

HTW Dresden
Fakultät Informatik
Diplomarbeit



Diplomarbeit

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

Autor: Wienert, Stefan

Seminargruppe 07/041/02

Betreuer Professor: Prof. Dr. Fritzsche

Datum: 5. September 2011

Inhaltsverzeichnis

| | |
|--------------------------------------------------------------------------|-----------|
| 1. Einleitung | 1 |
| 1.1. Motivation | 1 |
| 1.1.1. Die pludoni GmbH | 1 |
| 1.1.2. Arbeitsablauf in der pludoni GmbH | 3 |
| 1.2. Projektbeschreibung und Projektziele | 3 |
| 1.2.1. Anwendungsfälle | 5 |
| 1.2.2. Nichtfunktionale Anforderungen | 5 |
| 1.3. Aufbau der Arbeit | 6 |
| 2. Automatisierte Softwaretests | 6 |
| 2.1. Warum testen | 7 |
| 2.2. Arten von Tests | 7 |
| 2.3. Unitest | 9 |
| 3. Testgetriebene Entwicklung | 10 |
| 3.1. Motivation | 10 |
| 3.2. Ablauf TODO | 11 |
| 3.3. ATDD – Acceptance TDD | 14 |
| 3.4. Prinzip des Emergent Design – Evolutionäres Sofwaredesign | 14 |
| 3.5. Mocks und Stubs | 15 |
| 3.6. Behavior Driven Development | 16 |
| 3.7. Design Driven Testing | 17 |
| 4. Die Programmiersprache Ruby | 17 |
| 4.1. Einführung in Ruby | 17 |
| 4.2. Diskussion | 21 |
| 4.3. Testframeworks für Ruby | 22 |
| 4.3.1. Test::Unit und Minitest | 22 |
| 4.3.2. Cucumber | 24 |
| 4.4. Ruby on Rails | 25 |
| 4.4.1. Konzepte von Rails | 25 |
| 4.4.2. Diskussion | 29 |
| 5. Code-Metriken | 33 |
| 5.1. Überblick über Code-Metriken und Skalen | 33 |

| | |
|------------------------------------------------------------------------------------------------|-----------|
| 5.2. Code-Metriken für Tests | 35 |
| 5.2.1. Lines of Test / Lines of Code | 35 |
| 5.2.2. Testausführungsabdeckung | 35 |
| 5.2.3. Defect insertion | 37 |
| 5.3. Notwendigkeit von Code Metriken | 37 |
| 6. Auswahl der Entwicklungsstrategie und -Werkzeuge | 38 |
| 6.1. Herausbildung einer Entwicklungsstrategie für die Bedürfnisse der pludo-ni GmbH | 38 |
| 6.1.1. Einteilung der Features in Kategorien | 39 |
| 6.1.2. Weitere Bestandteile der Entwicklungsstrategie | 39 |
| 6.2. Auswahl der Entwicklungswerkzeuge | 40 |
| 6.3. Diskussion der Maßnahmen | 42 |
| 7. Anwendung der Testgetriebenen Entwicklung | 42 |
| 7.1. Implementierung von Unit-Tests (Modelltests) | 42 |
| 7.2. Implementierung von Controller-Tests (functional tests) | 48 |
| 7.3. Testen von externen Abhängigkeiten | 52 |
| 7.4. System und Akzeptanztests | 53 |
| 7.5. Testen von Javascript Ereignissen | 54 |
| 7.5.1. Test im Rahmen von Integrationstest | 54 |
| 7.5.2. Unitests mit Javascript | 54 |
| 8. Auswertung | 54 |
| 8.1. Eigenschaften erfolgreicher Tests | 54 |
| 8.2. Vorteile von TDD | 55 |
| 8.3. Nachteile und Grenzen von TDD | 55 |
| 9. Fazit | 56 |
| 9.1. Ausblick | 56 |
| A. Nutzung von Cucumber in Verbindung mit Selenium für Firefox und Guard ohne X-Server | 57 |
| B. Nutzung von deutschen Schlüsselwörtern für Cucumber Features | 59 |
| Abbildungsverzeichnis | 61 |

Acknowledgment

Der Autor möchte auch seinen Dank der Fakultät Informatik und Prof. Wiedemann für das zur Verfügung gestellte Büro, aussprechen. Der Bibliothek der HTW-Dresden und Prof. Nestler sei für das schnelle Bestellen von tagesaktueller Literatur zu danken.

Ich danke auch meiner Frau, die mich während des Schreibens unterstützt hat und bei der Erstellung der Grafiken Tipps gab. Für die orthografische Optimierung sei meinem guten Freund Stefan Koch gedankt.

Diese Arbeit entstand in Zusammenarbeit mit meiner Firma und unter Aufsicht von Jörg Klukas. Ich danke ihm für die Möglichkeit frei zu arbeiten und viele neue Dinge auszuprobieren und für das Vertrauen, dass er in mich gesteckt hat.

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

Glossar

Im folgenden werden einige oft-verwendete Begriffe näher erläutert.

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

Test oder Testfall ist eine, meist automatisierte, Prüfung des Programmverhaltens bei definierten Eingabeparametern

Test-Suite ist eine Sammlung von mehreren Testfällen für eine Komponente oder das gesamte System

Code-Qualität beinhaltet die Qualitäten Lesbarkeit, Testbarkeit, Wartbarkeit, Erweiterbarkeit, Geringe Komplexität

Metrik Eine Softwaremetrik ist das Ergebnis einer statischen oder dynamischen Codeanalyse zur Generierung von Informationen über den Source-Code. Beispiele: Testabdeckung, Anzahl Codezeilen, Anzahl Bad Smells pro Codezeile.

Testabdeckung auch: 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.

C0: Messung jeder Zeile, ob diese ausgeführt wurde

C1: Messung jedes Zweigs jeder Zeile, ob dieser ausgeführt wurde

C2: Messung jedes möglichen Codepfades, ob dieser ausgeführt wurde

Bad Smell oder Code-Smell. Ist ein Anzeichen für eine suboptimale Code-Stelle, die auch ein Hinweis auf ein größeres Design-Problem sein kann. Oft auch ein Kandidat für ein Refaktoring

TDD Test-Driven-Development/Test-Driven-Design, bezeichnet den Prozess der Testgetriebenen Entwicklung

BDD 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.

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 Gamme, Helm, Johnson, Vlissides (Gang of Four).

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

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 [3.5](#)

RoR, Rails ist eine gängige Kurzform für das Webframework Ruby on Rails

1. Einleitung

1.1. Motivation

Kurz nach der Firmengründung der pludoni GmbH absolvierte der Autor dort sein Pflichtpraktikum, und war bis zum heutigen Tag als Werkstudent tätig. Während dieser Zeit nahmen Programmierer verschiedener Erfahrungsstufen und insbesondere auch Praktikanten an der Neu- und Weiterentwicklung der Webserver-Software teil. Dies hat zur Folge, dass die Komplexität der Software inzwischen ein Level erreicht hat, dass das Maß an Regressionsfehlern¹ stark anstieg.

Da zum großen Teil keine automatisierten Softwaretests geschrieben wurden, lassen sich diese auch nur schwer aufspüren. 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 refaktoriert werden, aber da keine Tests vorhanden sind, ist dies aufgrund der Regressionsfehler riskant. Ein Teufelskreis!

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. Die testgetriebene Entwicklung ist dabei ein populärer Entwicklungsprozess der eigentlich als Ziel hat, gut testbaren, d.h. gut wartbaren und lesbaren Code zu schreiben. Quasi nebenbei erhält man aber auch eine umfassende Test-Suite, und kommt so dem Schritt, funktionierende Software zu entwickeln, einen Schritt näher.

1.1.1. Die pludoni GmbH

Die pludoni GmbH ist ein junges dynamisches Dresdner Unternehmen, dass sich zum Ziel gesetzt hat, lokale Job-Communitys 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.de².



pludoni GmbH

¹fehlerauslösender Quelltextänderungen

²<http://www.itsax.de/>, <http://www.itmitte.de>, [http://www.mintsax.de/](http://www.mintsax.de)

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

Tabelle 1: Übersicht über pludoni Communitys. Stand August 2011

| Community | Branche | Region | Mitglieder |
|----------------|------------------------------|-----------------------------------------------------------|------------|
| ITsax.de | IT, Software | Sachsen (Fokus Süden und Osten) | 26 |
| ITmitte.de | IT, Software | Mitteldeutschland (Thüringen, Sachsen-Anhalt, NW-Sachsen) | 20 |
| MINTsax.de | Maschinenbau, Elektrotechnik | Sachsen | 17 |
| OFFICESax.de | Vertrieb, Bürotätigkeiten | Sachsen | in Vorb. |
| OFFICEmitte.de | Vertrieb, Bürotätigkeiten | Mitteldeutschland | in Vorb. |

Diese Unternehmen, die einen jährlichen Mitgliedsbeitrag an die pludoni GmbH zahlen, dürfen Ihre für die Region relevanten Jobangebote auf dem jeweiligen Portal einstellen. Was die Jobcommunitys von pludoni von der Konkurrenz unterscheidet, ist das sogenannte **Empfehlungssystem**.

Viele der Personalchefs 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, eine Einstellung verhinderte. Hier setzt pludoni mit seinen Jobcommunitys an, und stellt eine Infrastruktur zur gegenseitigen Empfehlung dieser guten 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. Die Software löst intern den Empfehlungscode auf und bestätigt dieser Firma nun, dass der Bewerber empfohlen wurde.

Dieses Empfehlungssystem überzeugt die beteiligten Unternehmen. Aktuell wurden im letzten Jahr z.B. auf ITmitte.de über 800 Bewerberungen über das Portal versendet, von denen mehr als die Hälfte (440) mit Empfehlungscodes versehen waren [Klukas, 2011b]. Dies motivierte mittlerweile 54 Firmen bei den drei pludoni Communitys teilzunehmen [Klukas, 2011a].

1.1.2. Arbeitsablauf in der pludoni GmbH

Da der pludoni GmbH gegenwärtig weniger Büroplätze zu Verfügung stehen, als sie Mitarbeiter hat, findet ein Großteil der Arbeit in Heim- oder Telearbeit statt. Zur persönlichen Abstimmung findet aber mindestens einmal pro Woche ein Meeting statt, in welcher sich 2-4 der Mitarbeiter treffen, um alte Aufgaben abzunehmen und neue zu diskutieren. Die Abnahme erfolgt dabei durch den Chef Jörg Klukas.

Zentrales Kommunikationsmittel der pludoni GmbH ist, neben der E-Mail, die Online-Aufgaben- und Fehlerverwaltung, Redmine³. 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 ist eine Heimarbeit aber immer mit Nachteilen in der Kommunikation behaftet. Dies hat bei den Programmierern teils gravierende Auswirkungen auf die Produktivität. Einerseits, da mitunter Code-Teile anderer abwesender Programmierer benutzt werden müssen, andererseits, dass gleiche Funktionalität doppelt implementiert wird, weil der Überblick fehlt. Auch deswegen ist es leicht, bei Arbeiten am Code neue Fehler einzuführen. Für das kommende Projekt soll nun eine großflächige Testinfrastruktur erstellt werden, um das Regressionsrisiko zu minimieren und durch die Tests eine gute Quelltextdokumentation zu erhalten.

1.2. Projektbeschreibung und Projektziele

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

Als Ergänzung zu den lokalspezifischen Jobcommunitys mit strenger Mitgliederauswahl soll nun ein neues, allgemeines IT-Jobportal entwickelt werden. Der vorraussichtliche Name wird IT-Jobs-Und-Stellen.de⁴ sein.

Ziel soll es sein, eine konkurrenzfähige Alternative zu den Branchenprima stepstone.de, monster.de und jobscoach24⁵ zu entwickeln. Analog zu Abbildung 1 wurden die Anwendungsfälle wie folgt analysiert.

