Fachhochschule Köln
Cologne University of Applied Sciences

Testbarkeit von objektorientierten, dynamisch typisierten Programmen

MASTERTHESIS

ausgearbeitet von

Eduard Litau, 11038839

zur Erlangung des akademischen Grades
MASTER OF SCIENCE (M.SC.)

vorgelegt an der

FACHHOCHSCHULE KÖLN
CAMPUS GUMMERSBACH
FAKULTÄT FÜR INFORMATIK UND
INGENIEURWISSENSCHAFTEN

im Studiengang
MEDIENINFORMATIK

Erster Prüfer:	Prof. Dr. Mario Winter Fachhochschule Köln
Zweiter Prüfer:	M. Sc. Andreas Bade Fachhochschule Köln

Gummersbach, im August 2013

Adressen:

Eduard Litau
Venloer Str. 298
50823 Köln
eduard.litau@gmail.com

Prof. Dr. Mario Winter
Fachhochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
mario.winter@fh-koeln.de

Andreas Bade
Im Hilgersfeld 112
51427 Bergisch Gladbach
andi.bade@gmail.com

Inhaltsverzeichnis

Abbildungsverzeichnis iv

Listings v

Abkürzungsverzeichnis vi

1 Einleitung 1
1.1 Motivation 1
1.2 Ziel dieser Arbeit 1
1.3 Fragestellungen 2
1.4 Vorgehen 2
1.5 Abgrenzung 2

2 Grundlagen 4
2.1 Qualitätssicherung 4
2.1.1 Qualität in Benutzung 4
2.1.2 Produktqualität 5
2.2 Testen von Software 6
2.2.1 Testen im Entwicklungszyklus 7
2.2.2 Testmethoden 8
2.2.3 Testobjekt 11

3 Objektorientierte, dynamisch typisierte Programmiersprachen 12
3.1 Objektorientierung 12
3.2 Typen und Typsysteme 13
3.2.1 Verwendung von Typen 14
3.2.2 Statische Typisierung 14
3.2.3 Dynamische Typisierung und Duck-Typing 14
3.3 Introspektion und Metaprogrammierung 16
3.4 Überblick der Programmiersprache Ruby 17

4 Testbarkeit von Software 19
4.1 Definition 19

4.2 Vorteile durch Testbarkeit 20
4.3 Eigenschaften testbarer Software 21
4.3.1 Kontrollierbarkeit 21
4.3.2 Beobachtbarkeit 22
4.3.3 Softwaredesign 22
4.4 Analyse der Testbarkeit 23
4.5 Konstruktive Ansätze zur Verbesserung der Testbarkeit 24

5 Fehleranalyse 26
5.1 Untersuchungsobjekte 26
5.1.1 simfy.de 26
5.1.2 Spree 27
5.2 Kategorisierung der Fehler 27
5.2.1 Fehlerverteilung 28
5.2.2 Ausnahmen während der Programmausführung 33
5.3 Fehlermodell 34
5.3.1 Fehlerhafte Objektreferenzen 35
5.3.2 Fehlerhafte Parameter 38
5.3.3 Falsche Rückgabewerte von Methoden 39
5.3.4 Fehlender Aufruf einer Methode der Basisklasse 39
5.3.5 Mehrfachvererbung durch Verwendung von Mixins 40
5.3.6 Verletzung des liskovschen Substitutionsprinzips 41
5.3.7 Monkey Patching 41
5.3.8 Dynamische Auflösung von Konstanten und Methoden 42
5.3.9 Dynamische Generierung von Methoden 42

6 Identifikation der Schwachstellen 44
6.1 Einfluss dynamischer Typisierung auf die analytische Qualitätssicherung . 44
6.1.1 Statische Tests 44
6.1.2 Dynamische Tests 45
6.2 Identifikation der Schwachstellen 47
6.2.1 Fehlerhafte Objektreferenzen 47
6.2.2 Fehlerhafte Parameter 47
6.2.3 Falsche Rückgabewerte von Methoden 49
6.2.4 Fehlender Aufruf einer Methode der Basisklasse 50
6.2.5 Mehrfachvererbung durch Verwendung von Mixins 50
6.2.6 Verletzung des liskovschen Substitutionsprinzips 51
6.2.7 Monkey Patching 52
6.2.8 Dynamische Auflösung von Konstanten und Methoden 52
6.2.9 Dynamische Generierung von Methoden 53

