HTW Dresden
Fakultät Informatik/Mathematik
Diplomarbeit

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: 29. September 2011

Inhaltsverzeichnis

1. Einleitung **1**

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

2. Automatisierte dynamische Softwaretests **9**

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 14

3. Testgetriebene Entwicklung **16**

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 Sofwaredesign 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

4. Die Programmiersprache Ruby **26**

4.1. Einführung in Ruby 26
4.2. Diskussion 30
4.3. Ruby on Rails 32
4.3.1. Konzepte von Rails 32
4.3.2. Diskussion 35
4.4. Testframeworks für Ruby 38
4.4.1. Test/Unit 38

4.4.2. Cucumber 40

5. Code-Metriken **43**
5.1. Überblick über Code-Metriken und Skalen 43
5.2. Code-Metriken für Tests 45
5.2.1. Verhältnis von Lines of Test zu Lines of Code 45
5.2.2. Testausführungsabdeckung 46
5.2.3. Mutations/Pertubationstests – Defect insertion 47
5.3. Notwendigkeit von Code Metriken 48

6. Entwicklungsmethodik und -Werkzeuge **49**
6.1. Definition eines Entwicklungsmodells für die Bedürfnisse der pludoni GmbH 49
6.1.1. Einteilung der Features in Kategorien 50
6.1.2. Praktiken 50
6.2. Auswahl der Entwicklungswerkzeuge 52
6.3. Diskussion 53

7. Anwendung der Testgetriebenen Entwicklung **54**
7.1. Implementierung von Unit-Tests (Modelltests) 54
7.2. Implementierung von Controller-Tests (functional tests) 61
7.3. Testen von externen Abhängigkeiten 65
7.4. System/Akzeptanztests 73
7.5. Testen von JavaScript 78

8. Auswertung **80**
8.1. Entwicklungsstand IT-Jobs 80
8.2. Code-Quality-Benchmark mit anderen Ruby-Projekten 84
8.2.1. Vergleich von IT-Jobs mit eigenen Projekten 84
8.2.2. Vergleich von IT-Jobs mit anderen Rails-Projekten 85
8.2.3. Auswertung und Visualisierung 86

9. Fazit **90**
9.1. Ausblick 91

A. Eigenschaften erfolgreicher Tests **92**

B. Nutzung von Cucumber in Verbindung mit Selenium für Firefox und Guard ohne X-Server **93**

Abbildungsverzeichnis **96**

Quellcode-Listings **96**

Stichwortverzeichnis **99**

Literaturverzeichnis **100**

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 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.

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, 29. September 2011 _____

Unterschrift (Stefan Wienert)

Glossar

Im Folgenden werden einige oft verwendete Begriffe näher erläutert. Innerhalb des Hauptteils dieser Arbeit sind diese Begriffe mit einem ↑ 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. VI, 37, 78

CMS Content Management System (Inhaltsverwaltungssystem) ist eine, meist webbasierte Software, die es Nutzern ermöglicht einfach statische Inhalte anzulegen und zu bearbeiten. VI, 65, 80

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 ein Refaktoring. Informationen zu Smells und deren Refaktorisierung sind im Buch von M.Fowler zu finden [FBB+99]. VI, VII, 17, 44, 51

Code-Metrik Eine Softwaremetrik ist das Ergebnis einer statischen oder dynamischen Codeanalyse zur Generierung von Informationen über den Quelltext. Beispiele: Testabdeckung, Anzahl Codezeilen, Anzahl ↑Code Smells pro Codezeile. VI, 37, 43, 52

Code-Qualität beinhaltet die Software-Qualitäten Lesbarkeit, Testbarkeit, Wartbarkeit, Erweiterbarkeit, geringe Komplexität. VI

CRUD Create, Read, Update, Delete sind die vier Basisoperationen, die auf persistente Speicherung ausgeführt werden können.. VI, 33, 34, 49

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. VI, 38, 40

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). VI, VIII, 32

Gem als Gems (zu deutsch: Edelsteine) werden in der Ruby-Gemeinschaft Bibliotheken Dritter bezeichnet, die in einem zentralem Repository lagern. Das Bekannteste dieser

Repositorys ist rubygems.org. Nahezu alle Ruby-Bibliotheken lassen sich hier finden und innerhalb von Sekunden mittels des Kommandozeilenwerkzeuges gem installieren. Beispielsweise ließe sich ↑Ruby on Rails mittels `gem install rails` installieren. VI, 34, 37, 40, 52

MVC Model-View-Control, ist ein ↑Entwurfsmuster, das insbesondere bei GUI- und Web-Anwendungen beliebt ist. Rails basiert auf dem MVC-Muster. VI, 32, 54, 65

ORM Objektrelationales Mapping, ist eine Persistenztechnik, um Objekte transparent in einer Datenbank zu speichern und umgekehrt, Tabellenzeilen in Objekte wieder rückzuübersetzen. VI, 29, 32, 34, 55

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.. VI, 41, 52

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). VI, 56, 84

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. VI

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. VI, 7, 65

Ruby on Rails ist ein auf der Programmiersprache Ruby basierendes Web-Framework und ist Gegenstand des Kapitel 4.3 text. VI, VIII, 16, 30, 39, 41, 52

Software-Fehler oder Defekt, ist ein unerwartetes Verhalten der Software, der zu einem Versagen der Software führen kann. VI, 10, 12

Test oder Testfall ist eine, meist automatisierte, Prüfung des Programmverhaltens bei definierten Eingabeparametern. VI, VIII, 8, 17

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. VI, VIII, 12, 83

Test-Runner oder Testtreiber) ist eine Software, um ↑Tests aufzurufen und deren Ausführung zu überwachen und zu steuern. VI, VIII, 13

Test-Suite ist eine Gruppe von mehreren Testfällen für eine Komponente oder das für gesamte System. VI, 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 Tes) auch sogenannte ↑Test-Doubles und den ↑Test-Runner. VI, 13

Testabdeckung auch: Testausführungsabdeckung, Überdeckungsgrad, Testfallabdeckung. Eine dynamische Code-Metrik, die angibt, welche Codezeilen durch keinen Test abgedeckt wurde. Es wird unterschieden in die Stufen C0, C1 und C2 mit steigender Komplexität der Messung. Details im Abschnitt 5.2. VI, 7, 49, 52, 81

Testgetriebene Entwicklung englisch Test Driven Development/Test Driven Design, Ausführlich dargelegt in Abschnitt 3. VI, 16, 20, 21, 24, 48, 51, 54

Kapitel 1.

Einleitung

Testgetriebene Entwicklung...

Das Hauptthema dieser Diplomarbeit, die Testgetriebene Entwicklung, ist vielen Programmierern vom Namen her bekannt. Ungeachtet dessen, dass viele Studien über die (positiven) Auswirkungen über diesen Entwicklungsansatz 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[1].
Diese, teilweise mythenumrankte, Technik hat innerhalb des Entwicklerteams der pludoni GmbH ein großes Interesse geweckt und wird aus diesem Grund nun näher untersucht.

...von Webserveranwendungen...

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.

...auf Basis von Ruby on Rails

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.

Um die Auswirkungen der Testgetriebenen Entwicklung besser nachzuvollziehen 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 der Einsatz von Code-Metriken praktisch erfolgen, wird in dieser Arbeit im Detail betrachtet und mit eigenen praktischen Erfahrungen belegt.

[1] http://tamasgyorfi.wordpress.com/tag/excuses-to-tdd/

1.1. Motivation

Kurz nach der Firmengründung der pludoni GmbH 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[2] stark anstieg.

Da zum großen Teil keine automatisierten Softwaretests geschrieben wurden, lassen sich diese 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 (Spaghetticode), die schwer bis unmöglich zu testen sind. Hier müsste zuerst refaktorisiert 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 der Code-Qualität, um in Zukunft flexibel auf Änderungen reagieren zu können.

1.1.1. Die 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 zur Zeit ITsax.de, ITmitte.de und MINTsax.de[3].

pludoni GmbH

Funktionsweise der Communitys Die Communitys bestehen jeweils aus einer Anzahl mittelständischer Unternehmen einer Branche, Bildungseinrichtungen sowie Vertetern 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.

[2] fehlerauslösender Quelltextänderungen
[3] 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[4] 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 Empfehlungscodes

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 Infrastrukur zur gegenseitigen Empfehlung dieser guten

[4]Ein 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 Empfehlungscodes 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[5]. 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) 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.

Für das kommende Projekt soll nun eine großflächige Testinfrastruktur erstellt werden, um eine aktuelle Dokumentation des Quelltextes zu erhalten und die Code-Qualität verbessern und natürlich um das Risiko für Regressionsfehler zu minimieren.

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

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 vorraussichtliche Name wird IT-Jobs-Und-Stellen.de[6] 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[7]. Parallel zu dieser Arbeit wurden die Anforderungen analysiert und die in Abbildung 1.2 dargestellten Anwendungensfälle bestimmt.

1.2.1. Anwendungsfälle

Es gibt zwei verschiedene Nutzertypen. Zum einen, ein *Bewerber*, der das Ziel hat, einen Job zu suchen, zu finden und sich darauf zu bewerben. Zum anderen ein Mitarbeiter eines Kundenunternehmenes (*Arbeitgeber*), der Jobs 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 und 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 und Logo und Bild hinzufügen.
- Ein Arbeitgeber kann eine Stelle verlängern oder kopieren und als Vorlage für ein neues Stellenangebot nutzen.
- Ein Arbeitgeber kann eine laufende Stelle erneuern (refresh) und 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.

[6] http://www.it-jobs-und-stellen.de/
[7] http://www.stepstone.de - http://www.monster.de - http://www.jobscout24.de

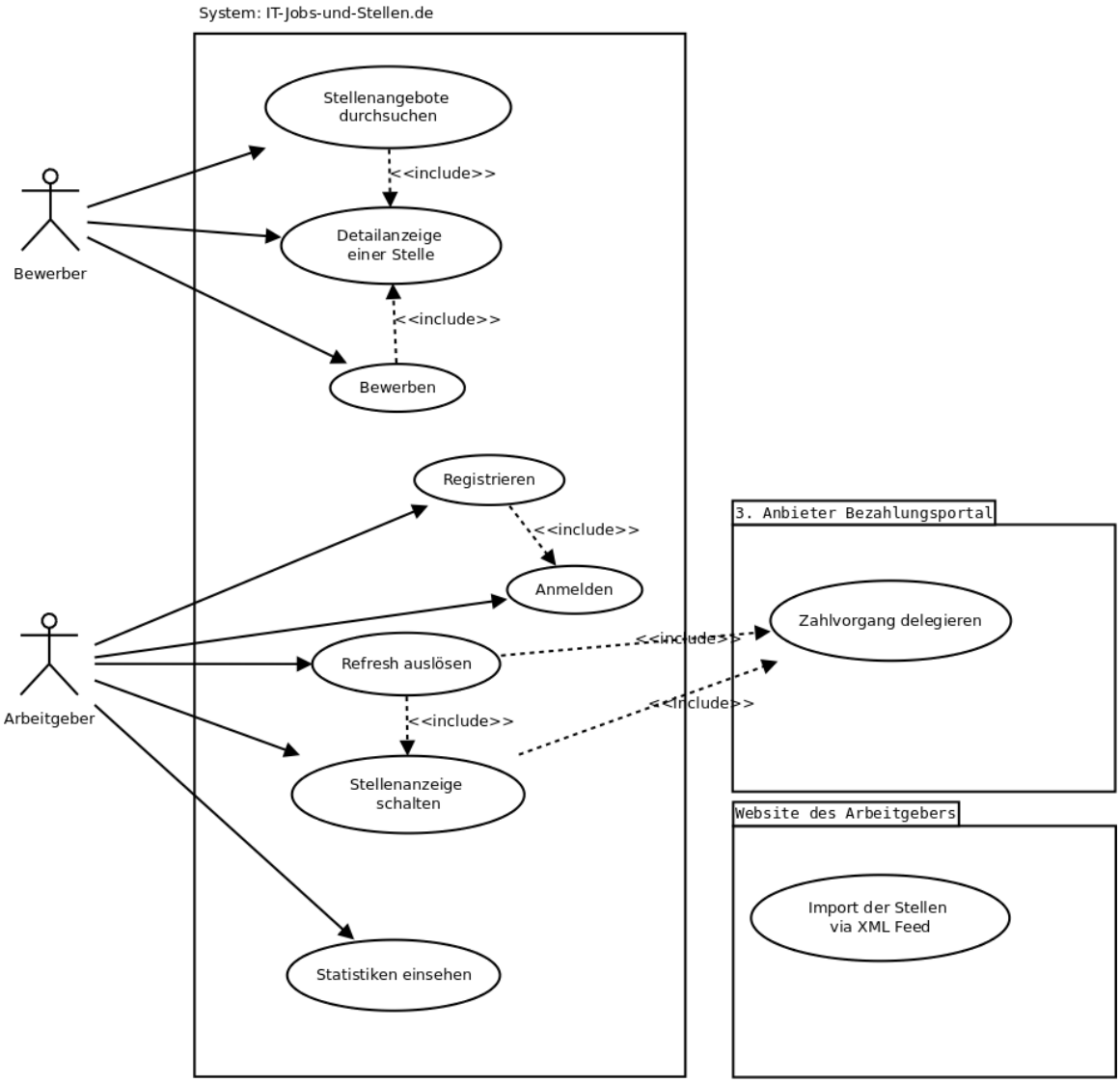


Abbildung 1.2.: Anwendungsfälle

- 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 6h automatisert 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 ↑RSS 2.0.

1.2.2. Nichtfunktionale Anforderungen

Folgende Rahmenbedingungen und zusätzliche Ziele wurden vereinbart.

- Eine hohe C0-↑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 Communityportale durch das neue System zu ersetzen, welche gegenwärtig auf dem Framework **Drupal** 5 (PHP) basieren
- Eine moderne Suchfunktion durch einen Suchdaemon, z.B. Sphinx oder Lucene
- 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, welches parallel zu dieser Arbeit enstand.

Da die Testgetriebene Softwareentwicklung auf dem dynamischen Unittest basiert, ist das Kapitel 2 – "*Automatisierte dynamische Softwaretests*" dem Testen im Allgemeinen gewidmet. 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 die Programmiersprache 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 kontrollieren.

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*" aus den 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 gibt es Hinweise für die Erstellung guter Tests aus der Literatur und die Anleitung, wie eine effektive Browsersimulationsumgebung eingerichtet wird. Am Ende der Arbeit erfolgt eine Auflistung der verwendeten Abbildungen, Quellcodes und das Literaturverzeichnis.

