.1 ist der Verlauf der Codezeilen und der Testzeilen abgebildet. Während in den ersten Tagen der Entwicklung, wegen des Einsatzes von Codegeneratoren und einer langsamen Gewöhnung an den TDD Ablauf, noch weniger Testcode als Programmcode geschrieben wurde, verlaufen die Linien ab dann nahezu proportional. Einen Ausreißer stellen die mit i. markierte Codezeilen dar: Hier ist eine Fehlkonfiguration der Statistikberechnung die Ursache dafür, dass Fremdcode mitgezählt wurde, der nicht Teil der Anwendung war. Eine andere interessante Erkenntnis ist, dass zwar die Lines Of Test monoton steigend ist, die Lines Of Code dagegen durch Refaktorisierungen abgefallen sind (z.B. mit ii. markierte Stellen). Die hohe Testdichte hat eine Refaktorisierung ermöglicht und in diesen Phasen konnte effektiv Design zu dem System hinzugefügt werden.

In Abbildung 8.2 ist der Verlauf der  $\uparrow$ Testabdeckung über den Verlauf der Entwicklung abgebildet. Insbesondere in den ersten Tagen gab es einige Probleme, die davon verursacht wurden, dass das Tool mit dem die Abdeckung gemessen wurde, RCov nicht kompatibel mit der aktuellen Ruby Version 1.9.2 ist. RCov lieferte deshalb falsche Werte liefert. Später

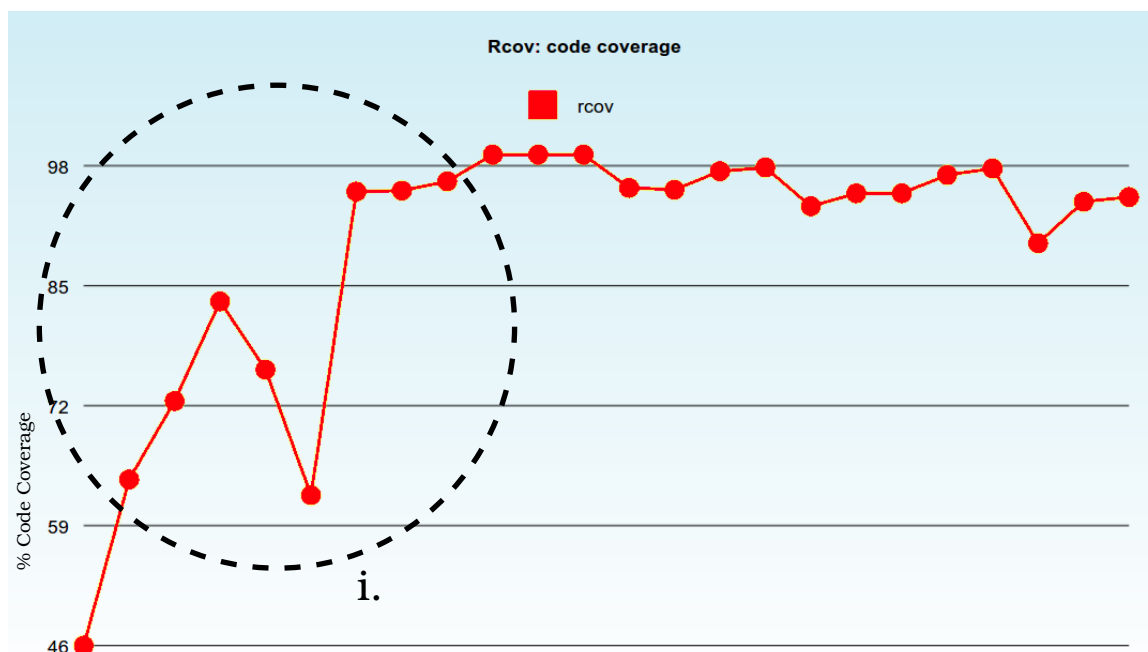


Abbildung 8.2.: CO-Testabdeckung über den Verlauf der Entwicklung

erfolgte eine Umstellung auf SimpleCov (bereits in Abschnitt 6.2 vorgestellt). Ab dann lag die Testabdeckung innerhalb von 90% bis 100%.

Ein Indikator für die  $\uparrow$ Code-Qualität sind die Anzahl und Arten von Code-Smells. Diese wurden ebenfalls über den Entwicklungszeitraum gemessen und sind in Abbildung 8.3 dargestellt. Insgesamt betrachtet, stiegen diese mit zunehmender Codemenge langsam an. Dank der Messung durch Code-Metriken konnten einige suboptimale Stellen nach einer gewissen Zeit refaktorisieren werden. So ist das wiederkehrende Muster zu erklären, dass die Anzahl der Vorkommen eines Codesmells erst anstieg und nach einigen Tagen wieder abfiel. Das Ergebnis gilt als positiv zu bewerten, da die Menge an Code-Smells langsamer anstieg, als die dazugehörige Menge an Code (LOC, vgl. Abbildung 8.1). Die Code-Smells wurden mit dem Werkzeug Reek gemessen und die Definition der Code-Smells sowie Informationen über das jeweilige Messverfahren entnehmen sie bitte [Rut10].

## Diskussion

Falls das Projekt in einem reinen TDD-Vorgehen entwickelt worden wäre, dann hätte die Testabdeckung 100% betragen müssen. Dies war aber nicht immer der Fall. Gründe hierfür sind:

**Nutzung von Codegeneratoren** Bei der Erstellung der Gerüste und der Programmierung einfacher Administrationsinterfaces waren nicht immer Testfälle mitgeneriert worden. Stellenweise wurde dieser generierte Code später neu in TDD geschrieben, allerdings nicht in allen Fällen

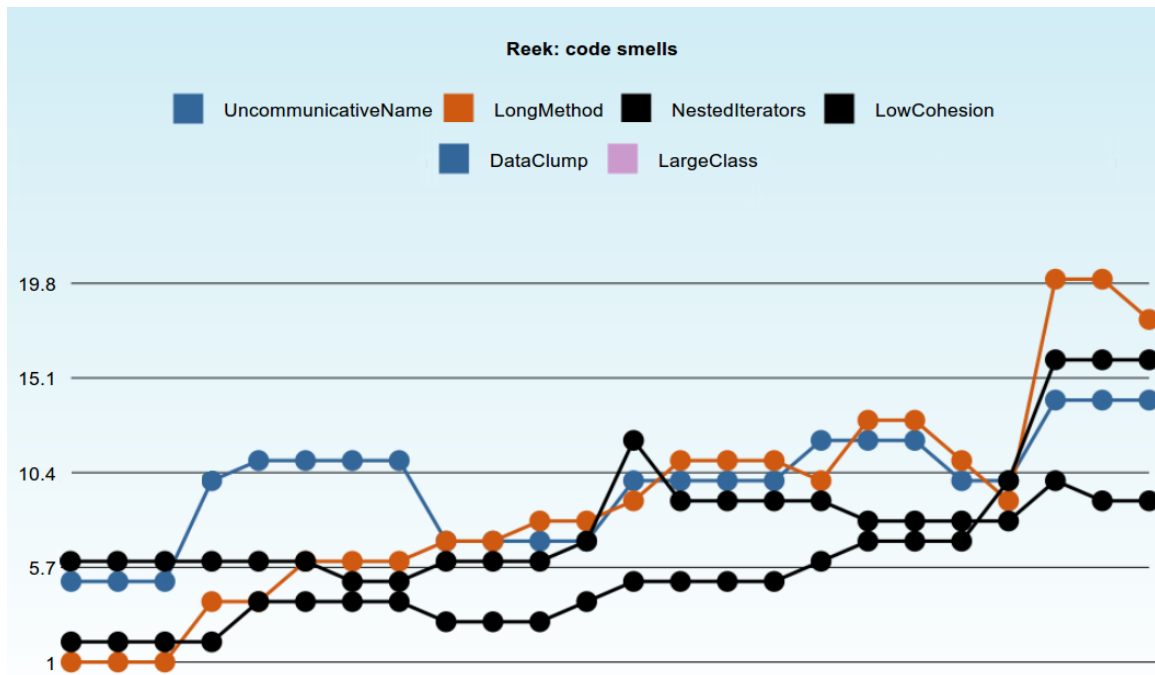


Abbildung 8.3.: Verlauf der verschiedenen Klassen an Code-Smells über den Entwicklungszeitraum

**Erfahrungsgewinn mit Rails** Vorher waren schon einige Erfahrung mit Rails vorhanden. Allerdings sind in der neusten Version von Rails einige Features dazu gekommen und wurden nun einige Bibliotheken verwendet, die vorher noch nicht bekannt waren. Aus der Unkenntnis entstand so viel Code, der eigentlich zum Experimentieren mit diesen neuen Features gedacht war. Diese Spikes (siehe Abschnitt 3.3) dienen dem Erkunden neuer Funktionalitäten und dem Prototypisieren. Diese sollten normalerweise nicht in den Hauptzweig der Entwicklung mit eingeecheckt werden. In zukünftigen Projekten, die nach dem Prinzip der kontinuierlichen Integration stattfinden werden (siehe Abschnitt 6.1.2), wird ein Einchecken dieser Spikephasen in den Hauptzweig der Entwicklung nicht erfolgen.