7 Prototypische Implementierung **54**
7.1 Funktionalität des Prototyps . . . 54
7.2 Architektur . . . 54
7.3 Evaluierungsmethode . . . 55
7.4 Evaluation der Analyseergebnisse . . . 56
7.4.1 simfy.de . . . 57
7.4.2 Spree . . . 59

8 Verbesserung der Testbarkeit **60**
8.1 Entwurfsmuster und -konzepte zur Steigerung der Testbarkeit . . . 60
8.1.1 Design By Contract . . . 60
8.1.2 Nullobjekt/Special Case Entwurfsmuster . . . 62
8.1.3 Bedingte Ausführung durch Blöcke . . . 63
8.2 Refaktorierungen zur Steigerung der Testbarkeit . . . 64
8.2.1 Separate Query from Modifier - Refaktorierung . . . 65
8.2.2 Extract Method, Class und Module - Refaktorierungen . . . 65
8.2.3 Reduktion von Metaprogrammierung . . . 66
8.3 Dynamische Typisierung und konstruktive Qualitätssicherung . . . 66

9 Fazit und Ausblick **68**

Literaturverzeichnis **71**

Anhang **77**

Fehlerkatalog der Plattform simfy.de **78**

Fehlerkatalog der Plattform Spree **79**

Eidesstattliche Erklärung **80**

Abbildungsverzeichnis

2.1 Produkt Qualitätsmodell nach ISO 25010 - SQuaRE, aus [ISO, 2010] . . . 5
2.2 Qualität im Lebenszyklus von Software nach ISO 25010 - SQuaRE [ISO, 2010] . . . 6
2.3 Allgemeines V-Modell aus [Spillner u. Linz, 2012, S. 42] . . . 7
2.4 Aussagekraft von Tests nach [Freeman u. Pryce, 2009, S. 11] . . . 8

3.1 Programmiersprachen und die verwendeten Typsysteme nach [Ebraert u. Vandewoude, 2005] . . . 16

4.1 Hauptfacetten der Testbarkeit nach [Binder, 1994] . . . 23

5.1 Verteilung der Fehler nach den relevanten und zusammengefassten Fehlermodellkategorien . . . 32
5.2 Verteilung der Ausnahmen, die an mehr als drei Stellen im Programm aufgetreten sind . . . 34
5.3 Ausschnitt der Sequenz zum Abrufen eines Teasers aus simfy.de . . . 38
5.4 Klassen- und Sequenzdiagramm mit Überschreiben der `start` Methode beim Vererben und fehlendem Aufruf der `important_task` Methode . . . 40

6.1 Auszug aus dem Meta-Modell der Programmiersprache Ruby . . . 48
6.2 Auszug aus dem Meta-Modell der Programmiersprache Ruby mit relevanten Methoden von `Module` . . . 51

7.1 Klassendiagram des Prototyps . . . 55

Listings

3.1 Namensschema für Variablen in Ruby 18

5.1 Beispiel für das Auslösen einer ArgumentError Ausnahme 33
5.2 Beispiel für das Auslösen einer TypeError Ausnahme 34
5.3 Aufruf einer Methode auf dem Nullwert 37
5.4 Testfall für die Methode country der Klasse TeaserRepository 37
5.5 Falscher Ausgangspunkt für die Auflösung einer Konstante 42

6.1 Monkey Patching einer Methode 52

7.1 Fehler durch Überschreiben einer fremden Instanzvariable im simfy.de Code 58
7.2 Erkannte Schwachstelle im Modul Order aus dem Spree Quelltext 59

8.1 Codebeispiel für die bedingte Ausführung durch Blöcke 64

Abkürzungsverzeichnis

DSL Domain Specific Language

IDE Integrated Development Environment 67
IEEE Institute of Electrical and Electronics Engineers 13
ISO International Organization for Standardization 4
OCL Object Constraint Language 48
SQuaRE Systems and software product Quality and Requirements 4
UML Unified Modeling Language 48

1 Einleitung

1.1 Motivation

Qualität spielt eine maßgebliche Rolle für den Erfolg von Softwareprodukten. Die Entwicklung von Applikationen ist eine komplexe Aufgabe und dabei entstehen unweigerlich Fehler, die die Qualität reduzieren können. Im Rahmen der Qualitätssicherung ist das Testen von Software eine bewährte Methode zum Auffinden von Fehlern sowie der Steigerung von Vertrauen in das Produkt. Jeder vorab erkannte und behobene Fehler reduziert den Aufwand und damit die Kosten für die Wartung der Software bei gleichzeitiger Verbesserung der Qualität in Benutzung.