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 $^{\uparrow}$ markiert, z.B. $^{\uparrow}$Test.
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 macht jeder Programmierer mindestens manuelle Tests. 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].

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

Neben den **dynamischen Tests**, gibt es 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, um 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 preiswer-

ter [Lig90, S.13].
Außerdem gilt getester Code im Allgemeinen als robuster, korrekter und leichter zu warten [Rap11]. Im Umkehrschluss bedeutet dies, dass ungetestete Software mit sehr hoher Wahrscheinlichkeit ↑Software-Fehler beinhaltet [Goo06].

2.2. Arten von Tests

Tests können nach verschiedenen Gesichtspunkten eingeteilt werden. In vielen konkreten Fällen reich eine eindimensionale Einordnung nicht aus und Test können ebenso Teil mehrerer Kategorien sein. Einige der gängisten 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, soviele Codeabschnitt wie möglich zu testen.
Blackbox-tests dagegen ignorieren den inneren Aufbau der Klassen 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. Diese werden ausführlich im nächsten Abschnitt behandelt

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 und Verifikation Testet den Fortschritt der Anwendung in Bezug auf die funktionalen Anforderungen. Dies ist meist ein Blackboxtest und testet das System als Ganzes (=Systemtest). Ein Spezialfall ist der **Akzeptanztest**. Hierbei nimmt der Kunde eine Anforderung/ein Feature ab. Auch diesem Testtyp ist ein Unterabschnitt gewidmet.

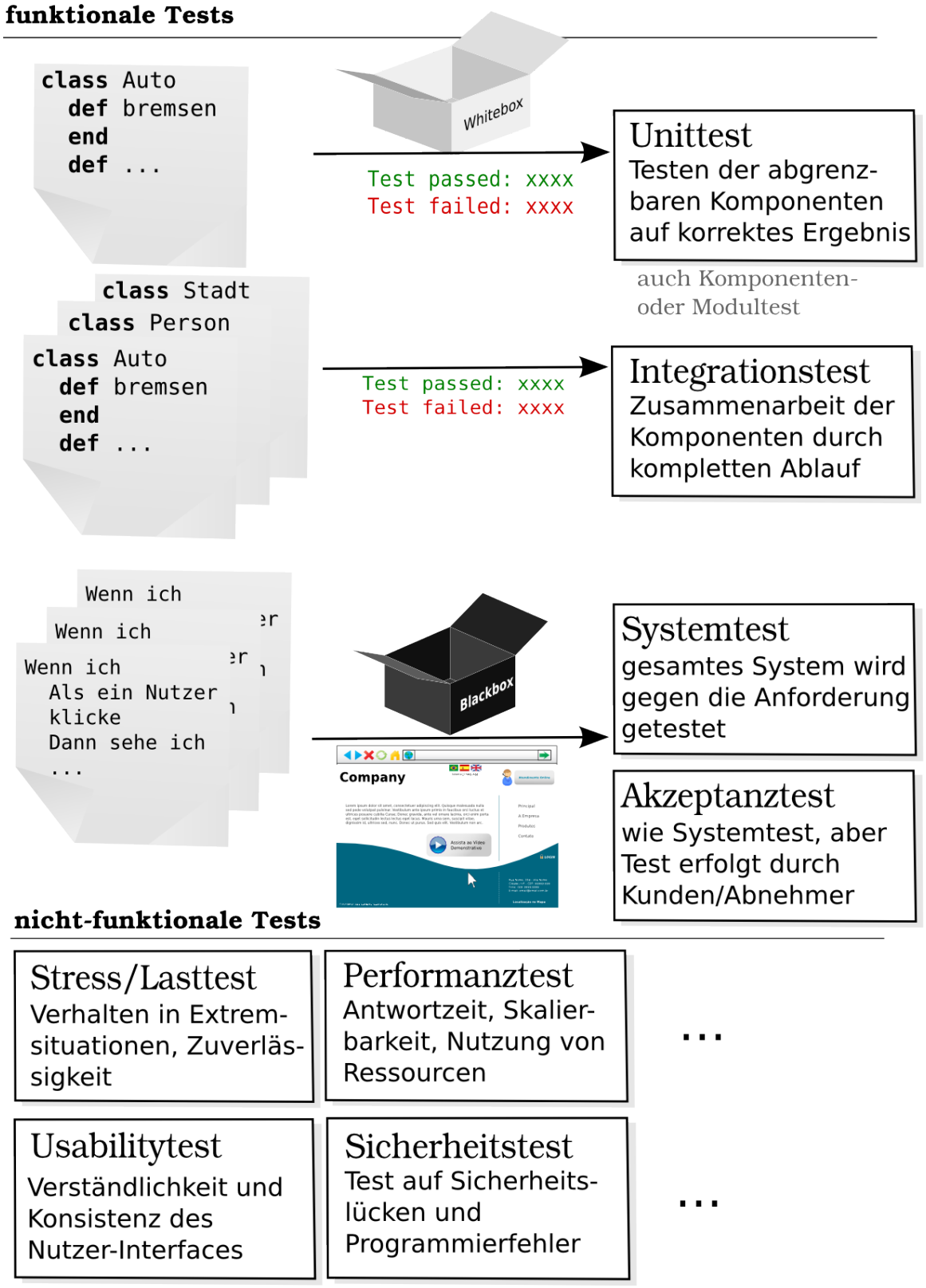


Abbildung 2.1.: Einteilung der Tests

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, dass 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 präperierten Umgebung (Usability Labor), geprüft.

Einteilung nach Testvollständigkeit Auf welche Art die sogenante Testvollständigkeit beurteilt wird und notwendige Testfälle generiert werden, ist ebenfalls eine Möglichkeit, Tests einzuordnen [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 gegeneinander. Dies beinhaltet z.B. Mutationentest, der später im Abschnitt 5.2.3 *Mutations/Pertubationstests – 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 ↑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 ↑Test-Doubles simuliert. Dies ist notwendig um sicherzustellen, dass gefundende Fehler von dem betreffenden Modul verursacht wurden und nicht durch äußere Einflüsse. Diese Isolierung macht das Testen einfacher [Goo06, Lig90].

Unittests werden fast immer durch sog. ↑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 solcher Test besteht in der Regel aus 4 Teilen [Rap11] [LO11, Karte 46]:

1. Initialisierung der ↑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 System under Test/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.

Komponententest Ziel des Komponententests ist es, verschiedene Units in Kombination als eine vollständige Komponente zu testen [Goo06].

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 Objekt, welches die 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**[1] 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 für unseren aktuellen Test keine Auswirkungen 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.

2.5. System- und Akzeptanztests

Der Unittest ist als Whitebox-Test auf Wissen über den Quelltext angewiesen und sein Zweck ist es, Fehlerfreiheit zu gewähren. 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 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

[1] http://mocha.rubyforge.org/

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**[2], 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].

[2] http://seleniumhq.org/

Kapitel 3.

Testgetriebene Entwicklung

> TDD is also good for geeks who form emotional attachments to code.
>
> [Bec02]

Extreme Programming ist eine Agile Software-Entwicklungs Methodik, welche das Ziel hat, die 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 bekannt geworden ist [Bec02].

Diese als Testgetrieben Entwicklung oder Test-Driven-Design 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 ↑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 ↑TDD näher beleuchtet und danach am Beispiel von Ruby/↑Ruby on Rails typische Testwerkzeuge aufzeigen.

3.1. Motivation

Das Erstellen einer gut-abdeckenden ↑Test-Suite ist für ein jedes größeres Softwareprojekt eine wichtige Voraussetzungen um interne Qualitäten, wie Wartbarkeit und Zuverlässigkeit zu aktivieren. ↑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 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 Co-

de ist [Bec02]. ↑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 beschreibt 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**[1] führen kann und damit eine effektivere Entwicklung ermöglicht [Bro08].

Falls TDD die Fehlerdichte 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 (Continuos 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 Unittest (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 ↑Test. Dies kann der erste eines neuen Features sein oder aber ein weiterer Test, um das aktuelle Feature umzusetzen.
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

[1] Schaffen-, Tätigkeitssrausch

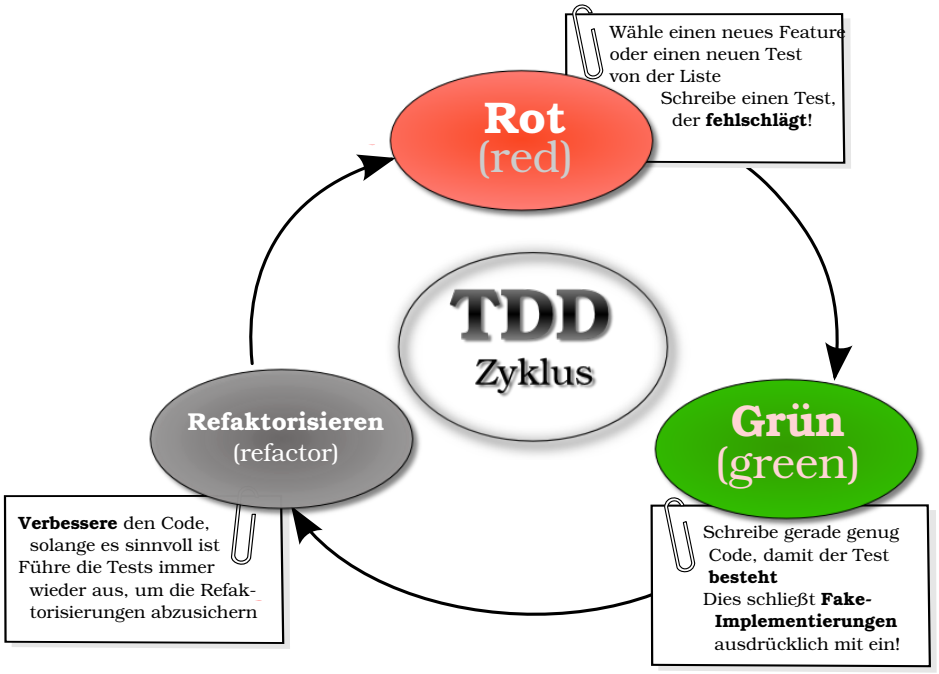


Abbildung 3.1.: Red-Green-Refactor: Der TDD Entwicklungszyklus

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.

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.

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 werden.

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 zustandegekommen sind.

It's about the design, not the tests.

Kent Beck

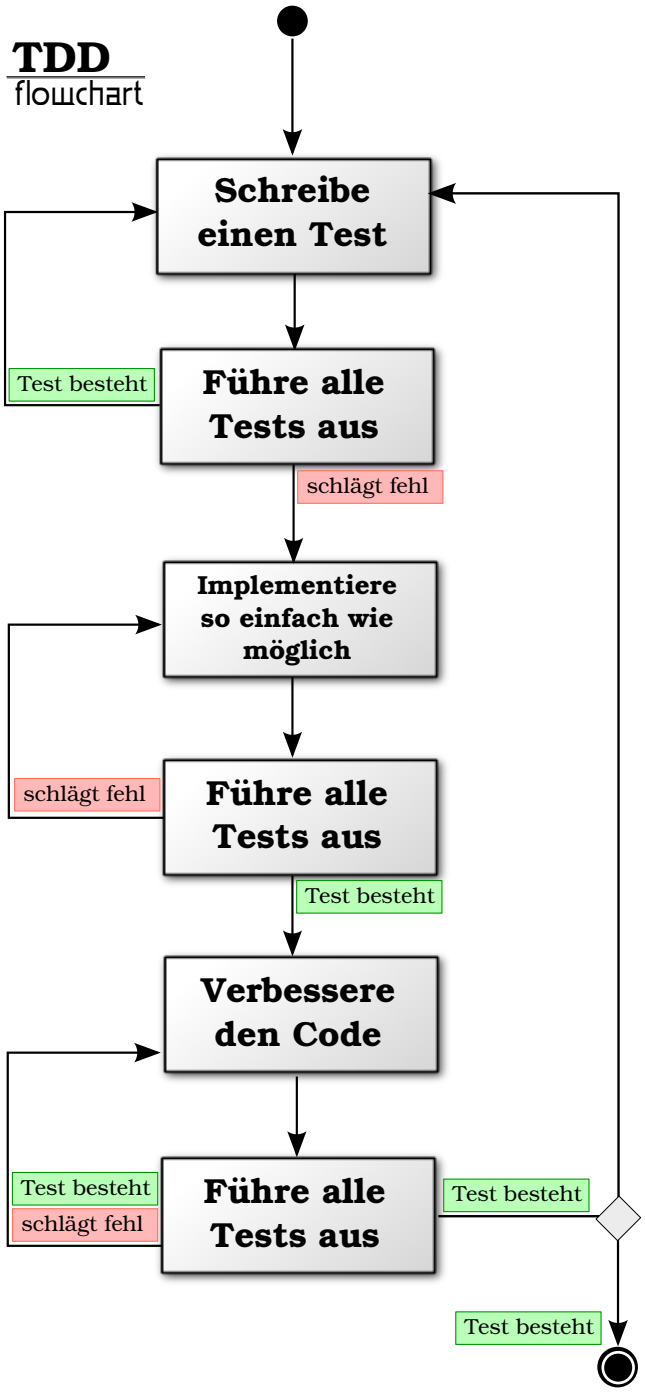


Abbildung 3.2.: Flussdiagram für TDD

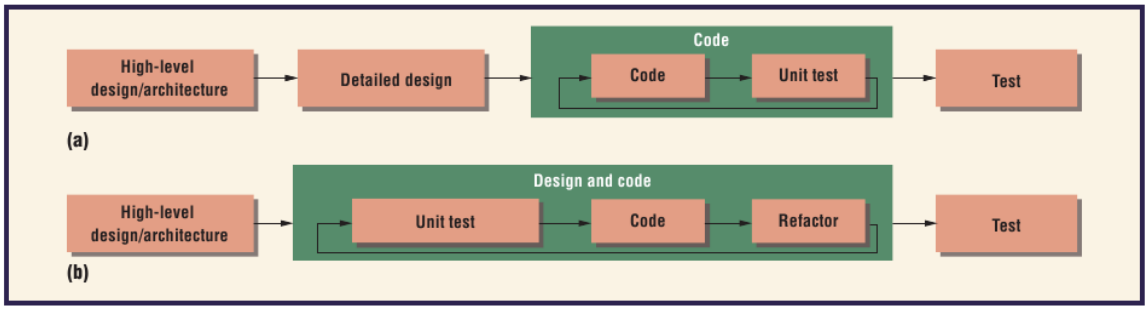


Abbildung 3.3.: Entwicklungsablauf

(a) Traditionelle Entwicklung, b) Testgetriebene Entwicklung

Quelle: [JS08]

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.