**Erfahrungsgewinn mit TDD** Trotz bisheriger Erfahrungen mit Ruby und Rails waren nur wenig Erfahrungen zum Thema Testen und Testgetriebene Entwicklung vorhanden. Insbesondere die Nutzung von <sup>†</sup>Test-Doubles musste erst gelernt werden, da ohne diese bestimmte Testszenarien schwierig bis überhaupt nicht zu testen sind.

Trotzdem ermöglichte die vorhandene Testdichte eine sichere Ausgangsbasis für ständige Refaktorisierungen. Ein Ergebnis dessen ist, dass die Anzahl der Code-Smells im akzeptablen Rahmen geblieben ist. Eine gewisse Menge an Komplexität lässt sich nicht vermeiden und Code-Smells können hier nur ein Indikator sein. In vielen Fällen muss direkt am Code entschieden werden, ob ein spezifischer Code-Smell für eine gewisse Quelltextstelle relevant ist oder nicht.

## 8.2. Code-Quality-Benchmark mit anderen Ruby-Projekten

Um die Ergebnisse besser einordnen zu können, vergleichen wir die Ergebnisse aus den Code-Quality-Benchmarks mit vergangenen Ruby-Projekten in der pludoni GmbH und mit bekannten Ruby/Rails-OpenSource-Applikationen und -Bibliotheken.

Eine vollständige Metrik wie oben an IT-Jobs war in vielen Fällen nicht möglich, da die Testwerkzeuge in vielen Fällen Inkompatibilitäten mit neueren Rubyversionen haben. Wir beschränken uns deshalb auf eine exemplarische Überblicksmetrik, bestehend aus:

**LOC** Anzahl der Quellcodezeilen, gemessen durch das <sup>↑</sup>Rake-Tasks `rake stats`.

**LOT** Anzahl der Quellcodezeilen aller Tests, ebenfalls gemessen durch das Rails-Kommando `rake stats`.

**LOT/TOC** Verhältnis aus Testzeilen und Quellcodezeilen.

**AVGComplex** Durchschnittliche Komplexität aller Klassen, gemessen durch **Flog**<sup>1</sup>. Flog besitzt ein eigenes Maß für Code-Komplexität. Es vergibt für jede Zuweisung, Verzweigung und Funktionsaufruf unterschiedlich viele Punkte und bildet so eine Summe je Funktion oder je Klasse. Dabei vergibt Flog besonders viele Punkte für schwer nachzuvollziehende Funktionsaufrufe, wie z.B. `eval(string)`, welches einen String als Ruby-Code auswertet.

**H5Complex** Komplexität oberen 5% komplexesten Klassen (mindestens 1 Klasse), nach Flog.

**DSmell** Anzahl der Code Smells nach **Roodi**<sup>2</sup>. Beinhaltet u.a.: hohe zyklomatische Komplexität in einer Methode (min. 4), lange Methoden, lange Parameterlisten, u.v.m.<sup>3</sup>

**DSmell/KLOC** Anzahl der Code Smells nach Roodi pro tausend Codezeilen.

**CSmell** Anzahl der Code Smells nach Reek. Diese beinhalten: geringe Kohäsion, Duplikation, unkommunikativer Name von Methoden/Variablen/Parameter, verschachtelte Iteratoren, u.v.m.<sup>4</sup> (vgl. Abschnitt 5.1).

**CSmell/KLOC** Anzahl der Code Smells nach Reek pro tausend Codezeilen.

### 8.2.1. Vergleich von IT-Jobs mit eigenen Projekten

Bisherige Projekte der pludoni GmbH und des Autors basierend auf Ruby/ Ruby on Rails:

**feedimport** Die fertiggestellte Bibliothek zum Parsen und Import von Stellenanzeigen, die im Rahmen der Entwicklung von IT-Jobs ausgelagert wurde, um den Community-Portalen bereits jetzt zur Verfügung zu stehen. Die Grundzüge wurden in Abschnitt 7.3 vorgestellt.

---

<sup>1</sup><https://github.com/seattlerb/flog>

<sup>2</sup><https://github.com/martinjandrews/roodi>

<sup>3</sup>[http://roodi.rubyforge.org/files/README\\_txt.html](http://roodi.rubyforge.org/files/README_txt.html)

<sup>4</sup><https://github.com/kevinrutherford/reek/wiki/Code-Smells>

**feedmerger** Eine Rails-Anwendung<sup>5</sup> zur Verwaltung, Cachen, Filtern und Zusammenfügen von RSS- und Atom-Feeds.

**pludonidb** Eine auf ActiveRecord basierende Bibliothek zur Anbindung der Datenbanken der Communityportale an Ruby-Skripte. Beinhaltet außerdem weitere Hilfsfunktionen für Berechnung von Tag-Wolken, Häufigkeiten u.ä.

**backlinks** Backlink und SEO-Success-Control ist eine Rails-2.3 Anwendung zum Messen der sogenannten Backlinks<sup>6</sup> der Communitymitglieder und der Platzierung für relevante Sucheingaben in den großen Suchmaschinen (konkret: Welchen Platz hatte ein Communityportal an einem bestimmten Tag für einen konkreten Suchbegriff). Details dazu sind im Praktikumsbericht des Autors zu finden.

**SiteAnalyzer (SAnalyzer)** Ein Werkzeuge zur Analyse von kompletten Websites/Webdomains mit dem Ziel, tote Links zu finden, alle Seiten auf HTML-Gültigkeit zu prüfen und insgesamt die Link-Architektur zu analysieren<sup>7</sup>.

Tabelle 8.1.: Vergleich von IT-jobs mit anderen Ruby Projekten des Autors/der pludoni GmbH

Projekt	ItJobs	Backlinks	Feedmerger	SAnalyzer	Feedimport	PludoniDb
Technologie	Rails3	Rails2	Rails3	Rails3	Ruby	Ruby
Testverfahren	TDD	manuell	manuell	manuell	TDD	manuell
LOC	1570	1884	724	214	571	1933
LOT	1750	75	137	0	865	126
LOT/LOC	1.11	0.04	0.19	0.00	1.51	0.07
AVGCmplx	8.00	19.90	10.50	14.80	16.80	13.70
H5Cmplx	43.80	70.60	26.90	39.70	25.10	113.20
DSemll	11	69	11	11	8	25
DSmell/KLOC	7.00	36.60	15.20	51.40	14.00	12.90
Csmell	53	152	52	13	28	113
Csmell/KLOC	33.80	80.70	71.80	60.70	49.00	58.50

### 8.2.2. Vergleich von IT-Jobs mit anderen Rails-Projekten

Weiterhin wird das Projekt mit folgenden beliebten Webprojekten, welche ebenfalls auf Ruby on Rails basieren, verglichen:

<sup>5</sup> Als OpenSource unter <https://github.com/zealot128/WenShanWenHai> zu finden.

<sup>6</sup>Backlinks sind Links anderer Webseiten auf die Eigene. In den Communityportalen sind die Mitglieder vertraglich verpflichtet, einen Backlink auf die Communityportale anzulegen.

<sup>7</sup>Auch dieses Projekt ist als OpenSource verfügbar: <https://github.com/zealot128/Site-Structure-Analyzer>

**lpp** Das Linkpartnerprogramm ist eine Studenteninitiative der TU Dresden, um ausländische Studenten mit deutschen Sprach- und Lernpartnern zusammenzubringen. Dazu wurde 2008 von einem Studententeam im Rahmen einer Semesterarbeit eine Rails-Webanwendung geschrieben, die unter <http://linkpartnerprogramm.de> zu finden ist. Der Autor dieser Arbeit war zwar nicht an der Entwicklung beteiligt, übernahm aber die Pflege und Weiterentwicklung selbiger.

**diaspora** Ein verteiltes soziales Netzwerk (vgl. Google+ und Facebook)

Code: <https://github.com/diaspora/diaspora>

**bucketwise (bucket)** Ein Web-basierter persönlicher Finanzmanager mit Budgetierung nach dem Briefumschlagsystem.

Code: <https://github.com/jamis/bucketwise>

**chiliproject (chilli)** Ein Fork<sup>8</sup> des sehr beliebten Bugtrackers Redmine<sup>9</sup>. Statt des originalen Redmines wurde dieser Fork verwendet, da er eine neuere Codebasis hat.

Code: <https://github.com/chiliproject/chiliproject>

**railscast (rCasts)** Der Code der Website <http://www.railscasts.com>, in welche allwöchentlich ein Screencast zum Thema Ruby und Rails veröffentlicht wird (bis dato 283 Episoden).

Code: <https://github.com/ryanb/railscasts>

**ActiveSupport (aSupport)** Beinhaltet Hilfsklassen und Erweiterungen der Ruby-Standardbibliothek und ist Kernbestandteil von Ruby on Rails.

Code: <https://github.com/rails/rails.git>

**ActionPack (aPack)** Ein Framework, um Anfragen und Antworten eines Webserverns zu verarbeiten und ein weiterer Kernbestandteil von Rails.

Code: <https://github.com/rails/rails.git>

Die Auswahl erfolgt z.T. willkürlich. Als Ansatzpunkt diente die Liste der beliebtesten Rails-Projekte, die auf Github gehostet sind<sup>10</sup>. Weiterhin wurde ActiveSupport und ActionPack ausgewählt, um die Code-Qualität von Rails selbst mit zu beurteilen.

### 8.2.3. Auswertung und Visualisierung

In den Abbildungen 8.4, 8.5 und 8.6 werden die gesammelten Ergebnisse für alle genannten Projekte visualisiert. Der obere Teil jeder Abbildung enthält die externen Projekte, der untere alle Projekte der pludoni GmbH.