Um die Qualität zu steigern, ist es hilfreich, wenn das Testen erleichtert wird und die Software somit auch unter dem Gesichtspunkt der Testbarkeit entwickelt wird. Nach [Spillner u. Linz, 2012, S. 15] beträgt der Testaufwand in der Regel zwischen 25 und 50 Prozent des Gesamtaufwandes eines Softwareentwicklungsprojektes. Durch die Verbesserung der Testbarkeit entstehen sowohl finanzielle als auch zeitliche Vorteile.

Gleichzeitig nimmt die Verwendung von dynamisch typisierten Sprachen wie Ruby und Javascript stetig zu. Diese Thesis beleuchtet die Auswirkungen dynamischer Typisierung auf die Testbarkeit von objektorientierten Programmen.

1.2 Ziel dieser Arbeit

Das Ziel dieser Arbeit besteht in der Identifizierung von Faktoren, die auf die Testbarkeit von objektorientierter, dynamisch typisierter Software Einfluss nehmen. Durch die Ermittlung und Beschreibung gängiger Fehlerquellen können analytische und konstruktive Qualitätssicherungsmaßnahmen daraufhin optimiert werden, so dass Effizienz und Effektivität des Testens gesteigert werden kann.

1.3 Fragestellungen

Die folgenden Fragen beziehen sich auf objektorientierte, dynamisch typisierte Programmiersprachen und sollen im Verlauf der Thesis diskutiert und beantwortet werden.

- Welche Schwierigkeiten im Allgemeinen begleiten die Erstellung von Tests?
- Was sind die typischen Fehlerquellen und welche Eigenschaften und Sprachkonstrukte begünstigen Fehler?
- Mit welchen Methoden können die Fehlerquellen erkannt werden und welche Schwierigkeiten gibt es dabei?
- Wie kann die Testbarkeit von Anwendungen verbessert werden, so dass die typischen Fehler reduziert oder zumindest leichter erkannt werden können?

1.4 Vorgehen

Die Grundlagen geben eine Einführung in die Thematik der Qualitätssicherung in der Softwareentwicklung. Kapitel 3 erläutert die grundlegenden Eigenschaften objektorientierter Sprachen und geht auf die Unterschiede zwischen dynamischen und statischen Typsystemen ein. Das darauffolgende Kapitel beleuchtet im Detail das Konzept der Testbarkeit. Die Eigenschaften testbarer Software werden beschrieben sowie gängige Methoden zur Analyse und Verbesserung der Testbarkeit dargestellt.

Die Kapitel 5 bis 8 stellen den Schwerpunkt der Thesis dar. Basierend auf Fehlern von Software, die sich in Produktivbetrieb befindet, wird ein Fehlermodell mit Schwachstellenbeschreibungen für objektorientierte, dynamisch typisierte Programme erstellt. Ebenso werden Methoden zur Identifizierung dieser Schwachstellen beschrieben, die in einem Prototyp zur Schwachstellenanalyse implementiert werden. Anschließend werden in Bezug zu den ermittelten Schwachstellen Wege zur Verbesserung der Testbarkeit dynamisch typisierter Programmiersprachen erläutert. Im Fazit erfolgt ein wertender Rückblick auf die gewonnenen Erkenntnisse und mögliche Ansatzpunkte für nachfolgende Arbeiten.

1.5 Abgrenzung

Testbarkeit wirkt sich auf fast alle Phasen der Softwareentwicklung aus. In der Planungsphase muss Testbarkeit als Teil der Anforderungen mitgeschätzt werden. In der

Designphase muss die Testbarkeit aller Artefakte beachtet werden und während der Entwicklung müssen testbare Komponenten entstehen. Die Effizienz der Tests ist direkt von der Testbarkeit betroffen und die Fehlerbehebung im laufenden Betrieb profitiert ebenfalls von der Testbarkeit des Systems.