3.3. Sonderfälle

Der oben gezeigt 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 ↑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, das 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 ↑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 dann aber immer gelöscht werden und danach

in originärer TDD-Manier neu entwickelt werden. Dies soll auch die gewählte Metapher einer Nadel aufzeigen: Schnell eine Nadel durch ein Brett bringen [Sho07], als provisorische Lösung, die später durch eine besser designte ersetzt wird. Ausführliche Informationen über das Wann und Wie eines Einsatzes von Spikes finden sie in dem Buch „The Art of Agile Development“ [Sho07][2].

Testen von privaten Methoden/Attributen Da die objektorienterte 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 sich 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 enstanden 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 ↑TDD positive Auswirkungen auf die Software-Entwicklung hat.

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

In dem Artikel des IEEE-Software-Journals stellen Janzen und Saiedian eine Studie vor, die akademische und industrielle Javaprojekte, die testgetrieben durchgeführt wurden (test first), mit denen, bei denen hinterher getestet wurde (test last), vergleicht. 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[3] [Mü06]. Aller-

[2] Der Autor hat sogar das betreffende Kapitel online verfügbar gemacht http://jamesshore.com/Agile-Book/spike_solutions.html

[3] Dies ist ein Maß, inwieweit der lokale Zustand einer Klasse/Methode von außen durch Parameter beeinflusst werden kann

dings 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 schwierg 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 unterstützen, aber nur 16% TDD in der Praxis einsetzen und nur 21% Testvollständigkeit messen. Auch verstehen anscheinend etwa die Hälfte der Befragten den Begriff TDD falsch, nämlich als die reine 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 Sofwaredesign

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. 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.

Gründe für eine Minimierung des Designs

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 Entwick-

lung. 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 Diszplinen hätten außerdem die Beschränkung, dass ein Build-Prozess äußerst teuer ist (man denke den Bau einer Brücke oder die Entwicklung von Flugzeugen), wohingegen Softwareentwicklung 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[4] soll das Design durch Entdeckung und Extrahieren aus dem Sourcecode gewonnen werden.

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 größe 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 Refaktoriserungen 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 einem passenden Refaktorisierungszyklus 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 sollte man aber beim Einführen neuer Architekturelemente so konversativ 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 Refaktoriserungsphase vor und zusammen mit einer Paarprogrammierung bietet es damit perfekte Vorraussetzungen für ein inkrementelles Design.

[4] (BDUF – Big Design Upfront)

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 ↑TDD ihre Anwendung. Statt der Unittests stehen hier die Akzeptanztests im Vordergrund (vgl. 2.5 zum Thema Akzeptanztests).

Die Entwicklung von Unittests lässt sich aber einbetten, wodurch man nun zwei Testebenen erhält [CAH+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, verfahre 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 1.
   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+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+10, S. 138]. Im Grunde enstehen neue Funktionen auch hier durch vorhergegangene Tests. Allerdings bezieht sich das Vokular weniger auf die Domän „Test“, sondern sucht ein gemeinsames Vokabular für Kunden und Programmierer.

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

- Zusicherung (assertion) → Erwartung (expectation), 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 Vokular nutzen.

3.6.3. Design Driven Testing

Design Driven Testing soll eine Umkehrung von Testgetriebener Software sein und wird als Alternative dazu vorgeschlagen [SR10]. Sie 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 ist eine Multiparadigma-Sprache

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 Statements haben einen Rückgabewert. Innerhalb einer Funktion ist dies immer das letzte 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 (Syntaktischer Zucker), um dem Programmierer mehrere Wege zur Lösung seines Problems zu ermöglichen.

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

Yukihiro Matsumoto

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 =>.

Interaktive Ruby Sitzung (IRB)

```
>> 2.even?
=> true
>> "hallo".upcase
=> "HALLO"
>> Date.today + 2
=> #<Date: 2011-06-30>
>> a = 4 + Math.sqrt(9)
=> 7.0
>> if (0..10).include? a
>>   puts "a liegt zwischen 0 und 10"
>> end
=> a liegt zwischen 0 und 10
```

Listing: Ruby Beispiele

In den ersten beiden Beispielen sieht man, dass Integer und String Objekte sind und über Instanzmethoden verfügen. Im ersten Beispiel wird geprüft, ob die Zahl gerade ist. Dabei existiert eine Konvention, dass boolsche Methoden mit einem Fragezeichen am Ende notiert werden. Im Dritten Beispiel wird eine Klassenmethode `today` auf die Klasse `Date` ausgeführt, welche ein Datumsobjekt konstruiert und zurückliefert. Da auch die Nutzung von Operatoren letzendlich nur syntaktischer Zucker für Methodenaufrufe sind, wird die Instanz-Methode `.+()` auf dieses Datums-Objekt 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 seien 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.

IRB

```
>> adder = lambda { |a,b|  a + b }
>> adder.call(1,2)
=> 3
```

```
# Sortiere nach Standardvergleichsoperator
>> [4,5,7,3].sort()
=> [3, 4, 5, 7]

# Es kann auch eine benutzerdefinierte Sortierfunktion
# angegeben werden
>> [ "string",  "rails",  "ruby" ].sort_by{ |item| item.length }
=> ["ruby", "rails", "string"]

# Die Quadratzahlen von 1 bis 5
>> (1..5).map{|element| element * 2}
=> [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

IRB

```
>> 2.class
=> Fixnum
>> Fixnum.class
=> Class
>> Class.class
=> Class

>> Fixnum.superclass
=> Integer
>> Fixnum.ancestors
=> [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.

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 beinhalteten 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 seine Methoden und Attributen bestimmt wird.

"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

Ruby verfügt über lexikalische und dynamische Bindung[1], 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.

IRB

```
>> class String
>>   def remove_whitespace
>>     self.gsub(/\s+/, "")
>>   end
>> end

>> "Dies ist ein Test".remove_whitespace
=> "DiesisteinTest"
```

Listing: Ruby Beispiel offene Klassen

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[2]-Datenbankframework ActiveRecord, um einfache SQL-Statements anhand des

[1] **Static Scoping**: Variablen werden zur Compilezeit gebunden ohne den aufrufenden Code zu berücksichtigen
Dynamic Scoping: Variablen-Bindung kann nur im Moment der Ausführung des Codes festgestellt werden

[2↑] Objekt-relationales Mapping (ORM)

Funktionsnamens zu erstellen (Ruby on Rails verwendet standardmäßig ActiveRecord als Schnittstelle zur Datenbank).

IRB

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

Listing: Demonstration von generischen

Die Methode `find_by_first_name` existiert nicht und wird zur Laufzeit auf Basis des Namens gebaut. Dies ist möglich, da Ruby sogennante 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 z.B. auch `method_added`, der aufgerufen wird, wenn in einer Klasse eine Methode definiert wird[3]. Auf diese Weise sind z.B. die Modifikatoren `private` und `public` implementiert.

Ein weiteres Beispiel ist die Definition der relationalen Beziehungen zwischen den einzelnen Modellen innerhalb von ↑Ruby on Rails (Ebenfalls unter der Verwendung von ActiveRecord).

app/models/job.rb

```
class Job < ActiveRecord::Base
  belongs_to :user
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 aufzuzeigen. Für eine weitere Vertiefung sei das Buch „Programming Ruby 1.9“ empfohlen, das im Detail auf die neuste Version der Programmiersprache eingeht [FD09].

4.2. Diskussion

Dynamische Typisierung und Performanz

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, implementiert in verschiedenen Sprachen, miteinander. So ergibt sich z.B. in der Gegenüberstellung von Ruby mit C ein

[3] Einen guten Überblick über die Callbacks die Ruby bereitstellt und was man damit kann, finden sie hier: http://www.khelll.com/blog/ruby/ruby-callbacks/

4-fach bis 300-fach langsamere Ausführungszeit. Dem gegenüber steht allerdings nur die Hälfte bis $1/7$ der Menge an benötigtem Code [Gam11].
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. Desweiteren gibt es gerade im Bereich Laufzeitoptimierung viel Bewegung innerhalb der Ruby-Implementierungen und fast alle Ruby-Implementierungen nehmen stetig an Geschwindigkeit zu [Can10].

... und fehlende Typsicherheit

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.
Auf das Testen hat dies eine direkte Auswirkung. Viele Meinungen belegen, dass eine dynamisch typisierte Sprache mehr Tests benötigt, als eine statisch typisierte [SH10].

Vorteile aus der dynamischen Typisierung

Ein Vorteil des Interpretierens, also der Übersetzung zur Laufzeit, ist eine hohe Plattformunabhängigkeit und ein leichterer 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].

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 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 2006 von der Firma 37signals[4] 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.

Viele professionelle Websites, die meist als Startup begannen, setzen bis heute auf Rails. Darunter Github, eine sehr beliebte Community für OpenSource Programmierer, Groupon, dem führenden Unternehmen bei Online-Gutscheinen, XING, einer deutschen 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, Nasa, Siemens oder Yahoo[5].

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 ↑Model-View-Controller (MVC) ↑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 ↑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 werden, 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.

[4] http://37signals.com/
[5] Weitere Firmen: http://www.workingwithrails.com/high-profile-organisations

Controller Klassen dieser Schicht vereinigen 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[6].
Standardmäßig stellt Rails die ↑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 bei einer bestimmten Aktion dem Clienten 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.

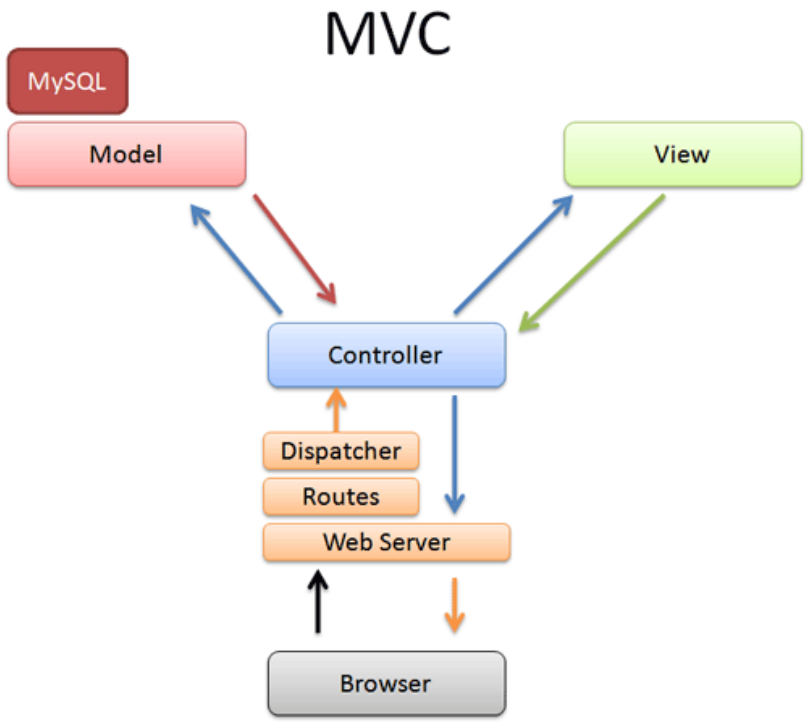


Abbildung 4.1.: MVC Modell von Rails

Quelle: betterexplained.com

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

[6]Authentifizierung: Wer ist der Nutzer?
Authorisierung: Was darf der Nutzer

Model Job stellen, den Job mit der ID 12 aus der Datenbank zu holen. Danach wird ein HTML Template zur Detailanzeige des Jobs generiert.

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

Convention over Configuration Rails ist so konzipiert, um als Framework komplett out-of-the-box zu funktionieren. Außer der Datenbankeinstellung wird keine Konfiguration im Vorfeld benötigt. Diese Methodologie zieht sich auch durch das Ökosystem von Ruby. Die meisten externen Bibliotheken, bei Ruby ↑Gems genannt, erschließen sich und funktionieren bereits binnen weniger Minuten. Dies macht das prototypische Entwickeln äußert 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 wiederrum 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 ↑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 benutzen Verben DELETE und PUT verwendet, um Aktionen auf eine Ressource zu definieren. Das Ziel ist ein sehr einfaches Design der URLs.

Eine Verwendung der ↑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[7] zu erstellen.

[7] Application Programming Interface – Eine Schnittstelle um extern mit der Anwendung zu kommunizieren

Der Scaffold (Gerüstbau)-Generator z.B. generiert alle notwendigen Elemente einer Ressource:

test/test_feed.rb