³<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*

⁴<http://www.it-jobs-und-stellen.de/>

⁵<http://www.stepstone.de> - <http://www.monster.de> - <http://www.jobscoach24.de>

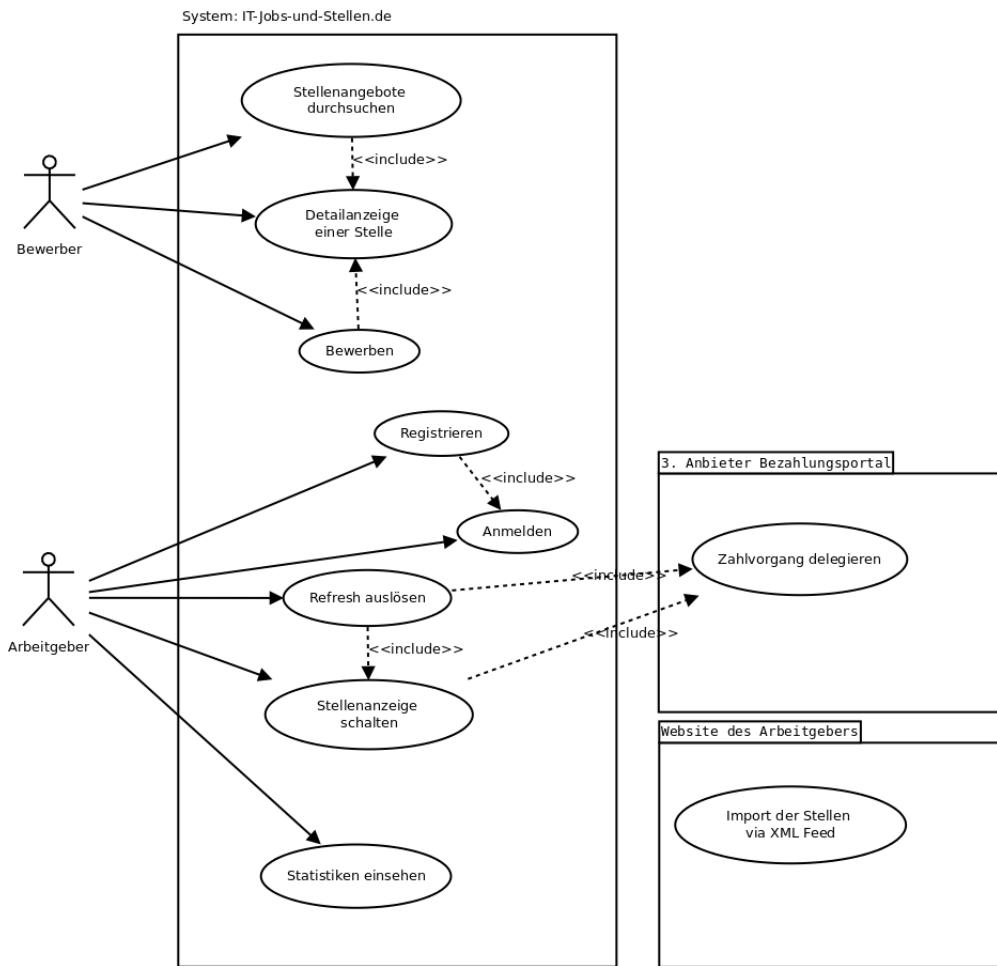


Abbildung 1: Anwendungsfälle

1.2.1. Anwendungsfälle

Es gibt im 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 Kundenunternehmens, 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, um so automatisch einen Lebenslauf generieren 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 einen Drittbieterportal (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 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
- Ein Arbeitgeber kann Statistiken über seine geschalteten Stellenanzeigen einsehen. Dies beinhaltet die Anzahl der Zugriffe pro Tag, 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 holt nun alle 6h liest nun alle Stellen aus diesem Feed ein. Das Format ist eine Modifikation von RSS2.0

1.2.2. Nichtfunktionale Anforderungen

Folgende Rahmenbedingungen und zusätzliche Ziele wurden vereinbart.

- Eine hohe C0-Testabdeckung von mindestens 95% als Grundlage für den TDD Prozess
- Eine hohe Erweiterbarkeit, um langfristig auch die bereits vorhandenen Jobcommunitys durch das neue System zu ersetzen, welche gegenwärtig auf Drupal 5 (PHP) basieren
- Eine moderne Suchfunktion durch einen Suchdaemon realisieren, 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

Diese Arbeit ist in 4 Teile mit insgesamt 11 Abschnitt gegliedert. Abschnitt 1 bis 3 haben einleitenden Charakter. In den darauffolgenden Abschnitten 4 bis 7 werden theoretische Grundlagen und Technologien, die für das die Aufgabenstellung relevant sind, näher beleuchtet. Dies beinhaltet eine Einführung in die Sprache Ruby, das Framework Ruby on Rails, sowie Automatisierte Softwaretests und den Prozess der Testgetriebenen Entwicklung. In den Abschnitten 8 bis 10 werden dann praktische Erfahrungen und Implementationsdetails dargestellt. Im 11. und letzten Abschnitt wird ein Fazit gezogen und ein Ausblick für weitere Forschungstätigkeit zum Thema dargeboten.

2. Automatisierte Softwaretests

Zum Prüfen der Korrektheit 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 müssen, um Regressionsfehler auszuschließen.

Stattdessen werden automatisierte Softwaretests instrumentalisiert, 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 in

kürzester Zeit zeitlich überlegen [Rappin, 2011]. Ein solcher Test besteht in der Regel aus 4 Teilen:

1. Initialisierung der Test-Umgebung und der Objekte
2. Ausführung der zu testenden Aktion, die den Systemzustand ändert
3. Spezifikation von Erwartungen (Assertions)
4. Aufräumen nicht mehr benötigter Objekte, File-Pointer, Sockets u.ä

2.1. Warum testen

Tests dienen in erster Linie dazu, das Vorhandensein bzw. Nichtvorhandensein von Software-Fehlern zu belegen [Goodliffe, 2006].

Getester Code gilt im Allgemeinen als robuster, korrekter und leichter zu warten [Rappin, 2011]. Im Umkehrschluss bedeutet dies, drastisch formuliert, dass ungetestete Software vorherbestimmt ist, Fehler zu haben [Goodliffe, 2006].

Die meisten Programmierer finden Testen besser als Debuggen. Testen führt zu einer Minimierung der Debugphase und macht den Software Entwicklungsprozess für Programmierer attraktiver und für Projektleiter leichter zu planen [Orsini, 2007].

2.2. Arten von Tests

Tests können nach verschiedenen Gesichtspunkten eingeteilt werden. In vielen Fällen ist die Einordnung sehr schwammig, und Tests können zu mehreren Kategorien eingeteilt werden.

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. Blackboxtests dagegen ignorieren den inneren Aufbau der Klassen und Testen entweder nur Schnittstellen oder das Gesamtsystem, fokussieren also Aktionen und Rückmeldungen des Systems. Das Ziel eines Blackbox-basierten Tests, ist die Korrektheit der Software gegenüber den Spezifikationen. Ein Spezialfall sind die sogenannten **Greybox**-Tests, die insbesondere bei der Testgetriebenen Entwicklung auftreten. Da der Test zuerst entwickelt wird, ist noch kein Wissen über den Zielquellcode vorhanden.

funktionale Tests

```
class Auto
def bremsen
end
def ...
```

```
class Stadt
class Person
class Auto
def bremsen
end
def ...
```

Whitebox
Test passed: xxxx
Test failed: xxxx

Unitest
Testen der abgrenzbaren Komponenten auf korrektes Ergebnis

auch Komponenten- oder Modultest

Test passed: xxxx
Test failed: xxxx

Integrationstest
Zusammenarbeit der Komponenten durch kompletten Ablauf

```
Wenn ich
Wenn ich
Wenn ich
Als ein Nutzer
klicke
Dann sehe ich
...
```



Systemtest
gesamtes System wird gegen die Anforderung getestet

Akzeptanztest
wie Systemtest, aber Test erfolgt durch Kunden/Abnehmer

nicht-funktionale Tests

Stress/Lasttest
Verhalten in Extrem-situationen, Zuverlässigkeit

Performanztest
Antwortzeit, Skalierbarkeit, Nutzung von Ressourcen

...

Usabilitytest
Verständlichkeit und Konsistenz des Nutzer-Interfaces

Sicherheitstest
Test auf Sicherheitslücken und Programmierfehler

...

Bildquelle: Der Author

Abbildung 2: *Einteilung der Tests*

Einteilung nach Testziel (nach Andrew Hunt [1999][S. 238ff])

Unittests Hierbei werden die atomaren Einheiten des Programmes auf ihr Verhalten getestet. Dies stellt die Basis für die meist darauffolgenden Integrationstests dar.

Integrationstests Im Grunde wie die Unittests, allerdings wird das Zusammenspiel zwischen Klassen getestet, welche ein gemeinsames Subsystem darstellen

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/Feature ab.

Ressourcennutzung, Performanz, Verhalten im Fehlerfall Die vorherigen Tests finden i.d.R. unter idealen Bedingungen statt. Diese Testkategorie 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 präparierten Umgebung (Usability Labor) geprüft.

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 atomaren Einheiten eines Programms. Dies können die Funktionen, oder bei einer objektorientierten Sprache, die Klassen sein. Unittests werden in strenger Isolation ausgeführt. Externe Abhängigkeiten wird durch ein Test Double simuliert. Dies ist notwendig um sicherzustellen, dass gefundene Fehler von dem betreffendem Modul verursacht wurden, und nicht durch äußere Einflüsse. Diese Isolierung macht das Testen einfacher [Goodliffe, 2006].

Unitests werden fast immer automatisiert ausgeführt. Verwendung finden dabei meist immer sogenannte Test-Frameworks. Eines 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#.

Komponententest Ziel des Komponententest ist es, verschiedene Units in Kombination als eine vollständige Komponente zu testen [Goodliffe, 2006].

3. Testgetriebene Entwicklung

Testgetriebene Entwicklung, im Englischen Test-Driven-Development (TDD) wurde erstmalig von Kent Beck 2003 im Detail erläutert. Zuvor war die Technik „Test-First“ aber schon seit 1999 im Kontext von Extreme Programming (XP) bekannt.

Test-Driven-Development wird als

$$\text{TDD} := \text{Test-First} + \text{Refaktorisieren}$$

beschrieben [Ambler, 2002]. So ist es Ziel, dass sich Test schreiben/Implementation und Refaktorisierungen, d.h. das konstante Verbessern des Systemdesigns und des Quellcodes, abwechseln. Damit wurde aus der Entwicklungs-Methode „Test-First“ ein Software-Entwicklungsprozess.

Im Folgenden werde ich diesen Prozess näher beleuchten und am Beispiel von Ruby on Rails typische Testwerkzeuge aufzeigen.

3.1. Motivation

Das Erstellen einer gut abdeckenden Test-Suite für ein jedes größeres Softwareprojekt ist eine wichtige Voraussetzung um interne Qualitäten, wie Wartbarkeit und Zuverlässigkeit zu aktivieren. TDD soll nicht dazu dienen, die Software zu verifizieren. Dies ist aber ein positiver Nebeneffekt. Das Hauptziel ist es, den Code in Einklang mit dem Test zu schreiben, so dass der Test den Code antreibt (Test drives thes code). Der messbare Effekt davon, ist ein gut-testbarer Code, welcher in der Regel auch ein gut-wartbarer und verständlicher Code ist. Bad-Smells, wie God-Methode und geringe Kohäsion, werden schon im Keim ersticken werden, 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“ Management von Angst. So hat Angst verschiedene Auswirkungen auf die Entwicklung. Sie mache zögerlich, führe zu weniger Kommunikation und Feedback und mache den Programmierer „mürrisch“ [Beck, 2002, S. xi].

Falls TDD die Fehlerdichte signifikant verringern würde und nur Code entstünde, der getestet wurde, so hätte dies auch soziale Auswirkungen auf das Entwicklerteam [Beck, 2002, 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

TDD fördert die Entwicklung in kleinen Schritten, und ermöglicht durch bestehende Tests kleine „Belohnungen“ für den Programmierer. Dadurch ist es leichter einen gewissen Arbeitsrhythmus zu erhalten, was stellenweise dem „Flow“⁶ ähnelt oder diesen strukturiert ergänzen kann [Brown, 2008].

3.2. Ablauf TODO

Ziel ist es, vor der Implementation eines Codes, einen Unitest zu implementieren. Davon ausgehend soll der geringstmögliche Code implementiert werden, damit der Test besteht. Zum Schluss wird refaktoriert, bei TDD auch als Designphase genutzt.

