

Behaviour-Driven Development

Verhaltensgetriebene Entwicklung
am Beispiel von Cucumber und RSpec

Semesterarbeit



Fachhochschule
Münster

vorgelegt beim Fachbereich Wirtschaft
im Modul Webengineering 1
des Studiengangs Wirtschaftsinformatik
der Fachhochschule in Münster

vorgelegt von

Frank Hohoff
Matrikelnr.: 661813
Gescherweg 84
48161 Münster

Lars Hoogestraat
Matrikelnr.: 659704
Gescherweg 84
48161 Münster

Abgabedatum: 17.06.2012

Prüfer: Prof. Dr. Ingo Bax

Inhaltsverzeichnis

II	Abkürzungsverzeichnis	iii
III	Abbildungsverzeichnis	iii
1	Einleitung.....	1
1.1	Motivation.....	1
1.2	Gliederung.....	2
1.3	Ziel der Semesterarbeit	2
2	Agile Softwareentwicklung.....	3
2.1	Prinzipien des agilen Manifest.....	3
2.2	Vorgehensmodell: Extreme Programming.....	4
3	TDD – Testgetriebene Entwicklung	5
3.1	Der TDD-Cycle: Red-Green-Refactor.....	5
3.2	Vorteile	6
3.3	Einschränkungen	7
4	Behaviour-Driven Development	7
4.1	Philosophie.....	8
4.2	Vorgehensweise.....	10
4.2.1	Features, User-Stories und Szenarien	10
4.2.2	Der BDD-Delivery-Cycle.....	11
5	Behaviour-Driven Development am Beispiel von Rails	13
5.1	Technische Anforderungen	13
5.2	Ausgangssituation	14
5.3	Cucumber.....	15
5.4	RSpec	17
6	Fazit	20
	Literaturverzeichnis	21
A	Anhang.....	22
A.1	Abstimmungsprobleme in Software-Projekten	22
A.2	BDD-Delivery Cycle mit RSpec und Cucumber.....	23
A.3	Cucumber: Feature-Dateien für das Sozialversicherungsbeispiel.....	24
A.4	Cucumber:Step-Definitions	25
A.4.1	Admin_Steps.rb.....	25
A.4.2	Social_Insurance_Contributions_Steps.rb	26
A.5	RSpec: Specs und Factories mit FactoryGirl.....	27

A.5.1	Factories	27
A.5.2	View-Specs	28
A.5.3	Controller-Specs	30
A.5.4	Model-Spec	32
A.5.5	Routing-Specs.....	33
Erklärung.....		34

II Abkürzungsverzeichnis

ATDP	Acceptance Test-Driven Planning
ATDD	Acceptance Test-Driven Development
BDD	Behaviour-Driven Development
DDD	Domain-Driven Design
DSL	Domain-specific language
MVC	Model-View-Controller
TDD	Test-Driven Design, Test-Driven Development
XP	Extreme Programming

III Abbildungsverzeichnis

1	TDD-Cycle.....	5
2	BDD-Delivery-Cycle.....	12

1 Einleitung

Traditionelle Software-Projekte, welche die Phasen Planung, Analyse, Design und Implementierung sequenziell durchlaufen, sind häufig zum Scheitern verurteilt. Eine Ursache sind die immer komplexeren Rahmenbedingungen eines Software-Projektes, u. a. aufgrund von volatilen Marktbedingungen in der IT. Ein weiteres Problem sind die dynamischen Anforderungen an eine Software, da der Kunde zu Beginn selten eine konkrete Vorstellung über das Endprodukt besitzt. Die Anforderungen können oft erst im Projektverlauf mit ggf. wachsender Erfahrung anhand von Software-Prototypen genauer spezifiziert werden. Weiterhin hat der Kunde, als Stakeholder, eine fachliche Sicht, welche häufig nicht deckungsgleich zu der technischen Sichtweise der Analysten und Entwickler ist. Die daraus resultierenden Abstimmungsprobleme haben zur Folge, dass falsch interpretierte Funktionalitäten implementiert werden (Siehe Anhang A.1). Weiterhin können dadurch Release Termine oder Projektbudgets oftmals nicht eingehalten werden.

1.1 Motivation

Um diesem Trend entgegenzuwirken werden in der Software-Entwicklung oft agile statt traditionelle Vorgehensweisen eingesetzt. Agile Vorgehensweisen sind leichtgewichtiger als traditionelle, da u. a. die Planung nicht allumfassend ist. Dadurch kann auch mit wandelbaren Anforderungen zum Projektbeginn gestartet werden. Das Extreme Programming (XP) ist eines der agilen Vorgehensmodelle, welches unter anderem die Praktiken der test- und verhaltensgetriebenen Entwicklung beinhalten. Diese Methodiken der Software-Entwicklung beschränken sich nicht nur auf die Definition von Tests, sondern sind gesamtheitliche Konzepte für qualifizierten, gut strukturierten Programmcode in einem agilen Software-Projekt

Die verhaltensgetriebene Entwicklung (engl. Behaviour-Driven Development, kurz BDD) ist als Erweiterung der testgetriebenen Entwicklung (Test-Driven Development oder Test-Driven Design, kurz TDD) anzusehen. Die beiden Entwicklungsansätze haben gemein, dass vor der Implementierung der einzelnen Funktionalitäten die dazugehörigen Tests definiert und erstellt werden. Bei TDD liegt der Fokus auf den internen Strukturen einer Anwendung und beschreibt granulare Objekte, nicht deren

Verhalten. Weiterhin bietet TDD keine einheitliche, bereichsübergreifende Sprache. Abstimmungsprobleme zwischen der fachlichen und technischen Sicht sind weiterhin möglich. Infolgedessen wird TDD häufig nur als Test-Suite wahrgenommen und genutzt. Behaviour-Driven Development hingegen beschreibt das Verhalten einer Anwendung (außen) bis zu dem Verhalten der einzelnen Objekte (innen). Zudem wird eine allgemein verständliche Sprache aller Sichtweisen (engl. domain-specific language, kurz DSL) benutzt. BDD geht somit verstärkt auf die Anforderungen agiler Software-Projekte ein.

Dieses Dokument soll, nachdem die grundlegenden Merkmale der testgetriebenen Entwicklung erläutert wurden, auf die verhaltensgetriebene Entwicklung eingehen.

1.2 Gliederung

Nach der Einführung in Kapitel 1, wird zunächst in Kapitel 2 die agile Software-Entwicklung anhand des agilen Manifest und dem Vorgehensmodell Extreme Programming erläutert. Anschließend wird in Kapitel 3 das Verfahren der testgetriebenen Entwicklung beschrieben. Dabei liegt der Fokus auf den Gemeinsamkeiten zum verhaltensgetriebenen Ansatz. Weiterhin werden die Einschränkungen der testgetriebenen Entwicklung aufgezeigt. Im nächsten Kapitel wird das verhaltensgetriebene Entwicklungsmodell erläutert. Dabei wird auf die Philosophie, die Vorgehensweise und die domain-specific language eingegangen.

Anschließend wird im Kapitel 5 die verhaltensgetriebene Entwicklung anhand eines Beispiels in einer Ruby-on-Rails-Webanwendung mit den BDD-Frameworks Cucumber und Rspec vorgestellt. Im letzten Kapitel wird ein Fazit gegeben.

1.3 Ziel der Semesterarbeit

Ziel dieser Semesterarbeit ist es, einen Überblick über agile Vorgehensweisen, TDD und BDD zu verschaffen. Weiterhin soll hierbei BDD von der testgetriebenen Entwicklung abgegrenzt werden. Zusätzlich soll eine Sensibilisierung anhand eines Praxis-Beispiels für das verhaltensgetriebene Entwicklungsmodell geschaffen werden.

2 Agile Softwareentwicklung

Agile Vorgehensmodelle der Softwareentwicklung unterscheiden sich von herkömmlichen Softwareentwicklungsmodellen u. a. durch ihre Flexibilität. Bei diesen Vorgehensmodellen wird die Entwicklung nicht in Phasen, sondern in fortlaufenden Iterationen unterteilt. Diese beinhalten wiederum die üblichen Phasen der Software-Entwicklung. Dadurch kann die Entwicklung jederzeit auf Änderungen eingehen.

Grundlage aller agilen Vorgehensmodelle bildet das agile Manifest, welches 2001 aus dem Wunsch einer gemeinsamen Plattform für agile und iterative Methoden entstand.