```
1 rails generate scaffold job title:string description:text \
2   start_date:datetime active:boolean user:references
```

Damit wird das Model Job, eine Migration zur Erstellung der Datenbanktabelle „jobs“, ein Controller „jobs“ mit den REST-Standardaktionen und entsprechenden Beispielviews sowie Testfälle für Unit- und Funktionale Tests angelegt.

Full-Stack Webframework Rails bringt out-of-the-box alles mit, was zur Webentwicklung benötigt wird. Im Gegensatz zu anderen Webframeworks wurde für Datenbankanbindung, Templatesystem, JavaScriptframework, 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 Teil-Frameworks lassen sich zwar leicht austauschen, Rails selbst aber behauptet „opinionated“, also rechthaberisch/eigensinnig, zu sein und den 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 zu empfehlen:

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 Sourcecode 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-Dokumentationen, die nahezu alle Aspekte von Rails beleuchten und anhang von praktischen Beispielen erläutert.
http://guides.rubyonrails.org/getting_started.html

4.3.2. Diskussion

Nach einem kurzen Überblick über Rails, sollen nun die Eigenschaften des Frameworks diskutiert werden und welche Auswirkungen sich dadurch auf das Testen ergibt.

Nachteile Oft wird angeführt, dass Ruby als Skriptsprache und Rails als darauf aufbauendes Framework eine schlechte Performanz hat und dadurch ungeeignet für große Weban-

Performanz

wendungen ist.
Anderseits existieren Erfahrungen, dass eine clevere Architektur und Caching für skalierende Anwendung entscheidender ist, als die letztendliche Ausführungszeit [Hai]. Das 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].

Auswahl der Komponenten

Da Rails ein komplettes Webframework ist, wurden bei der Auswahl der einzelnen Komponenten bereits Entscheidungen getroffen. Erst jüngst 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.

Integration in bestehende Software

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 ist, wenn man die Konventionen von Rails ignoriert, z.B. wegen äußerer Umstände und Anforderungen.
Rails verzichtet auf die Verwendung von Constraints innerhalb der Datenbank. In einigen Fällen, z.B. Bankensoftware, sind diese aber unbedingt erforderlich.

Mangel an Entwicklern

Ein Nachteil aus Sicht des Projektmanagement ist der Mangel an freien Ruby-on-Rails Programmierern auf dem Arbeitsmarkt[8]. So kann bei einem langfristig angesetzten Projekt u.U. die Wartung nicht garantiert werden.

Software-Lebenszyklus

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 sich benutzen und Konfigurationsparametern zu Caching und Performanz. Weitere Umgebungen (z.B. Staging) können jederzeit definiert werden.

Flexible Datenanbindung

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 zudem auch sicherer. Bei relationalen Datenbanken verwendet ActiveRecord standardmäßig Transaktionen, insofern die Datenbank dies unterstützt.

Sicherheit

Rails bietet eine gute Ausgangsbasis um sichere Websoftware zu entwickeln. Das Verwenden eines Datenbankframeworks macht SQL-Injections unmöglich.
Cross-Site-Request-Forgery und Session-Angriffe werden erschwert, da Session und Cookie-

[8] z.B. http://rubypays.blogspot.com/2011/04/rubyruby-on-rails-development-job.html

Variablen standardmäßig verschlüsselt werden und bei Nutzung der Formulargeneratoren ein CSRF-Token generiert wird, um Replay-Angriffe zu unterbinden.

Testen mit Rails Rails bietet ausgezeichnete Vorraussetzungen zum Softwaretest. Dafür sprechen, dass...

- benötigte Bibliotheken bereits mitgliefert 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 (↑Behavior Driven Development-Testframework), Rcov (Testabdeckung), diverse Mockbibliotheken (mocha, FlexMock, RR, Rspec Mocks), Tools zum Generieren und Bereitstellen von Testdaten (Fixtures, Factories, Faker) und ↑Code-Metriken (metric-fu)
- das Testen einen sehr hohen Stellenwert in der Ruby und Rails-Comumnity hat. Nahezu alle namhaften Ruby-Programmierer schreiben umfassende Tests [DN08]. Das Resultat ist, dass auch fast alle ↑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. Es können auch damit auch gleichzeitig View-Tests unternommen werden, also z.B. die Untersuchung, ob ein bestimmtes HTML-Element auf der Web-Seite 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
Zielstellung: Herausfinden von Performanz-Flaschenhälsen in allen Ebenen.
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 ↑Domain-Specific-Language (DSL)[9] für die Definition von Spezifikationen.

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 denselben 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 wiederrum wird eine Klasse mit Namen `TestJob` definiert.

Eine solche Testklasse kann wie folgt aussehen:

Testen mit Test/Unit

```ruby
require "job"

class TestJob < Test::Unit::TestCase
  def setup
    @job = Job.create
  end

  def teardown
    Job.delete_all
  end

  def test_job_exists
    @job.title =  "Ruby on Rails in Entwickler"
    @job.add_location_to_title( "Dresden")

    assert_equal( "Ruby on Rails Entwickler in Dresden",  Job.first.title)
  end
end
```

Listing: Testen mit Test/Unit in Ruby

[9] Eine Domain-Spezifische-Sprache, welche durch Kunden nachvollzogen werden kann. Siehe http://martinfowler.com/bliki/BusinessReadableDSL.html

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. 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 gleich sind, hinsichtlich des ==-Operators[10]

`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 dem spezifizierten regulären Ausdruck matcht

Natürlich lassen sich beliebige weitere Zusicherungen definieren. ↑Ruby on 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 ↑Ruby on Rails das Management, d.h. Laden und Löschen dieser, selbständig. 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, insofern die verwendete Datenbank dies unterstützt.
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 der selben Datei wie der Test zu finden, womit man Tests nur verstehen kann, wenn man die Fixtures kennt. Eine mögliche Lösung ist es, stattdessen **Factories**[11]

[10] 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
[11] http://www.dan-manges.com/blog/38

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 Domän (z.B. für Stresstests) existieren ebenfalls Lösungen. Mittels der ↑Gems „populator“ und „faker“ lassen sich beispielsweise eine beliebige Menge an gültig-anscheinenden Personendaten (Name, Vorname, Adresse, E-Mail-Adresse, Passwort,...) oder Blind-Texten generieren[12].

4.4.2. Cucumber

Cucumber ist ein relativ neues Framework (2008), um mittels einer domainspezifischen Sprache (↑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 darunterliegenden werden diese Schritte in echten Testcode implementiert.
Im Folgenden sei ein Trivialbeispiel einer Anwendung, die Addieren implementiert, gezeigt.

features/addition.feature

```
# language: de
Funktionalität: Addition zweier Zahlen
  Hier würde eine grobe Beschreibung des Businessvalues
  und der Rahmenbedingungen stehen
  Szenario: Addition von ganzen positiven Zahlen
    Wenn ich "1" für a und "2" für b eingebe
    Und auf "Addieren" klicke
    Dann sehe ich "3"
```

Listing: Cucumber: Defintion 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

```
Wenn /^ich "([^"]*)" für a und "([^"]*)" für b eingebe$/ do |arg1, arg2|
  @addierer = Addierer.new(arg1, arg2)
end

Wenn /^auf "([^"]*)" klicke$/ do |arg1|
  @result = @addierer.add()
end

Dann /^sehe ich "([^"]*)"$/ do |arg1|
  assert_equal( arg1.to_f, @result)
```

[12]Eine sehr gute Erklärung zur Nutzung beider Gems ist im Railscast #128 zu finden http://railscasts.com/episodes/126-populating-a-database

```
11 end
12
```

Listing: Cucumber: Implementierung der Additionstestschritte 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.
Cucumber ist von der Syntax so generisch, dass damit beliebige Anwendungen getestet werden, 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 ↑Ruby on 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. Den eigentlichen 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. ↑**RackTest**) und automatisierten Browser (Firefox, Chrome, etc. durch eine weitere Middleware: Selenium) je nach Bedarf zu wechseln, ohne große Änderungen am Testcode.

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

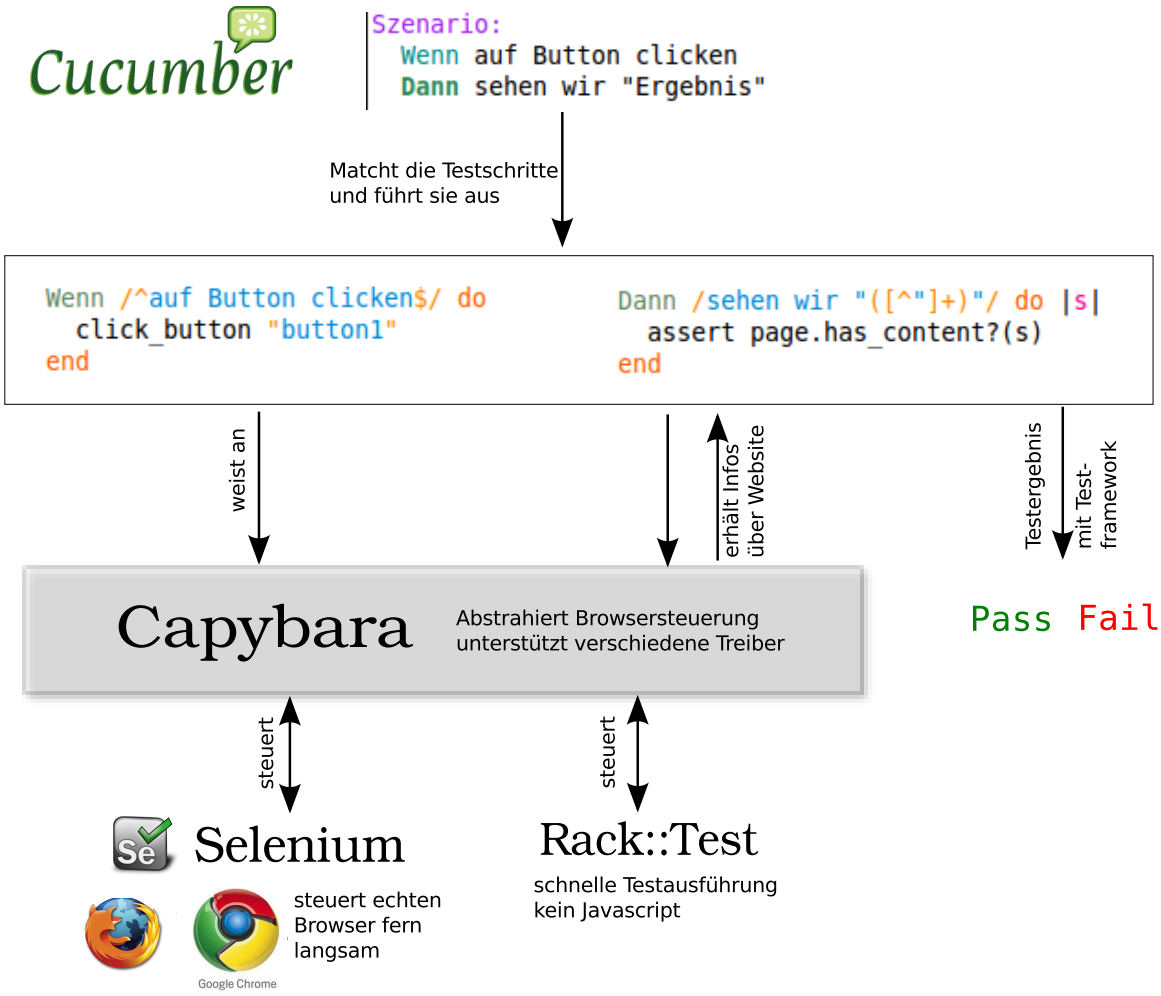


Abbildung 4.2.: Ablauf beim Akzeptanztest mit Cucumber und Capybara

Kapitel 5.

Code-Metriken

Eine ↑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.

> 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]

Dem Verwenden von Code-Metriken liegt der Wunsch zugrunde, komplexe Codeteile auf einfache Zahlen automatisiert beurteilen zu lassen, 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 wiederrum 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 die 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 tieferliegendes 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. 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 zuviel über die internen Strukturen einer anderen Klasse und implementiert Funktionalität, die eigentlich in jene Klasse gehören sollten.
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 ihren unmittelbaren Nähe kommunizieren und nicht etwa in Nachrichtenketten, wie z.B.:

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

Beim Law of Demeter ist eine solche Kette bis maximal Länge 1 erlaubt.

Duplikation 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[1] festgestellt werden.
Weitere Informationen zu Smells und deren Beseitigung finden sie in dem Buch „Refactoring“ von M. Fowler [FBB+99].

5.2. Code-Metriken für Tests

Tests haben (auch) die Aufgabe, ein Programm oder Codestück auf Korrektheit[2] 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, 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:

[1] https://github.com/kevinrutherford/reek/wiki/Code-Smells
[2] 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 macht keine Aussage über den Vollständigkeit oder die Qualität der Tests

Sollte dieser deutlich kleiner als 1 sein, so ist dies ein Symptom für zu wenige Tests. Diese Zahl ist von dem Testframework und dem Programmframework stark abhängig. Gute Projekte sollten (deutlich) mehr Test-Code, 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[3] verwendet

C2 (Pfadüberdeckung, Path Coverage) prüft, ob jeder mögliche Codepfad durchlaufen wurde. Ein Codepfad sei 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**[4], für Ruby ab 1.9.1 **simple-cov**[5], welche beide die C0-Testabdeckung bestimmen können. Zum aktuellen Zeitpunkt sind keine weiteren Tools bekannt, um C1 oder C2 Abdeckungen zu bestimmen.

[3] if-then-else in einer Zeile: int a = (1==1) ? 5 : 3
[4] http://relevance.github.com/rcov/
[5] https://github.com/colszowka/simplecov

Wieviel Testabdeckung ist sinnvoll oder notwendig Beim Messen der Abdeckung stellt sich schnell die Frage, wieviel Testabdeckung sinnvoll oder gar notwendig ist. Zuerst sei 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 müssste die C0 Testabdeckung immer 100% sein [Bec02]. Für ein Rails Projekt sei es auch relativ leicht, 100% oder nahe 100% zu erreichen [Rap11]. Die Zahl „100%“ sei für sich genommen nutzlos, aber sie zu erreichen, sei für den Prozess der Testgetriebenen Entwicklung nützlich [Rap11, S. 270]. Vielen Autoren bringen aber zum Ausdruck, dass es von der Situation abhängt wieviel 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 gäbe [Els07]. Zudem gebe eine hohe Abdeckung keinen Aufschluss darüber, dass die Tests „gut“ sind. Aber eine niedrige Testabdeckung zeigt 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 Verzweigung 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/Pertubationstests – 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 seien die Tests zu oberflächlich.

Für Ruby gibt es ein Werkzeug, **Heckle**[6], 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. Sollten dennoch in einer Mutation kein Test fehlschlagen, dass ein Test fehle, so ist die Annahme des Testverfahrens.

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.

[6] http://ruby.sadi.st/Heckle.html

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 psychlogische Auswirkungen dürfen nicht unterschätzt werden. Alleine der Fakt, dass Codemetriken in einem Unternehmen regelmäßig verwendet werden, motiviert den Programmierer keinen sogenannten „Big Ball of Mud“[7] zu schreiben. Insbesondere in kleinen Projektteams, die keine dedizierte Qualitätssicherung haben, sind Codemetriken als kostengünstiges Kontrollinstrument unerlässlich. Studien zeigen, dass der konsequente Einsatz von Code-Metriken und Analysebenchmarks die Fehlerdichte und Entwicklungskosten stark verringen kann [BCSV11, S.10f].

Für die Testgetriebene Software 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 ↑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 Team-Leiter gelesen und besprochen werden. Allerdings besteht bei einer zu hohen Beobachtung 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 würden.

[7] 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 *V-Modell*[1] oder *Rational Unified Process*[2], finden ihre Anwendung in großen Projektteams. Für mittelgroße 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. Zu eigen machen sie sich dabei kurze Releasezyklen, welche regelmäßig Feedback geben. Damit wird der klassische GAU am Ende des Projektes, wenn die Wünsche des Kunden mit den tatsächlichen Umsetzungen doch nicht einher gehen, vermieden. Aber viele dieser Methoden, wie z.B. *SCRUM*[3], benötigen 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 1 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% ↑Testabdeckung ist nicht erforderlich. Wichtige Systemlogiken, wie Bezahlvorgang und Suche müssen dagegen getestet werden. Offensichtliche ↑CRUD-Operationen müssen nicht getestet werden

[1] http://v-modell.iabg.de/
[2] http://www.ibm.com/software/awdtools/rup/
[3] 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, welche in der Ansicht für Kunden und Besucher der Website sichtbar sind → Detailansichten, Listen, Bezahlvorgänge, ...

B. Features, welche nur dem Admin sichtbar sind oder welche im Backend ausgeführt werden → Reporting, Statistiken, Indizierung der Datenbank, Cron-Scripte, Caching, ...

Features der **Kategorie A** sollen in Zukunft Akzeptanztestgetrieben entwickelt werden. Die Entwicklung verläuft nach dem Schema, dass 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 wiederholt zu werden.

Der Vorteil dieser Outside-In Entwicklung ist, dass er auf den Kunden ausgerichtet ist. Die Verwendung der domänspezifischen Sprache 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**[4], ist allgemein für Software-Projekte obligatorisch. Bei der Kontinuierlichen Integration wollen wir sicherstellen, dass der Hauptzweig unserer Entwicklung stets ein lauffähigs Produkt enthält und das laufende Änderungen so oft wie möglich integriert werden sollen. Das Vorhandensein einer großen Test-Suite ermöglicht es, diese beim Einchecken in den Hauptzweig immer komplett auszuführen, um so sicherstellen, dass auf dort stets eine lauffähige Version zu finden ist.

[4] http://git-scm.com/

In großen Projekten ist es üblich, komplexe Testpläne zu erstellen. Anscheinend sind aber automatisierte Tests, die bei jeden 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 ↑Code Smells abliefern, so macht die Code-Analyse dies sichtbar. Dies dient in erster Linie nicht, um den Programmierer zu maßregeln, sondern ihm dabei zu helfen, den ↑TDD-Prozess zu lernen und seinen Programmierstil ständig zu verbessern. Die Erfahrungen zeigen, dass die Programmierer meist selbst unzufrieden mit schlechtem Code, den sie geschrieben haben, sind. Code-Metriken können dabei helfen, dem Programmierer schnell ein Feedback zu seinem Code zu geben, wie es ein Code-Audit durch Andere in der Geschwindigkeit und Effizienz nie könnte.

Regelmäßige Paar-Programmierung 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 Paarprogramming eine effektive Methode [HA05, S. 9]. Erfahrungsgemäß führt sie zu besser dokumentierten Code, kann die Anzahl der Fehler verringern und zu einer höheren Arbeits-Effektivität führen [HA05]. Die in Abschnitt 1.1.2 angesprochene Dezentralisierung der Zusammenarbeit erschwert eine regelmäßige Paarprogrammierung. Nichtsdestotrotz sollten in regelmäßigen Abständen Features zu zweit in wechselnder Zusammensetzung entwickelt werden.

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 Entwufsphase auf ein Minimum zu reduzieren, z.B. ein grobes ERM[5] und Mock-Ups[6] zu erstellen.
Da allerdings nicht immer Paarprogrammierung möglich ist, so sind regelmäßige Abstimmungen unter den Programmierern erforderlich, um neue Designideen zu diskutieren. Dafür sollten feste Termine anberaumt werden.

[5] Entity-Relationship-Model
[6] Wegwerfprototypen der Oberfläche

6.2. Auswahl der Entwicklungswerkzeuge

Für die zukünftige Entwicklung vorrangig von Webanwendungen, werden folgende Werkzeuge berücksichtigt.

Werkzeuge für Tests Die Spezifikationssprache **Cucumber** dient als Schnittstelle zwischen dem Kunden und die vom vom Programmierer entwickelten Testschritte. Diese könnten in einem von vielen Testframeworks geschrieben werden. Die Entscheidung viel 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 so 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 lassen sich z.B. **Selenium** ansteuern, welches wiederrum 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 ↑**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 gibt es z.B. **autotest** und **guard**. 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.

Durch **spork** lässt sich eine ↑Ruby on Rails-Anwendung 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 ↑Code-Metriken dient das ↑Gem **metric-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 ↑Testabdeckung.

Die Testabdeckung wird durch **simple-cov** berechnet, welches eine C0-Code-Coverage bestimmt.

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 dies zu einer verminderten Ausführungsrate und ist damit hinderlich für die Entwicklung von sauberen Code. Dazu ist es notwendig, dass die Tests so 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 Paarprogrammierungen 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 einen solchen Zeitpunkt, kann z.B. der Einsatz der Code-Metriken reduziert werden.

Kapitel 7.

Anwendung der Testgetriebenen Entwicklung

In den nachfolgenden Abschnitten wird examplarisch 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 ↑TDD gelegt werden. Zu dessen Verdeutlichung ist am Dokumentenrand die jeweilige Phase innerhalb des TDD-Zyklus zufinden (Red, Green, Refactor), dem die im Text gezeigten Codeabschnitte zuzuordnen sind.

Ziel dieses Kapitels wird es sein, einen Überblick über die Art und Weise zu erhalten, mit denen die verschiedenen Teilbereiche einer Webanwendung testgetrieben entwickelt werden können. Die ersten beiden Abschnitte richten sich an zwei der Grundbausteine einer ↑MVC (vgl. Abschnitt 4.3.1). Webanwendung, den 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. 32 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 Validierungskritieren 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). Weiterhin sollten natürlich alle weiteren, selbstdefinierten, Methoden getestet werden.

Diese Validierungen werden durch das ↑ORM-Framework ActiveRecord, welches Rails standardmäßig nutzt, bereitgestellt. Bevor wir weiter auf die

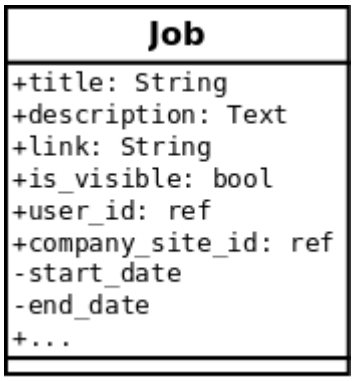


Abbildung 7.1.: Attribute des Modells „Job“

1. Der Anfang Während der Analyse wurden die benötigten Attribute bestimmt. In Abbildung 7.1 sei z.B. ein Fragment des Grobdesigns, in dem die Basisattribute der Tabelle dargestellt werden. 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 *MODELLNAME* spalte1:datentyp1 spalte2:datentyp2 ...

generieren wird 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-Model (erkennbar an

dem `invoke active_record`). Dieser wiederrum 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 usw.
- 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.
- Die dazugehörige Testklasse (`app/unit/job_test.rb`)
- und Fixtures-Datei (`test/fixtures/jobs.yml`), zur Definition von Testdaten.

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 ↑**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[1].

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`.