Abbildung 8.4 zeigt das Verhältnis aus Testcode zu Programmcode. Der Median über alle Projekte ist 0.95. Ausreißer sind hierbei vier der internen Projekte pludonidb, SiteAnalyzer,

<sup>8</sup>In der (Open-Source)-Software-Entwicklung ist ein Fork eine legale Kopie eines bestehenden Software-Produktes, um von nun an eine unabhängige Entwicklung zu betreiben.

<sup>9</sup><https://github.com/edavis10/redmine>

<sup>10</sup><https://github.com/search?q=rails&type=Repositories>

Tabelle 8.2.: Vergleich von IT-jobs mit Ruby/Rails Projekten aus der Community

Projekt	itjobs	bucket	lpp	chili	aPack	aSupport	diaspora	rCasts
Technol.	Rails3	Rails2	Rails1	Rails2	Rails3	Rails3	Rails3	Rails3
LOC	1570	1979	7116	21201	12995	9407	7466	653
LOT	1750	1684	1557	20127	32570	13590	10072	748
LOT/LOC	1.11	0.85	0.22	0.95	2.51	1.44	1.35	1.15
AVGCmplx	8.00	12.80	17.10	19.10	11.10	10.90	13.10	11.00
H5Cmplx	43.80	32.20	181.80	84.40	64.00	47.90	54.00	34.60
DSmell	11	37	250	651	370	166	99	2
DSmell /KLOC	7.00	18.70	35.10	30.70	28.50	17.60	13.30	3.10
CSmell	53	129	386	1535	982	291	324	42
CSmell /KLOC	33.80	65.20	54.20	72.40	75.60	30.90	43.40	64.30

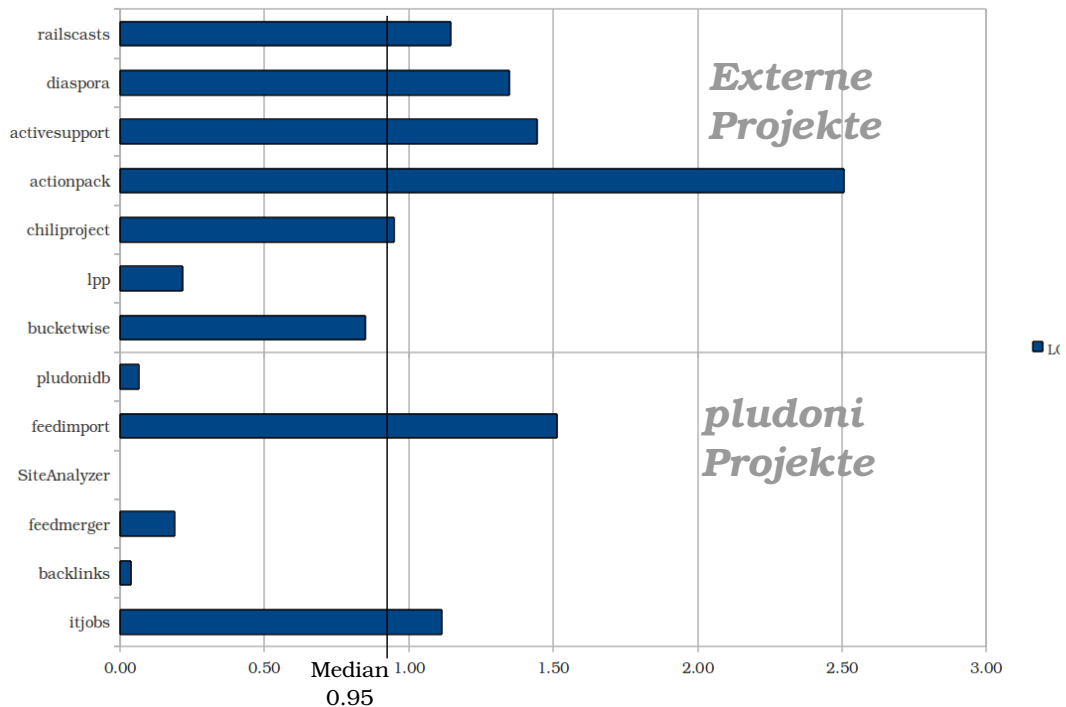


Abbildung 8.4.: Vergleich des Verhältnisses aus Anzahl Testcodezeilen / Anzahl Codezeilen

feedmerger und backlinks, die wenig bis gar keine automatisierten Tests besitzen. Als Gegenbeispiel sei ActionPack zu nennen, dass über 2.5x soviel Testcodezeilen wie Programmcode verfügt. Gegenüber den bisherigen Projekten in der pludoni GmbH kann man anhand der Projekte Feedimport und IT-Jobs, die im Rahmen dieser Diplomarbeiten entstanden sind, sehen, dass die bloße Anzahl an Tests stark zugenommen hat. Beide Projekte wurden mittels TDD entwickelt. Gegenüber dem Feedimport nutzt IT-Jobs aber Rails als Grundlage. Dies bedeutet, dass dadurch viel Logik durch das Framework implementiert und vorgegeben wurde. Der Feedimport dagegen besitzt eine komplett eigene Objekthierarchie und hat darum mehr Tests benötigt als IT-Jobs.

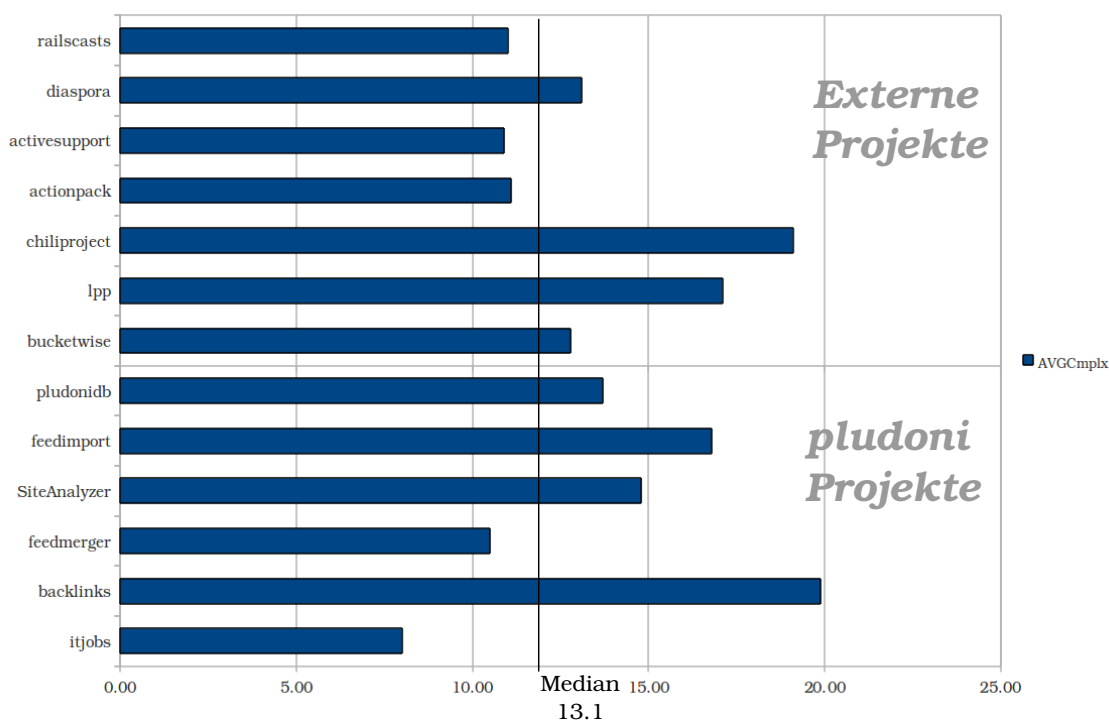


Abbildung 8.5.: Vergleich der durchschnittlichen Komplexität

In Abbildung 8.5 wurde die durchschnittliche Komplexität nach dem Flog-Verfahren gegenübergestellt. Der Median war hier 13,1. Unter den eigenen Projekten sind hier Backlinks und der Feedimport die komplexesten Projekte. IT-Jobs und Feedmerger, die beide Rails-Projekte sind, haben dagegen eine geringere Code-Komplexität. Die Ursache ist hier, dass während der Entwicklung von IT-Jobs ständig die Code-Qualität durch die Metriken gemessen wurde und im Falle von Ausreißern gegengesteuert wurde. Bei dem Feedimport, der ebenfalls durch TDD entwickelt wurde, wurde eine solche Analyse erst am Ende der Entwicklung durchgeführt. Für uns ist das ein Indiz, dass die Nutzung einer Testgetriebenen Entwicklung kein alleiniger Garant für eine gute Code-Qualität ist.

Bei den externen Projekten war die Bugtracker-Anwendung Chiliproject/Redmine das komplexeste Projekt, dicht gefolgt von der Sprachpartnervermittlung Linkpartnerprogramm. Gerade wenn man das Verhältnis aus Testcode zu Programmcode mit einbezieht, kann man



eine Tendenz ableiten: beide Projekte haben gegenüber den anderen Externen eine relativ geringe Anzahl an Testcode und beide haben eine relativ hohe Komplexität.