2.1 Prinzipien des agilen Manifest

Das agile Manifest beinhaltet Werte und Prinzipien für die agile Softwareentwicklung. Es werden Rahmenbedingungen für die Erstellung von Software formuliert. Folgende Werte werden unterstützt:

- | | | |
|---------------------------------|----------|--|
| • „Individuen und Interaktionen | mehr als | Prozesse und Werkzeuge |
| • Funktionierende Software | mehr als | umfassende Dokumentation |
| • Zusammenarbeit mit dem Kunden | mehr als | Vertragsverhandlung |
| • Reagieren auf Veränderung | mehr als | das Befolgen eines Plans“ ¹ |

Hieraus resultieren 12 agile Prinzipien, siehe Anhang A.2. Unter anderem ergibt sich daraus eine starke Kundenorientierung. Des Weiteren werden, auch spät in der Entwicklung, Anforderungsänderungen befürwortet. Späte Änderungen sind für agile Prozesse keine Herausforderung. Vielmehr sollen dadurch Wettbewerbsvorteile für den Kunden geschaffen werden. Weiterhin ist definiert, dass funktionierende Software oder Prototypen in kurzen Zeitspannen (Iterationen) dem Kunden zur Abstimmung vorgelegt werden, da funktionierende (Teil-)Software ein wichtiger Aspekt für den Projekterfolg ist. In den agilen Vorgehensmodellen wird dieses Prinzip z. B. durch eine kontinuierliche Integration mit Versions-kontrollsystemen und automatisierten Tests und Deployment sichergestellt.

¹ <http://agilemanifesto.org/iso/de/>, abgerufen am: 13.06.2012

2.2 Vorgehensmodell: Extreme Programming

Extreme Programming (XP) bezeichnet ein agiles iteratives Vorgehensmodell. Der Grundgedanke von XP ist, dass die Anforderungen des Kunden zu Projektbeginn weder konkret noch vollständig oder verbindlich vorliegen. Infolgedessen reduziert XP die anfängliche Planungsphase auf ein Minimum. Die Anforderungen werden somit erst im Projektverlauf konkretisiert, was einen hohen Kommunikationsaufwand für den Kunden, Analysten und den Entwicklern bedeutet. Aus den Anforderungen werden mit Hilfe des Kunden Funktionalitäten definiert. Eine gewünschte Funktion wird anschließend in einer oder mehreren User Stories beschrieben. User Stories werden nach folgendem Muster aufgebaut:

„Als [ROLLE] kann ich [FUNKTION] ausführen, um [ZIEL] zu erreichen.“.

User Stories sollten in einer Iteration umgesetzt werden können. Die Festlegung in welcher Iteration eine User Story implementiert wird, geschieht durch Prioritätswerte und Aufwandsabschätzungen. Eine User Story ist erst vollständig integriert, wenn dazu alle erforderlichen Aufgaben abgeschlossen sind. Zu einer Aufgabe gehören, neben der implementierten Funktion, Testdurchläufe und Refactoring zur Sicherstellung qualifizierten Programmcodes. Eine Iteration, welche die traditionellen Phasen einer Softwareentwicklung für die selektierten User Stories durchläuft, ist mit Erstellung eines ausführbaren Prototyps abgeschlossen (kontinuierliche Integration). Dadurch hat das Testen in XP einen hohen Stellenwert. Durch die Praktiken TDD und BDD werden die Anforderungen des ausreichend getesteten und qualifiziert erstellten Teilmoduls sichergestellt.

Die XP-Praktik der testgetriebenen Entwicklung soll durch die Prämisse der Testerstellung vor der Implementierung sicherstellen, dass nur funktionsfähige und fehlerfreie User Stories dem Kunden vorgestellt werden. Das Modell berücksichtigt jedoch nur die internen Programmstrukturen und stellt dabei keine direkte Verbindung zu den User Stories her. Der verhaltensgetriebene Ansatz zur Softwareentwicklung erweitert das testgetriebene Modell um diese Verbindung und stellt einen direkten Bezug zu den User Stories her. Die vollständige Integration einer User Story wird in dem verhaltensgetriebenen Modell durch Akzeptanz- und Integrationstest sichergestellt. Dabei wird das Verhalten der Anwendung, nicht deren Zustand getestet und mit einer DSL dokumentiert.

3 TDD – Testgetriebene Entwicklung

Wie bereits erwähnt werden Tests bei TDD nicht, wie in traditionellen Vorgehensmodellen parallel und unabhängig, sondern zum Anfang erstellt. Durch strikte Einhaltung dieser Praktik sollen alle Funktionen einer Anwendung durch Tests abgedeckt werden. TDD ist auf Unit-Tests (granulare Tests an Objekten) ausgelegt. Die Tests sind möglichst vollständig automatisiert und werden in der Gesamtheit ausgeführt. TDD definiert dafür den sogenannten TDD-Cycle, der für jede zu implementierende Funktion durchlaufen werden soll. Der TDD-Cycle beinhaltet die drei Phasen Red, Green und Refactor.

3.1 Der TDD-Cycle: Red-Green-Refactor

Red: In dieser initialen Phase wird zunächst für jede zu implementierende Funktion (engl. Feature). ein Test definiert. Der Test wird auf Grundlage des „Codes you wish you had“² erstellt. Demnach werden in den Tests die noch nicht verfügbaren Objekte und deren Methoden über die Methodensignaturen angesprochen. Die Tests schlagen zunächst fehl. Anschließend wird mit der Implementierung begonnen.

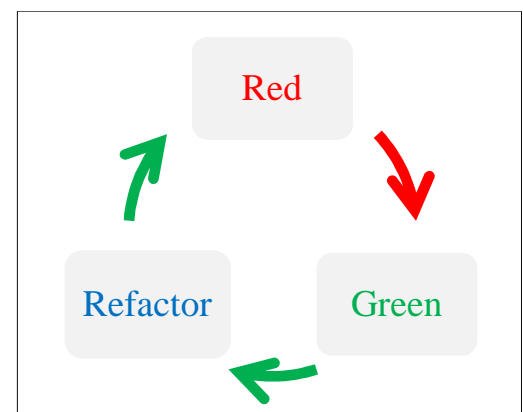


Abbildung 1: TDD-Cycle

Green: Nach der Implementierung wird der Testlauf wiederholt. Falls dieser erneut fehlschlägt, muss der geschriebene Programmcode korrigiert oder die Methodensignatur im Test angepasst werden. Sobald die Tests erfolgreich durchlaufen, werden keine neuen Funktionalitäten mehr programmiert (Green), da jede Funktion ihren eigenen TDD-Cycle durchläuft.

Anschließend wird zur Sicherstellung eines qualifizierten Programmcodes Refactoring betrieben.

Refactor: Beim Refactoring wird der vorhandene Programmcode auf sogenannte Code-Smells untersucht. Bei einem Code-Smell handelt es sich um einen schlecht

² David Chelimsky, The RSpec Book (2010), Seite 40

strukturierten und schwer verständlichen Programmcode. Ein Code-Smell ist zunächst kein Logik-Fehler, kann aber einen solchen offenlegen.

Beispielhafte Code-Smells sind:³

- **Duplizierter Code:** (engl. Don't repeat yourself, DRY) Doppelt vorkommender Programmcode soll nach Möglichkeit in einer Methode ausgelagert werden.
- **Lange Methoden:** Diese Methoden sind zu lang und besitzen gegebenenfalls einen zu großen Zuständigkeitsbereich. In der objektorientierten Programmierung sollte eine Methode nur eine Aufgabe besitzen (SRP, Single Responsibility Principle).⁴
- **Lange Klassen:** Dieser Code-Smell besitzt wie „lange Methoden“ einen zu großen Zuständigkeitsbereich (SRP), zu viele Instanzvariablen und eventuell duplizierten Code.
- **Lange Parameterliste:** Die Methodensignaturen beinhalten Parameter, die nicht benötigt werden.
- **Faule Klasse:** Eine faule Klasse hat einen zu kleinen Aufgabenbereich. Die entsprechenden Funktionen sind unter Umständen in andere Klassen einzugliedern. Die faule Klasse sollte anschließend aufgelöst werden.

Nach dem Refactoring müssen die Tests erneut ausgeführt werden um sicherzustellen, dass die Funktionalität nicht beeinträchtigt wurde. Eventuell sind dabei die Methodenaufrufe in den Tests anzupassen.