```
$ rake db:migrate

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

[1] Jede Umgebung verfügt normalerweise über eine eigene Datenbank. standardmäßig befinden wir uns in der „Development“-Umgebung

Danach können wir die Rails-Test-Suite mithile eines weitere Rake-Tasks auch schon ausführen. Dieser Rake-Task legt selbsständig eine Testdatenbank an, führt offene Migrationen darauf aus und führt dann alle vorhanden Tests aus.

```
$ 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
```

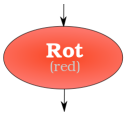
Listing: Standardtest generiert durch Rails

2. Testen auf Validierung Ein Feature von Rails umfassen die sogenannten Validierungen. Diese stellen sicher, dass eine Instanz eines Modells nur dann gespeichert ist, wenn es gewissen Kritieren 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 instanziieren wir einen Job und geben ihm explizit einen leeren Titel, um das Testziel nochmal herauszustellen. Danach rufen wir die `save`-Methode auf, die prüft, ob alle Validierungskritierien erfolgt sind und speichert das Objekt persistent in der Datenbank im Erfolgsfall. Dann gibt `save` ein `true` zurück, anderfalls, d.h. wenn die Validierung fehlschlug, `false`. Der Ablauf ist in der Abbildung 7.2 noch einmal erläutert.

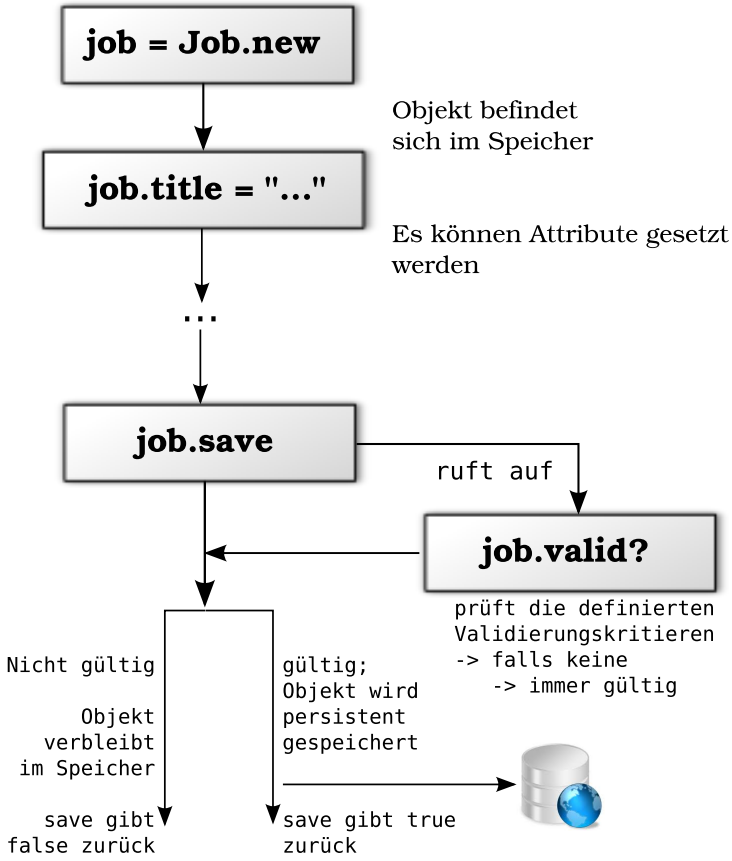


Abbildung 7.2.: Funktionsweise von save bei ActiveRecord Objekten

Da wir noch keine Validierungskritieren implementiert haben, schlägt dieser Test fehl, da das Objekt gespeichert wurde.

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 ei-

ne Liste an Validierungskritieren. Hier ist das Kritierum `presence`, also das Vorhandensein eines nicht-leeren Attributs. Weitere Kritieren sind z.B. Format, Länge, Minimum, Maximum oder 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

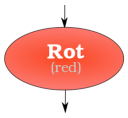
```ruby
test "ein Job muss einen Titel haben" do
  job = Job.new :title => nil
  assert !job.save
end
```

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

```ruby
test "ein Job muss eine gültige E-Mail haben" do
  job = Job.new :email => "invalid_email"
  assert !job.save
end
```



Die Implementierung wäre dann:

app/models/job.rb

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

Eine Refaktorisierung ist aufgrund der Einfachheit der Beispiel 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/job.rb und app/models/job.rb

```ruby
# config/initializers/regex.rb
REGEX_EMAIL_FORMAT = /^[\w\d_\-]+@[\w\d_\-]\.[\w\d]{2,3}$/

# app/models/job.rb
class Job < ActiveRecord::Base
  validates :email, :format => REGEX_EMAIL_FORMAT
    ...
end
```

Listing: Auslagerung des Regulären Ausdrucks in einen Initalisierer

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 seinn" do
3   job = Job.new :title => "Rails Entwickler", :email => "info@stefanwienert.net"
4   assert_valid job
5 end
```

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

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 => "info@stefanwienert.net"
11   assert_valid job
12 end
```

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[2]. 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:

test/fixtures/jobs.yml

```
1  valid_job:
2    title: Rails Entwickler
3    email: info@stefanwienert.net
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

[2]Im ersten Test ist nicht nur der Titel nicht gesetzt, sondern auch die E-Mail entspricht nicht dem Format

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

test/unit/job_test.rb

```
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.
>
> Jamis 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 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.

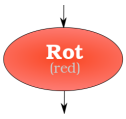
test/functional/jobs_controller_test.rb

```ruby
require 'test_helper'

class JobsControllerTest < ActionController::TestCase
  test "Gast Nutzer kann Stellen betrachten" do
    session[:user_id] = nil
    job = jobs(:valid_job)

    get :show, :id => job.id

    assert_response :success
    assert_equal job, assigns(:job)
  end
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 aus (Die Aktion `show` mit der ID des Jobs). Nun erwarten wir, dass wir einen HTTP-Status Code 200 (Erfolg) erhalten und dass der Controller eine Variable `@jobs` bereitstellt, die mit unserem Fixture identisch ist.

Die Implementation dieser Anforderung könnte wie folgt umgesetzt werden:

app/controllers/jobs_controller.rb

```ruby
class JobsController < ApplicationController
  ...
  def show
    @job = Job.first
  end
  ...
end
```

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

Das Laden des erstem Jobs aus unserer Datenbank 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 wir 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:

app/controllers/jobs_controller.rb

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

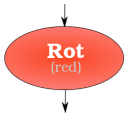
Listing: JobsController: Ersetzung der Fake-Implementierung

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

test/functional/jobs_controller_test.rb

```
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. Dieses mal erwarten wir einen HTTP Statuscode 301 (Redirect/Weiterleitung) und dass unser Controller eine Hinweisnachricht generiert.

app/controllers/jobs_controller.rb

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

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

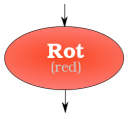
Falls der aktuelle Job nicht sichtbar ist, dann erfolgt eine Weiterleitung auf die Startseite und die Bereitstellung des Hinweistextes. Da auch hier der Quelltext wieder sehr kurz ist, ist ein Refaktorisieren nicht notwendig.

Nun möchten wir, dass ein Kunde dieser Anwendung, also ein Unternehmen 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 Authen-

tifizierung 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 zur Zeit 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 and 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 zur Zeit nicht sichtbar"
5   end
6 end
```

Listing: JobsController: Anwendung der Refaktoriserung

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 controllerglobalen 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 zur Zeit nicht sichtbar"
7   end
8 end
```

Listing: JobsController: Nutzung der neuen Methode current_user

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

- HTTP Statuscodes und Weiterleitungen,
- das Vorhandensein von Statusmeldungen, gennant „Flash“-Messages
- dass ein bestimmtes Template geladen 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.