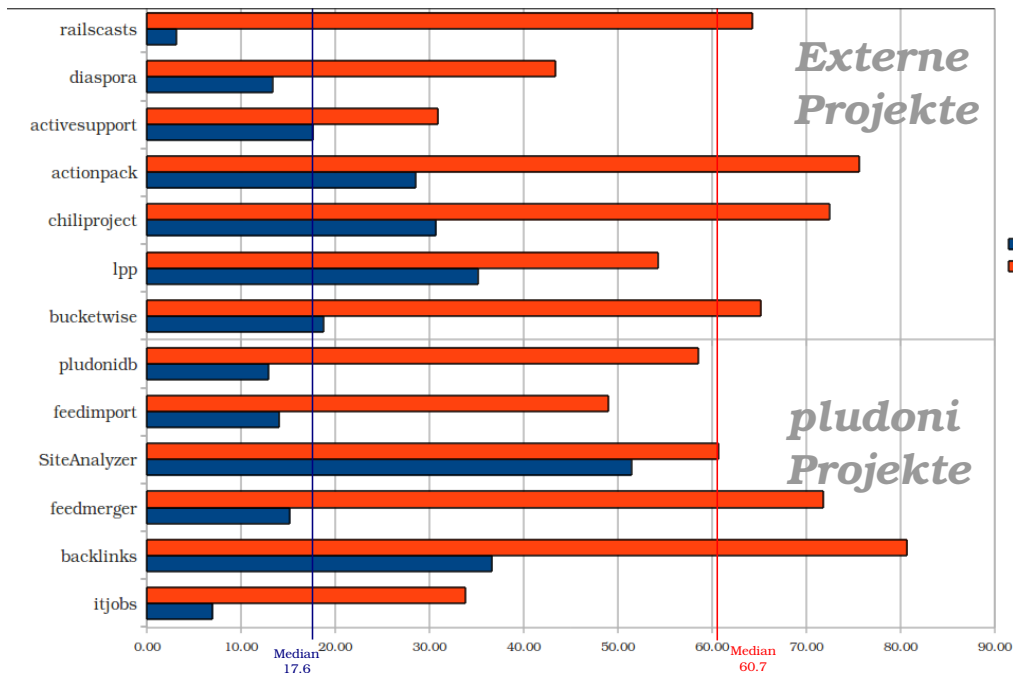


Abbildung 8.6.: Vergleich der Anzahl Smells pro KLOC

Die Abbildung 8.6 zeigt zwei verschiedene Arten von Smells: einerseits durch Reek gemessen (Rot) und andererseits durch Roodi gemessen (Blau). Zu bemerken ist, dass IT-Jobs eine deutlich geringere Code-Smell-Dichte (1/2 Median) besitzt. Auch der Feedimport ist noch im Rahmen und leicht unter dem Schnitt. Dagegen hat das interne Projekt Backlinks die höchste Dichte an Code Smells. Interessanterweise hat dieses Projekt auch so gut wie gar keine automatisierten Tests, was eine nachträgliche Refaktorisierung sehr erschwert. Für das Projekt Feedimport existiert allerdings eine gute Testabdeckung und so ließen sich leicht weitere Refaktorisierungen durchführen, sollte das gewünscht sein.

Actionpack weist trotz der großen Menge an Tests und der relativ geringen Komplexität unter den externen Projekten die höchste Smell-Dichte auf. Da Actionpack auch das umfangreichste Projekt unter den Untersuchten ist, läge hier ein guter Ansatzpunkt für weitere Untersuchungen in der Sinnhaftigkeit gewisser Code-Smells.

Die mit Abstand häufigsten Code-Smells hierbei waren geringe Kohäsion bzw. starke Koppelung unter den Klassen. Eventuell lassen sich einige der Smells bei großen Modulen nicht vermeiden, da diese zwangsweise untereinander gekoppelt sein müssen.

Für statistisch signifikante Ergebnisse reichen die hier präsentierten Daten nicht aus, stattdessen sollte nur eine Tendenz untersucht werden. Insbesondere der Vergleich mit den internen Projekten hinsichtlich Code-Qualität ist für weitere Projektplanungen interessant. Das Gefühl der Programmierer, dass z.B. das Projekt Backlinks eine relativ schlechte Code-

Schlussfolgerung

basis verfügt, wurde durch die hier gezeigten Metriken bestätigt. Der aktuelle Stand von IT-Jobs dagegen lässt eine gute Ausgangsbasis für eine Weiterentwicklung vermuten.

# Kapitel 9.

## Fazit

Write the tests you wish you had. If you don't, you will eventually break something while refactoring. Then you'll get bad feelings about refactoring and stop doing it so much. Then your design will deteriorate. You'll be fired. Your dog will leave you. You will stop paying attention to your nutrition. Your teeth will go bad. So, to keep your teeth healthy retroactively test before refactoring.

---

Kent Beck [Bec02]

In diesem abschließendem Kapitel werden erreichte Ergebnisse zusammengefasst und ein Ausblick für weitere Forschungen auf dem Gebiet der Testgetriebenen Entwicklung gegeben.

### 9.1. Zusammenfassung

Die im Rahmen dieser Diplomarbeit entwickelte Web-Anwendung ist zum gegenwärtigen Zeitpunkt noch nicht fertiggestellt. Aufgrund einer Repriorisierung innerhalb der Firma wird diese voraussichtlich 2012 weiterentwickelt werden. Gleichwohl kann die Entwicklung als Erfolg gewertet werden, da alle Qualitätskennzahlen eine wesentliche Verbesserung gegenüber den bisherigen Projekten aufzeigen. Im Großen ist dies der Testgetriebenen Entwicklung und der ständigen Beobachtung durch Code-Metriken zu verdanken. Dank der Testgetriebenen Entwicklung entstand relativ lose gekoppelter Code und ein feinmaschiges Testnetz, durch das die Hinweise aus den Code-Metriken in Form von gezielten Refaktorisierungen leicht umgesetzt werden konnten.

Entwicklungsstand  
IT-Jobs

Allerdings hat sich gezeigt, dass die Umstellung auf eine Testgetriebene Entwicklung nicht von heute auf morgen stattfinden kann. Das konstante Testen-Vor-Implementieren bedarf insbesondere in der Anfangsphase einer hohen Disziplin, da es leicht ist, in das alte Schema zurückzufallen. Auch das Schreiben der Tests sollte gelernt werden. Eine große Anzahl an Tests ist zwar wünschenswert, aber lange nicht ausreichend für eine Beurteilung der Testvollständigkeit oder ein gar Beweis für eine Korrektheit des untersuchten Programmes.

Lernkurve von  
TDD

Die Testgetriebene Entwicklung ist allerdings kein Allheilmittel („Silver Bullet“) für alle

Grenzen von  
TDD

Probleme der Software-Entwickler. Der Einsatz von TDD erfordert vom Programmierer ein hohes Maß an Eigenverantwortung und ein hohes Wissen über Software-Design, um Refaktorisierungen sinnvoll durchführen zu können. TDD kann auch nicht davor schützen, ein falsches Produkt zu entwickeln, weil die Anforderungen falsch verstanden wurden. Außerdem stellt die Testgetriebene Entwicklung keinen Ersatz für andere Arten von Tests wie Performanz-, Stress- und Usability-Tests dar. Auch lassen sich einige Probleme schlecht mittels TDD testen und implementieren, darunter Nebenläufigkeit oder Software-Sicherheit. Auf kurze Sicht betrachtet benötigt TDD mehr Zeit bei der Entwicklung, insbesondere in den ersten Wochen. Dafür kann es auf lange Sicht Zeit beim Debuggen sparen, da die Tests schnell Anhaltspunkte über Ursachen der Fehler geben.

Die Effektivität einer Testgetriebenen Entwicklung ist auch maßgeblich von den vorhandenen Werkzeugen innerhalb der Programmiersprache und Framework abhängig. So verfügen die weitverbreiteten Sprachen bereits über ein Vielzahl von Werkzeugen, die den Prozess unterstützen, z.B. Code-Metriken, parallelisierbare Test-Runner oder hochentwickelte Test-Frameworks, während bei weniger weitverbreiteten Sprachen diese Werkzeuge wohlmöglichst erst entwickelt werden müssten.

### Positive Effekte

Die Erfahrung und bisherige Studien haben aber gezeigt, dass durch den konsequenten Einsatz der Testgetriebenen Entwicklung im Schnitt deutlich besserer Code hinsichtlich der Komplexität und des Umfangs produziert wird und auch die Testung selbst stark vereinfacht wird, da Code und Test gemeinsam entstehen und nicht, wie sonst üblich, Tests im Nachhinein. Dadurch ist das Testen wesentlich einfacher, als durch den traditionellen Ansatz, da man gezwungen ist, gegen die Testspezifikation zu programmieren und nicht die Testspezifikation dem Code anzupassen.

Dies hat sich ebenfalls in den beiden Projekten IT-Jobs und Feedimport, die im Rahmen dieser Arbeit nach TDD entstanden, gezeigt (vgl. Abschnitt 8.2.3).

Beide Projekte lassen eine deutliche Verbesserung hinsichtlich der qualitativen Kennzahlen vermuten, verfügen über eine hohe Testabdeckung und sind im Falle des Feedimports bereits im produktiven erfolgreichen Einsatz.

Insgesamt ist TDD eine Technik, die es nach Ansicht des Autors wert ist, von jedem Programmierer gelernt zu werden. Bisherige Studien und Erfahrungen zeigen fast überwiegend positive Auswirkungen von TDD. Die Technik ist zwar schnell beschrieben („Red“, „Green“, „Refactor“), allerdings bedarf es viel Übung, diesen Zyklus beizubehalten und scheinbar schwierig zu testende Probleme anzugehen.