3.2 Vorteile

Durch die permanenten automatisierten Testdurchläufe ist ein schnelles Feedback gegeben. Die Fehlerquellen werden dadurch besser lokalisierbar. Weiterhin wird ein qualifiziert strukturierter Programmcode durch das kontinuierliche Refactoring produziert.

³ vgl. Martin Fowler, Refactoring (2011), Seite 63 - 71

⁴ vgl. David Chelimsky, The RSpec Book (2010), Seite 80

3.3 Einschränkungen

Die Tests dokumentieren zwar den technischen Aspekt der Anwendung, bieten jedoch keinen Bezug zu den User Stories. Die durch Tests dokumentierten Funktionen sind zudem nicht allgemein verständlich. Weiterhin werden nur Objektzustände und nicht deren Verhalten getestet, Integrations- und Systemtests werden durch TDD nicht unmittelbar berücksichtigt.

Da die Tests eine nur technische Dokumentation der implementierten Funktionen bieten und diese nicht allgemeinverständlich sind, wird TDD oft nur als eine Test-Suite, zur Verbesserung der Programmcodequalität und Steigerung der Testabdeckung missverstanden. Das volle Potential von TDD als ein Design-Prozess wird dadurch selten ausgeschöpft.⁵

4 Behaviour-Driven Development

Dan North stellte im Jahre 2003 erstmalig die verhaltensgetriebene Entwicklung (BDD) vor.⁶ Das Scheitern traditioneller Projekte war ursächlich die Motivation für Dan North ein neues Modell zu entwickeln. Durch BDD sollen folgende, im Kapitel 1 erwähnte, Auswirkungen vermieden werden:⁷

- Projekt-Releases und / oder Projekt-Budgets werden nicht eingehalten
- Anforderungen werden nicht fachgemäß und / oder fehlerlos umgesetzt
- Das Produkt ist instabil und / oder lässt sich durch eine schlechte Strukturierung schwer warten

Dafür erweitert Dan North, wie schon erläutert, die testgetriebene Entwicklung u. a. um eine natürliche, allgemein verständliche Sprache (DSL). BDD Entwickler nutzen ihre eigene native Sprache in Verbindung mit einer ubiquitären Sprache, der DDD (Domain-Driven Design). Des Weiteren liegt der Fokus nicht primär auf den internen Strukturen der Anwendung (Unit-Tests) sondern zunächst auf den User-Stories (Akzeptanz-Tests). Die Akzeptanz-Tests stellen das erwartete Verhalten der Anwendung dar. Durch die DSL sind die Akzeptanz-Tests von allen Stakeholdern

⁵ ASTELS, DAVE: A new look at tdd, 2006, http://blog.daveastels.com/files/BDD_Intro.pdf

⁶ vgl. <http://dannorth.net/introducing-bdd/>

⁷ vgl. David Chelimsky, The Rspec Book (2010), Seite 109ff

eines Projekts zu verstehen und können somit zwischen Kunden und Analysten gemeinsam erarbeitet werden. Dabei werden die gewünschten Funktionen / Features einer Anwendung in Form von User-Stories mit dem Kunden definiert. Anschließend werden auf Grundlage der User-Stories die Akzeptanz-Kriterien und die dazugehörigen Tests erarbeitet.

Nachfolgend wird der Red-Green-Refactor-Cycle für die Akzeptanztests (äußerer Cycle) durchlaufen. Nach der Phase Red wird die Funktion zunächst nicht implementiert, sondern durchläuft zuerst auf Objektebene einen eigenen Red-Green-Refactor-Cycle (innerer Cycle). Hierbei werden Unit-Tests für den entsprechenden Akzeptanz-Test geschrieben. Dazu wird wie bei den Akzeptanz-Tests das Objektverhalten und nicht deren Zustand getestet. Das Vorgehen wird im Abschnitt „4.2.2 Der BDD-Delivery-Cycle“ detaillierter erläutert.

Die Definition der Akzeptanz- und Unit-Tests dokumentieren die zu entwickelnde Anwendung nicht nur auf der technisch-funktionalen Ebene. Stattdessen wird zusätzlich eine Beziehung von den User-Stories zu den einzelnen Funktionen hergestellt.

4.1 Philosophie

Dan North beschrieb BDD auf der „Agile specifications, BDD and Testing eXchange“ am 21. November 2009 in London wie folgt:

„BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology.“⁸

Diese Definition basiert auf folgende Eigenschaften und Prinzipien von BDD:⁹

- **Second-generation:** BDD definiert eine nächste Generation, da es sich aus verschiedenen, bereits bewährten Praktiken ableitet. Dazu gehören neben dem Domain-Driven Design aus XP die testgetriebene Entwicklung und die kontinuierliche Integration. Weiterhin verwendet BDD das Modell des „Acceptance Test-Driven Planning und Development“ (ATDP, ATDD) bei der eine akzeptanztestgetriebene Planung und Entwicklung durchgeführt wird.

⁸ <http://skillsmatter.com/podcast/agile-testing/how-to-sell-bdd-to-the-business>

⁹ vgl. David Chelimsky, The Rspec Book (2010), Seite 5-7, 123-124

- **Outside-in:** Mit „outside-in“ bezeichnet bei BDD die bereits sehr frühe Integration des Design-Modells. Im Gegensatz zu der testgetriebenen Entwicklung ist BDD nicht auf die Objektzustände beschränkt. Viel mehr findet eine Unterstützung schon auf einer abstrakteren Ebene statt, da Features, User-Stories sowie Akzeptanzkriterien explizit in BDD definiert werden.
- **Pull-based:** Der Grundgedanke ist, dass alles einen abnehmenden Ertrag im Verhältnis zum Aufwand besitzt. BDD definiert hieraus das erste Prinzip:
 - 1) **Enough is enough:** Die Planung, die Analyse und das Design sollen sich nur auf das Nötigste beschränken. Des Weiteren soll nicht mehr Funktionalität als nötig integriert werden (Lean Software Development)
- **Multiple-stakeholder:** BDD ist nicht auf eine spezifische Zielgruppe beschränkt, sondern involviert alle Projektbeteiligten (Stakeholder). Dan North unterscheidet zwischen „Core-Stakeholdern“ und „Incidental-Stakeholdern“.¹⁰ Die Core-Stakeholder sind die Personen mit der Vision und dem größten Interesse an dem Ergebnis. Die Incidental-Stakeholder beschreiben die Personen, welche zur Ergebniserreichung beitragen. Durch die multiple-stakeholder methodology definiert BDD das zweite Prinzip:
 - 2) **Deliver stakeholder value:** Es sollen nur diejenigen Arbeiten ausgeübt werden, die auch einen Nutzen für den Core-Stakeholder darstellen. Hierunter ist eine Orientierung an der Umsetzung der zuvor festgelegten Features und User-Stories zu verstehen.
- **Multiple-scale:** Die verhaltensgetriebene Entwicklung ist nicht nur auf einer Ebene beschränkt. Es wird das Anwendungsverhalten auf der oberen Ebene sowie das Objektverhalten auf der unteren Ebene beschrieben. Hieraus leitet sich das dritte Prinzip ab:
 - 3) **It's all behaviour:** Der zentrale Aspekt von BDD. Auf allen Ebenen kann das Verhalten der Anwendung getestet werden. Dies umfasst Akzeptanz- und Integrationstests, sowie Unit-Tests.
- **High automation:** Durch automatisierte Tests und kontinuierlicher Integration sind Fehler schnell lokalisierbar (rapid Feedback). Außerdem ist die Anwendung immer Release-fähig.
- **Agil:** BDD unterstützt alle Prinzipien des agilen Manifests, sowie einige in Extreme Programming definierte Werte und Richtlinien.

¹⁰ David Chelimsky, The RSpec Book (2010), Seite 124

4.2 Vorgehensweise

BDD beschreibt einen Kreislauf mehrerer Interaktionen mit ordentlich definierten und strukturierten Ergebnissen, welche zu einer stabilen und getesteten Software-Auslieferung führt. Dieser Kreislauf wird als BDD-Delivery-Cycle bezeichnet. Bevor dieser genauer erläutert wird, werden zunächst die BDD-spezifischen Begriffe Feature, User-Story, Akzeptanz-Kriterium und Szenario beschrieben.