Im Detail sind das also folgende Phasen, vgl. Abbildung 3:

1. Schreibe einen neuen Test. Dies kann der erste eines neuen Features sein, oder aber ein Test, um Funktionalität zum aktuellen Feature hinzuzufügen
2. Red: Führe alle Tests aus, um sicherzugehen, dass der Test fehlschlägt. Andernfalls ist der Test überflüssig.
3. Green: Nachdem der Test fehlschlägt, implementiere nun den einfachsten Code, damit der Test besteht

⁶Schaffen-, Tätigkeitssrausch

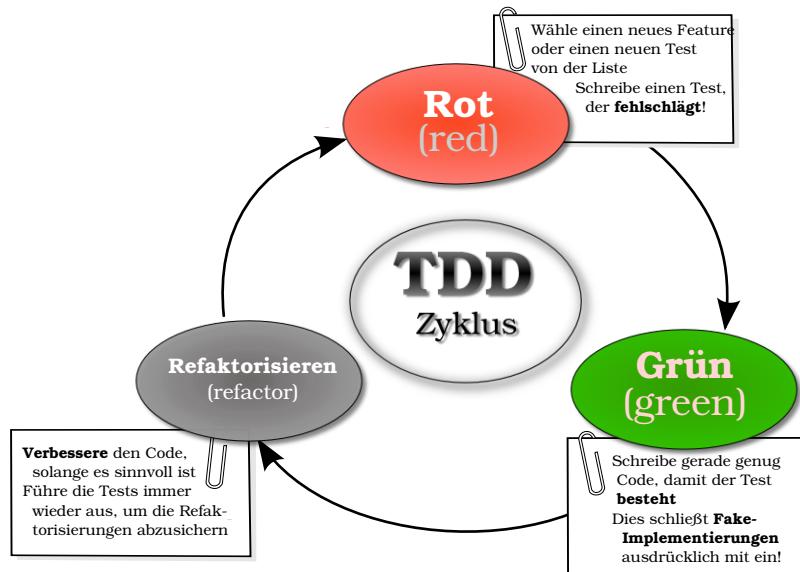


Abbildung 3: Red-Green-Refactor: Der TDD Entwicklungszyklus

Bildquelle: Der Autor

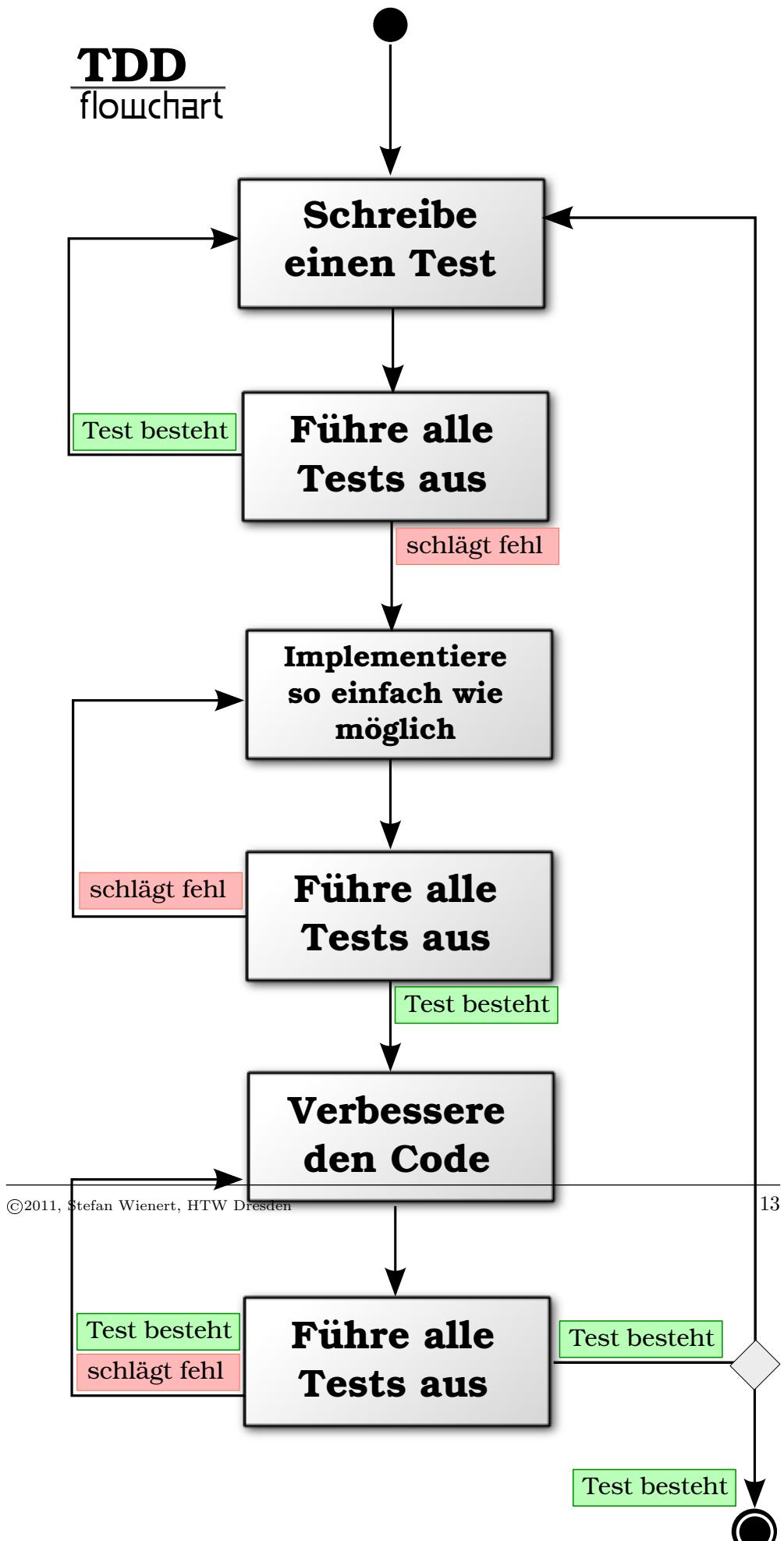
Dies kann ausdrücklich auch eine Fake-Implementierung sein, also z.B. die Rückgabe eines konstanten Wertes anstelle einer Berechnung. Wichtig ist, dass diese Phase so schnell wie möglich verlassen wird.

4. Refactor: Nachdem der Test bestanden wird, folgt nun die **wichtigste Phase**, die Refaktorisierungsphase.

Da wir bereits einen Test haben, der unser gewünschtes Systemverhalten widerspiegelt, können wir gefahrlos refaktorieren, d.h. meist Duplikation eliminieren. In dieser Phase findet das Design des Codes statt. Man macht sich Gedanken, wie die vorhanden Klassen optimal refaktorieren können, um Code-Smells zu eliminieren, und welches Entwurfsmuster angewendet werden kann.

Ein genau spezifizierter Ablauf ist in Abbildung 4 zu finden.

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



Im Prinzip soll jeder Änderung der Programmlogik ein fehlgeschlagener Test vorausgehen. Auch bei der Behebung von Defekten, also dem Bugfixing, soll zuerst ein Test geschrieben werden, der genau das Verhalten des Bugs widerspiegelt, und erst dann gefixt werden.

3.3. ATDD – Acceptance TDD

Die Akzeptanztest-getriebene Entwicklung ist eine Modifikation von TDD. Statt der Unitests, stehen hier die Akzeptanztests im Vordergrund

Das ganze lässt sich auch in den übergeordneten Prozess zum Entwickeln eines Features einordnen.

1. Schreibe einen Akzeptanztest/Systemtest um das aktuelle Feature zu implementieren
2. Implementiere die Teilschritte, die notwendig sind, um den Akzeptanztest bestehen zu lassen. Verfahre bei der Implementation nach dem TDD Schema zur Implementierung der benötigten Units wie oben.
 - a) Schreibe einen Unittest
 - b) Prüfe, ob der Test fehlschlägt, andernfalls zurück zu 1.
 - c) Implementiere mit so wenig wie möglich so, dass der Test besteht
 - d) Refaktorisiere
3. Nachdem der Akzeptanztest besteht, Prüfe etwaige Refaktorisierungen für die Anwendungsebene.

Somit werden 2 Testebenen erstellt, die Akzeptanz- und die Unitests.

3.4. Prinzip des Emergent Design – Evolutionäres Sofwaredesign

Ein Konzept von TDD ist das, des sich Herausbildenden Designs. Gegenüber traditionellen Entwicklungsansätzen erfolgt die Entwicklungsphase (Design) nicht als eigenständige Phase, sondern ist streng in den Entwicklungsprozess integriert. Immer wenn ein Zyklus beim Refaktorisieren angelangt ist, findet 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 benötigten Design suchen, im Gegensatz zu einem Software-Design, das im Vorfeld bedacht wurde.

Durch die vielen Iterationen und die darausfolgenden zahlreichen Refaktorisierungen tritt nach und nach das Design hervor, welches optimal für das System ist.

Einige Software-Architekten (Ford, Reeves und Vanderburg) proklamieren, dass Software-Engineering eigentlich keine ordinäre Ingenieursdisziplin sei. Ingenieure sind für die Planung und das Design verantwortlich, welches in aller Regel im Vorfeld der Implementierung stattfindet. Das klassische Software-Engineering empfindet diesen Prozess nach, was sich z.B. in dem Wasserfallmodell äußert. Diese Autoren äußern nun, dass dies für die meisten Softwareprojekte nicht ideal sei, da sich die Businessanforderungen meist im Laufe einer Entwicklung ändern. Traditionelles Softwaredesign abstrahiere zu früh und spekuliere ohne die letztendlichen Fakten zu kennen. Traditionelle Ingenieurswissenschaftliche Disziplinen haben außerdem die Beschränkung, dass ein Build-Prozess äußerst teuer ist (man denke an Brücken oder Chips in Flugzeugsystemen), wohingegen dies bei Software in der Regel fast nichts kostet. Dadurch kann die Software-Entwicklung ein iteratives Vorgehen nutzen [Ford, 2010], [Vanderburg, 2010], [Reeves, 1992]. Statt eines großen Designs am Anfang (BDUF – Big Design Upfront) soll das Design durch Entdeckung und Extrahieren aus dem Sourcecode gewonnen werden.

Ein Emergent Design kann auch ohne TDD verwendet werden. Allerdings ist dies ohne das Vorhandensein von einer guten Test-Suite ein risikoreiches Unterfangen. Mit seiner iterativen Herangehensweise passen Emergent Design in Kombination mit TDD dagegen perfekt in den Entwicklungszyklus der Agilen Software Entwicklung.

3.5. Mocks und Stubs

Beim Testen im Allgemeinen und bei TDD im Besonderen wird das Testen durch externe Abhängigkeiten erschwert. Dies können z.B. Klassen sein, die noch gar nicht implementiert wurden, externe Ressourcen (Netzwerkzugriffe, Versenden von Mails) oder externe Prozesse (Bezahlen in einem Onlineshop) sein. In diesen Situationen ist es angebracht, auf sogenannte Test-Doubles, meist Mocks und Stubs zurückzugreifen.

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

```
def test_report_failed_payment
    Payment.stubs(:pay).returns(false)

    bestellung = Bestellung.new()
    bestellung.commit()
```

```
assert bestellung.errors.present?  
end
```

Mit dem oben angegeben (Pseudo-) Rubycode würde man z.B. mittels des Mock-Frameworks „mocha“, ein Mockobjekt erzeugen, welches den Bezahlprozess nachahmt, und die Methode „pay“ ersetzt, so dass sie immer „false“ zurückgibt. So kann z.B. das Objekt Bestellung gefahrlos Bezahlungen auslösen. Der genaue Ablauf innerhalb des Bezahlprozesses ist die Bestellung unwichtig, lediglich der Rückgabewert, ob die Bezahlung erfolgreich war oder nicht.

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 als Testfall. Ein Mock wartet darauf, ob die Methode, wie sie definiert wurde, auch tatsächlich aufgerufen wurde.

Hier z.B. ein Mock, um statt des heutigen Datums das von vor einer Woche zurückzugeben

```
def test_always_fail  
  Date.mocks(:today).returns( 7.days.ago)  
end
```

Der gezeigte Programmcode wird jedesmal fehlschlagen, da von einem Mock erwartet wird, dass er während des Tests genau einmal aufgerufen wird. Ist dies nicht der Fall, gilt der Test als nicht bestanden. Mocks fungieren somit als zusätzliche Möglichkeit Interna des Programmflusses zu testen.

3.6. Behavior Driven Development

Eine oft erwähnte Variante von TDD ist die Verhaltensgetriebene Entwicklung (Behavior-Driven-Development – BDD). Dabei dienen hier Akzeptanztests als treibende Kraft der Entwicklung. Diese beschreiben ein Verhalten (Behavior). Der Fokus liegt also nicht auf Implementationsdetails, sondern soll, in einer domainspezifischen Sprache, das Verhalten und die daraus resultierenden Erwartungen des Systems beschreiben.