Intern nutzen Controller-Tests viele Stubs, um HTTP-Anfragen und Antworten durch eigene TestKlassen 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 ↑RSS 2.0 Format[3] bereit, der ein häufig verwendetes Format zum Austausch von Informationen ist und durch eine Vielzahl von Werkzeugen und ↑Content Management System unterstützt wird. Dabei wird der Inhalt

[3] Spezifikation des RSS 2.0 Formats: http://cyber.law.harvard.edu/rss/rss.html

des Haupttextfeldes „description“ um weitere Informationen in einem Subdialekt angereichert.

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

beispiel_job.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>RSS Feed für Jobangebote </title>
    <language>de</language>
    <item>
      <title>Softwareentwickler Java/JEE (m/w)</title>
      <description>
        <![CDATA[
        <!--
        <nummer>example_job_01</nummer>
        <tags>Java,Webentwickler,Softwareentwickler</tags>
        <ort>Dresden</ort>
        <kontakt>Max Mustermann bewerbung@example.com</kontakt>
        <link>http://www.example.com/jobs/512.html</link>
        -->
        Zur Verstärkung unseres Teams suchen wir zum nächstmöglichen
        Zeitpunkt einen Softwareentwickler Java/JEE (m/w) zur Festanstellung.<br />
        Ihre Aufgaben: ...
        ]]>
      </description>
      <link>http://www.example.com/jobs/512.html</link>
      <pubDate>Wed, 25 Mar 2011 13:30:00 +0100</pubDate>
      <guid>example_job_01</guid>
    </item>
  </channel>
</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.

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. Initialier Test Bevor man anfängt zu implementieren, ist es sinnvoll sich Gedanken darum zu machen, was von den zu implementieren 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

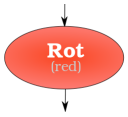
```ruby
require "test_helper"    # Stuff that we need for convenient tests
require "imported_feed" # Object under Test

class TestImportedFeed < ActiveSupport::TestCase

  test "get an feed through httparty" do
    HTTParty.expects(:get).with("http://www.example.com/feed.xml")
    ImportedFeed.new("http://www.example.com/feed.xml")
  end
```

Listing: Feed Test I.

Hier definieren wir einen ersten Test für den ImportedFeed. Für die HTTP-Zugriffe wollen wir die Bibliothek HTTParty[4] 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 Konstruktur 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')
```

[4]https://github.com/jnunemaker/httparty

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

Die Implementierung wäre:

lib/imported_feed.rb

```ruby
require "httparty"

class ImportedFeed
  def initialize(url)
    HTTParty.get(url)
  end
end
```

Listing: Feed Implementation I.

Die Funktion initialize stellt innerhalb von Ruby den Konstruktor dar. Dort rufen wir unseren Netzwerkzugriff auf, 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.

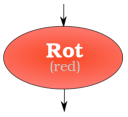
test/test_imported_feed.rb

```ruby
test "really get content from an feed" do
    fake_response = OpenStruct.new
    fake_response.code = 200  # HTTP OK!
    fake_response.body = "<?xml version='1.0'?><Hallo/>"

    HTTParty.expects(:get).with("http://www.example.com/feed.xml").
      returns(fake_response)

    import = ImportedFeed.new(@url)
    assert_match "Hallo", import.body
  end
```

Listing: Feed Test II



Wir bilden das Antwortobjekt, das HTTParty normalerweise generieren würde, beschränken uns hierbei aber nur auf die für uns notwendigen Methoden von „body“ und „code“ (Dem HTTP-Status Code). Wir nutzen dazu die Klasse OpenStruct, die Getter und Setter für das Objekt beim Benutzen erstellt. Unserem Mock können wir dann anweisen, diese Antwort zurückzugeben.

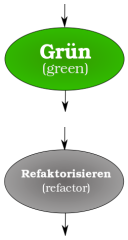
Bei Ausführung des Tests stellen wir fest, dass zwar die Erwartung erfüllt wurde, aber unser ImportedFeed noch kein Attribut „body“ besitzt (Fehler) und dass dieser keine String beinhaltet.

app/models/job.rb

```ruby
class ImportedFeed
  attr_reader :body
```

```
3   def initialize(url)
4     response = HTTParty.get(url)
5     @body = response.body
6   end
7 end
```

Listing: Feed Implementation II



Mitfhile 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 zusammenmergen. Diese Art der Parameterübergabe ist ein sehr gebräuchliches Idiom innerhalb der Ruby-Community.

Der Aufruf unsere neuen Hilfsfunktion erfolgt dann mittels:

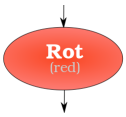
test/test_imported_feed.rb

```
1    def setup
2      @url = "http://example.com/feed.xml"
3    end
4
5    test "get an feed through httparty" do
6      should "perform a get request when initializing" do
7        mock_feed :url => @url
8        ImportedFeed.new(@url)
9      end
10
11     test "really get content from an feed" do
```

```
12        mock_feed :url => @url, @body => "`Hallo"
13        import = ImportedFeed.new(@url)
14        assert_match "Hallo", import.body
15      end
```

Listing: Feed Test IIb nach Refaktorisierung

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



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

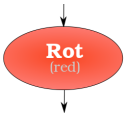
```
1 ...
2   def valid?
3   end
```

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:

ib/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

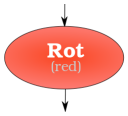
Zu bemerken ist, dass wir den Funktionsrückgabewert nicht explizit mit „return “ kennzeichnen müssen. Bei Ruby hat jeder Ausdruck einen Rückgabewert. Innerhalb einer Funktion ist dies das letzte Statement, falls nicht mit return spezifiziert.

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 resistent 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
```

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 setzten 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 Softwareentwicklung (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 Browser-Engine. Cucumber lädt automatisch alle Ruby-Dateien in den beiden Verzeichnissen.

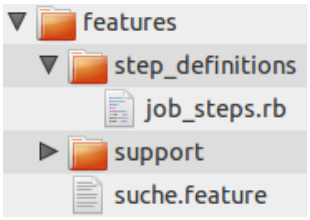


Abbildung 7.3.: Aufbau eines Cucumber- Verzeichnisses

1. Definition des gesamten Akzeptanztest Zusammen mit dem Kunden oder basierend auf den Anforderungen entwickeln wir zuerst eine Spezifikation für ein Feature und darauf aufbauend, Da die Akzeptanztests in erster Linie dazu dienen, die Software gegenüber den Anfordungen des Kunden zu validieren und auch als Kommunikationsmittel genutzt wird, orientert 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.

features/job.feature

```
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 („Funktionaliät“) 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:

```
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 uns schon ein paar Codeschnipsel generiert, mit denen wir unsere eigenen 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_defintions/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 unserem simulierten Browser die Anweisung, die Startseite zu besuchen. Der Zweite Testschritt spezifiziert einen Testschritt mit 2 variablen Texten, die an unseren Block 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 implementiert den Klick auf einen HTML-Button mit dem angegebenen Inhalt. Während die ersten 3 Schritte Aktionen ausführen, hat der 4. eine andere Funktion: Er spezifiert eine Zusicherung, dass auch unserer aktuellen Webseite irgendwo ein gewisser Inhalt steht.

features/step_defintions/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

Der Testschritt für die Implementation des Testschrittes mit der Tabelle, ist etwas umfangreicher. Wir bekommen von Cucumber ein Array von Hashes übergeben, die die geparste Tabelle beinhaltet. Über diese können wir nun mit dem Iterator „each“ iterieren und die definierten Attribute mit denen unserer Job-Fixture[5] vereinigt wird, um somit einen neuen Job in der Testdatenbank anzulegen. Wir haben den Testschritt so allgemein gehaltet, 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:

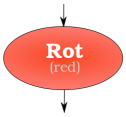
```
Wenn wir "Rails" für "search" eintippen    # features/step_definitions/job_steps
    .rb:5
  cannot fill in, no text field, text area or password field with id, name, or
      label 'search' found (Capybara::ElementNotFound)
```

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

```
      (eval):2:in `fill_in'

Failing Scenarios:
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/appplication.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 Suchfeld 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[6], 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   ...
```

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 „?“. 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.