4.2.1 Features, User-Stories und Szenarien

Ein Feature definiert eine umzusetzende Funktionalität. Diese werden vom Kundenwunsch geprägt. Ein Feature enthält einen oder mehrere User-Stories. Jede User-Story wiederum besitzt mehrere Akzeptanz-Kriterien (Szenarien).

Überwiegend enthält ein Feature allerdings genau eine User-Story mit den dazugehörigen Szenarien. Eine User-Story muss jedoch mit wenig Aufwand und in einer Iteration umgesetzt werden können. Falls ein Feature für eine Iteration zu zeitintensiv ist, wird diese in mehreren User-Stories unterteilt.¹¹

Eine User Story wird in BDD mit einer ubiquitären Sprache und folgendem Muster ausgedrückt:

As a [Stakeholder] I want [feature] So that [benefit]

Oder synonym dazu:

in order to [benefit] a [stakeholder] wants to [feature]

Die Akzeptanzkriterien (Szenarien) werden ebenfalls mit einer ubiquitären Sprache und folgendem Muster definiert:

Given some context
When some event occurs
Then I expect some outcome

¹¹ David Chelimsky, The Rspec Book (2010), Seite 129

4.2.2 Der BDD-Delivery-Cycle

In einem ersten Meeting legt der Kunde (Core-Stakeholder) zunächst grundlegende Anforderungen, Use-Cases und Ziele einer Anwendung fest. Der Business-Analyst erarbeitet dann mit Hilfe des Stakeholders aus den Anforderungen die Features, sowie die User-Stories. Diese werden priorisiert und für Iterationen und Releases selektiert.

Anschließend werden die Entwickler in dem Projekt einbezogen. Der Stakeholder, Business-Analyst und der Entwickler entwerfen gemeinsam den Geltungsbereich einer User-Story (Story-Scope). Darunter wird die Frage verstanden: „Was muss die Anwendung zur Erfüllung einer User Story leisten?“ Hier wird speziell das Prinzip: „Enough is enough“ beachtet. Der Fokus soll ausschließlich auf die Erfüllung der User-Story liegen. Dan North erläuterte dazu: „Any more detail is waste, any less is risk!“¹² Anschließend erfolgt eine spezifische Festlegung der Akzeptanzkriterien (Szenarien). Hierzu überlegt sich der Entwickler Randbedingungen einer User-Story, die sogenannten Edge-Cases. Diese werden in Szenarien festgehalten, welche später als Akzeptanztests dienen. Daraufhin priorisiert der Kunde die Szenarien und weist ihnen gegebenenfalls Tags zu. Die Akzeptanztests werden so in Gruppen unterteilt und können gruppenweise ausgeführt werden.

Zur Integration einer User-Story durchlaufen alle dazugehörigen Szenarien ihren eigenen BDD-Cycle. Der BDD-Cycle, welcher nachfolgend erklärt und anschließend skizziert wird, besteht aus einem äußeren und einem inneren TDD-Cycle (Red-Green-Refactor).

1. Mit der initialen Erstellung eines Szenarios und Ausführung des Tests beginnt der äußere TDD-Cycle mit der Phase Red. Akzeptanztests testen das Anwendungsverhalten für eine konkrete User-Story.
2. Anschließend wird zunächst eine Menge an Unit-Tests erstellt, welche das spezifische Objektverhalten testet (innerer TDD-Cycle). Hierbei sollten für alle Objekte, die zum erfolgreichen Durchlauf des übergeordneten Akzeptanztests nötig sind, Unit-Tests erstellt werden. Jeder Unit-Test durchläuft dabei einen eigenen Cycle.

¹² http://www.jfok.us/se/jfok.us10/preso/jf-10_Behaviour-DrivenDevelopment.pdf

3. Wie in Kapitel 3.1 beschrieben, werden nun die Schritte des TDD-Cycles durchlaufen (Red, Green, Refactor).
4. Sobald alle Unit-Tests erfolgreich durchlaufen, sowie Refactoring betrieben wurde, wird der Akzeptanztest erneut ausgeführt. Dieser sollte mittlerweile erfolgreich durchlaufen. (Phase Green des äußeren Cycle). Falls dieser weiterhin fehlschlägt, sind nicht alle Objekte mit Unit-Tests abgedeckt. Diese müssen in einem weiteren inneren Cycle erstellt werden.
5. Anschließend wird auf abstraktere Ebene Refactoring betrieben und mit einem erneuten Durchlauf des Akzeptanztests überprüft.

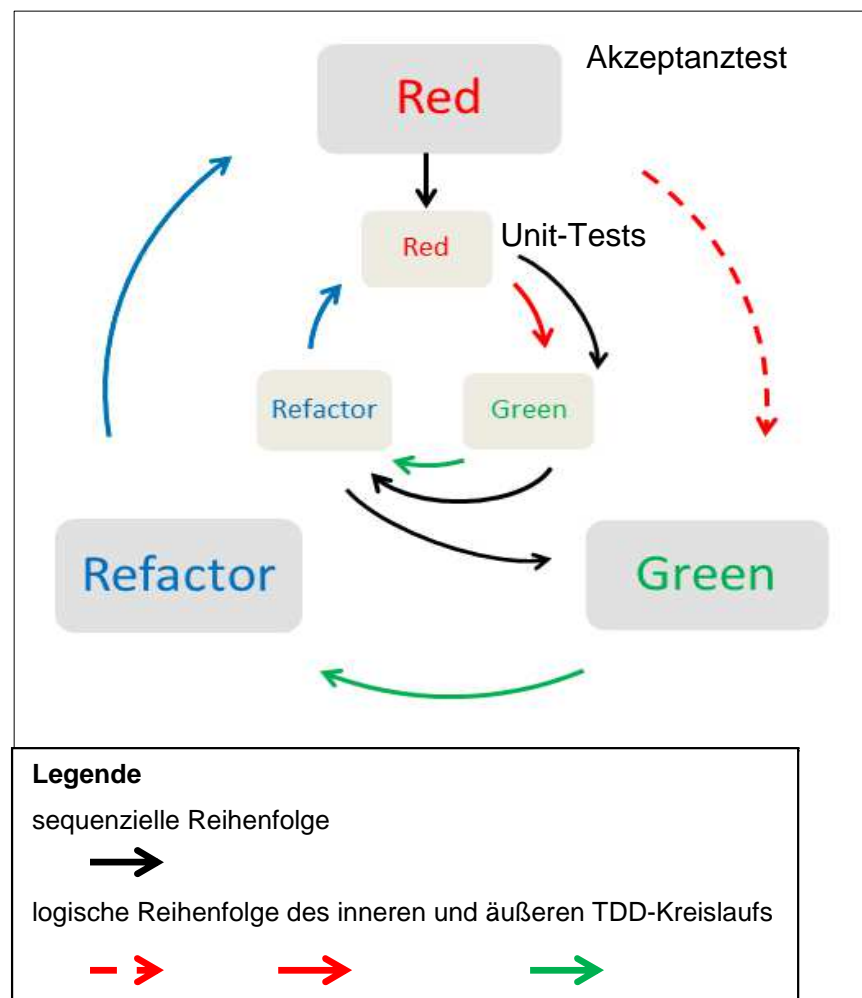


Abbildung 2: BDD-Cycle

5 Behaviour-Driven Development am Beispiel von Rails

Zur Verdeutlichung soll eine Ruby-on-Rails Webanwendung mit BDD entwickelt werden. Dazu wird eine kleine Webanwendung erstellt, welche Sozialversicherungsbeiträge kalkuliert. Anhand dieses Beispiels wird gezeigt wie eine erste Iteration einer Webanwendung mit einem ausführbaren Prototypen verhaltensgetrieben entwickelt werden kann. Das „bdd_demonstration“-Beispiel kann auf Github.com eingesehen und mit „bundle install“ und „rake db:setup“ lauffähig deployed werden.¹³

Im Beispiel werden die BDD-Frameworks Cucumber und RSpec genutzt. Mit Cucumber werden Features, User-Stories sowie automatisierte Akzeptanztests definiert. Hiermit kann der äußere Cycle toolgestützt realisiert werden. Mit Hilfe von Rspec können Unit-Tests auf der gesamten MVC-Ebene (Model-View-Controller) automatisiert angefertigt werden. Im Anhang A.3 befindet sich der BDD-Delivery-Cycle im Zusammenspiel mit Cucumber und RSpec.