Dies drückt sich meist auch in dem Vokabular aus. Während bei klassischen Unit-Tests, und damit auch bei TDD, die Begriffsdomain „Zusicherungen“ (assertions) und „Tests“ beinhaltet, so hat BDD stattdessen „Erwartungen“ (expectations) und „Spezifikationen“ (specs/specifications), und verwendet oft das Modalverb „sollte“ (should).

3.7. Design Driven Testing

Design Driven Testing soll eine Umkehrung von Testgetriebener Software sein und wird Stephens und Rosenberg als Alternative dazu vorgeschlagen [Stephens and Rosenberg, 2010]. Sie kritisieren, dass TDD in Reinform betrieben, lediglich Unitests, aber keine Dokumentation oder höhere Tests höherer Levels produziert. 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 proklamieren die Nutzung von Akzeptanz- anstelle der Unitests. 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 Projektteam von it-jobs-und-stellen.de ein sehr kleines ist, wird auf diesen Entwicklungsprozess nicht näher eingegangen.

4. Die Programmiersprache Ruby

Ruby ist eine Programmiersprache, die ab 1993 von Yukihiro Matsumoto 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 und imperative Programmierung ermöglicht [Team, 2011].

Eine vollständige Einführung in Ruby zu geben würde den Rahmen dieser Diplomarbeit sprengen, weswegen ich mich auf die Herausstellung der Hauptmerkmale und Unterschiede zu anderen Sprachen konzentrieren werde, und was die Auswirkungen auf das Testgeschehen sind.



Ruby ist eine Multiparadigma-Sprache

4.1. Einführung in Ruby

Ruby ist 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

Ruby ist eine interpretierte Sprache, auch Skriptsprache genannt. Dies heisst, dass der Programmcode zur Laufzeit analysiert und ausgeführt wird.

Ruby nimmt für sich in Anspruch, eine Sprache für Menschen, und nicht für Maschinen zu sein. Dies drückt sich durch eine Syntax, die oft laut als englische Sprache gelesen werden kann, aus. Auch hält Ruby eine Vielzahl von redundanten Keywords bereit (Syntaktischer Zucker), um dem Programmierer mehrere Wege zur Lösung seines Problems zu ermöglichen.

Im Nachfolgenden einige Beispiele für die Verwendung von Ruby, insbesondere die „Alles ist ein Objekt“-Philosophie.

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

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

Yukihiro
Matsumoto

Listing: Ruby Beispiele

In den ersten beiden Beispielen sieht man, dass Integer und String Objekte sind, und über Memberfunktionen 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 letztendlich in Methodenaufrufe gemappt werden, wird die Methode „.+ (2)“ auf dieses Objekt ausgeführt, und liefert ein neues Datumsojekt, welches sich um 2 Tage unterscheidet, zurück. Im Vierten Beispiel wird der Einsatz von Variablen demonstriert. Das letzte Beispiel zeigt den Einsatz von Kontrollstrukturen. Als Besonderheit seien hier auf die Range „(0..10)“, die ein Intervall für den Integerzahlenbereich von 0 bis einschließlich 10 liefert. Die

Methode „.include?(a)“ tests nun, ob die Variable „a“ in diesem Intervall existiert. Bei Eindeutigkeit können 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.

Listing 2: Ruby Beispiel: Lambdas

```
>> 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]
```

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

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

>> Fixnum.superclass
=> Integer
>> Fixnum.ancestors
=> [Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]
```

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

oder Mixins unterstützt, kann eine Klasse auch eine Menge von Oberklassen haben. Die gezeigte Klasse Fixnum verfügt somit standardmäßig sogar über 7 Oberklassen.

Ruby ist dynamisch stark typisiert, d.h. dass die Zuweisung des Typs einer Variable zur Laufzeit des Programms geschieht. Der Typ einer Variable ergibt sich damit aus ihrem beinhalteten Objekt. Durch die starke Typisierung ist es aber nicht möglich, invalide Operationen auf inkompatible getypte Daten 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 seinen Methoden und Attributen bestimmt wird.

Ruby verfügt über eine lexikalische und dynamische Bindung⁷, letztere wird allerdings seltener verwendet. Im der Basissyntax verwendet Ruby statische Bindung. Es gibt eine im Ruby-Core enthaltene Bibliothek „Dynamic“ zum dynamischen binden.

Reflektion und Introspection 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. Ruby „merkt“ sich allerdings die überschriebenen Methoden und ein Aufruf der überschriebenen Methoden ist stets mittels „super“ möglich.

”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

Listing 3: Ruby Beispiel offene Klassen

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

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

Diese Beispiele sollten als kurzer Einstieg in Ruby dienen, und einen Querschnitt durch die Besonderheiten der Sprache aufzuzeigen.

Für eine weiter Vertiefung sei das Buch „Programming Ruby 1.9“ empfohlen, das im Detail auf die neuste Version der Programmiersprache eingeht [?].

⁷ **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

4.2. Diskussion

Ruby als Skriptsprache Dynamisch typisierte Sprachen, wie Ruby, haben gegenüber klassischen statisch typisierten Sprachen einige Nachteile. Zu allererst wird oft der Geschwindigkeitsnachteil angesprochen, den der Prozess des Interpretierens und das fehlende statische Typsystem verursachen. Allerdings hat dies in der Regel gravierende Geschwindigkeitsnachteile. Der genaue Faktor variiert extrem, je nach Algorithmus. 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 4-300 fache langsamere Ausführungszeit. Dem gegenüber steht allerdings jeweils nur die Hälfte bis 1/7 der Menge an Code [Game, 2011].

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 [Meijer and Drayton, 2005] [Ousterhout, 1998].

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 [Spiewak and Harrop, 2010].

Generische Programmierung Ruby ermöglicht es Programmcode zu schreiben, der wiederum Programmcode schreibt. Dies ermöglicht es Probleme effektiv zu lösen, die andernfalls nur mit erheblichem Aufwand, oder gar nicht zu lösen sind. So verwendet das beliebte Objektrelationale Datenbankframework ActiveRecord dies, um einfache SQL-Statements zu erstellen. Ruby on Rails verwendet standardmäßig ActiveRecord als Schnittstelle zur Datenbank.

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

Die Methode `find_by_first_name` existiert nicht, und wird zur Laufzeit auf Basis des Namens gebaut.

Auch das sehr beliebte Testframework RSpec, verwendet Metaprogrammierung, um Testfälle und Zusicherungen wie fast in der englischen Sprache zu formulieren. Dazu werden sämtliche Objekte von Ruby um Funktionen erweitert. Dies ist möglich, da die Klasse „Objekt“, die Basisklasse (fast) aller Ruby-Klasse ist, um diese Methoden erweitert wurde.

```
>> 4.should == 3
RSpec::Expectations::ExpectationNotMetError: expected: 3
```

All diese Methoden können, richtig angewendet, zur Verbesserung der Lesbarkeit der Programme, und damit zur Erhöhung der Wartbarkeit, führen.

Schlussfolgerung Die Verwendung von Ruby und anderen dynamischen Sprachen birgt durchaus Risiken, die zu beachten sind. Falls man diese Risiken im Kopf behält, und die Möglichkeiten der Sprache nutzt, um die Lesbarkeit zu verbessern, sind sie gerechtfertigt. Gerade bei der Entwicklung kleinerer Entwicklerteams oder Projekten mit engem Budget können dynamische Sprachen ihre Vorteile ausspielen, da sie eine schnellere Entwicklung ermöglichen. Im Gegensatz zu den meisten auf C basierten Sprachen, ist die Syntax von Ruby äußerst leserlich, da nur wenige Sonderzeichen verwendet werden. Auch bietet Ruby mehr Funktionalität pro Programmzeile, da die Deklaration entfällt und es viel sogenannten syntaktischen Zucker gibt. Auch dies kann, richtig angewendet, der Lesbarkeit zuträglich sein.

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

4.3. Testframeworks für Ruby

4.3.1. Test::Unit und Minitest

Test::Unit (Ruby 1.8.7) und Minitest (1.9.2) sind die Testbibliotheken, die Ruby standardmäßig mitbringt. Beide basieren auf dem xUnit-, bzw. SUnit-Design von Kent Beck, und sind für Nutzer von JUnit oder NUnit leicht nachvollziehbar.

Für eine zu testende Klasse wird eine analoge 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 `test-job.rb` (Ruby Standard) oder `job-test.rb` (Rails Standard) erstellt. Dort wiederrum wird eine Klasse mit Namen `TestJob` definiert.

Ein Beispieltest sieht z.B. so aus:

Listing 4: Testen mit Test::Unit in 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
```

Unsere Klasse `TestJob` erbt von der `TestUnit` Basisklasse. Sie beinhaltet die Methoden „`setup`“ und „`teardown`“, die jeweils vor, respektive 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

Danach können nun beliebig viele Testmethoden folgen, deren Namen mit `test_` beginnen müssen. Jede Testmethode besteht in der Regel aus einer Initialisierung (kann in die `setup`-Methode ausgelagert werden), der Ausführung einer zu testenden Aktion und dem Prüfen der danach geltenden Eigenschaften mittels Assertions. Diese Zusicherungen sind Prädikate die oft Gleichheit oder Boolesche Rückgabewerte prüfen.

Rails erledigt das Anlegen und Löschen von Testdaten selbstständig. Diese werden als Fixtures bezeichnet und extern definiert. Alternativ ist der Einsatz sogenannter Factories möglich, um schnell Objekte mit bestimmten Eigenschaften zu erstellen. In jedem Fall setzt Rails die Datenbank nach jedem einzelnen Test zurück.

4.3.2. Cucumber

Cucumber ist ein relativ neues Framework (2008), um mittels einer domainspezifischen Sprache verständliche automatisierte Tests zu schreiben. Dabei gibt es 2 Ebenen. In der obersten werden Tests 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 unterstützt gezeigt.

Listing 5: Cucumber: Additionsfeature in Deutsch

```
# 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"
```

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:

Listing 6: Cucumber: Implementierung der Additionstestschritte in Ruby

```
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)
end
```

Wie man sehen können werden die Testschritte über Regex gemacht. Jeder Schritt 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 Originaldialekt. Die Einteilung in klare Testschritte fördert die Wiederverwendbarkeit der Testschritte selbst.

Der Vorteil von Cucumber ist nun, dass diese Feature-Datei zusammen mit dem Kunden durchgesprochen werden kann, und am Ende eine funktionale Validierung durchzuführen. Auch hilft es, nur diejenigen Features zu implementieren, die letztendlich einen Wert für das Geschäft haben werden.

In der Regel werden mit Cucumber Webanwendungen getestet. Dabei kann z.B. auch ein echter Browser ferngesteuert werden, um den Test so authentisch am echten Nutzungsprozess wie möglich zu orientieren.

Neben Ruby, wird auch die Implementierung der Testschritte in JVM und .NET-Sprachen unterstützt.

4.4. Ruby on Rails

Für das Projekt IT-jobs-und-stellen.de soll das Webframework Ruby-on-Rails verwendet werden. Rails wurde 2006 von der Firma 37signals unter der Leitung von David Heinemeier Hansson entwickelt und erlangte seitdem eine wachsende Popularität. Rails inspirierte viele andere Frameworks, wie z.B. cakePHP, Groovy on Grails, Symfony und ASP.NET MVC.



Viele professionelle Websites, die meist als Startup begannen, setzen bis heute auf Rails. Darunter z.B. Yellow Pages, die Gelben Seiten der USA, Github, eine sehr beliebte Community für OpenSource Programmierer, Groupon, dem führenden Unternehmen bei Online-Gutscheinen und XING, einer deutschen Online-Community für Business-Kontakte.

Im Folgenden werden die Grundzüge von Ruby on Rails näher erläutert.

4.4.1. Konzepte von Rails

Rails ist ein Webframework, das auf dem Model-View-Controller-Pattern basiert, welches eine 3-schichten Architektur darstellt. Jede Schicht hat fest definierte Aufgaben. Diese bilden normalerweise ein Dreigespann, bei Rails „Ressource“, genannt. Im folgenden werden die Schichten kurz erläutert, und am Beispiel einer Ressource „Job“,

Model In Klassen dieser Schicht werden Zugriffe auf die Persistenz vorgenommen.

Meist geschieht dies durch Ausführung von SQL-Befehlen. Innerhalb von Rails ist dies aber meist nicht notwendig, da das ORM⁸ ActiveRecord häufig verwendete SQL-Befehle abstrahiert. Auch die Geschäftslogik soll per Definition zu großem Teil in dieser Schicht erfolgen.