Gegenstand der Thesis ist die Betrachtung der Testbarkeit beim Einsatz von objektorientierten, dynamisch typisierten Programmiersprachen. Hieraus ergibt sich eine Fokussierung auf die Phasen Entwicklung und Test, da hier die Typisierung eine maßgebliche Rolle spielt.

Um den Rahmen der Thesis nicht zu sprengen, erfolgt die gesamte Betrachtung der Thematik am Beispiel der Programmiersprache Ruby. Die Erkenntnisse sind im Regelfall auf andere objektorientierte, dynamisch typisierte Sprache übertragbar.

Das Programmierparadigma hat starken Einfluss auf die Art der möglichen Fehler eines Programms. Um Missverständnisse zu vermeiden, wurde bereits im Titel der Rahmen dieser Ausarbeitung auf objektorientierte Programme begrenzt. Zur besseren Lesbarkeit wird im Folgenden auf die komplette Ausschreibung verzichtet und in den meisten Fällen von dynamisch typisierten Programme gesprochen.

2 Grundlagen

Dieses Kapitel skizziert die thematischen Grundlagen dieser Thesis. Testbarkeit wird in den größeren Rahmen des Qualitätsmodells der International Organization for Standardization (ISO) gestellt und die Bausteine von Softwaretests erläutert.

2.1 Qualitätssicherung

Der internationale Standard ISO 25010 beschreibt im Rahmen der Systems and software product Quality and Requirements (SQuaRE)[1] zwei Qualitätsmodelle für Softwareprodukte. Das Modell für die Qualität in Benutzung besteht aus fünf Charakteristika, die sich auf die Interaktionen zwischen dem Produkt und dem Benutzer beziehen. Das Produktqualitätsmodell beschreibt die internen und externen Charakteristika von Softwareprodukten, die sich sowohl auf statische Eigenschaften der Software als auch auf dynamische Eigenschaften des Computersystems beziehen, siehe Abbildung 2.1. Testbarkeit ist im Qualitätsmodell der ISO als Subcharakteristik der Wartbarkeit von Softwareprodukten angesiedelt. Wartbarkeit wird hierbei als Grad der Effektivität und Effizienz, mit der das Produkt geändert werden kann, definiert.

2.1.1 Qualität in Benutzung

Die Qualität in Benutzung ergibt sich erst aus den unterschiedlichen Kontexten, in denen die Benutzer das Softwaresystem verwenden. Dazu zählen vor allem die Ziele und Aufgaben, die die Benutzer verfolgen, wobei das System sie dahingehend unterstützen soll. Die qualitätsbestimmenden Aspekte beziehen sich hierbei vor allem auf das Gebiet der Mensch-Computer-Interaktion und werden im Rahmen dieser Thesis nicht weiter behandelt.

[1] Die internationale Norm ISO/IEC 25000 ersetzt seit 2005 die Norm ISO/IEC 9126