5.1 Technische Anforderungen

Ausgangssituation für dieses Beispiel ist ein bereits installiertes Ruby-on-Rails. Das Beispiel wurde mit Rails in der Version 3.2.6 und Ruby in der Version 1.9.4 erstellt. Zunächst muss ein neues Rails-Projekt mit

„rails new bdd_demonstration“

angelegt. Anschließend muss das Gemfile um die Gems für folgende BDD-Tools erweitert werden. Ein Gem ist ein Programmpaket der Programmiersprache Ruby.

```
[...]
group :development, :test do
  gem 'rspec-rails'
  gem 'factory_girl_rails'
  gem 'capybara'
  gem 'cucumber-rails'
  gem 'selenium-webdriver'
  gem 'database_cleaner'
end
[...]
```

¹³ https://github.com/fhohoff/bdd_demonstration.git

- **rspec-rails:** Das Paket beinhaltet das Rspec-Framework. Es wird, wie bereits erwähnt, für die Unit-Tests benötigt
- **factory_girl_rails:** Factory Girl stellt eine Alternative der Fixtures (Testdatensätze) dar. Bei Factory Girl sind die Datensätze allerdings nicht global verfügbar. Damit sollen Kollisionen verhindert werden.
- **capbara:** Mit Capybara können Benutzereingaben in Cucumber simuliert werden.
- **cucumber-rails:** Cucumber-rails stellt ein Rails-spezifisches Testframework für die Akzeptanztests zur Verfügung.
- **selenium-webdriver:** Selenium-Webdriver ist ein Browser-Testwerkzeug für Ruby, auf das capybara zurückgreift
- **database_cleaner:** Database_cleaner sorgt für eine gesäuberte Testdatenbank vor jedem Durchlauf anhand auswählbarer Cleaning-Strategien.

Die Gems werden anschließend mit „*bundle install*“ installiert.

5.2 Ausgangssituation

In einem ersten Meeting werden zunächst alle Anforderungen für einen Sozialversicherungsrechner aufgenommen. Unter anderem sollen folgende Funktionen umgesetzt werden:

- 1) Kalkulieren der Sozialversicherungsbeiträge für Arbeitnehmer unter Angabe des Brutto-Gehalts
- 2) Berücksichtigung der Zusatzbeiträge für Kranken- und Pflegeversicherung unter Angabe des Alters und der Kinderanzahl
- 3) Für Beitragssatzänderungen müssen die Prozentsätze der Versicherungen änderbar sein
- 4) Berücksichtigung der länderspezifischen Versicherungssätze
- 5) Berücksichtigung der Beitragsbemessungssätze
- 6) Sozialversicherungsbeiträge für Arbeitgeber

Anschließend werden die Anforderungen priorisiert und in Iterationen eingeteilt. Für die erste Iteration sollen die ersten drei Anforderungen umgesetzt werden.

Nachfolgend wird der BDD-Delivery-Cycle für die erste Iteration durchlaufen. Dazu werden zunächst aus den Anforderungen die Features und User-Stories definiert. Anschließend werden die Szenarien erstellt, welche das Verhalten der Anwendung testen soll.

5.3 Cucumber

Zur Umsetzung mit Cucumber wurden Feature-Dateien, welche eine User-Story sowie die Akzeptanz-Tests enthalten, erstellt. Die Feature-Dateien befinden sich im Anhang A.3 und haben folgenden Aufbau:

- **Titel:** Beinhaltet aussagekräftigen Titel des Features.
- **Dokumentation:** In ubiquitärer Sprache wird die User-Story definiert, welche nicht weiter von Cucumber beachtet wird. Sie dient nur zur Dokumentation.
- **Background:** Beschreibt die allgemeine Situation in dem sich jedes Szenario befindet. Definierte Steps unter Background werden für jedes Szenario wiederholt ausgeführt.
- **Scenario:** Beschreibt das erwartete Verhalten der Anwendung.
 - Given: Definiert eine eindeutige Ausgangslage bei einem Test
 - When: Aktion, die ausgeführt wird
 - Then: erwartetes Verhalten nach Ausführung der Aktion. Schlägt fehl, falls die Anwendung sich anders verhält.
- **Scenario Outline:** Um gleiche Szenarien mit unterschiedlichen Werten zu erstellen, können Szenario Outlines benutzt werden. Dadurch müssen die Szenarien nicht wiederholt werden. Stattdessen gibt es eine sogenannte Outline für alle Szenarien. Darin sind die Steps mit Variablen definiert. Ein spezielles Szenario enthält dann nur noch die Werte der Variablen (Siehe Anhang A.3 unten).

Anschließend wird das Feature erstmalig mit dem Konsolenauf Ruf

```
„cucumber ./feature/admin.feature“
```

ausgeführt.

Folgende Terminal-Ausgabe wird dabei generiert (gekürzt):

```
Using the default profile...
Feature: Administrate the social insurance percentages

  As an Administrator
  [...]

  Background:
    Given I am on the admin page
      Undefined step: "I am on the admin page" (Cucumber::Undefined)
      ./features/admin.feature:8:in `Given I am on the admin page'

  Scenario: Successfull update social insurance percentage
    [...]
  Scenario: Failed to update social insurance percentage
    [...]

2 scenarios (2 undefined)
8 steps (8 undefined)
0m0.537s

You can implement step definitions for undefined steps with these
snippets:

Given /^I am on the admin page$/ do
  pending # express the regexp above with the code you wish you had
end
```

Für jedes Feature wird nun eine Datei in „./features/step_definitions/“ angelegt. Die „Snippets“ müssen nun in der neuen Datei

(z.B. „./features/step_definitions/admin_steps.rb“)

kopiert werden. Anschließend wird „pending“ durch das gewünschte Verhalten ersetzt. Dabei sollte bei der Definition der Ausgangslage (Given) direkt auf das Model zugegriffen werden (Direct Model Access). Die Aktionen und Ereignisse (When, Then) sollten stattdessen mit einem simulierten Browser (Webrat) ausgeführt bzw. abgefragt werden. Die beispielhafte Umsetzung der Step-Definitionen für den Sozialversicherungsrechner befindet sich im Anhang A.4.

Nachdem die Step-Definitionen auf Grundlage des „Code you wish you had“ geschrieben wurden, wird der Test mit

```
„cucumber ./features/admin.feature“
```

erneut ausgeführt.

Die Akzeptanztests / Szenarien schlagen erwartungsgemäß fehl (Phase Red im äußeren Cycle):

```
Using the default profile...
Feature: Administrate the social insurance percentages

  As an Administrator
  I want to edit the social insurance percentages between 0 and 30 %
  So that I can keep the calculator-page up-to-date

  Background:
    Given I am on the admin page
    No route matches [GET]
    "/admin/social_insurances" (ActionController::RoutingError)

  Scenario: Successfull update social insurance percentage
    Given I want to update "Krankenversicherung"
    When I update the percentage to "5.0"
    Then I should see on the admin page "5.0" for "Krankenversicherung"

  Scenario: Failed to update social insurance percentage
    Given I want to update "Krankenversicherung"
    When I update the percentage to "55"
    Then I should receive an error

Failing Scenarios:
cucumber ./features/admin.feature:11

2 scenarios (1 failed, 1 skipped)
8 steps (1 failed, 7 skipped)
0m0.586s
```

Anschließend werden mit RSpec alle benötigten Objekte getestet.

5.4 RSpec

Mit RSpec können Models, Controllers, Views sowie Routen getestet werden. Ein einzelner Test wird auch Spec genannt. Der grundlegende Aufbau eines Specs ist wie folgt:

```
describe KLASSE do
  describe METHODE do
    it „should do something“ do
      [do something]
      [check if it worked]
    end

    it „should do something else“ do
      [do something else]
      [check if it worked]
    end
  end
end
```

- **describe(„Klassen- / Methodenname“, &block):** Ein Spec lässt sich mit describe gliedern und beliebig tief verschachteln. Ein Block kann somit weitere describe-Methoden enthalten.
- **it („Testbeschreibung“, &block):** Ein it – Block umfasst einen einzelnen Testfall. Aus der Testbeschreibung kann später eine Dokumentation generiert werden
- **[do something]:** Hier werden die Aktionen auf Grundlage des „Code you wish you had“ ausgeführt. Zum Beispiel können gewünschte Funktionen, Actions, HTTP-Requests, Model-Interaktionen oder Routen aufgerufen werden, welche zum Zeitpunkt der Testerstellung noch nicht vorhanden sind.