⁸Objektrelationale Framework

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 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.

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 (Wer darf was).

Standardmäßig stellt Rails die CRUD⁹-Operationen bereit, welche in Form eines REST¹⁰

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 wäre das eine View für die Liste aller Jobs, einen Job im Detail anzeigen sowie das Formular zum Anlegen und Bearbeiten eines Jobs.

In Abbildung 5 ist der Ablauf einer Anfrage an den Server dargestellt. Die Anfrage des Browsers an die Website <http://localhost/jobs/12> wird über den Webserver, z.B. Apache2, an die Railsanwendung gestellt. Innerhalb von Rails wird dieser Anforderungsstring anhand der Routen, die die Anwendung anbietet, gematcht. In unserem Falle würde /jobs/12 auf den Controller jobs aufgelöst werden. Innerhalb dieses Controllers wird eine Methode (Aktion) show erwartet. Diese Methode wird nun ihrerseits eine Anfrage an das Model Job stellen, den Job mit der ID 12 aus der Datenbank zu holen. Danach wird ein HTML Template zur Detailanzeige des Jobs generiert.

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 die Datenbankeinstellung wird keine Konfiguration im Vorderfeld benötigt. Diese Methodology zieht sich auch durch das

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.

⁹Create Read Update Delete

¹⁰Representational State Transfer die HTTP-Methoden GET, POST, PUT, DELETE werden in Kombination mit einem definierten URL-Schema direkt auf die Aktionen Auflisten, Anzeigen, Bearbeiten, Löschen, Neu anlegen gemappt. http://en.wikipedia.org/wiki/Representational_State_Transfer

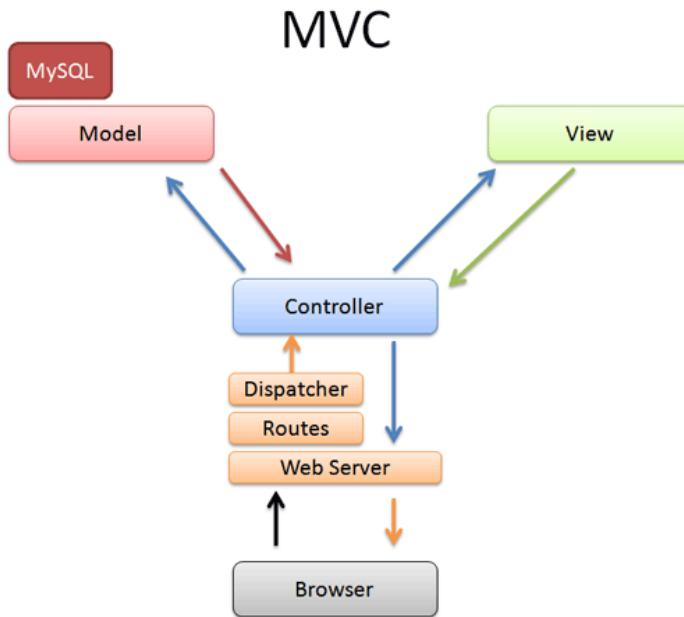


Abbildung 5: MVC Modell von Rails

Quelle: betterexplained.com

Ökosystem durch. Die meisten externen Bibliotheken, bei Ruby Gems genannt, funktionieren bereits nach wenigen Kommandos. Dies macht das prototypische Entwickeln äußert effektiv. Weiterhin ist die Struktur eines Railsprojektes extrem 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, und das ständige Auslagern und Refaktorieren des Codes, um bei Änderungen nur an einer Stelle ansetzen zu müssen. Ein weiteres Beispiel ist die Definition der Spalten des ORMs. Im Gegensatz zu anderen ORM-Frameworks ist diese bei Rails nicht notwendig. Rails erstellt automatisch Getter und Setter für die in der Datenbank definierten Tabellenspalten.

REST Representational State Transfer ist Software-Architektur für HTTP-WebServices.

Dabei werden neben den Standard HTTP Methoden GET und POST auch die selten benutzen Verben DELETE und PUT verwendet, um Aktionen auf einer Ressource zu definieren. Das Ziel ist ein sehr einfaches Design der URLs. Hier folgt die Auflistung der vier CRUD Operationen plus Auflistenvon REST am Beispiel einer Ressource „jobs,, eines Webservice

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 wiederrum 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 es gilt als die bevorzugte Methode in der Community, APIs zu bauen.

Codegeneratoren Rails bietet viele Codegeneratoren an, um schnell benötigte Klassen und Datenbanktabellen anzulegen. Im Railsprojekt reicht z.B.:

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

Damit wird das Model Job, eine Erstellung der Tabelle „jobs,, ein Controller „jobs,, mit den REST-Standardaktionen und entsprechenden Beispielviews, sowie Testfälle für Unit- und Funktionale Tests angelegt. Weiterhin sei zu bemerken, dass durch die Anweisung `user references` eine Spalte `user_id` angelegt wird und eine 1:n-Beziehung zum Modell „user,, hergestellt wird.

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, ERB, JQuery, sowie Test::Unit und Rack. Die meisten dieser Teil-Frameworks lassen sich zwar leicht austauschen, Rails selbst aber proklamiert „opinionated,, also rechthaberisch/eigensinnig, zu sein, und den Entwickler Standards vorzugeben [Hansson, 2011].

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

David Heinemeier Hansson, Begründer von Rails

Eine Einführung und Programmierung in Rails soll nicht Bestandteil dieser Diplomarbeit werden. Für eine weitere Einarbeitung seien die folgende Quellen insbesondere empfohlen:

Rails for Zombies Dies ist ein moderner, interaktiver Onlinekurs. Greg Pollack und das Team von RailsEnv verpackt die Lektionen in humorige interaktive Lernerfahrungen. Jeweils eingeleitet durch ein Video muss der Teilnehmer Aufgaben direkt im Sourcecode lösen. Mithilfe dieses Kurses gelang es mir, meinen Arbeitskollegen einen guten Überblick über Rails zu verschaffen. Die Teilnahme ist kostenlos.

<http://railsforzombies.org/>

Agile Webdevelopment with Ruby on Rails Das quasi-Standardwerk. Wird meist parallel mit einer neuen Rails-Version in einer neuen Auflage gedruckt, aktuell die Dritte [Dave Thomas, 2009].

Rails Guides Die von Ruby-on-Rails herausgegebenen „Rails Guides“ sind eine gut strukturierte, kostenlose Online-Dokumentation, die nahezu alle Aspekte von Rails beleuchten.

http://guides.rubyonrails.org/getting_started.html

4.4.2. Diskussion

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

Vorteile Dank der Modularität können als Persistenzgrundlage sowohl relationale Datenbank, 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 95% der Fälle überflüssig und zudem auch sicherer. Hier ein Beispiel, wie das Anlegen und Auslesen einer Instanz „Vertragsart“, erfolgt. Parallel dazu ist Kommentar die SQL Kommandos, die ActiveRecord im Hintergrund ausführt.

```

c = ContractType.new
c.name = "Vollzeit"
c.save
=> false
#   SQL (0.1ms)  BEGIN
#   SQL (0.3ms)  SELECT 1 FROM `contract_types`
#     WHERE (`contract_types`.`name` = BINARY 'Vollzeit') LIMIT 1
#   SQL (0.1ms)  ROLLBACK
>> c.errors
=> {:_name=>["Name bereits vorhanden. Der Name muss einmalig sein"]}
>> c.name = "Praktikum"
>> c.save
=> true
#   SQL (0.1ms)  BEGIN
#   SQL (0.4ms)  SELECT 1 FROM `contract_types`
#     WHERE (`contract_types`.`name` = BINARY 'Praktikum') LIMIT 1
#   SQL (0.4ms)  describe `contract_types`
#   AREL (0.4ms)  INSERT INTO `contract_types` ('name') VALUES ('Praktikum')
#   SQL (7.1ms)  COMMIT

```

Hierbei ist auch schön zusehen, wie in Rails standardmäßig Transaktionen verwendet werden. Auch sieht man, wie eine Validierung funktioniert. Für das Modell VertragsArt wurde eine Einmaligkeit des Attributs name vereinbart. Dies prüft Rails vor dem Speichern und bricht das Speichern ab, falls eine Prüfung fehlschlug.

Die Implementation dieser Prüfung ist denkbar einfach.

```

class ContractType < ActiveRecord::Base
  validates :name, :uniqueness => true, :presence => true

  has_and_belongs_to_many :jobs
end

```

Wir vereinbaren so, dass das Attribut Name einmalig sein muss (uniqueness), und ausgefüllt sein muss (presence). Die andere Zeile mit has_and.belongs_to_many definiert eine n:m Beziehung mit dem Modell Job, d.h. ein Job hat mehrere Vertragsarten. Der Rest geschieht dann durch Metaprogrammierung.

Mit minimalem Code lassen sich so komplexe Probleme abbilden. Dies macht Rails zu dem hochproduktivem Framework, als das es entwickelt wurde.

I needed to be way more productive...

David Heinemeier Hansson

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

Nachteile Interaktion mit Legacy-Software ist nicht immer möglich. ActiveRecord reserviert ein paar 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.

Performance Oft wird angeführt, dass Ruby als Skriptsprache und Rails als darauf aufbauendes Framework eine schlechte Performance hat, und dadurch ungeeignet für große Webanwendungen ist.

Anderseits gibt es Anzeichen dafür, dass eine clevere Architektur und Caching für skalierende Anwendung entscheidender ist, als die letztendliche Ausführungszeit.

Das dies möglich ist, zeigen z.B. Groupon, der führende Online-Coupon-Anbieter mit mehr als 50 Mio Abonnenten und Twitter, die jeweils Rails verwenden.

Rails und Tests Rails bietet ausgezeichnete Voraussetzungen zum Softwaretest. Dafür sprechen, dass...

- benötigte Bibliotheken bereits mitgeliefert werden. Dies umfasst einen Test-Runner, vorkonfigurierte Test-Datenbanken (auf Basis von SQLite) und das Testframework Minitest,
- 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 (BDD¹¹-Testframework), Rcov (Testabdeckung), diverse Mockbibliotheken (mocha, FlexMock, RR, RSpec Mocks), Tools zum Generieren und Bereitstellen von Testdaten (Fixtures, Factories, Faker) und Codemetriken (metric-fu)

¹¹Behavior Driven Development

- das Testen einen sehr hohen Stellenwert in der Ruby und Rails-Community hat. Nahezu alle namhaften Ruby-Programmierer schreiben umfassende Tests [DeVries and Naberezny, 2008]. Das Resultat ist, dass auch fast alle OpenSource Bibliotheken (Gems) bei Ruby eine “solid suite of tests,” haben [DeVries and Naberezny, 2008].

Dabei werden mehrere verschiedene Testarten unterstützt und definiert.

Unittests oder Modelltests (model test). Zielstellung: Hier wird die Logik einer Modelklasse untersucht

Testziel: alle (komplexeren) Methoden die das Modell anbietet. Am Beispiel Job kann das die Aussage sein, wann ein Job gültig ist, d.h. welche Bedingungen für die einzelnen Attribute gelten sollen.

Testart: Whitebox

Funktionale Tests Untersuchungsgegenstand sind die Controller, also die Schnittstellen zum Nutzer.

Testziel: Getestet wird meist der Arbeitsablauf innerhalb eines Controllers, also Weiterleitungen, Benachrichtigungen und welches Template gerendert wird.

Testart: Whitebox Es können auch oberflächliche View-Tests unternommen werden, also die Aussage ob ein bestimmtes HTML-Element auf der 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

Testart: Graybox oder Blackbox

Performanz-Tests Eine Testart, die alle Methoden aus Unittests und Funktionalen Tests beinhaltet

Zielstellung: Herausfinden von Performanz-Flaschenhälzen in allen Ebenen.

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

Testart: Graybox

5. Code-Metriken

Eine Codemetrik 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

IEEE [1998]

Dem Verwenden von Code-Metriken liegt der Wunsch zugrunde, komplexe Codeteile auf einfache Zahlen automatisiert beurteilen zu lassen, um potenziell suboptimale Codestellen zu finden, welche Defekte verursachen könnten. Aus Business-Sicht stellen Code-Metriken eine Methode dar, Entwicklungsfortschritt zu messen und qualitativ 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. Diese Größe erhält dann 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. Es werden alle Zeilen der Datei gezählt, die nicht leer und keine Kommentare sind. Kommentare wiederrum können als eigene Metrik verwendet werden, um den Grad der Quelltextdokumentation zu bestimmen.