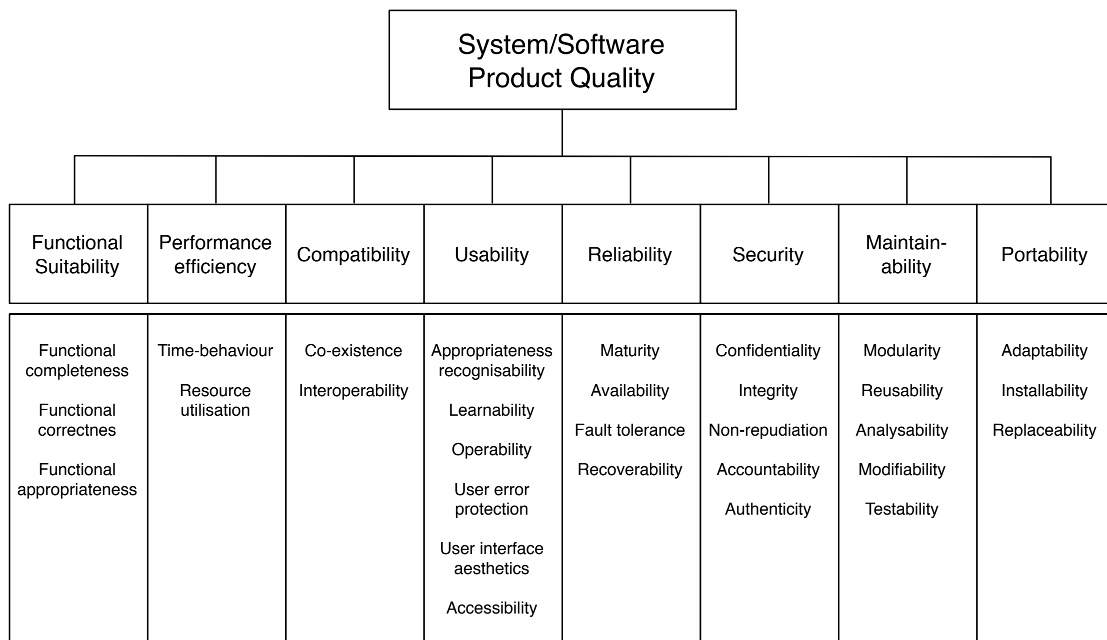

Abbildung 2.1: Produkt Qualitätsmodell nach ISO 25010 - SQuaRE, aus [ISO, 2010]

2.1.2 Produktqualität

Die interne Qualität wird hauptsächlich durch den Entwicklungsprozess beeinflusst. Es werden konstruktive Methoden zur Verbesserung herangezogen, so dass bereits während der Entwicklung das Entstehen von Fehlern minimiert wird. Die externe Qualität wird wiederum maßgeblich durch die interne Qualität beeinflusst. Der Zusammenhang wird durch Abbildung 2.2 verdeutlicht. Die Prozessqualität, also die Gestaltung der entwicklungsbezogenen Tätigkeiten, wirkt sich direkt auf die interne Produktqualität aus. So kann sich z.B. ein testgetriebener Entwicklungsprozess positiv auf die Komplexität und Wiederverwendbarkeit von Komponenten auswirken [Turhan u. a., 2010, S. 212]. Weist die Software eine hohe interne Qualität auf, hat das positive Auswirkungen auf die externe Produktqualität und schließlich auch auf Qualität in Benutzung. Die Steigerung von internen Produktmerkmalen, wie z.B. der Wartbarkeit, ist somit nicht nur für die Entwicklung der Software förderlich.

Die externe Qualität wird durch die Ausführung der Software während des Testens gemessen und zielt somit auf das beobachtbare Verhalten ab. Jede Abweichung von der Spezifikation sowie unerwartete Fehler zeigen Mängel der Qualität auf. Testbare Software erleichtert damit die Prüfung der externen Qualität.

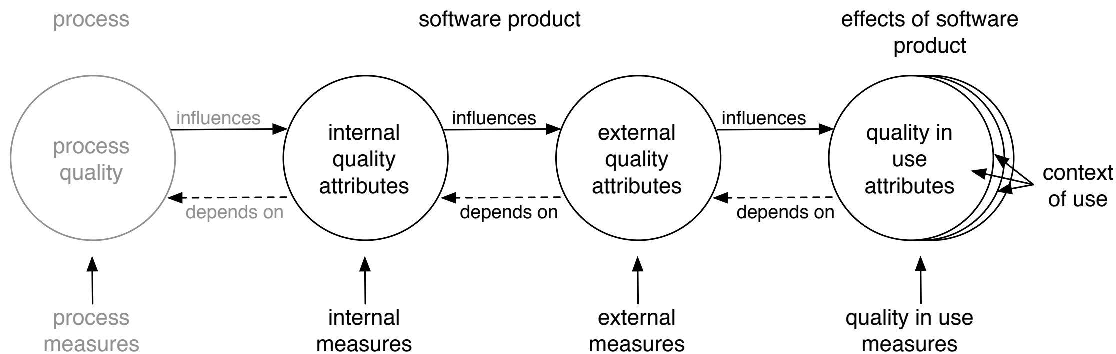

Abbildung 2.2: Qualität im Lebenszyklus von Software nach ISO 25010 - SQuaRE [ISO, 2010]

Die Messung der internen Qualität erfolgt durch Betrachtung statischer Attribute der Software, wozu unterschiedliche Artefakte wie z.B. Entwurfsdokumente und der Quelltext zählen. Testbarkeit ist als interne Qualitätscharakteristik anzusehen, womit die Messung der Testbarkeit durch statische Analyse der Artefakte erfolgt und direkt von der internen Struktur der Artefakte abhängig ist (→ Abschnitt 4.4).