Falls Models oder Controller und deren Actions (Klassen und Methoden) ausgeführt werden sollen, die nicht in der aktuellen Iteration umgesetzt werden, können sogenannte Mock-Objekte und Stub-Methoden eingesetzt werden. Hierbei handelt es sich um Dummy-Objekte / -Methoden, welche einen festgelegten Return-Wert zur Laufzeit zurückliefern.

```
social_insurance = mock(SocialInsurance)
social_insurance.stub!(:name).and_return(„Rentenversicherung“)
```

Da in der ersten Iteration jedoch auch das Model erstellt werden soll, ist dieses Verfahren nicht notwendig.

- **[check if it worked]:** Rspec erweitert alle Klassen um die Methode „should“ und „should_not“, um eine Testbedingung definieren zu können.
 - **.should(matcher, message)** Die Methode gibt einen boolean-Wert zurück, je nachdem ob die Testbedingung, definiert durch (matcher, message), erfolgreich durchläuft
 - **matcher:**
 - eql(): untersucht Objekte auf ihre inhaltliche Gleichartigkeit
 - be_methodname: dynamischer boolean-Matcher, z.B. wird aus „Me.male?“ die Testbedingung „Me.should be_male“
 - include(): überprüft, ob ein Hash, Array oder String ein Element oder Teilstring enthält
 - have(int).items überprüft Elemente auf ihre
have_at_least().items Länge
have_at_most().items

- `match(/RegEx/)` untersucht einen String nach einem regulären Ausdruck
- `raise_error` überprüft eine Methode auf einem geworfenen Fehler. Zur besseren Syntax wird statt `should` die Methode `to` verwendet.

Die Specs für den Sozialversicherungsrechner befinden sich im Anhang A.5. Nachdem die Testfälle auf Unit-Ebene erstellt worden sind, werden diese ebenfalls mit „`rspec ./spec`“ ausgeführt und schlagen erwartungsgemäß fehl:

```
FFFFFFFFFFFFFFFFFFFFF
Failures:

  1) SocialInsurancesController routing routes to #index
     Failure/Error:
       get("admin/social_insurances").should
         route_to("social_insurances#index")
       No route matches "/admin/social_insurances"

[...]

Finished in 0.2572 seconds
19 examples, 19 failures
```

Anschließend wird die Programmlogik implementiert und Refactoring betrieben. Das DRY-Prinzip kann in Ruby-on-Rails-Anwendungen u.a. mit Methoden-Auslagerungen in einer „`before(:each) {&block}`“-Methode umgesetzt werden. Der Programmcode für die Programmlogik befindet sich zum Nachschlagen auf Github.

Als Nächstes werden die Unit-Tests mit „`rspec ./spec`“ erneut ausgeführt:

```
.....
Finished in 0.57238 seconds
19 examples, 0 failures
```

Sobald alle Specs für einen Akzeptanztest erfolgreich durchlaufen, wird der Akzeptanztest in cucumber erneut ausgeführt.

```
„cucumber ./features/admin.feature“
```

```
Using the default profile...
Feature: Administrate the social insurance percentages

  As an Administrator
  I want to edit the social insurance percentages between 0 and 30 %
  So that I can keep the calculator-page up-to-date

  Background:
    Given I am on the admin page

  Scenario: Successfull update social insurance percentage
    Given I want to update "Krankenversicherung"
    When I update the percentage to "5.0"
    Then I should see on the admin page "5.0" for "Krankenversicherung"

  Scenario: Failed to update social insurance percentage
    Given I want to update "Krankenversicherung"
    When I update the percentage to "55"
    Then I should receive an error

2 scenarios (2 passed)
8 steps (8 passed)
0m1.143s
```

6 Fazit

Die verhaltensgetriebene Entwicklung hat, durch die Erweiterung der testgetriebenen Entwicklung, einen bereits bewährten Entwicklungsprozess um eine abstrakte, kundennahe Sicht ergänzt. Durch BDD wird somit eine sehr gute Testabdeckung mit Kundenbezug umgesetzt. Des Weiteren wird durch Refactoring und Tests gut strukturierter und dokumentierter Programmcode generiert. Außerdem kann auf Projektänderungen flexibel eingegangen werden, da das agile Manifest mit seinen Prinzipien unterstützt wird.

Kritisch zu betrachten sind jedoch benötigte Tests, die nicht oder nur schwer in User-Stories definiert werden können. Darunter sind zum Beispiel Sicherheitsaspekte einer Webanwendung oder Vorgaben für Latenzzeiten zu verstehen. Ohne den Bezug zu einer User-Story können solche technischen Tests leicht übersehen werden.

Literaturverzeichnis

David Chelimsky et.al., The RSpec Book (2010), ISBN: 9781-93435-637-1

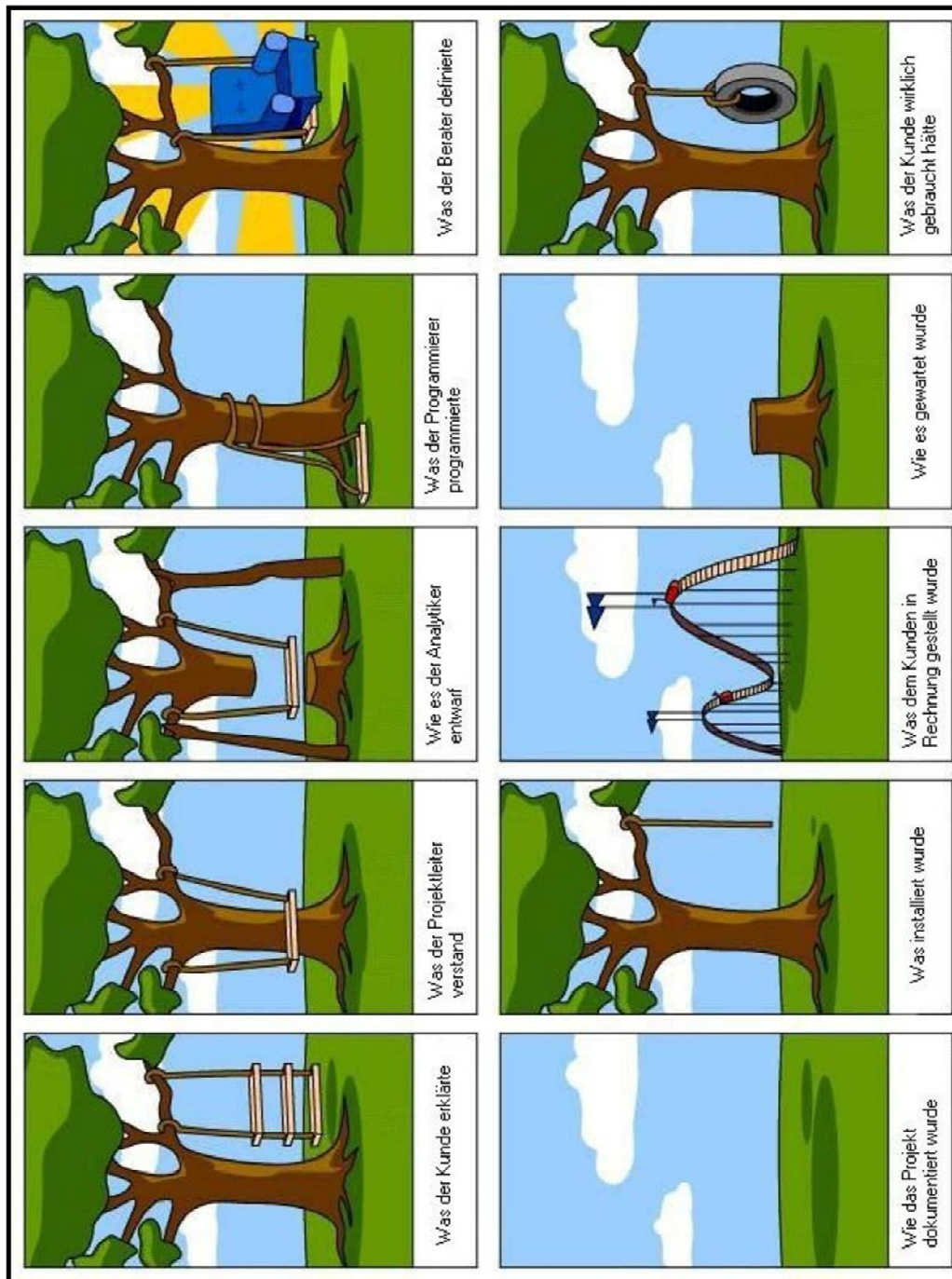
Martin Fowler, Refactoring: Improving the Design of Existing Code (2011),
ISBN: 978-0201485677