Zyklomatische Komplexität (CC) ist ein Indikator für die Komplexität auf Basis des Kontrollflussgraphen eines Programms. Gemessen wird die Anzahl der linear unabhängigen Programmfpade. 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 [McCabe, 1976, 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 [McCabe, 1976, S. 318]. Dadurch erreicht man Branch-Coverage (C1) siehe dazu weiter unten.

Anzahl Bad Smells ist eine aggregierte Metrik über alle suboptimale Codestellen. Da es viele verschiedene Bad Smells gibt, können ebensoviele Metriken davon abgeleitet werden. Für eine Übersicht genügt aber auch einfach die Summe. Welche Bad Smells für die Entwicklung entscheidend sind hängt von der gewählten Sprache, dem damit einhergehenden Programmierparadigma und manchmal auch den verwendeten Frameworks. Hier seien einige der häufig gebrauchten Smells für Ruby vorgestellt [Rutherford, 2010]:

Geringe Kohäsion insb. „Feature Envy“ (deutsch: Neid), ist für alle objektorientierten Programme anzuwenden. Eine Klasse weiß zuviel über die internen Strukturen einer anderen Klasse, und implementiert Funktionalität, die eigentlich in jene Klasse gehören würde. Im Beispiel würde diese Berechnung 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.

Diese Smells können mit dem Tool reek¹² festgestellt werden.

Duplikation ist das Auftreten von gleichen oder ähnlichen Codeteilen an mehr als einer Stelle im Programm. Fortgeschritten Analysemethoden betrachten die Baumstruktur und finden ähnliche Teile unabhängig von Bezeichnernamen.

5.2. Code-Metriken für Tests

Der Programmcode wird i.d.R. durch die geschriebenen Tests abgesichert. Die Tests allerdings haben ihrerseits keine Tests. Um also die Nützlichkeit der eigenen Tests zu bestimmen, kann man sich aber zumindest auf Code-Metriken stützen. 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 messen.

5.2.1. Lines of Test / Lines of Code

Neben den Lines of Code kann die gleiche Metrik auf die Test-Suite angewendet werden. Daraus ergibt sich ein Quotient der grob den Grad der Testabdeckung beschreibt. 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 mehr Test Code, als Programmcode besitzen, um so die Zahl der Defekte gegen 0 zu reduzieren [Andrew Hunt, 1999][S. 238].

5.2.2. Testausführungsabdeckung

Die Testabdeckung, im Englischen „Code Coverage“ bezeichnet, misst den Grad inwieweit ein Programm getestet wurde. Die Angabe erfolgt in Prozent, wobei 100% bedeuten, „das Programm wurde durch die Tests komplett ausgeführt“, und 0% „Das Programm wurde die die Tests überhaupt nicht berührt“ Dabei wird die vorhandene Test-Suite ausgeführt und währenddessen der entsprechende Quellcode beobachtet. Es wird festgehalten, welche Anweisungen ausgeführt wurden. Allerdings gibt es 3

¹²<https://github.com/kevinrutherford/reek/wiki/Code-Smells>

Abstufungen, diese Abdeckung zu beobachten (Mit steigender Komplexität des Messverfahrens):

C0 (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 (Branch Coverage) prüft zusätzlich, ob jeder Zweig jeder Zeile ausgeführt wurde. Dies ist wichtig, falls man ternäre Ausdrücke¹³ verwendet

C2 (Path Coverage) prüft, ob jeder mögliche Codepfad durchlaufen wurde. Ein Codepfad sei eine einmalige Abfolge von Zweigen innerhalb einer Funktion von Eintritt bis Rücksprung [Cornett, 1996]. 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 [Powell, 2008], in anderen Fällen aber bei C1 [Cornett, 1996].

Für Ruby 1.8.7 gibt es das Tool rcov¹⁴, für Ruby ab 1.9.1 simple-cov¹⁵, welche beide die C0 Testabdeckung bestimmen können. Zum aktuellen Zeitpunkt sind keine weiteren Tools bekannt, um C1 oder C2 Abdeckungen zu bestimmen.

Wieviel Testabdeckung ist sinnvoll oder notwendig Beim Messen der Abdeckung stellt man sich schnell die Frage, wieviel Testabdeckung notwendig ist. Zuerst sei die Art des Messverfahrens, also C0 bis C2, wichtig. Je komplexer das Messen erfolgte, desto geringer kann also die Testabdeckung am Ende ausfallen [Powell, 2008].

Falls dem TDD-Prozess minutiös gefolgt wurde, so ist die C0 Testabdeckung immer 100% [Beck, 2002]. Für ein Rails Projekt sei es auch relativ leicht, 100% oder nahe 100% zu erreichen [Rappin, 2011]. Die Zahl „100%“ sei für sich genommen nutzlos, aber sie zu erreichen sei für den Prozess der Testgetriebenen Entwicklung nützlich [Rappin, 2011, S. 270]. Vielen Autoren bringen zum Ausdruck, dass es von der Situation abhängt [Elssamadisy, 2007]. Test-Anfänger sollten sich zuerst überhaupt ans Testen gewöhnen, und erfahrene Entwickler sollte wissen, dass es keine einzige einfache Antwort auf diese Frage gebe [Elssamadisy, 2007]. Zudem gebe eine Hohe Abdeckung keinen Aufschluss darüber, dass gut getestet wurde. Aber eine niedrige Zahl zeigt deutlich auf Missstände hin. Einem pragmatischen Ansatz von Savoia folgend, kann

¹³if-then-else in einer Zeile: int a = (1==1) ? 5 : 3

¹⁴<http://relevance.github.com/rcov/>

¹⁵<https://github.com/colszowka/simplecov>

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 [Savoia, 2007].

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. Defect insertion

Eine weiter Methode, um die Qualität von Testcode zu messen, ist die Defect insertion. Hierbei werden (automatisiert oder manuell) nacheinander alle Zeilen des Programm-codes geändert, und geprüft, ob danach ein Test fehlschlägt [Beck, 2002].

Für Ruby gibt es ein Tool, Heckle¹⁶, welches dieses Verfahren implementiert. Im Detail wird aus Bedingungen negiert, konstante Zahlen und Funktionsaufrufe verändert, Zuweisungen verändert usw. [Sadists, 2010]. Dabei wird immer eine Änderung (Mutation) vorgenommen, und dann alle Tests ausgeführt. Sollten dennoch in einer Mutation alle Tests bestehen, so ist die Prämisse der Autoren von Heckle, dass ein Test fehle.

5.3. Notwendigkeit von Code Metriken

Code-Metriken geben dem Programmierer automatisiert und schnell ein Feedback über die Qualität seiner Arbeit. Sie helfen dabei, Probleme frühzeitig zu erkennen und die Wartbarkeit durch gezielte Refaktorisierungen nachhaltig zu verbessern. Auch psychologische Auswirkungen dürfen nicht unterschätzt werden. Alleine der Fakt, dass Codemetriken in einem Unternehmen regelmäßig verwendet werden, motiviert den Programmierer keinen sogenannten „Big Ball of Mud“ zu schreiben. Insbesondere in kleinen Projektteams, die keine dedizierte Qualitätssicherung haben, sind Codemetriken als kostengünstiges Kontrollinstrument unerlässlich.

Für die Testgetriebene Software dient insbesondere die Testabdeckung als Kontrollinstrument, um zu prüfen, ob man sich diszipliniert an den Prozess hält. Für erfahrene Programmierer in TDD mag dies nicht notwendig sein, für alle anderen ist dies anfangs sehr wichtig.

Nach Erfahrungen in der pludoni GmbH sind Code-Metriken ein wichtiges Feedbackinstrument, und damit auch eine unmittelbare Belohnung für das Schreiben sauberen Codes. Wichtig ist, dass die Metriken regelmäßig berechnet werden, entweder

¹⁶<http://ruby.sadi.st/Heckle.html>

als Cronjob oder nach jedem Einchecken in den Hauptentwicklungszweig der Versionsverwaltung.

6. Auswahl der Entwicklungsstrategie und -Werkzeuge

6.1. Herausbildung einer Entwicklungsstrategie für die Bedürfnisse der pludoni GmbH

Viele der gängigen Entwicklungsstrategien, wie V-Modell oder Rational Unified Process, finden ihre Anwendung in großen Projektteams. Für mittelgroße Projektteams gibt es seit ca. 10 Jahren die agilen Prozesse. 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, benötigen eine Schulung für das gesamte Team, die nicht immer finanziert werden kann. Für die Arbeit von sehr kleinen Teams mit weniger als 4 Mitgliedern, wird nun eine Entwicklungsstrategie auf Basis der Testgetriebenen Entwicklung mit Ruby on Rails vorgestellt, die 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 CRUD¹⁷-Methoden müssen nicht getestet werden

¹⁷Create Read Update Delete - Die 4 Standardmethoden, die auf Ressourcen ausgeführt werden können

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 3.3 vorgestellt wurde. Die Akzeptanztests sollen in Cucumber geschrieben werden. Die Websiteinteraktion soll mithilfe von Rack::Test im Falle einer Rails-Anwendung, und mit Capybara andernfalls simuliert werden. 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 möglicherweise nicht-technische Kunde verstehen kann.

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

6.1.2. Weitere Bestandteile der Entwicklungsstrategie

Der oben genannte Teil bezieht sich in erster Linie auf den Kunden. Weitere Praktiken, die für den Programmieralltag wichtig sind, umfassen:

Kontinuierliche Integration Das Vorhandensein einer großen Test-Suite ermöglicht es, diese beim Einchecken in den Hauptzweig komplett auszuführen. Damit lässt sich sicherstellen, dass auf dem Hauptzweig eine immer lauffähige Version vorhanden ist. Die Verwendung einer Versionsverwaltung, z.B. git, ist obligatorisch.

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 [Andrew Hunt, 1999][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 äußerst schlechten Code abgeliefert haben, 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 oben angesprochene Personalsituation der Telearbeit erschwert eine regelmäßige Paarprogrammierung (Pair-Programming [PP]). Nichtsdestotrotz sollten in regelmäßigen Abständen Features zu zweit entwickelt werden. Erfahrungsgemäß führt Pair-Programming zu besser dokumentierten Code, kann die Anzahl der Fehler verringern und zu einer höheren Arbeits-Effektivität führen [Hulkko and Abrahamsson, 2005]. Insbesondere beim Lösen schwieriger Aufgaben und beim Anlernen neuer Teammitglieder ist Pair-Programming eine effektive Methode [Hulkko and Abrahamsson, 2005][S. 9].

6.2. Auswahl der Entwicklungswerkzeuge

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

Werkzeuge für Tests Die formale Beschreibungssprache **Cucumber** dient als Schnittstelle für die vom Programmierer entwickelten Testschritte. Diese könnten in einem von vielen Testframeworks geschrieben werden. Die Entscheidung viel hierbei auf Minitest (Test::Unit in Ruby 1.9), 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 wiederum Mozilla Firefox, Internet Explorer oder Google Chrome fernsteuern kann. Dies ist allerdings sehr langsam, da ein kompletter Browser gestartet und ferngesteuert wird. Daher ist die Nutzung von Selenium nur für das Testen von möglicherweise problematischen Interaktionen und das Testen von Javascript notwendig. Für alle anderen Fälle kann man auf Rack-Test zurückgreifen, welches extrem schnell eine Rack-Anwendung¹⁸ simuliert. 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 RoR-Anwendung starten und im Hintergrund halten, so dass eine erneute Testausführung deutlich schneller von statthen geht, als wenn die komplette Anwendung neu geladen werden müsste.

Werkzeuge für Code-Metriken Für die Generierung von Code-Metriken dient das Ruby-Gem „metric-fu“, welches seinerseits über verschiedene Code-Metriken Zusammenfassungen bildet und diese auch zeitlich darstellen kann. 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.

Texteditoren Innerhalb der Entwickler der pludoni GmbH besteht ein Konsens für die Verwendung von vim, da hier bereits eine große Basis an Plugins gesammelt wurde, die die Entwicklung von Webanwendungen und insbesondere Ruby on Rails erleichtern. Ein weiterer Vorteil ist es, dass vim ohne ein graphisches Interface auskommt, und so direkt von der Shell auf dem Webserver ausgeführt werden kann. Nichtsdestotrotz sei

¹⁸Rack ist ein minimales Interface zwischen Webserver und Webanwendung – Ruby on Rails ist eine solche Rack Anwendung

es zukünftigen Entwicklern freigestellt, eine IDE, wie Eclipse mit dem Plugin RadRails oder Netbeans zu verwenden.