2.2 Testen von Software

Das Testen von Software ist *die* Methode zur Sicherstellung und Steigerung der Qualität. Nach [Spillner u. Linz, 2012, S. 9] werden beim Testen mehrere Ziele verfolgt. Durch das Ausführen des Programms im Rahmen des Testens werden Fehlerwirkungen nachgewiesen, die Qualität bestimmt und das Vertrauen in das Programm erhöht. Zusätzlich kann durch die Analyse bei der Entwicklung entstehender Artefakte Fehlerwirkungen vorgebeugt werden.

Edger W. Dijkstra machte bereits Ende der sechziger Jahre des letzten Jahrhunderts deutlich, dass Softwaretests nur zum Aufzeigen von Fehlern, jedoch nicht zum Widerlegen der Anwesenheit von Fehlern geeignet sind [Dijkstra, 1970]. Zusätzlich ist durch die praktisch endlose Fülle an Möglichkeiten die Anweisungen eines Programms auszuführen, „*der Testaufwand nach Risiko und Prioritäten zu steuern*“ [Spillner u. Linz, 2012, S. 37].

„*Oft finden sich in nur wenigen Teilen eines Testobjekts die meisten Fehlerwirkungen, […]*“ [Spillner u. Linz, 2012, S. 37], womit es sinnvoll ist zu ermitteln, welche Bestandteile oder Konstrukte eines Programms besonders fehleranfällig sind. Aus diesem Grund wird in Kapitel 5 ein spezifisches Fehlermodell für dynamisch typisierte Programmiersprachen erstellt, so dass der Testaufwand gezielt eingesetzt werden kann.

Die folgenden Abschnitte beschreiben die unterschiedlichen Testarten, zunächst nach Entwicklungsphase und anschließend nach dem Bezug zum Testobjekt.

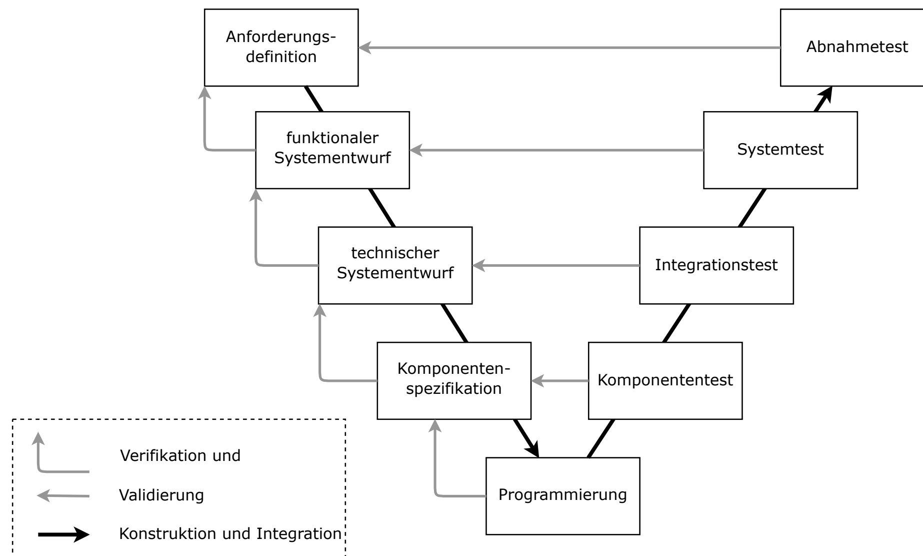

Abbildung 2.3: Allgemeines V-Modell aus [Spillner u. Linz, 2012, S. 42]

2.2.1 Testen im Entwicklungszyklus

Spillner und Linz beschreiben in [Spillner u. Linz, 2012, S. 41ff.] das allgemeine V-Modell, welches jeder typischen Entwicklungsarbeit die entsprechenden Testaktivitäten gegenüber stellt. Abbildung 2.3 zeigt das V-Modell. Die Tests der jeweiligen Ebene zielen auf die Verifikation und Validierung unterschiedlicher Qualitäten ab. Dabei werden die Tests, ebenso wie die Entwicklungsschritte und deren Artefakte, immer spezifischer. Der Abnahmetest findet im Optimalfall mit den Benutzern des Systems statt und testet damit die Qualität in Benutzung. Integrationstests überprüfen die Korrektheit der Zusammenarbeit mehrerer Komponenten und Komponententests verifizieren einzelne, isolierte Komponenten gegenüber ihrer Spezifikation.