Matt Wynne, Aslak Hellesoy, The Cucumber Book: Behaviour-Driven
Development for Testers and Developers (2012), ISBN: 978-1934356807

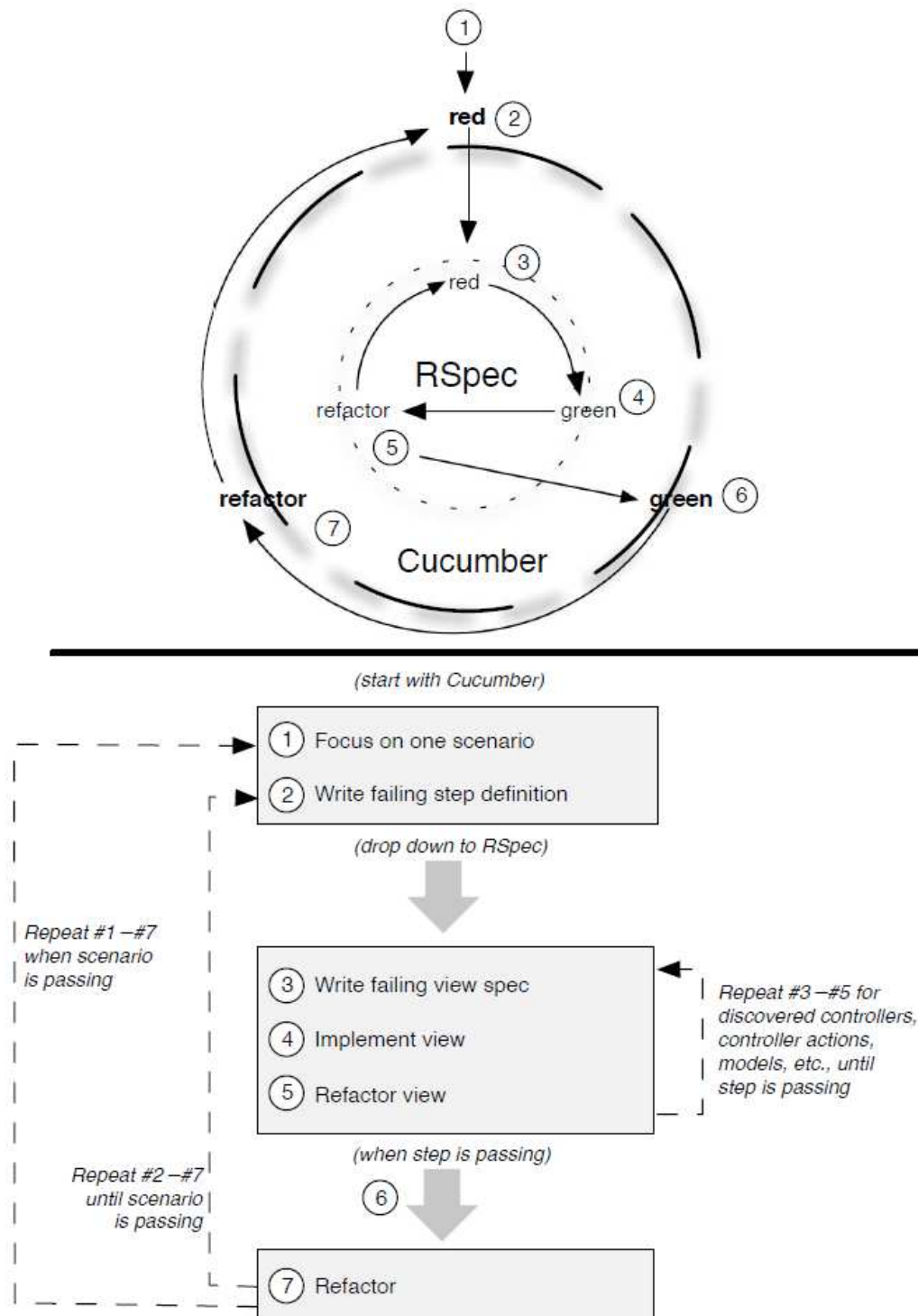
Stefan Sprenger, Kieran Hayes, Ruby on Rails 3.1 Expertenwissen:
Eine praxisorientierte Einführung in die Entwicklung mit Ruby on Rails (2011)
ISBN: 978-3898646970

A Anhang

A.1 Abstimmungsprobleme in Software-Projekten



A.2 BDD-Delivery Cycle mit RSpec und Cucumber¹⁴



¹⁴ David Chelimsky, The RSpec Book (2010), Seite 279

A.3 Cucumber: Feature-Dateien für das Sozialversicherungsbeispiel

./features/admin.feature

Feature: Administrate the social insurance percentages

As an Administrator

I want to edit the social insurance percentages between 0 and 30 %
So that I can keep the calculator-page up-to-date

Background:

Given I am on the admin page

Scenario: Successfull update social insurance percentage

Given I want to update "Krankenversicherung"

When I update the percentage to "5.0"

Then I should see on the admin page "5.0" for "Krankenversicherung"

Scenario: Failed to update social insurance percentage

Given I want to update "Krankenversicherung"

When I update the percentage to "55"

Then I should receive an error

./features/social_insurance_calculator.feature

Feature: Calculate social insurance contributions

As a Employee

I want to calculate my social insurance contributions
based on my gross salary
So that I can make plans with my net salary

Background:

Given I am on the calculate page

And I want to calculate with the social insurance percentages of 2012

Scenario Outline: calculate social insurance contributions

Given I am "<years>" old

And I have "<children>" child / children

When I enter "<gross_salary>" gross salary

And I click on "Berechne..."

Then the outcome should be "<total_contributions>"

Scenarios: No additional nursing care insurance contribution

gross_salary	years	children	total_contributions
3000	22	0	614,25 EUR
3500	23	1	716,63 EUR
4000	25	1	819,00 EUR

Scenarios: Additional nursing care insurance contribution

gross_salary	years	children	total_contributions
3000	23	0	621,75 EUR
3500	23	0	725,38 EUR
4000	25	0	829,00 EUR

A.4 Cucumber:Step-Definitions

A.4.1 Admin_Steps.rb

./features/step_definitions/admin_steps.rb

```
Given /^I am on the admin page$/ do
  visit "/admin/social_insurances"
end

Given /^I want to update "(.*?)"/ do |social_insurance|
  FactoryGirl.create(:health_insurance)
  [...] # FactoryGirl für :nurse_care_insurance, etc.
  @social_insurance = SocialInsurance.find_by_label(social_insurance)
  insurance_path =
    "/admin/social_insurances/" + @social_insurance.id.to_s + "/edit"
  visit insurance_path
end

When /^I update the percentage to "(.*?)"/ do |percentage|
  fill_in "social_insurance_percentage", :with => percentage
  click_button "Ändern"
end

Then /^I should see on the admin page "(.*?) for "(.*?)"/ do
  |percentage, social_insurance|
  page.should have_content "Der Beitragssatz der " + social_insurance + "
  wurde erfolgreich auf " + percentage + " % geändert!"
end

Then /^I should receive an error$/ do
  page.should_not have_content "geändert!"
end
```

A.4.2 Social_Insurance_Contributions_Steps.rb

./features/step_definitions/social_insurance_contributions_steps.rb

```
#encoding: utf-8
Given /^I am on the calculate page$/ do
  visit input_path
end

Given /^I want to calculate with the social insurance percentages of
(\d+)$/ do |arg1|
  FactoryGirl.create(:nurse_care_insurance)
  FactoryGirl.create(:additional_nurse_care_insurance)
  FactoryGirl.create(:health_insurance)
  FactoryGirl.create(:unemployment_insurance)
  FactoryGirl.create(:pension_insurance)
end

Given /^I am "(.*?)" old$/ do |years|
  fill_in "age", :with => years
end

Given /^I have "(.*?)" child \// children$/ do |number_children|
  if number_children.to_i > 0
    check "have_children"
  else
    uncheck "have_children"
  end
end

When /^I enter "(.*?)" gross salary$/ do |salary|
  fill_in "gross_salary", :with => salary
end

When /^I click on "(.*?)"$/ do |calculate_button|
  click_button calculate_button
end

Then /^the outcome should be "(.*?)"$/ do |contributions|
  page.should have_content "Die Sozialversicherungsbeiträge betragen
insgesamt: " + contributions
end
```