6.3. Diskussion der Maßnahmen

TODO Zusammenfassung evtl ins Fazit oder Auswertung

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. Andernfalls, so die Erfahrung, führt dies zu einer verminderten Ausführungsrate, und ist damit hinderlich für die Entwicklung von sauberen Code.

Durch die gewählte Entwicklungsstrategie kann sogar die Anforderungsanalyse und Abnahme testgetrieben durchgeführt werden.

7. Anwendung der Testgetriebenen Entwicklung

In den nachfolgenden Abschnitten wird exemplarisch an dem Objekt „Job“, also einer Stellenanzeige, die Testgetriebene Entwicklung näher erläutert.

7.1. Implementierung von Unit-Tests (Modelltests)

Ein Modell 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 überprüft Validierungskriterien seines korrespondierenden Modells, d.h. wann eine Instanz dieses Modells gültig ist und damit gespeichert werden darf (man denke z.B. an Pflichtfelder für ein Modell „Nutzer“, oder die Validierung des Formates seiner E-Mail-Adresse). Daneben sollten natürlich alle weiteren, selbstdefinierten, Methoden getestet werden.

1. Der Anfang Während der Analyse wurden die benötigten Attribute bestimmt. In Abbildung 6 seien die Basisattribute der Tabelle dargestellt. 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.

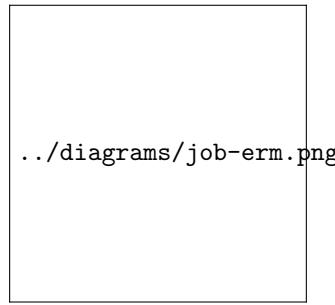


Abbildung 6: Attribute des Modells „Job“

In der Regel erfolgt nun eine Generation der Testklasse und des Testmodells mittels der mitgelieferten Codegeneratoren.

```
~/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. 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!)
- Die Modelklasse (`app/models/job.rb`)
- Die dazugehörige Testklasse (`app/unit/job_test.rb`)
- und Fixtures-Datei (`test/fixtures/jobs.yml`), zur Definition von Testdaten.

Nach einer Erstellung der (SQLite)-Datenbank und Ausführung der Migration kann die Rails-Test-Suite nun auch schon ausgeführt werden:

```
$ rake db:migrate && rake test
```

```

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

(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/unit/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: Listing Test

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 Kriterien entspricht. Viele der Validations sind vergleichbar mit den Datenbank-Constraints einiger Datenbanken. Rails nutzt diese standardmäßig nicht, da es auch andere Persistenzsysteme unterstützt, wie z.B. Key-Value-Store, sogenannte NoSQL Datenbanken. So stellt Rails die Konsistenz und referentielle 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/unit/job_test.rb -----
1 require 'test_helper'
2
3 class JobTest < ActiveSupport::TestCase

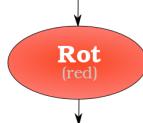
```

```

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 sichern wir zu, dass eine Speicherung nicht erfolgt, also „false“ ist.

Nach Ausführung des Tests, schlägt dieser wie zu erwarten fehl.

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

```

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

```

Listing: Implementierung der Validierung in die Klasse Job



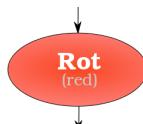
Nach erneuter Ausführung der Testsuite, besteht der Test nun. Jetzt folgt die Refaktorisierungsphase. Der Programmcode lässt sich nicht weiter vereinfachen. Aber der Testcode ist ausdrücklich nicht von Refaktorisierungen befreit, und eine Refaktorisierung wäre z.B.:

```

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

```

Listing: refaktorierter Test



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

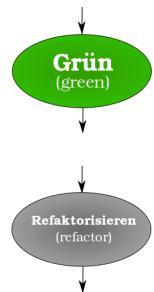
```

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

```

Die Implementierung wäre dann:

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

Listing: Auslagerung des Regulären Ausdrucks in einen Initialisierer

Ein erneutes Ausführen der Tests bestätigt 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 Redundanzen:

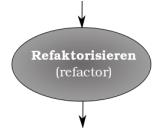
```
test/unit/job_test.rb
1 test "ein Job muss einen Titel haben" do
2   job = Job.new :title => nil
3   assert !job.save
```

```

4 end
5 test "ein Job muss eine gültige E-Mail haben" do
6   job = Job.new :email => "invalid_email"
7   assert !job.save
8 end
9 test "ein vollstaendiger Job muss gueltig seinn" do
10  job = Job.new :title => "Rails Entwickler", :email => "info@stefanwienert.net"
11  assert_valid job
12 end

```

Listing: Alle bisherigen Testmethoden in der Klasse JobTest



In allen drei Methoden wird ein Job instanziert, 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 fehlt¹⁹. Es ist also höchste Zeit, die Tests zu refaktorieren. 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

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

```

¹⁹Im ersten Test ist nicht nur der Titel nicht gesetzt, sondern auch die E-Mail entspricht nicht dem Format

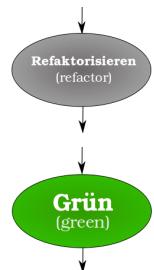
```

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



7.2. Implementierung von Controller-Tests (functional tests)

Neben den Unitests stellt Ruby on Rails eine weitere Testart nativ bereit. Technisch gesehen handelt es sich bei diesen Functional Tests aber auch um Unitests, da deren Testobjekt ein Controller ist. Ein Controller hat bei Ruby on Rails die Aufgabe, Anfragen für bestimmte Routen, also Web-Adressen, anzunehmen, die Arbeit an einer 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
require 'test_helper'

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

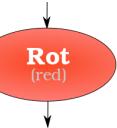
```

```

    get :show, :id => job.id

    assert_response :success
    assert_equal job, assigns(:job)
  end
end

```



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 (success) 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
class JobsController < ApplicationController
  ...
  def show
    @job = Job.first
  end
  ...
end

```



Das Laden des ersten 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 wie möglich einen funktionierenden Test zu erhalten mit dem man arbeiten kann.

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

```

# app/controllers/jobs_controller.rb
def show
  @job = Job.find(params[:id])
end

```

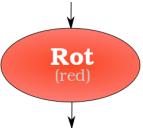


Nun wollen wir testen, ob ein Guest von einer nicht-sichtbaren Stellenanzeige weitergeleitet wird und einen Hinweis erhält.

```
test "Gast Nutzer kann nicht-sichtbare Stellen nicht betrachten" do
  session[:user_id] = nil
  job = jobs(:invisible_job)

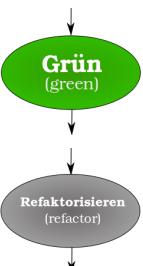
  get :show, :id => job.id

  assert_response :redirect
  assert flash[:notice].present?
end
```



Wir laden unser zweites definiertes Fixture, dass eine unsichtbaren Stellenanzeige. Dieses mal erwarten wir einen HTTP Statuscode 301 (Redirect), und dass unser Controller eine Hinweisnachricht generiert.

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



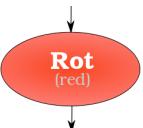
Falls der aktuelle Job nicht sichtbar ist, dann erfolgt eine Weiterleitung auf die Startseite und die Bereitstellung des Hinwestextes. Da auch hier der Quelltext wieder sehr kurz ist, ist ein Refaktorisieren nicht notwendig.

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 "Ein Kunde darf aber seine unsichtbaren Jobs betrachten" do
  job = jobs(:invisible_job)
  session[:user_id] = job.user_id

  get :show, :id => job.id

  assert_response :success
end
```



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

```

def show
  @job = Job.find(params[:id])
  if !@job.visible? and @job.user != User.find(session[:user_id])
    redirect_to root_path, :notice => "Diese Stelle ist zur Zeit nicht
                                         sichtbar"
  end
end

```



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
class Job < ActiveRecord::Base
  ...
  def visible_for_user?(user)
    self.visible and self.user == user
  end
end

# app/controllers/jobs_controller.rb
def show
  @job = Job.find(params[:id])
  unless @job.visible_for_user?(User.find(session[:user_id]))
    redirect_to root_path, :notice => "Diese Stelle ist zur Zeit nicht
                                         sichtbar"
  end
end

```



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
...
def current_user
  User.find(session[:user_id])
end

# app/controllers/jobs_controller.rb
def show

```

```

@job = Job.find(params[:id])
unless @job.visible_for_user? current_user
  redirect_to root_path, :notice => "Diese Stelle ist zur Zeit nicht
  sichtbar"
end
end

```

Funktionale Tests und deren Controllerimplementierungen 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, genannt „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.

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

Jamis Buck,
Programmierer
bei 37signals

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 Format²⁰ bereit, der ein häufig verwendetes Format zum Austausch von Informationen ist, und durch eine Vielzahl von Werkzeugen und Content-Management-Systemen unterstützt wird. Dabei wird der Inhalt des Haupttextfeldes „description“ um weitere Informationen in einem Subdialet angereichert.

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

²⁰Spezifikation des RSS 2.0 Formats: <http://cyber.law.harvard.edu/rss/rss.html>

```

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

```

Listing: Feedimport Beispiel-XML Datei mit einem Job

Der RSS-Feed in dem oben genannten Beispiel enthält eine Stellenanzeige (item). Die description beinhaltet einen HTML-Kommentar, in dem nummer, tags, ort, kontakt und link für die Stellenanzeige definiert werden. Das ganze wurde mit einem Kommentar, und nicht mit einer Erweiterung der Syntax durch eine DDT oder XSD, realisiert, da sich eine Eingliederung der Syntaxelemente mittels DDTs und XSDs bei einigen der CMS der Kunden als problematisch herausgestellt hat.

mittels Mocks und Stubs am Beispiel Feedimport

7.4. System und Akzeptanztests

Cucumber Outside-In Development

7.5. Testen von Javascript Ereignissen

7.5.1. Test im Rahmen von Integrationstest

Systemtests mittels Selenium fuer vereinzeltes JS in Form von

7.5.2. Unitests mit Javascript

Unitests: Jasmine fuer komplexe Objekte und Javascript Anwendungen

8. Auswertung

* Fortschritt in der Entwicklung

* Diskussion der Metriken

8.1. Eigenschaften erfolgreicher Tests

TODO Ueberarbeitung und merging mit MSDNA Tipps und eigene Erfahrungen

Das Vorhandensein von zahlreichen Tests reicht nicht, um das Testen erfolgreich abzuschließen. Zur Beurteilung der Brauchbarkeit einer Testsuite genügen die folgenden Kriterien [Rappin, 2011, S.272-279].

Unabhängigkeit (Independence) 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 [Beck, 2002].

Wiederholbarkeit (Repeatability) Ein Test wird als wiederholbar bezeichnet, wenn er mehrmals hintereinander ausgeführt werden kann, und dabei jedes mal dasselbe Ergebnis liefert. Problematisch sind dabei z.B. Datum und Zeit, sowie Zufallsfunktionen

Klarheit (Clarity) Ein Test ist klar, wenn sein Zweck sofort verständlich wird. Damit wird einerseits die Lesbarkeit gemeint. Andererseits schließt dies auch ein, ob der Test genau eine Eigenschaft testet und nicht redundant zu anderen Tests ist. Dies hat zur Folge, dass die Tests wartbarer werden und als Code Dokumentation 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. 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

8.2. Vorteile von TDD

Die Test-Suite, die durch TDD entsteht, kann als zusätzliche Dokumentation dienen, die nie veraltet, im Gegensatz zu einer geschriebenen Dokumentation [Palermo, 2006].

Die Software-Engineering Literatur ist sich einig, dass ein Bug teurer wird, je später er gefunden wird [Andrew Hunt, 1999][S. 238]. TDD hilft somit, einen Bug so frühzeitig wie möglich zu entdecken, und hat so das Potenzial, Budget einzusparen.

Softwaresysteme, die durch TDD entstehen, tendieren dazu deutlich besser designt, lose gekoppelt und besser wartbar zu sein [Beck, 2002] [Palermo, 2006], da Refaktorierungen mit hoher Zuversicht durchgeführt werden können

Zusammenfassend führt TDD zu einer erhöhten Produktivität, da das Maß an manuellen Tests reduziert wird und Debuggen deutlich weniger wird [Palermo, 2006].

8.3. Nachteile und Grenzen von TDD

Es gibt bestimmte Programmieraufgaben, die nicht allein durch die testgetriebene Entwicklung implementiert werden können. So seinen Nebenläufigkeit oder Software Sicherheit genannt, in denen TDD als Zielgeber nicht ausreiche [Beck, 2002, S. xii].

Zudem stelle die Testgetriebene Entwicklung kein Ersatz für andere Arten von Tests, wie Performanz/Stress und Usability-Test [Beck, 2002, S. 86].

9. Fazit

9.1. Ausblick

Weitere Arbeiten an IT-jobs-und-stellen.de in 2012 geplant.

Rspec als Testframework evaluieren

Virtualisierung der Entwicklungsumgebung um globalen Zustand zu minimieren, a la „On my machine it works“

A. 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 Testausführungs geschwindigkeit, 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 16: 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 17: 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/.+\$}) { |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 18: 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

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.

B. Nutzung von deutschen Schlüsselwörtern für Cucumber Features

In dieser Diplomarbeit wurde die deutsche Modifikation für Cucumber verwendet. Da Cucumber in erster Linie als Kommunikationsmittel zum Kunden dient, ist es hilfreich, die Features in dessen Sprache zu schreiben.

Dazu die Datei `features/support/i18n.yml` anlegen.

Listing 19: `features/support/i18n.yml`