Freeman und Pryce verdeutlichen in [Freeman u. Pryce, 2009, S. 11], dass die Testebenen eine unterschiedliche Aussagekraft, bezogen auf die interne und externe Qualität, aufweisen. Komponententests geben Aufschluss über die interne Qualität, also ob z.B. einzelne Klassen ihrer Spezifikation entsprechen, können jedoch nicht dazu benutzt werden, Aussagen zum Gesamtsystem zu geben. Die externe Qualität wird wiederum gut durch Abnahme- und Systemtests validiert. Abbildung 2.4 verdeutlicht diesen Zusammenhang.

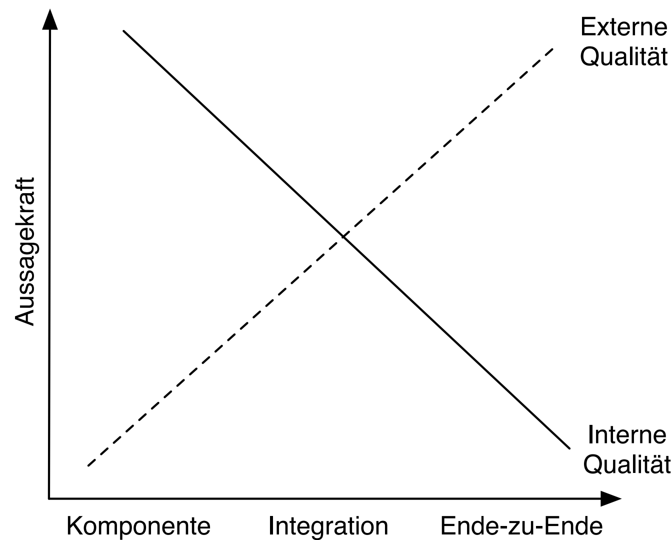

Abbildung 2.4: Aussagekraft von Tests nach [Freeman u. Pryce, 2009, S. 11]

Umgekehrt sollte ein Komponententest nicht zu viel vom restlichen System „kennen“, also Abhängigkeiten dazu aufweisen, was wiederum nur durch geringe Abhängigkeiten und klare Verantwortlichkeiten der Komponente selbst erreicht werden kann. Freeman und Pryce beschreiben es wie folgt: „*So, for a class to be easy to unit-test, the class must have explicit dependencies that can easily be substituted and clear responsibilities that can easily be invoked and verified.*“ [Freeman u. Pryce, 2009, S. 11] Dementsprechend sollte auch ein weit oben im V-Modell gelagerter Test möglichst wenig von der internen Struktur der Komponenten abhängig sein.

2.2.2 Testmethoden

Beim Testen wird grundsätzlich zwischen statischen und dynamischen Tests unterschieden. Bei statischen Tests wird, im Gegensatz zu dynamischen, das Testobjekt nicht ausgeführt, sondern nur auf Inhalt und Struktur hin überprüft. Die folgenden Abschnitte beschreiben beide Varianten basierend auf [Spillner u. Linz, 2012].