[6] Embedded Ruby, eine Syntax für Rails Views

app/views/jobs/index.html.erb

```
1
2 <% @jobs.each do |job| %>
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 ein paar Testschritte implementiert haben, ist die Entwicklung von weiteren Szenarien leichter. Die Tests wurden bisher durch einen 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, 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 Webanwendung 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**[7], also durch asynchrone Kommunikation mit einem Server und Aktualisieren der Seite ohne dass diese neugeladen werden muss. Durch sehr gute Frameworks, wie z.B. **jQuery**, dass ab Rails 3 standardmäßig Teil der Rails-Distribution ist, sind solche Aktualierungen 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/Akzeptanztest, die bereits im voherigen Abschnitt Thema waren. Dafür müssen wir nur eine Browser-Engine wählen, die Javascript ausführen kann. Für die Browsersimulation nahmen 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 letztendlichen Realen des Benutzers sehr nahe kommt.

Dedizierte Unittests für JavaScript Falls eine Anwendung nicht nur einfache Ajax-Funktionen der Art „Klicke Link – Sende Anfrage an Server – setzte fertiggenerierten HTML Code der Antwort direkt in den DOM 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. das aktuelle Twitter-Layout 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[8], einem Vertreter der xUnit-Reihe und Jasmine[9], einem Vertreter aus dem ↑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 waren.

[7] Asynchrones JavaScript und XML (oder JSON)
[8] http://www.jsunit.net/
[9] http://pivotal.github.com/jasmine/

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 JavaScriptanwendungen sind aber unbedingt dedizierte Unittests erforderlich.

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, welche im Verlauf der Entwicklung regelmäßig festgehalten wurden, näher betrachten. Im weiteren Teil 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: ↑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. E-Mail Benachrichtigung im Fehlerfalle 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: Bezahlsystem: Beim Anlegen eines Jobs wird dieser 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 beteiligt waren 2 Programmierer, einschließlich des Autors. 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.

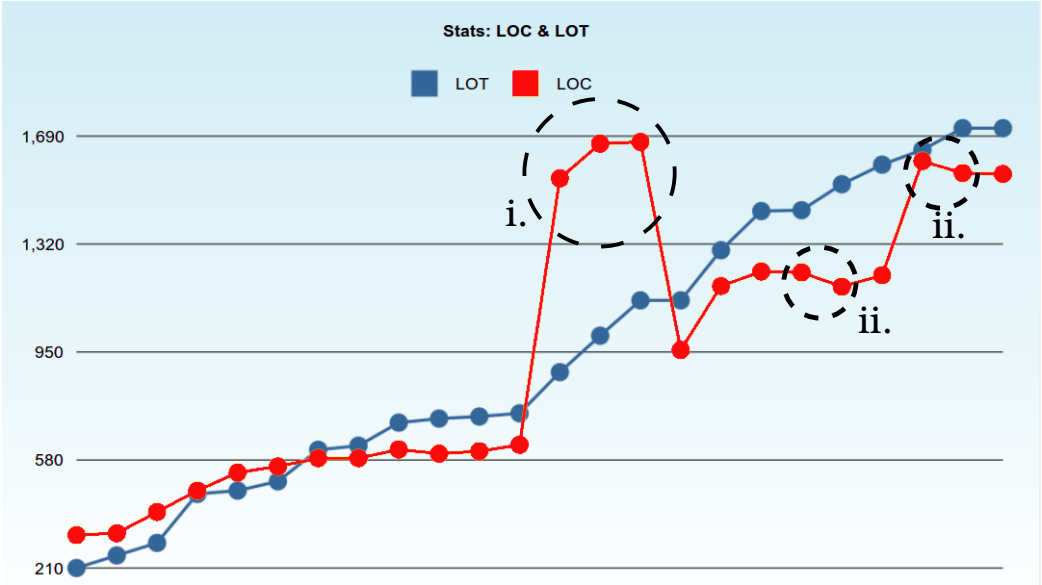


Abbildung 8.1.: Entwicklung des Umfangs des Test- und Programmcodes

In Abbildung 8.1 ist der Verlauf der Codezeilen und der Testzeilen abgebildet. Während in den ersten Tagen der Entwicklung, wegen dem 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 Refaktoriserung ermöglicht und in diesen Phasen konnte effektiv Design zu dem System hinzugefügt werden.

In Abbildung 8.2 ist der Verlauf der [↑]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 erfolgte eine Umstellung auf SimpleCov (bereits in Abschnitt 6.2 vorgestellt). Ab dann lag die Testabdeckung innerhalb von 90% bis 100%.

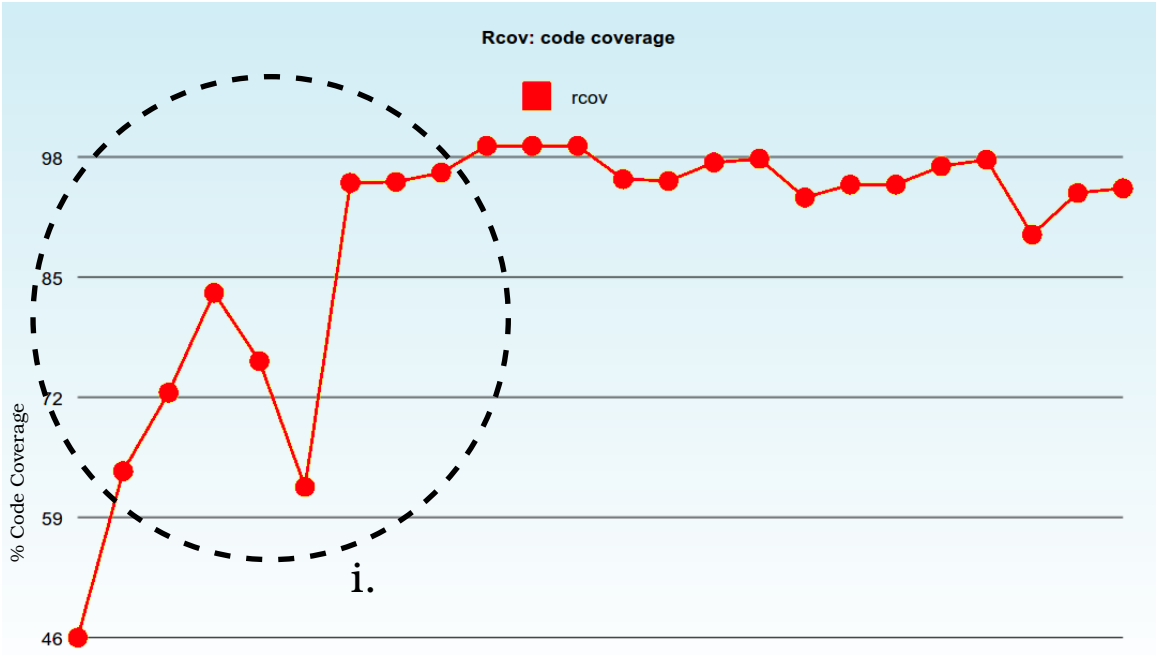


Abbildung 8.2.: C0-Testabdeckung über den Verlauf der Entwicklung

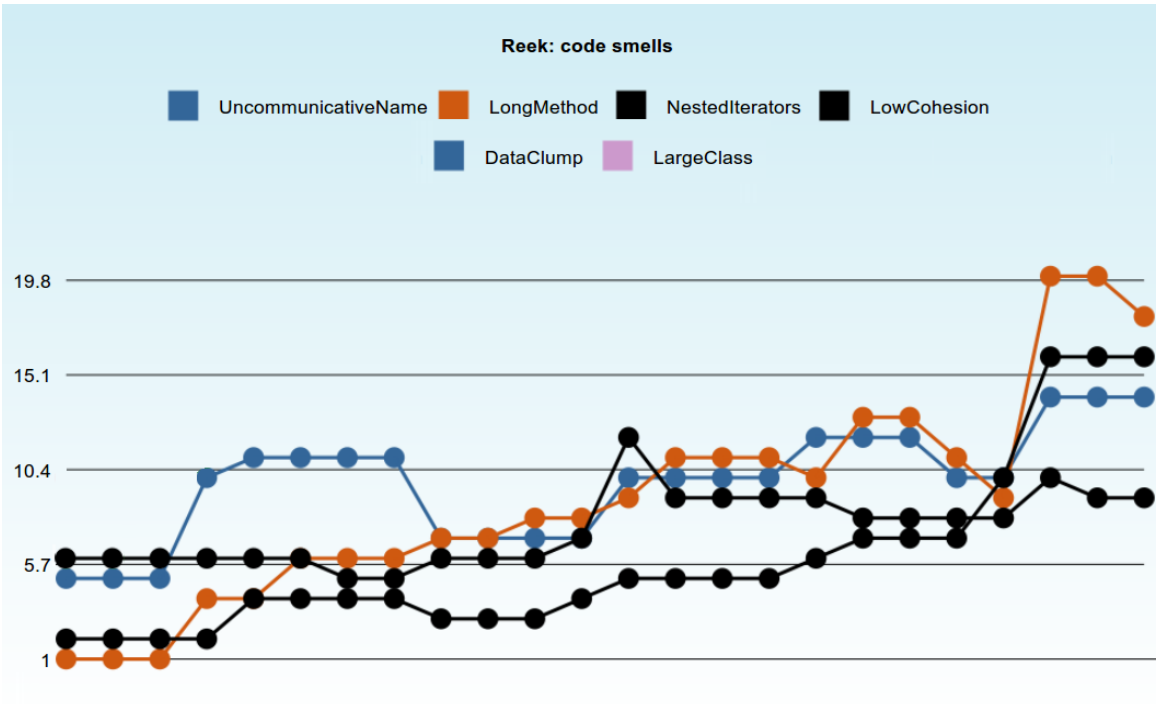


Abbildung 8.3.: Verlauf der verschiedenen Klassen an Code-Smells über den Entwicklungszeitraum

Ein Indikator für die 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 konnte einige suboptimale Stellen nach einer gewissen Zeit refaktorisiert werden. So ist das wiederkehrende Muster, 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. Ist 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
- 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 eingecheckt 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.
- Trotz bisheriger Erfahrungen mit Ruby und Rails waren nur wenig Erfahrungen zum Thema Testen und Testgetriebene Entwicklung vorhanden. Insbesondere die Nutzung von ↑Test-Doubles musste erst gelernt werden, da ohne eine solche bestimmte Testszenarien schwierig bis überhaupt nicht zu testen sind.

Trotzdem ermöglichte die vorhande 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 gewisser Code-Smell für eine gewisse Stelle 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-Benchmark 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 ↑Rake-Tasks „rake stats“.

LOT Anzahl der Quellcodezeilen aller Tests, gemessen durch das Rails-Kommando „rake stats“.

LOT/TOC Verhältnis aus Testzeilen und Quellcodezeilen

AVGCmplx Durchschnittliche Komplexität aller Klassen, gemessen durch **Flog**[1]. Flog besitzt ein eigenes Maß für Code-Komplexität. Es vergibt für jede Zuweisungen, Verzweigungen und Funktionsaufrufe unterschiedlich viele Punkte und bildet so eine Summe per Funktion oder per Klasse. Dabei vergibt Flog besonders viele Punkte für schwer nachzuvollziehende Funktionsaufrufe, wie z.B. „eval(string)“, welches einen String als Ruby-Code auswertet.

H5Cmplx Komplexität der 5% komplexesten Klassen, nach Flog

DSmell Anzahl der Code-Smells nach Roodi. Beinhaltet u.a.: hohe Cyclomatische Komplexität in einer Methode (min. 4), lange Methoden, lange Parameterlisten, u.v.m.[2]

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

CSmell Anzahl der Code-Smells nach Reek. Diese beinhalten: Geringe Kohäsion, Duplikation, Control-Couple, Unkommunikativer Name von Methoden/Variablen/Parameter, verschachtelte Iteratoren, u.v.m [3]

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 Ist die fertiggestellte Bibliothek, 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.

[1] https://github.com/seattlerb/flog
[2] http://roodi.rubyforge.org/files/README_txt.html
[3] https://github.com/kevinrutherford/reek/wiki/Code-Smells

feedmerger Eine Rails-Anwendung[4] zur Verwalten, Cachen, Filtern und Zusammenfügen von RSS und Atom-Feeds.

pludonidb Ist 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[5] der Communitymitglieder und der Platzierung für relevante Sucheingaben in den großen Suchmaschinen (konkret: Welchen Platz hatte ein Communityportal an einem bestimmten Tag für „it jobs dresden“, usw.). Details dazu sind im Praktikumsbericht des Autors zu finden.

SiteAnalyzer (SAnalyzer) ist 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[6].

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:

[4] Als OpenSource unter https://github.com/zealot128/WenShanWenHai zu finden.
[5] Backlinks sind Links anderer Webseiten auf die eigenen. In den Communityportalen sind die Mitglieder vertraglich verpflichtet, einen Backlink auf die Communityportale anzulegen.
[6] Auch dieses Projekt ist als OpenSource verfügbar: https://github.com/zealot128/Site-Structure-Analyzer

lpp 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. Der Autor dieser Arbeit war zwar nicht an der Entwicklung beteiligt, übernahm aber die Pflege und Weiterentwicklung selbiger.

diaspora Ist ein verteiltes soziales Netzwerk.
Code: https://github.com/diaspora/diaspora

bucketwise (bucket) ist ein web-basierter persönlicher Finanzmanager mit Budgetierung nach dem Briefumschlagsystem
Code: https://github.com/jamis/bucketwise

chiliproject (chilli) Ist ein Fork[7] des sehr beliebten Bugtrackers Redmine[8]. Statt des originalen Redmines wurde dieser Fork genommen, da er eine neuere Codebasis hat
Code: https://github.com/chiliproject/chiliproject

railscast (rCasts) Ist der Code der Website http://www.railscasts.com, in welche allwöchentlich ein Screencast zum Thema Ruby und Rails veröffentlich 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) ist ein Framework um Anfragen und Antworten eines Webservers zu verarbeiten und ist ein weiterer Kernbestandteil von Rails.
Code: https://github.com/rails/rails.git

Die Auswahl erfolgt z.T. sehr willkürlich. Als Ansatzpunkt diente die Liste der beliebtesten Rails-Projekte, die auf Github gehostet sind. 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, feedmerger und backlinks, die wenig bis gar keine automatisierten Tests besitzen. Als Gegenbeispiel sei ActionPack zu nennen, dass über 2.5x soviel Testcodezeilen wie Pro-

[7] In der (Open-Source) Software-Entwicklung ist ein Fork eine legale Kopie eines bestehendes Software-Produktes, um von nun an eine unabhängige Entwicklung zu betreiben
[8] https://github.com/edavis10/redmine

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

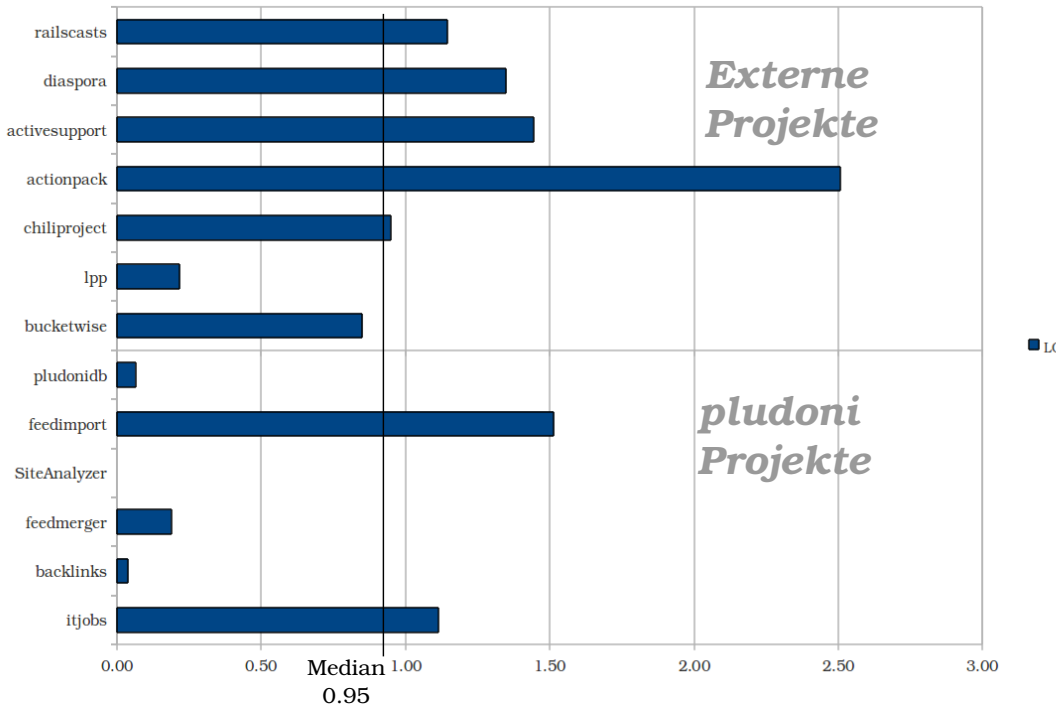


Abbildung 8.4.: Vergleich des Verhältnisses aus Anzahl Testcodezeilen / Anzahl Codezeilen

grammcode verfügt. Gegenüber den bisherigen Projekten in der pludoni GmbH kann man anhand der Projekte Feedimport und IT-Jobs, die im Rahmen dieser Diplomarbeiten enstanden, sehen, dass die bloße Anzahl an Tests stark zugenommen hat. Beide Projekte wurden mittels TDD entwickelt. Gegenüber dem Feedimport verfügt IT-Jobs aber über mehr Code, der durch das Framework bereits zur Verfügung gestellt wird, während der Feedimport eine weitestgehend eigene Objekthierarchie besitzt und damit auch mehr Tests bedurfte.

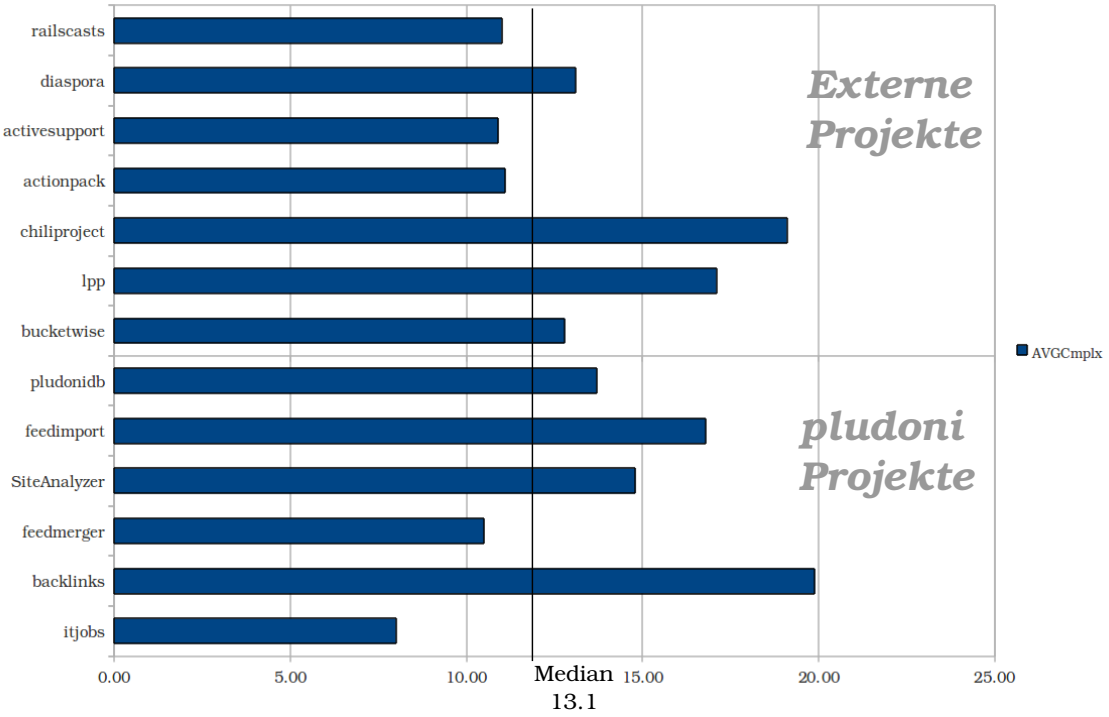


Abbildung 8.5.: Vergleich der durchschnittlichen Komplexität

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 dagegeben eine geringere Code-Komplexität. Die Ursache ist hierbei, 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 zwar durch TDD entwickelt wurde, wurde eine solche Analyse erst am Ende der Entwicklung durchgeführt. Für uns ist das ein Indiz, dass das Nutzen einer Testgetriebenen Entwicklung kein alleiniger Garant für eine gute Code-Qualität ist.
Bei den externen Projekten war der Bugtracker-Anwendung Chiliproject/Redmine die komplexeste Anwendung, 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.

Die Abbildung 8.6 zeigt zwei verschiedene Arten von Smells: Einerseits durch Reek gemessen (Rot) und andererseits durch Roodi gemessen (Blau). Zu bemerken sei, dass IT-Jobs

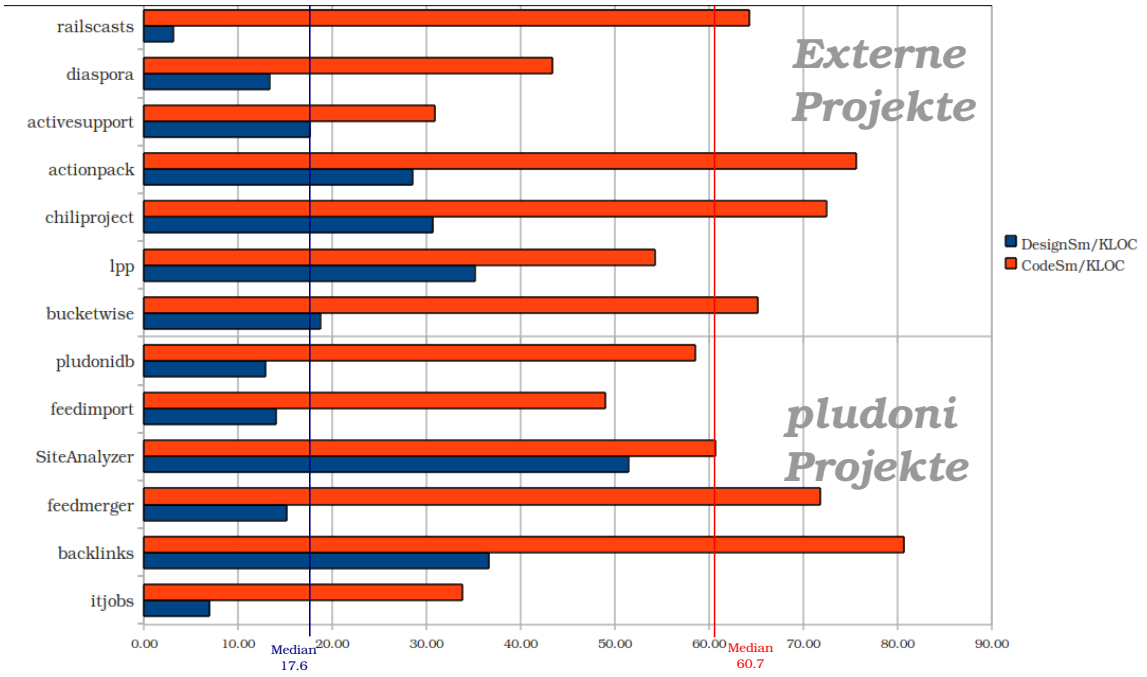


Abbildung 8.6.: Vergleich der Anzahl Smells pro KLOC

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 verfügt unter den externen Projekten die höchste Smell-Dichte, trotz der großen Menge an Tests und der relativ geringen Komplexität. Da Actionpack auf das umfangreichste Projekt unter den untersuchten ist, wäre hier ein guter Ansatzpunkt für weitere Untersuchungen in der Sinnhaftigkeit gewissen Code-Smells. Die mit Abstand größte Code-Smell hierbei war Geringe Kohäsion bzw. starke Kopplung 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. Das Gefühl der Programmierer, das z.B. das Projekt Backlinks eine relativ schlechte Codebasis 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