## 9.2. Ausblick

### Weiterentwicklung von IT-Jobs

Wie erwähnt ist die Entwicklung der Anwendung noch nicht abgeschlossen und wird zu gegebener Zeit fortgesetzt. In diesem Zusammenhang werden dann häufiger Akzeptanztests

als treibende Kraft eingesetzt und deren Auswirkungen untersucht werden (vgl. 2.5, 3.6.1 und 7.4).

Der hier gezeigte Ansatz eines Benchmarks der Code-Qualität innerhalb der Ruby/Rails-Community kann so ausgebaut werden, dass weitere Parameter untersucht werden und deutlich mehr Projekte einfließen, um so letztendlich repräsentative Kennzahlen darüber zu erhalten, was eine gute Code-Qualität für Railsanwendungen ausmacht. Dazu wird in Zukunft ein Benchmarking-Tool entwickelt, welches die Messungen halbautomatisch vornehmen wird. Leider ist die Situation bei den Metrik-Werkzeugen nicht optimal; viele Werkzeuge funktionieren nur in der älteren Ruby Version 1.8.7. Allerdings nutzen viele der aktuellen Anwendungen bereits Features, die nur die Ruby Version 1.9.x unterstützt. Hier wäre eine teilweise Erweiterung oder Neuentwicklung einiger Messwerkzeuge sinnvoll.

Ausbau der  
Benchmarks

Interessant wäre, die Effektivität von TDD in Abhängigkeit von der eigenen Erfahrung mit dieser Technik zu untersuchen, um daraus Schlussfolgerungen zu ziehen, wie lange ein Programmierer im Durchschnitt benötigt, um effektiv testgetrieben zu entwickeln. Ebenfalls wären Studien über die psychologischen Auswirkungen von TDD für die pludoni GmbH interessant, da erfahrene TDD-Entwickler zwar gefühlsmäßig wissen, dass TDD einen Fluss erzeugt und richtig angewendet Spaß macht, aber dies nicht anhand von empirischen Zahlen belegen können.

Erfahrungen  
mit TDD

Innerhalb der pludoni GmbH werden außerdem zukünftige Ruby-Projekte mittels TDD durchgeführt, um weitere Erfahrungen zu sammeln. Falls diese Erkenntnisse als Fallstudie für die Wissenschaft interessant sein sollten, so werden sie veröffentlicht.

# Anhang A.

## Eigenschaften erfolgreicher Tests

Das Vorhandensein von zahlreichen Tests reicht nicht aus, um die <sup>†</sup>Code-Qualität der Tests zu beurteilen. Stattdessen existieren einige Qualitätskriterien, um Tests auf ihre Sinnhaftigkeit und Wartbarkeit zu untersuchen:

**Unabhängigkeit (Independence) und Isolation** Ein Test ist unabhängig, falls er nicht durch andere Tests beeinflusst wird. Auch die Reihenfolge, in der die Tests ausgeführt werden, darf auf das Ergebnis keinen Einfluss üben. Siehe auch [Bec02], [Pal06] sowie [LO11, Karte 45].

**Wiederholbarkeit (Repeatability)** Ein Test wird als wiederholbar bezeichnet, wenn er mehrmals hintereinander ausgeführt werden kann und dabei jedes Mal dasselbe Ergebnis liefert [LO11, Karte 45] [Rap11]. Problematisch sind dabei z.B. Datum und Zeit sowie Zufallsfunktionen.

**Klarheit (Clarity)** Ein Test ist klar, wenn sein Zweck sofort verständlich wird [Rap11], [Pal06]. Damit wird einerseits die Lesbarkeit gemeint, andererseits schließt dies auch ein, ob der Test genau eine Eigenschaft testet und nicht redundant zu anderen Tests ist. Dies hat zur Folge, dass die Tests wartbarer werden und als Codedokumentation dienen können.

**Präzise (Conciseness)** Ein Test ist präzise, wenn er so wenig Code und so wenige Objekte wie möglich benötigt, um sein Ziel zu erreichen [Pal06] [Rap11]. Eine Auswirkung ist, dass der Test schneller wird.

**Robustheit (Robustness)** Ein Test ist robust, wenn es eine direkte Korrelation zum zu testenden Code gibt: Ist der Code korrekt, so ist der Test erfolgreich. Ist der Code falsch, so schlägt der Test fehl. Nicht-robuste Tests werden auch „zerbrechlich“ (brittle) genannt. Dazu zählen auch sogenannte tautologische Tests, die immer erfolgreich verlaufen und keine Aussage über den zugrunde liegenden Programmcode geben.

**Schnelligkeit** Gute Tests sollten schnell sein, da die Test-Suite ansonsten deutlich seltener ausgeführt wird [LO11] [Pal06] [NMBW08]. Siehe auch Präzision.

**Zeitliche Nähe** Gute Tests entstehen in zeitlicher Nähe zu dem zugrunde liegenden Programmstück [LO11]. Innerhalb von TDD ist dieses Kriterium immer erfüllt.

## Anhang B.

# Nutzung von Cucumber in Verbindung mit Selenium für Firefox und Guard ohne X-Server

Für ein effektives Test-Setup wurden folgende Testtools benutzt:

**Guard** automatische Testausführung bei Änderungen der Dateien.

**Xvfb** Ein X-Server, der ohne Grafikausgabe arbeitet. Er eignet sich also für die Ausführung von GUI-Programmen wie Mozilla Firefox auf Remote-Servern.

**Selenium** Interface zur Fernsteuerung von Browsern.

**Spork** Erhöht die Testausführungsgeschwindigkeit, da Teile von Rails im Hintergrund gehalten werden und zwischen den Tests nicht neu geladen werden müssen.

### Installation

Es muss bereits installiert sein: firefox, xvfb, Rails > 3.0

1. In das Gemfile folgendes eintragen:

Listing B.1: RAILS\_ROOT/Gemfile

```
group :test, :cucumber do
  gem "capybara"
  gem 'cucumber'
  gem "cucumber-rails"
  gem 'database_cleaner'
  gem "launchy"
  gem "guard"
  gem 'guard-cucumber'
  gem 'guard-spork'
  gem "rb-inotify" # Für Linux
  gem "spork", :git => "git://github.com/timcharper/spork.git"
end
```

Danach folgendes ausführen:

```
bundle install
rails g cucumber:install
```

2. Eine Datei Guardfile anlegen. Hier kommen alle Anweisungen zur Steuerung von guard hinein.

Listing B.2: RAILS\_ROOT/Guardfile

```
group "cucumber" do
  guard 'spork' do
    watch('config/application.rb')
    watch('config/environment.rb')
    watch('features/support/env.rb')
    watch(%r{^config/environments/.+\.rb$})
    watch(%r{^config/initializers/.+\.rb$})
    watch('spec/spec_helper.rb')
  end
  guard 'cucumber', :cli => '--no-profile --color --format pretty --strict
    RAILS_ENV=test' do
    watch(%r{^features/.+\.feature$})
    watch(%r{^features/support/.+$}) { 'features' }
    watch(%r{^features/step_definitions/(.+)_steps\.rb$}) { |m| Dir[File.
      join("**/#{m[1]}.feature")][0] || 'features' }
  end
end
```

3. In der Datei config/cucumber.yml die Option -drb für „default“ und „wip“ setzen.
4. Die Datei features/support/env.rb anpassen, um sie mit Spork kompatibel zu machen:

Listing B.3: features/support/env.rb

```
require 'cucumber/rails'
require "spork"
require 'test/unit/testresult'
Test::Unit.run = true

Spork.prefork do
  ENV["RAILS_ENV"] ||= "test"
  require File.expand_path(File.dirname(__FILE__) + '/../../config/
    environment')
  require 'cucumber/formatter/unicode'
  require 'cucumber/rails'
```



```
require 'test/unit/testresult'
ActionController::Base.allow_rescue = false
begin
  DatabaseCleaner.strategy = :truncation
rescue NameError
  raise "You need to add database_cleaner to your Gemfile (in the :test
    group) if you wish to use it."
end
end

Spork.each_run do
  require 'cucumber/rails/world'
end
```

## Nutzung

### 1. Aktivierung von Xvfb auf Displayport 99

```
Xvfb :99 -ac
```

### 2. Aktuelle Shell auf diesen Displayport einstellen

```
export DISPLAY=:99
```

### 3. Guard starten

```
guard -g cucumber
```

Nun werden automatisch alle Cucumber-Features getestet. Falls Selenium verwendet wird, dann wird Firefox im Hintergrund gestartet, ohne dass dies sichtbar wird.



# Glossar

Im Folgenden werden einige oft verwendete Begriffe näher erläutert. Innerhalb des Hauptteils dieser Arbeit sind diese Begriffe mit einem <sup>†</sup> gekennzeichnet.