A.5 RSpec: Specs und Factories mit FactoryGirl

A.5.1 Factories

./spec/factories/social_insurance.rb

```
FactoryGirl.define do
  factory :nurse_care_insurance, :class => SocialInsurance do
    label "Pflegeversicherung"
    percentage 1.95
  end

  factory :health_insurance, :class => SocialInsurance do
    label "Krankenversicherung"
    percentage 15.5
  end

  factory :unemployment_insurance, :class => SocialInsurance do
    label "Arbeitslosenversicherung"
    percentage 3.00
  end

  factory :pension_insurance, :class => SocialInsurance do
    label "Rentenversicherung"
    percentage 19.6
  end

  factory :additional_nurse_care_insurance, :class => SocialInsurance do
    label "Zusatzbeitrag Pflegeversicherung"
    percentage 0.25
  end
end
```

A.5.2 View-Specs

A.5.2.1 View-Specs: Calculator

./spec/views/calculator/input.html.erb_spec.rb

```
require 'spec_helper'

describe "calculator/input.html.erb" do
  it "show the calculator form" do
    render
    assert_select "form", :action => output_path(:have_children => true, :age
=> 23, :gross_salary => 3000), :method => "post"
  end
end
```

./spec/views/calculator/output.html.erb_spec.rb

```
require 'spec_helper'

describe "calculator/output.html.erb" do
  it "renders the outcome of the calculation" do
    render
    assert_select "tr>td", :text => "Krankenversicherung".to_s, :count => 1
    assert_select "tr>td", :text => "Pflegeversicherung".to_s, :count => 1
  end
end
```

A.5.2.2 View-Specs: Social Insurances

./spec/views/social_insurances/edit.html.erb_spec.rb

```
require 'spec_helper'

describe "social_insurances/edit" do
  before(:each) do
    @social_insurance = assign(:social_insurance, stub_model(SocialInsurance,
      :percentage => "9.99"
    ))
  end

  it "renders the edit social_insurance form" do
    render

    # Run the generator again with the --webrat flag if you want to use webrat
    # matchers
    assert_select "form", :action =>
social_insurances_path(@social_insurance), :method => "post" do
      assert_select "input#social_insurance_percentage", :name =>
"social_insurance[percentage]"
    end
  end
end
```

A.5.2.2 View-Specs: Social Insurances

./spec/views/social_insurances/index.html.erb_spec.rb

```
require 'spec_helper'

describe "social_insurances/index" do
  before(:each) do
    assign(:social_insurances, [
      stub_model(SocialInsurance,
        :label => "Pflegeversicherung",
        :percentage => "9.99"
      ),
      stub_model(SocialInsurance,
        :label => "Krankenversicherung",
        :percentage => "9.99"
      )
    ])
  end

  it "renders a list of social_insurances" do
    render
    assert_select "tr>td", :text => "Krankenversicherung".to_s, :count => 1
    assert_select "tr>td", :text => "Pflegeversicherung".to_s, :count => 1
    assert_select "tr>td", :text => "9.99".to_s, :count => 2
  end
end
```


A.5.3 Controller-Specs

A.5.3.1 Calculator

./spec/controllers/calculator_controller_spec.rb

```
require 'spec_helper'

describe CalculatorController do

  before(:each) do
    FactoryGirl.create(:health_insurance)
    FactoryGirl.create(:nurse_care_insurance)
    FactoryGirl.create(:unemployment_insurance)
    FactoryGirl.create(:pension_insurance)
    FactoryGirl.create(:additional_nurse_care_insurance)
  end

  describe "GET 'input'" do
    it "returns http success" do
      get 'input'
      response.should be_success
    end
  end

  describe "POST 'output'" do
    it "returns http success" do
      post 'output', {:age => 23, :gross_salary => 3000, :have_children =>
false}
      response.should be_success
    end
  end
end
```

A.5.3.2 Social Insurance

./spec/controllers/social_insurances_controller_spec.rb

```
require 'spec_helper'

describe SocialInsurancesController do
  before(:each) do
    FactoryGirl.create(:health_insurance)
    FactoryGirl.create(:nurse_care_insurance)
    FactoryGirl.create(:unemployment_insurance)
    FactoryGirl.create(:pension_insurance)
    FactoryGirl.create(:additional_nurse_care_insurance)
  end

  describe "GET index" do
    it "should render index of social insurance" do
      get :index
      social_insurances = SocialInsurance.all
      response.should render_template("index")
      assigns(:social_insurances).should eq(social_insurances)
    end
  end

  describe "GET edit" do
    it "should render edit form for health insurance" do
      social_insurance = SocialInsurance.find_by_label("Krankenversicherung")
      get :edit, {:id => social_insurance.to_param}
      assigns(:social_insurance).should eq(social_insurance)
    end
  end

  describe "PUT update" do
    describe "with valid params" do
      it "updates the requested social_insurance" do
        social_insurance = SocialInsurance.find_by_label("Krankenversicherung")
        put :update, {:id => social_insurance.id, :social_insurance =>
                      {'percentage' => '3'}}
      end

      it "redirects to the social_insurances" do
        social_insurance = SocialInsurance.find_by_label("Pflegeversicherung")
        put :update, {:id => social_insurance.id, :social_insurance =>
                      {'percentage' => '5'}}
        response.should redirect_to(social_insurances_path)
      end
    end

    describe "with invalid params" do
      it "should render edit form" do
        social_insurance = SocialInsurance.find_by_label("Krankenversicherung")

        put :update, {:id => social_insurance.id, :social_insurance =>
                      {'percentage' => '55'}}
        response.should render_template("edit")
      end
    end
  end
end
```

A.5.4 Model-Spec

./spec/models/social_insurance_spec.rb

```
require 'spec_helper'

describe SocialInsurance do

  before(:each) do
    @SocialInsurance = FactoryGirl.create(:health_insurance)
  end

  describe "validations" do
    context "given a valid percentage of a social_insurance" do
      it "has a percentage between 0.2 and 30" do
        @SocialInsurance.percentage = 30
        @SocialInsurance.should have(:no).errors_on(:percentage)

        @SocialInsurance.percentage = 0.2
        @SocialInsurance.should have(:no).errors_on(:percentage)
      end
    end
    context "given a invalid percentage of a social_insurance" do
      it "has a percentage with 0" do
        @SocialInsurance.percentage = 0
        @SocialInsurance.should have(1).error_on(:percentage)
      end

      it "has a percentage over 30" do
        @SocialInsurance.percentage = 35
        @SocialInsurance.should have(1).error_on(:percentage)
      end
    end
  end
end
```

A.5.5 Routing-Specs

./spec/routing/social_insurances_routing_spec.rb

```
require "spec_helper"

describe SocialInsurancesController do
  describe "routing" do

    it "routes to #index" do
      get("admin/social_insurances").should
      route_to("social_insurances#index")
    end

    it "routes to #edit" do
      get("admin/social_insurances/1/edit").should
      route_to("social_insurances#edit", :id => "1")
    end

    it "routes to #update" do
      put("admin/social_insurances/1").should
      route_to("social_insurances#update", :id => "1")
    end

  end
end
```

./spec/routing/calculator_routing_spec.rb

```
require "spec_helper"

describe CalculatorController do
  describe "routing" do

    it "routes to #input" do
      get("calculator/input").should route_to("calculator#input")
    end

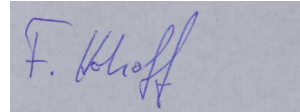
    it "routes to #output" do
      post("calculator/output").should route_to("calculator#output")
    end

  end
end
```

Erklärung

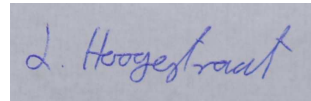
Hiermit erklären wir, dass wir die vorliegende Arbeit selbständig angefertigt haben. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut haben wir als solches kenntlich gemacht.

Münster, 17.06.2012 ____
Datum, Ort



Unterschrift

Münster, 17.06.2012 ____
Datum, Ort



Unterschrift