Die im Rahmen dieser Diplomarbeit entwickelte Web-Anwendung ist zum gegenwärtigen Zeitpunkt noch nicht fertiggestellt. Aufgrund einer Repriorisierung innerhalb der Firma, wird diese vorraussichtlich 2012 weiter entwickelt werden. Gleichwohl kann die Entwicklung als Erfolg gewertet werden, da alle Qualitäts-Kennzahlen einen wesentlichen Unterschied gegenüber den bisherigen Projekten aufzeigt. Im Großen ist dies der Testgetriebenen Entwicklung und der ständigen Beobachtung der Code-Metriken zu verdanken. Dank der Testgetriebenen Entwicklung enstand relativ lose gekoppelter Code und ein feinmaschiges Testnetz, dank dessen Hinweise aus den Code-Metriken leicht umgesetzt werden konnten.

Lernkurve von TDD

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 wünschenswerte, aber lange nicht ausreichend für eine Beurteilung der Testvollständigkeit oder ein gar Beweis für eine Korrektheit des untersuchten Programmes.

Grenzen von TDD

Die Testgetriebene Entwicklung ist allerdings kein Allheilmittel („Silver Bullet“) für alle 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, da die Anforderungen falsch verstanden wurden. Außerdem stellt die Testgetriebene Entwicklung keinen Ersatz für andere Arten von Tests, wie Performanz/Stress und Usability-Test, dar. Auch lassen sich einige Progblemme schlecht mittels TDD testen und implementieren, darunter Nebenläufigkeit oder Sicherheit. Auch benötigt TDD, auf kurze Sicht betrachtet, mehr Zeit bei der Entwicklung, 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 vorhanden Werkzeugen innerhalb der Programmiersprache und Framework abhängig. So verfügen die weit-verbreiteten Sprachen über ein Vielzahl von Werkzeugen, die den Prozess unterstützen, z.B. Code-Metriken, parallelisierbare Test-Runner oder hochentwickelte Test-Frameworks.

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/Umfangs produziert wird und auch die Testung selbst stark vereinfacht wird, da Code und Test gemeinsam entstehen und nicht im Nachhinnein.

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.

9.1. 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 werden und deren Auswirkungen untersucht werden (vgl. 2.5, 3.6.1 und 7.4).

Ausbau der Benchmarks

Der hier gezeigte Ansatz eines Benchmarks der Code-Qualität innerhalb der Ruby/Rails-Community kann so ausgebaut werden, dass weitere Parameter untersucht und deutlich mehr Projekte einfließen, um so letztendlich repräsentative Kennzahlen zu erhalten, was eine gute Code-Qualität für Railsanwendungen ausmacht. Dazu wird in Zukunft ein Benchmarking-Tool entwickelt, welcher 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 Neuentwicklung einiger Messwerkzeuge sinnvoll.

Erfahrungen mit TDD

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 Durschnitt benötigt, um effektiv testgetrieben zu entwickeln. Ebenfalls wären Studien über die psychologische 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

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 Qualität der Tests zu beuteilen. Stattdessen existieren einige Qualitätskriterien, um Tests auf ihre Sinnhafigkeit 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. Anderseits 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 Code Dokumenation 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 würd [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 Kritierium 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 Ist 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 der Firefox im Hintergrund gestartet, ohne dass dies sichtbar wird.

Abbildungsverzeichnis

1.1. Funktionsweise des Empfehlungscodes 3
1.2. Anwendungsfälle 6

2.1. Einteilung der Tests 11

3.1. Red-Green-Refactor: Der TDD Entwicklungszyklus 18
3.2. Flussdiagram für TDD 19
3.3. Entwicklungsablauf 20

4.1. MVC Modell von Rails 33
4.2. Ablauf beim Akzeptanztest mit Cucumber und Capybara 42

7.1. Attribute des Modells „Job“ 55
7.2. Funktionsweise von save bei ActiveRecord Objekten 58
7.3. Aufbau eines Cucumber- Verzeichnisses 73

8.1. Entwicklung des Umfangs des Test- und Programmcodes 81
8.2. C0-Testabdeckung über den Verlauf der Entwicklung 82
8.3. Verlauf der verschiedenen Klassen an Code-Smells über den Entwicklungszeitraum 82
8.4. Vergleich des Verhältnisses aus Anzahl Testcodezeilen / Anzahl Codezeilen . 87
8.5. Vergleich der durchschnittlichen Komplexität 88
8.6. Vergleich der Anzahl Smells pro KLOC 89

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 28
4.4. Ruby Beispiel offene Klassen 29
4.5. Demonstration von generischen 30
4.6. Nutzung von Metaprogrammierung zur Erstellung von Objektbeziehungen . 30
4.7. Testen mit Test/Unit in Ruby 38
4.8. Cucumber: Defintion eines Additionsfeature (in Deutsch) 40
4.9. Cucumber: Implementierung der Additionsteststschritte in Ruby 41

7.1. Standardtest generiert durch Rails 57
7.2. Test auf Vorhandensein eines Titels 58
7.3. Implementierung der Validierung in die Klasse Job 58
7.4. refaktorisierter Test 59
7.5. Auslagerung des Regulären Ausdrucks in einen Initalisierer 59
7.6. Alle bisherigen Testmethoden in der Klasse JobTest 60
7.7. Fixtures Testdaten für zwei Jobs 60
7.8. Finale Job-Test Klasse nach Refaktorisierung 61
7.9. JobsController-Test: Gast Nutzer darf Stellen betrachten 62
7.10. JobsController: Erfüllung des Tests durch eine Fake-Implementierung 62
7.11. JobsController: Ersetzung der Fake-Implementierung 62
7.12. JobsController-Test: Gast Nutzer dürfen nicht-sichtbare Stellen nicht betrachten 63
7.13. JobsController: Weiterleitung, falls ein Job nicht sichtbar ist 63
7.14. JobController-Test: Neuer Testfall 63
7.15. JobsController: Implementierung der Weiterleitung, falls Stelle unsichtbar . 64
7.16. Job: Einführung einer Methode zur Bestimmung der Sichtbarkeit eines Jobs im Job-Modell 64
7.17. JobsController: Anwendung der Refaktoriserung 64
7.18. ApplicationController: Einführung einer controllerglobalen Methode current_user 64
7.19. JobsController: Nutzung der neuen Methode current_user 65

7.20. Feedimport Beispiel-XML Datei mit einem Job 66
7.21. Feed Test I. 67
7.22. Feed Implementation I. 68
7.23. Feed Test II 68
7.24. Feed Implementation II 69
7.25. Zentrale Implementierung des Mocks in der Test Helper 69
7.26. Feed Test IIb nach Refaktorisierung 70
7.27. Feed Test III 70
7.28. Feed Implementation III - Fake Implementierung 70
7.29. Feed Test IV – Test auf nicht-erfolgreichen HTTP-Status Code 70
7.30. Feed Implementation IV - Implementation der Validierung 71
7.31. Feed Test V - Testen der Robustheit gegen Exceptions 71
7.32. Feed Implementation V 71
7.33. Feed Implementation Vb - nach Refaktorisierung 72
7.34. Definition eines Szenarios in einem Cucumber-Feature 74
7.35. Implementierung der Testschritte zur Interaktion mit einem Browser 75
7.36. Testschritt zum Anlegen von Testdaten auf Basis der Cucumber Definition . 75
7.37. Implementation eines Suchfeld im Layout der Webseite 76
7.38. JobsController am Ende der Testphase 76
7.39. View-Implementation für eine Liste von Jobs 77
7.40. Auslagerung der Suchlogik in die Job-Klasse 77
7.41. Finaler Jobcontroller nach Refaktorisierung 77

B.1. RAILS_ROOT/Gemfile 93
B.2. RAILS_ROOT/Guardfile 94
B.3. features/support/env.rb 94

Stichwortverzeichnis

– **A** –

ActiveRecord 29, 30, 32, 35, 36, 55, 56, 58, 76, 85
Akzeptanztest 10, 14, 15, 24, 38, 41, 50, 53, 54, 73, 77, 91

– **B** –

Browser
  Automatisierter 15, 41
  Simulierter 8, 15, 41, 50, 52, 75, 77

– **C** –

Capybara 41, 50, 52, 73, 74, 77, 78
Code-Generator 35, 55, 81, 83
Code-Metrik IX, 1, 8, 37, 43, 45, 48, 52, 53, 81, 83, 90
Code-Smell ..... VII, 44, 45, 52, 83, 88, 89
Cucumber .. 38, 40, 41, 50, 52, 73–75, 77, 95
  Testschritt 40, 41, 52, 73–75, 77

– **D** –

Datenbank ... VIII, 7, 13, 29, 32, 34–36, 39, 50, 54–58, 60, 62, 76, 80, 85

– **G** –

Gem ..... VII, 34, 37, 40, 52

– **I** –

IT-Jobs-Projekt 5, 7, 66, 78, 80, 84, 88, 89, 91

– **J** –

JavaScript .. 7, 15, 33, 35, 36, 52, 54, 78, 79

– **R** –

Rails
  Modell 40
Refaktorisierung VII, 16–18, 20–23, 48, 59, 61, 62, 72, 81, 83, 89, 90
Ruby .. 13, 16, 26–31, 34–36, 38, 39, 41, 44, 46, 47, 52, 66, 68, 69, 71, 84, 91
  Rake 56, 57, 84
Ruby-on-Rails ... VIII, 1, 4, 7, 8, 30, 32–38, 47, 49, 51, 52, 54–57, 59–61, 65, 66, 76, 78, 80, 81, 83–86, 93
  Controller . 33–35, 37, 54, 61–65, 76, 77
  Modell 21, 30, 32, 37, 54–57, 65
  Validierung 32, 37, 39, 54, 55, 57, 58, 60
  View 33, 34, 37, 61, 65, 76

– **S** –

Selenium 15, 41, 52, 78, 93, 95

– **T** –

TDD VII, 1, 5, 7, 8, 10, 12, 15, 17, 18, 20–25, 47, 49–51, 53, 54, 60, 61, 67, 72, 81, 83, 88, 90–92
Test VIII, 2, 10, 12–15, 17, 18, 20, 39, 41, 44–47, 50, 52, 56–62, 64, 67–70, 73, 76, 81, 91, 92
  Akzeptanztest 22, 24, 25, 78
  Test-Suite 23, 46, 50, 57, 73, 92
  Testabdeckung . IX, 37, 46–49, 52, 81, 83, 89
  Testdaten .. 9, 13, 37, 39, 56, 57, 60, 75
  Unittest 7, 10, 12–14, 17, 18, 24, 25, 37, 50, 54, 61, 78, 79
Test-Double ..... VIII, 12, 83
  Mock ..... VIII, 14, 37, 67–69, 71, 72
  Stub ..... VIII, 13, 14, 65
Test/Unit-Framwork 13, 35, 38, 41, 52

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. Version: April 2008, Abruf: 07.08.2011

[CAH+10] CHELIMSKY, David ; ASTELS, Dave ; HELMKAMP, Bryan ; NORTH, Dan ; DENNIS, Zach ; HELLESOY, 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/. 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. 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. 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+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

[Gam11] GAME, Computer Language B.: *Ruby 1.9 Speed - C/GNU GCC speed*. 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

[Hai] 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/, 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. 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 Modulvertifikation. 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. 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

[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