**Behavior Driven Development** (Verhaltensgetriebene Entwicklung). Umformulierung von TDD zur Ausrichtung auf Businessprozesse. Das Vokabular zielt auf die Spezifikation von Erwartungen im Systemverhalten, anstatt Definition nachträglicher Tests. 38, 81

**CMS** Content Management System (Inhaltsverwaltungssystem) ist eine, meist Web-basierte Software, die es Nutzern ermöglicht einfach statische Inhalte anzulegen und zu bearbeiten. 67, 82

**Code Smell** oder Bad Smell. Ist ein Anzeichen für eine suboptimale Quelltextstelle, die auch ein Hinweis auf ein größeres Designproblem sein kann. Oft auch ein Kandidat für eine Refaktorisierung. Informationen zu Smells und deren Refaktorisierung sind im Buch von M.Fowler zu finden [FBB<sup>+</sup>99]. 17, 18, 45, 52, 101

**Code-Metrik** Eine Softwaremetrik ist das Ergebnis einer statischen oder dynamischen Codeanalyse zur Generierung von Informationen über den Quelltext. Beispiele: Testabdeckung, Anzahl Codezeilen, Anzahl <sup>†</sup>Code Smells pro Codezeile. 38, 44, 54

**Code-Qualität** beinhaltet die Software-Qualitäten Lesbarkeit, Testbarkeit, Wartbarkeit, Erweiterbarkeit, geringe Komplexität. 2, 16, 84, 96

**CRUD** Create, Read, Update, Delete sind die vier Basisoperationen, die auf persistente Speicherung ausgeführt werden können. 34, 35, 50

**DSL** Eine Domain-spezifische Sprache ist eine auf eine spezielle Problemdomäne ausgerichtete Programmier- oder Spezifikationssprache. Vertreter sind z.B. SQL, die Statistiksprache R oder die Hardwarebeschreibungssprache VHDL. Sie stehen damit den Allzweck-Programmiersprachen gegenüber. 39, 41, 51

**Entwurfsmuster** oder Design Patterns sind bewährte Vorlagen, um häufig wiederkehrende Probleme zu lösen. Weitere Informationen im Buch Design Patterns: Elements of Reusable Object-Oriented Software von Gamma, Helm, Johnson, Vlissides (Gang of Four). 18, 33, 102

**Gem** als Gems (zu Deutsch: Edelsteine) werden in der Ruby-Gemeinschaft Bibliotheken Dritter bezeichnet, die in einem zentralen Repository lagern. Das Bekannteste dieser Repositorien ist `rubygems.org`. Nahezu alle Ruby-Bibliotheken lassen sich hier finden und innerhalb von Sekunden mittels des Kommandozeilenwerkzeuges `gem` installieren. Beispielsweise ließe sich `↑Rails` mittels `gem install rails` installieren. 35, 38, 41, 54

**MVC** Model-View-Control, ist ein `↑Entwurfsmuster`, das insbesondere bei GUI- und Web-Anwendungen beliebt ist. Rails basiert auf dem MVC-Muster. 33, 38, 55, 66

**ORM** Objektrelationales Mapping, ist eine Persistenztechnik, um Objekte transparent in einer Datenbank zu speichern und umgekehrt, Tabellenzeilen in Objekte wieder zu rückübersetzen. 30, 33, 35, 56

**RackTest** ist eine Test-API für Rack-Anwendungen. Rack ist ein minimales Interface zur Kommunikation mit Webservern, Middlewares und Webframeworks, wie z.B. Rails. RackTest kommuniziert direkt mit dem Rack-Interface, verzichtet auf HTTP Traffic und ist damit sehr schnell. 42, 53

**Rake** ist ein Build-Programm, das im Gegensatz zu `make` mit Ruby programmiert wird. Es ist modular mit eigenen Tasks erweiterbar. Innerhalb von Rails dient es auch als weitere Schnittstelle zur Anwendung, um Wartungsaufgaben auszuführen (z.B. Datenbankbackup oder Cronjobs). 57, 86

**Refaktorisieren** Ist eine Modifikation des Programmcodes ohne Modifikation des externen Verhaltens um nicht-funktionale Eigenschaften des Quellcodes zu verbessern, wie z.B. Lesbarkeit, Wiederverwendbarkeit, Wartbarkeit. 18

**RSS** RDF Site Summary, ist ein standardisierter XML-Dialekt zur maschinenlesbaren Verteilung und Veröffentlichung von Inhalten. Es existiert in den Versionen 0.9 bis 2.0.1, die sich nur in Details, wie z.B. Einbindung von Rich-Media (Podcasts, ..) und Namespaces unterscheiden. 7, 67

**Ruby on Rails** ist ein auf der Programmiersprache Ruby basierendes Web-Framework und ist Gegenstand des Kapitel 4.3. 16, 30, 40, 42, 54, 102

**Software-Fehler** ist jede Abweichung der Implementation von der Spezifikation [Fri05, IEEE/ANSI]. Diese Abweichungen können zu fehlerhaftem Verhalten der Anwendung führen. 10, 12

**Test-Double** haben die Aufgabe, komplexe Objekte in einem isolierten Test zu simulieren, in dem statt komplexer Berechnungen oder externer Zugriffe konstante Werte geliefert werden. Vertreter dieser Test Doubles sind die Mocks und Stubs, siehe Abschnitt 2.4. 12, 85, 103

**Test-Runner** oder Testtreiber) ist eine Software, um <sup>†</sup>Tests aufzurufen und deren Ausführung zu überwachen und zu steuern. 13, 103

**Test-Suite** ist eine Gruppe von mehreren Testfällen für eine Komponente oder für das gesamte System. 16

**Test-Umgebung** ist eine spezielle Konfiguration der Hard- und Software, um Tests unter kontrollierten und bekannten Bedingungen auszuführen. Dies beinhaltet neben dem zu testenden Objekten (Object under Test) auch sogenannte <sup>†</sup>Test-Doubles und den <sup>†</sup>Test-Runner. 13

**Testabdeckung** auch: Testausführungsabdeckung, Überdeckungsgrad, Testfallabdeckung. Eine dynamische Code-Metrik, die angibt, welche Codezeilen durch keinen Test abgedeckt wurden. Es wird unterschieden in die Stufen C0, C1 und C2 mit steigender Komplexität der Messung. Details im Abschnitt 5.2. 7, 50, 54, 83

**Testfall** oder ist eine, meist automatisierte, Prüfung des Programmverhaltens bei definierten Eingabeparametern. 8, 17, 103

**Testgetriebene Entwicklung** englisch Test Driven Development/Test Driven Design, Ausführlich dargelegt in Abschnitt 3. 16, 20, 21, 24, 49, 52, 55

# Abbildungsverzeichnis

1.1. Funktionsweise des Empfehlungscode . . . . .	3
1.2. Anwendungsfälle . . . . .	6
2.1. Einteilung der Tests . . . . .	11
3.1. Red-Green-Refactor: Der TDD Entwicklungszyklus . . . . .	18
3.2. Flussdiagramm für TDD . . . . .	19
3.3. Entwicklungsablauf . . . . .	20
4.1. MVC-Modell von Rails am Beispiel einer Anfrage /jobs/12 . . . . .	34
4.2. Ablauf beim Akzeptanztest mit Cucumber und Capybara . . . . .	43
7.1. Attribute des Modells „Job“ . . . . .	56
7.2. Funktionsweise von save bei ActiveRecord Objekten . . . . .	59
7.3. Aufbau eines Cucumber- Verzeichnisses . . . . .	75
8.1. Entwicklung des Umfangs des Test- und Programmcodes . . . . .	83
8.2. CO-Testabdeckung über den Verlauf der Entwicklung . . . . .	84
8.3. Verlauf der verschiedenen Klassen an Code-Smells über den Entwicklungs- zeitraum . . . . .	85
8.4. Vergleich des Verhältnisses aus Anzahl Testcodezeilen / Anzahl Codezeilen .	89
8.5. Vergleich der durchschnittlichen Komplexität . . . . .	90
8.6. Vergleich der Anzahl Smells pro KLOC . . . . .	91

# Listings

2.1. Beispiel für den Einsatz eines Stubs in einem Bestellprozess . . . . .	14
2.2. Beispiel für den Einsatz eines Mocks zum Test eines Netzwerkzugriffes . . .	14
4.1. Ruby Beispiele . . . . .	27
4.2. Ruby Beispiel: Blöcke . . . . .	28
4.3. Klassenhierarchien . . . . .	29
4.4. Ruby Beispiel offene Klassen . . . . .	29
4.5. Demonstration von generischen Funktionen bei ActiveRecord . . . . .	30
4.6. Nutzung von Metaprogrammierung zur Erstellung von Objektbeziehungen .	30
4.7. Testen mit Test/Unit in Ruby . . . . .	40
4.8. Cucumber: Definition eines Additionsfeature (in Deutsch) . . . . .	41
4.9. Cucumber: Implementierung der Additionstestschritte in Ruby . . . . .	42
7.1. Shell . . . . .	57
7.2. Shell . . . . .	58
7.3. Standardtest generiert durch Rails . . . . .	58
7.4. Test auf Vorhandensein eines Titels . . . . .	59
7.5. Implementierung der Validierung in die Klasse Job . . . . .	60
7.6. refaktorisierter Test . . . . .	60
7.7. Auslagerung des Regulären Ausdrucks in einen Initialisierer . . . . .	60
7.8. Alle bisherigen Testmethoden in der Klasse JobTest . . . . .	61
7.9. Fixtures Testdaten für zwei Jobs . . . . .	62
7.10. Finale Job-Test Klasse nach Refaktorisierung . . . . .	62
7.11. JobsController-Test: Gast Nutzer darf Stellen betrachten . . . . .	63
7.12. JobsController: Erfüllung des Tests durch eine Fake-Implementierung . . . .	63
7.13. JobsController: Ersetzung der Fake-Implementierung . . . . .	64
7.14. JobsController-Test: Gast Nutzer dürfen nicht-sichtbare Stellen nicht betrachten	64
7.15. JobsController: Weiterleitung, falls ein Job nicht sichtbar ist . . . . .	64
7.16. JobController-Test: Neuer Testfall . . . . .	65
7.17. JobsController: Implementierung der Weiterleitung, falls Stelle unsichtbar .	65
7.18. Job: Einführung einer Methode zur Bestimmung der Sichtbarkeit eines Jobs im Job-Modell . . . . .	65
7.19. JobsController: Anwendung der Refaktorisierung . . . . .	66