```
# encoding: UTF-8
"de":
  name: German
  native: Deutsch
  feature: Funktionalität|Feature
  background: Grundlage
  scenario: Szenario
  scenario_outline: Szenariogrundriss
  examples: Beispiele
  given: "*|Angenommen|Gegeben sei"
  when: "*|Wenn"
  then: "*|Dann"
  and: "*|Und"
  but: "*|Aber"
```

Dazu einen Loader programmieren, der die Sprachdatei vor der Ausführung von Cucumber lädt.

Listing 20: `features/support/i18n.loader.rb`

```
# Include custom "german" language File for Cucumber
AfterConfiguration do |config|
  languages = YAML.load_file(File.expand_path(File.dirname(__FILE__) + '/i18n.yml'))
  Gherkin::I18n::LANGUAGES["de"] = Gherkin::I18n::LANGUAGES["de"].merge(
    languages["de"])
end
```

Abbildungsverzeichnis

| | | |
|----|----------------------------------------------------------|----|
| 1. | Anwendungsfälle | 4 |
| 2. | Einteilung der Tests | 8 |
| 3. | Red-Green-Refactor: Der TDD Entwicklungszyklus | 12 |
| 4. | Flussdiagramm für TDD | 13 |
| 5. | MVC Modell von Rails | 27 |
| 6. | Attribute des Modells „Job“ | 43 |

Listings

| | | |
|-----|-----------------------------------------------------------------------|----|
| 1. | Ruby Beispiele | 18 |
| 2. | Ruby Beispiel: Lambdas | 19 |
| 3. | Ruby Beispiel offene Klassen | 20 |
| 4. | Testen mit Test::Unit in Ruby | 23 |
| 5. | Cucumber: Additionsfeature in Deutsch | 24 |
| 6. | Cucumber: Implementierung der Additionstestschritte in Ruby | 24 |
| 7. | Listing Test | 44 |
| 8. | Test auf Vorhandensein eines Titels | 45 |
| 9. | Implementierung der Validierung in die Klasse Job | 45 |
| 10. | refaktorierter Test | 45 |
| 11. | Auslagerung des Regulären Ausdrucks in einen Initialisierer | 46 |
| 12. | Alle bisherigen Testmethoden in der Klasse JobTest | 47 |
| 13. | Fixtures Testdaten für zwei Jobs | 47 |
| 14. | Finale Job-Test Klasse nach Refaktorisierung | 48 |
| 15. | Feedimport Beispiel-XML Datei mit einem Job | 53 |
| 16. | RAILS_ROOT/Gemfile | 57 |
| 17. | RAILS_ROOT/Guardfile | 58 |
| 18. | features/support/env.rb | 58 |
| 19. | features/support/i18n.yml | 60 |
| 20. | features/support/i18n_loader.rb | 60 |

Literaturverzeichnis

- [Ambler 2002] AMBLER, Scott: *Introduction to Test Driven Design (TDD)*. Letzter Zugriff: 2011-08-28 08:07:09. 2002. – URL <http://www.agiledata.org/essays/tdd.html>
- [Andrew Hunt 1999] ANDREW HUNT, David T.: *The Pragmatic Programmer: From Journeyman to Master*. 1. Addison-Wesley Professional, Oktober 1999. – ISBN 020161622X
- [Baggen u. a. 2011] BAGGEN, Robert ; CORREIA, José P. ; SCHILL, Katrin ; VISSER, Joost: Standardized code quality benchmarking for improving software maintainability. In: *Software Quality Journal* (2011), Mai. – URL <http://www.springerlink.com/content/vm77272104085276/>. – ISSN 0963-9314
- [Balzert 1997] BALZERT, Helmut: *Lehrbuch der Software-Technik, Bd.2, Software-Management, Software-Qualitätssicherung und Unternehmensmodellierung, m. CD-ROM*. Spektrum-Akademischer Vlg, November 1997. – ISBN 3827400651
- [Beck 2002] BECK, Kent: *Test Driven Development. By Example*. Addison-Wesley Longman, Amsterdam, November 2002. – ISBN 0321146530
- [Brown 2008] BROWN, Roger: *Test Driven Development and Flow*. Web 2011-08-07 18:31:57. April 2008. – URL <http://www.agilecoachjournal.com/post/Test-Driven-Development.aspx>
- [Cornett 1996] CORNETT, Steve: *Code Coverage Analysis*. Letzter Zugriff: 2011-08-07 20:53:34. 1996. – URL <http://www.bullseye.com/coverage.html>
- [Dave Thomas 2009] DAVE THOMAS, David Heinemeier H.: *Agile Web Development with Rails, Third Edition*. Third Edition. Pragmatic Bookshelf, April 2009. – ISBN 1934356166
- [DeVries and Naberezny 2008] DEVRIES, Derek ; NABEREZNY, Mike: *Rails for PHP Developers*. 1. Pragmatic Bookshelf, Februar 2008. – ISBN 1934356042
- [Elssamadisy 2007] ELSSAMADISY, Amr: *InfoQ: 100% Test Coverage?* Letzter Zugriff: 2011-06-19 10:08:32. Mai 2007. – URL http://www.infoq.com/news/2007/05/100_test_coverage
- [Ford 2010] FORD, Neal: *Vortrag: Emergent Design And Evolutionary Architecture*. März 2010. – URL <http://www.thoughtworks.com/emergent-design>

- [Game 2011] GAME, Computer Language B.: *Ruby 1.9 Speed - C/GNU GCC speed.* <http://shootout.alioth.debian.org/u32q/benchmark.php?p=all&lang=yarv&lang2=gcc>. Juni 2011. – URL <http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=yarv&lang2=gcc>
- [Goodliffe 2006] GOODLIFFE, Pete: *Code Craft: The Practice of Writing Excellent Code.* 1st ed. No Starch Press, August 2006. – ISBN 1593271190
- [Grood 2008] GROOD, Derk-Jan de: *TestGoal: Result-Driven Testing.* 1. Springer, Juli 2008. – ISBN 354078828X
- [Hansson 2011] HANSSON, David H.: *RailsConf 2011 Keynote.* Mai 2011. – URL <http://www.rubyinside.com/dhh-keynote-streaming-live-from-railsconf-2011-right-here-right-now-4769.html>
- [Hoffmann 2008] HOFFMANN, Dirk W.: *Software-Qualität.* Springer, März 2008. – ISBN 9783540763222
- [Hulkko and Abrahamsson 2005] 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, S. 495. – URL <http://dl.acm.org/citation.cfm?id=1062545>
- [IEEE 1998] IEEE: IEEE 1061-1998 Standard for a Software Quality Metrics Methodology. In: *IEEE Computer Society* (1998), Nr. 31 Dec. 1998
- [Klukas 2011a] KLUKAS, Jörg: *Referenzen | pludoni GmbH - the community experts.* <http://www.pludoni.de/referenzen>. Juni 2011. – URL <http://www.pludoni.de/referenzen>
- [Klukas 2011b] KLUKAS, Jörg: *Startseite - Aktuelles.* <http://www.itmitte.de/>. Juni 2011. – URL <http://www.itmitte.de/>
- [Madeyski 2009] MADEYSKI, Lech: *Test-Driven Development: An Empirical Evaluation of Agile Practice.* 1st Edition. Springer, Dezember 2009. – ISBN 9783642042874
- [McCabe 1976] McCABE: A Complexity Measure. In: *IEEE Transactions on Software Engineering* (1976), Dezember, S. 308–320. – URL <http://www.literateprogramming.com/mccabe.pdf>

- [Meijer and Drayton 2005] MEIJER, E. ; DRAYTON, P.: Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, 2005
- [Orsini 2007] ORSINI, Rob: *Rails Cookbook (Cookbooks.* 1st Ed. O'Reilly Media, Januar 2007. – ISBN 0596527314
- [Ousterhout 1998] OUSTERHOUT, J. K.: Scripting: Higher level programming for the 21st century. In: *Computer* 31 (1998), Nr. 3, S. 23–30
- [Palermo 2006] PALERMO, Jeffrey: *Guidelines for Test-Driven Development.* Letzter Zugriff: 22.08.2011. Mai 2006. – URL [http://msdn.microsoft.com/en-us/library/aa730844\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx)
- [Powell 2008] POWELL, Catherine: *Abakas: Code Coverage Complexity.* Letzter Zugriff: 2011-08-07 20:59:33. April 2008. – URL <http://blog.abakas.com/2008/04/code-coverage-complexity.html>
- [Rappin 2011] RAPPIN, Noel: *Rails Test Prescriptions.* 1. Pragmatic Bookshelf, März 2011. – ISBN 9781934356647
- [Reeves 1992] REEVES, Jack W.: Three Essays by Jack W. Reeves - I. What Is Software Design? In: *C++ Journal* (1992), Nr. Herbst 1992. – URL http://www.developerdotstar.com/mag/articles/reeves_design.html
- [Rutherford 2010] RUTHERFORD, Kevin: *Code Smells - Reek.* Letzter Zugriff: 09.08.2011. 2010. – URL <https://github.com/kevinrutherford/reek/wiki/Code-Smells>
- [Rätzmann 2004] RÄTZMANN, Manfred: *Software-Testing & Internationalisierung: Rapid Application Testing, Softwaretest, Agiles Qualitätsmanagement.* 2. Galileo Computing, September 2004. – ISBN 3898425398
- [Sadists 2010] SADISTS, Ruby: *Confessions of a Ruby Sadist sudo gem install heckle.* Letzter Zugriff: 2011-08-07 21:15:44. 2010. – URL <http://ruby.sadi.st/Heckle.html>
- [Savoia 2007] SAVOIA, Alberto: *The Code C.R.A.P. Metric Hits the Fan - Introducing the crap4j Plug-in.* Letzter Zugriff: 2011-08-07 21:56:27. Oktober 2007. – URL <http://www.artima.com/weblogs/viewpost.jsp?thread=215899>

[Schwartz u. a. 2009] SCHWARTZ, Randal ; BECK, Kent ; LAPORTE, Leo: *FLOSS Weekly 87: Extreme Programming With Kent Beck.* September 2009. – URL <http://twit.tv/floss87>

[Simon u. a. 2006] SIMON, Frank ; SENG, Olaf ; MOHAUPT, Thomas: *Code Quality Management: Technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht.* 1., Aufl. Dpunkt Verlag, Mai 2006. – ISBN 3898643883

[Spiewak and Harrop 2010] SPIEWAK, Daniel ; HARROP, Jon: *Dynamic type languages versus static type languages - Stack Overflow.* <http://stackoverflow.com/questions/125367/dynamic-type-languages-versus-static-type-languages>. 2010. – URL <http://stackoverflow.com/questions/125367/dynamic-type-languages-versus-static-type-languages>

[Stephens and Rosenberg 2010] STEPHENS, Matt ; ROSENBERG, Doug: *Design Driven Testing: Test Smarter, Not Harder.* Apress, November 2010. – ISBN 9781430229438

[Team 2011] TEAM, Ruby Visual I.: *About Ruby.* <http://www.ruby-lang.org/en/about/>. Juni 2011. – URL <http://www.ruby-lang.org/en/about/>

[Vanderburg 2010] VANDERBURG, Glenn: *Vortrag: Real Software Engineering.* August 2010. – URL <http://confreaks.net/videos/282-lsrc2010-real-software-engineering>