7.20. ApplicationController: Einführung einer controller-globalen Methode <code>current_user</code> . . . . .	66
7.21. JobsController: Nutzung der neuen Methode <code>current_user</code> . . . . .	66
7.22. Feedimport Beispiel-XML Datei mit einem Job . . . . .	68
7.23. Feed Test I. . . . .	69
7.24. Feed Implementation I. . . . .	69
7.25. Feed Test II . . . . .	70
7.26. Feed Implementation II . . . . .	70
7.27. Zentrale Implementierung des Mocks in der Test Helper . . . . .	71
7.28. Feed Test IIb nach Refaktorisierung . . . . .	71
7.29. Feed Test III . . . . .	72
7.30. Feed Implementation III - Fake Implementierung . . . . .	72
7.31. Feed Test IV – Test auf nicht-erfolgreichen HTTP-Status Code . . . . .	72
7.32. Feed Implementation IV - Implementation der Validierung . . . . .	72
7.33. Feed Test V - Testen der Robustheit gegen Exceptions . . . . .	73
7.34. Feed Implementation V . . . . .	73
7.35. Feed Implementation Vb - nach Refaktorisierung . . . . .	74
7.36. Definition eines Szenarios in einem Cucumber-Feature . . . . .	75
7.37. Implementierung der Testschritte zur Interaktion mit einem Browser . . . . .	77
7.38. Testschritt zum Anlegen von Testdaten auf Basis der Cucumber Definition . . . . .	77
7.39. Implementation eines Suchfeldes im Layout der Webseite . . . . .	78
7.40. JobsController am Ende der Testphase . . . . .	78
7.41. View-Implementation für eine Liste von Jobs . . . . .	79
7.42. Auslagerung der Suchlogik in die Job-Klasse . . . . .	79
7.43. Finaler Jobcontroller nach Refaktorisierung . . . . .	79
 B.1. RAILS_ROOT/Gemfile . . . . .	 97
B.2. RAILS_ROOT/Guardfile . . . . .	98
B.3. features/support/env.rb . . . . .	98



# Stichwortverzeichnis

## – A –

ActiveRecord .... 30, 33, 36, 37, 56, 57, 60, 78, 87  
Akzeptanztest 10, 15, 24, 39, 42, 51, 54, 55, 74, 75, 79, 94

## – B –

Browser  
    Automatisierter ..... 15, 42  
    Simulierter .... 8, 15, 42, 51, 53, 77, 80

## – C –

Capybara ..... 42, 51, 53, 74, 76, 80  
Code-Generator ..... 56, 83, 84  
Code-Metrik 1, 8, 38, 44, 46, 49, 54, 83, 84, 93, 94, 103  
Code-Smell ..... 45, 46, 54, 84, 85, 91, 101  
Cucumber .. 39, 41, 42, 51, 53, 74–77, 79, 99  
    Testschritt ..... 41, 42, 53, 75–77, 79

## – D –

Datenbank 7, 13, 30, 33–37, 40, 51, 55–59, 61, 63, 78, 82, 87, 102

## – G –

Gem ..... 35, 38, 41, 54, 102

## – I –

IT-Jobs-Projekt 5, 7, 68, 81, 82, 86, 90–92, 94

## – J –

JavaScript .. 7, 15, 34, 36, 37, 53, 55, 80, 81

## – R –

Refaktorisierung . 16–18, 20, 21, 23, 49, 60–62, 64, 74, 83, 85, 91, 94, 101

Ruby .. 13, 16, 26–31, 35–37, 39, 40, 42, 45, 47, 48, 53, 68, 70, 71, 73, 86, 95

Rake ..... 57, 58, 86

Ruby-on-Rails 1, 4, 7, 8, 30, 33–39, 48, 50, 52, 53, 55–58, 60, 61, 63, 66, 68, 78, 80, 82, 83, 85–88, 97, 102

Controller . 34, 35, 38, 55, 62–67, 78, 79

Modell ..... 30, 33, 38, 41, 55–58, 66

Validierung ... 33, 38, 40, 55, 56, 58–61

View ..... 34, 35, 38, 63, 66, 78

## – S –

Selenium ..... 15, 42, 53, 80, 97, 99

## – T –

TDD 1, 5, 7, 8, 10, 12, 15, 17, 20–22, 24, 25, 48, 50–52, 54, 55, 61, 62, 69, 74, 83–85, 90, 93–96, 101

Test . 2, 10, 13–15, 17, 18, 21, 40, 42, 45–49, 51, 53, 57–63, 65, 69–72, 75, 78, 83, 94, 96, 102, 103

Akzeptanztest ..... 22, 24, 25, 80

Test-Suite ..... 24, 47, 51, 58, 75, 96

Testabdeckung . 38, 47–50, 54, 83, 84, 91, 103

Testdaten 9, 13, 38, 40, 41, 57, 58, 61, 77

Unittest .. 7, 10, 12, 13, 15, 20, 24, 25, 38, 51, 55, 63, 80, 81

Test-Double ..... 12, 85, 103

Mock ..... 14, 38, 69–71, 73, 74, 102

Stub ..... 13, 14, 67, 102

Test/Unit-Framework ... 13, 36, 39, 42, 53

# Literaturverzeichnis

- [37s11] 37SIGNALS: *Ruby on Rails: Applications*.  
<http://rubyonrails.org/applications>. Version: 2011, Abruf:  
15.09.2011
- [BCSV11] BAGGEN, Robert ; CORREIA, José P. ; SCHILL, Katrin ; VISSER, Joost:  
Standardized code quality benchmarking for improving software  
maintainability. In: *Software Quality Journal* (2011), Mai.  
<http://dx.doi.org/10.1007/s11219-011-9144-9>. – DOI  
10.1007/s11219-011-9144-9. – ISSN 0963-9314
- [Bec02] BECK, Kent: *Test Driven Development. By Example*. Addison-Wesley Longman,  
Amsterdam, 2002. – ISBN 0321146530
- [Bro08] BROWN, Roger: *Test Driven Development and Flow*. [http://www.  
agilecoachjournal.com/post/Test-Driven-Development.aspx](http://www.agilecoachjournal.com/post/Test-Driven-Development.aspx).  
Version: April 2008, Abruf: 07.08.2011
- [CAH<sup>+</sup>10] CHELIMSKY, David ; ASTELS, Dave ; HELMKAMP, Bryan ; NORTH, Dan ;  
DENNIS, Zach ; HELLESØY, Aslak: *The RSpec Book: Behaviour Driven  
Development with Rspec, Cucumber, and Friends*. 1. Pragmatic Bookshelf, 2010.  
– ISBN 9781934356371
- [Can10] CANGIANO, Antonio: *The Great Ruby Shootout (July 2010)*.  
[http://programmingzen.com/2010/07/19/  
the-great-ruby-shootout-july-2010/](http://programmingzen.com/2010/07/19/the-great-ruby-shootout-july-2010/). Version: Juli 2010, Abruf:  
01.09.2011
- [Car08] CAROLI, Paulo: *Agile Tips: Testing private methods, TDD and Test-Driven  
Refactoring*. [http://agiletips.blogspot.com/2008/11/  
testing-private-methods-tdd-and-test.html](http://agiletips.blogspot.com/2008/11/testing-private-methods-tdd-and-test.html). Version: November  
2008, Abruf: 15.09.2011
- [CD10] CLARK, Kevin ; DAVIS, Ryan: *Confessions of a Ruby Sadist - heckle*.  
<http://ruby.sadi.st/Heckle.html>. Version: 2010, Abruf: 10.09.2011

- [Coo11] COOPER, Peter: *Rails 3.1 Adopts CoffeeScript, jQuery, Sass and.. Controversy*.  
<http://www.rubyinside.com/rails-4669.html>. Version: April 2011,  
 Abruf: 15.09.2011
- [Cor96] CORNETT, Steve: *Code Coverage Analysis*.  
<http://www.bullseye.com/coverage.html>. Version: 1996, Abruf:  
 08.08.2011
- [DN08] DEVRIES, Derek ; NABEREZNY, Mike: *Rails for PHP Developers*. 1. Pragmatic  
 Bookshelf, 2008. – ISBN 1934356042
- [Els07] ELSSAMADISY, Amr: *InfoQ: 100% Test Coverage?*  
[http://www.infoq.com/news/2007/05/100\\_test\\_coverage](http://www.infoq.com/news/2007/05/100_test_coverage).  
 Version: Mai 2007, Abruf: 19.08.2011
- [Eva03] EVANS, Eric: *Domain-Driven Design: Tackling Complexity in the Heart of  
 Software*. 1. Addison-Wesley Professional, 2003. – ISBN 0321125215
- [FBB<sup>+</sup>99] FOWLER, Martin ; BECK, Kent ; BRANT, John ; OPDYKE, William ; ROBERTS,  
 Don: *Refactoring: Improving the Design of Existing Code*. 1. Addison-Wesley  
 Professional, 1999. – ISBN 0201485672
- [FD09] FOWLER, Chad ; DAVE THOMAS, Andrew H.: *Programming Ruby 1.9: The  
 Pragmatic Programmers' Guide*. 3rd. Pragmatic Bookshelf, 2009. – ISBN  
 9781934356081
- [For10] FORD, Neal: *Vortrag: Emergent Design And Evolutionary Architecture*.  
<http://www.thoughtworks.com/emergent-design>. Version: März  
 2010, Abruf: 01.09.2011
- [Fri05] FRITZSCHE, Prof. H.: *Software-Qualitätssicherung – Skript*.  
[http://www.informatik.htw-dresden.de/~fritzsche/QSM/qsm\\_](http://www.informatik.htw-dresden.de/~fritzsche/QSM/qsm_script.html#k1)  
[script.html#k1](http://www.informatik.htw-dresden.de/~fritzsche/QSM/qsm_script.html#k1). Version: November 2005, Abruf: 05.10.2011
- [Gam11] GAME, Computer Language B.: *Ruby 1.9 Speed - C/GNU GCC speed*.  
[http://shootout.alioth.debian.org/u32q/benchmark.php?](http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=yarv&lang2=gcc)  
[test=all&lang=yarv&lang2=gcc](http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=yarv&lang2=gcc). Version: Juni 2011, Abruf:  
 01.09.2011
- [Goo06] GOODLIFFE, Pete: *Code Craft: The Practice of Writing Excellent Code*. 1st ed. No  
 Starch Press, 2006. – ISBN 1593271190
- [HA05] HULKKO, Hanna ; ABRAHAMSSON, Pekka: A multiple case study on the impact  
 of pair programming on product quality. In: *Proceedings of the 27th  
 international conference on Software engineering - ICSE '05*. St. Louis, MO,  
 USA, 2005, 495

- [Hai10] HAINES, Kirk: *Ruby Scales, AND It's Fast – If You Do It Right!* | Engine Yard Ruby on Rails Blog. <http://www.engineyard.com/blog/2010/architecture-wins-varnish-and-more/>. Version: März 2010, Abruf: 15.09.2011
- [Han11] HANSSON, David H.: *RailsConf 2011 Keynote*. <http://www.rubyinside.com/dhh-4769.html>. Version: Mai 2011, Abruf: 01.09.2011
- [Her05] HERRMANN, Dr. C.: *Metaprogrammierung*. In: *Funktionale Programmierung WS2005/2006* (2005). <http://www.infosun.fim.uni-passau.de/cl/lehre/sips1-ss06/Uebung/Metaprogrammierung.pdf>
- [HQ10] HQ, Selenium: *Selenium Contributors*. <http://seleniumhq.org/about/contributors.html>. Version: 2010, Abruf: 09.09.2011
- [HT09] HANSSON, David H. ; THOMAS, Dave: *Agile Web Development with Rails, Third Edition*. Third Edition. Pragmatic Bookshelf, 2009. – ISBN 1934356166
- [IEE98] IEEE: IEEE 1061-1998 Standard for a Software Quality Metrics Methodology. In: *IEEE Computer Society* (1998), Nr. 31 Dec. 1998
- [Inc07] INC., Stelligent: *Stelligent Survey Reveals Majority of Organizations Do Not Practice Test-Driven Development*. [http://www.businesswire.com/portal/site/google/index.jsp?ndmViewId=news\\_view&newsId=20071127006009&newsLang=en](http://www.businesswire.com/portal/site/google/index.jsp?ndmViewId=news_view&newsId=20071127006009&newsLang=en). Version: November 2007, Abruf: 09.09.2011
- [JS08] JANZEN, David ; SAIEDIAN, Hossein: Does Test-Driven Development Really Improve Software Design Quality? In: *IEEE Software* 25 (2008), Nr. 2, S. 77–84. – ISSN 0740–7459
- [Klu11a] KLUKAS, Jörg: *Referenzen* | pludoni GmbH - the community experts. <http://www.pludoni.de/referenzen>. Version: Juni 2011, Abruf: 01.09.2011
- [Klu11b] KLUKAS, Jörg: *Startseite - Aktuelles*. <http://www.itmitte.de/>. Version: Juni 2011, Abruf: 01.09.2011
- [Lig90] LIGGESMEYER, Peter: *Modultest und Modulverifikation. State of the Art*. BI-Wiss.-Verl., 1990 (Angewandte Informatik 4). – ISBN 3411143614
- [LO11] LANGR, Jeff ; OTTINGER, Tim: *Agile in a Flash: Speed-Learning Agile Software Development*. Crds. Pragmatic Bookshelf, 2011. – ISBN 1934356719

- [Mad09] MADEYSKI, Lech: *Test-Driven Development: An Empirical Evaluation of Agile Practice*. 1st Edition. Springer, 2009. – ISBN 9783642042874
- [McC76] MCCABE, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* (1976), Dezember, 308–320.  
<http://www.literateprogramming.com/mccabe.pdf>
- [MD05] MEIJER, E. ; DRAYTON, P.: Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, 2005
- [Mü06] MÜLLER, Matthias M.: The Effect of Test-Driven Development on Program Code. In: *PROC. INT'L CONF. EXTREME PROGRAMMING AND AGILE PROCESSES IN SOFTWARE ENG. (XP 06)* (2006), 94—103. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.8244>
- [NMBW08] NAGAPPAN, Nachiappan ; MAXIMILIEN, E ; BHAT, Thirumalesh ; WILLIAMS, Laurie: Realizing quality improvement through test driven development: results and experiences of four industrial teams. In: *Empirical Software Engineering* 13 (2008), Juni, Nr. 3, 289–302.  
<http://dx.doi.org/10.1007/s10664-008-9062-z>. – ISSN 1382–3256
- [Ors07] ORSINI, Rob: *Rails Cookbook (Cookbooks)*. 1st Ed. O'Reilly Media, 2007. – ISBN 0596527314
- [Ous98] OUSTERHOUT, J. K.: Scripting: Higher level programming for the 21st century. In: *Computer* 31 (1998), Nr. 3, S. 23–30
- [Pal06] PALERMO, Jeffrey: *Guidelines for Test-Driven Development*. [http://msdn.microsoft.com/en-us/library/aa730844\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx).  
Version: Mai 2006, Abruf: 22.08.2011
- [Pow08] POWELL, Catherine: *Abakas: Code Coverage Complexity*. <http://blog.abakas.com/2008/04/code-coverage-complexity.html>.  
Version: April 2008, Abruf: 07.08.2011
- [Rap11] RAPPIN, Noel: *Rails Test Prescriptions*. 1. Pragmatic Bookshelf, 2011. – ISBN 9781934356647
- [Ree92] REEVES, Jack W.: Three Essays by Jack W. Reeves - I. What Is Software Design? In: *C++ Journal* (1992), Nr. Herbst 1992. [http://www.developerdotstar.com/mag/articles/reeves\\_design.html](http://www.developerdotstar.com/mag/articles/reeves_design.html)
- [Rub11] RUBY VISUAL IDENTITY TEAM: *About Ruby*.  
<http://www.ruby-lang.org/en/about/>. Version: Juni 2011, Abruf: 01.09.2011

- [Rut10] RUTHERFORD, Kevin: *Code Smells - Reek*.  
<https://github.com/kevinrutherford/reek/wiki/Code-Smells>.  
 Version: 2010, Abruf: 09.08.2011
  
- [Sav07] SAVOIA, Alberto: *The Code C.R.A.P. Metric Hits the Fan - Introducing the crap4j Plug-in*.  
<http://www.artima.com/weblogs/viewpost.jsp?thread=215899>.  
 Version: Oktober 2007, Abruf: 07.08.2011
  
- [SH10] SPIEWAK, Daniel ; HARROP, Jon: *Dynamic type languages versus static type languages - Stack Overflow*.  
<http://stackoverflow.com/questions/125367/>. Version: 2010,  
 Abruf: 01.09.2011
  
- [Sho07] SHORE, James: *The Art of Agile Development*. 1. O'Reilly Media, 2007. – ISBN 0596527675
  
- [SP08] SKEET, Jon ; PREUSS, Ilja: *testing - Is there a difference between TDD and Test First Development (or Test First Programming)? - Stack Overflow*.  
<http://stackoverflow.com/questions/334779/>. Version: 2008,  
 Abruf: 01.09.2011
  
- [SR10] STEPHENS, Matt ; ROSENBERG, Doug: *Design Driven Testing: Test Smarter, Not Harder*. Apress, 2010. – ISBN 9781430229438
  
- [TH99] THOMAS, David ; HUNT, Andrew: *The Pragmatic Programmer: From Journeyman to Master*. 1. Addison-Wesley Professional, 1999. – ISBN 020161622X
  
- [Van10] VANDERBURG, Glenn: *Vortrag: Real Software Engineering*.  
<http://confreaks.net/videos/282-lsrc2010-real-software-engineering>. Version: August 2010,  
 Abruf: 01.09.2011
  
- [WG07] WASMUS, Hans ; GROSS, Hans: *Evaluation of Test Driven Development*. In: *Technical Report Series* (2007), Nr. TUD-SERG-2007-014.  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.100.6964>

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich erkläre mich einverstanden, dass meine Diplomarbeit an Personen, die nicht mittelbar oder unmittelbar an meiner Prüfung beteiligt sind, ausgeliehen wird.

Dresden, 6. Oktober 2011 \_\_\_\_\_

Unterschrift (Stefan Wienert)