Statische Tests

Bei statischen Tests gibt es grundsätzlich zwei Möglichkeiten der Untersuchung. Bei Reviews durch Personen wird das Testobjekt intensiv betrachtet. Als Testobjekte können hierbei bereits frühe Entwürfe der Software dienen, so dass Abweichungen gegenüber

der Spezifikation oder gängigen Normen erkannt und beseitigt werden können. [Fagan, 1976] hebt die Bedeutung von Reviews auch in Bezug auf gesteigerte Produktivität hervor: „*We can conclude from experience that inspections increase productivity and improve final program quality.*“

Weiterhin dienen diese Prüfungen der Prozessoptimierung und sollen vor allem möglichst früh Fehler und Abweichungen aufdecken und damit präventiv wirken. „*Ein weiteres Ziel ist die Ermittlung von Messgrößen oder Metriken, um eine Qualitätsbewertung durchführen und somit Qualität messen und nachweisen zu können.*“ [Spillner u. Linz, 2012, S. 99]

Wie bereits erläutert, ist ein vollständiger Test eines größeren Programms nicht praktisch umsetzbar. Als Teil der analytischen Qualitätssicherung helfen statische Tests bei der Planung von dynamischen Tests, so dass diese durch besondere Beachtung der typischen Schwachstellen oder gegebenen Anforderungen effizienter und effektiver Fehler aufdecken können.

Die zweite Variante von statischen Tests ist die werkzeugbasierte Analyse. Hierbei müssen die Testobjekte in formalisierter Form vorliegen, wie z.B. Entwürfe in UML Notation oder der Quelltext eines Programms. Ein Beispiel dafür ist die Datenflussanalyse. „*Hierbei wird die Verwendung von Daten auf Pfaden durch den Programmcode untersucht*“ [Spillner u. Linz, 2012, S. 102]. Durch dieses Verfahren können Hinweise auf fehlerhafte Nutzung von Variablen gewonnen werden, z.B. wenn eine Variable vor ihrer Initialisierung benutzt wird.

Eine weitere statische Analyseform ist die Ermittlung von Softwaremetriken, die im nächsten Abschnitt erläutert wird.

Softwaremetriken

Softwaremetriken, wie sie z.B. in [Chidamber u. Kemerer, 1994] vorgestellt werden, ermitteln durch rein statische Betrachtung Informationen über den Quelltext. Zu den gewonnenen Informationen zählen Aussagen über Komplexität, Kohäsion, Kupplung, Abhängigkeiten uvm. „*Software metrics can be used to understand applications, to get an overview of a large system and identify potential design problems*“ [Lanza u. a., 2005]. Die Analyse findet hierbei vor oder während der Implementierung statt, um „*die Einhaltung von Richtlinien und Programmierkonventionen zu prüfen*“ [Spillner u. Linz, 2012, S. 99].

Die Ermittlung statischer Metriken stellt ein etabliertes, einfaches Verfahren zur Aufdeckung von Fehlern im Design objektorientierter Programme und damit Wartbarkeitsdefi-

ziten dar [Bruntink u. van Deursen, 2004]. Die statische Analyse zielt also hauptsächlich auf die Prüfung der internen Produktqualität ab. Neben der Betrachtung von einzelnen Komponenten kann aber auch die Beziehung zwischen Komponenten analysiert werden, so dass auch Erkenntnisse auf Integrationsebene gewonnen werden können.

> *„In this study, we collected data about faults found in object-oriented classes. Based on these data, we verified how much fault-proneness is influenced by internal (e.g., size, cohesion) and external (e.g., coupling) design characteristics of OO classes.“* [Basili u. a., 1996]

Im praktischen Einsatz liefern die Werkzeuge meist eine lange Liste von erkannten Problemen, da eine eindeutige Identifikation von Schwachstellen und die Einschätzung des damit verbundenen Risikos nur sehr schwer feststellbar ist. „*It is important not to be blinded by metrics. [...] Metrics are a tool with power but also with limits. There are many aspects of design and its quality that are difficult to measure.*“ [Lanza u. a., 2005, S. 3] Die Analyse der internen Struktur liefert jedoch zumindest wertvolle Informationen über mögliche Fehlerquellen, die für andere Testaktivitäten genutzt werden können.

Ebenso können Metriken dynamische Tests nicht ersetzen, sind jedoch ein wichtiger Baustein bei der Einschätzung der Qualität und der Ermittlung von Schwachstellen. Weiterhin ist der Grad der Formalisierung des Testobjektes von Bedeutung. Je mehr Daten, wie z.B. Typangaben (→ Abschnitt 3.2.1), in die Analyse einfließen können, desto bessere Aussagen über die Qualität und etwaige Fehlerquellen können getroffen werden.

Dynamische Tests

In [Binder, 1999, S. 1085] wird dynamisches Testen als „*Defect prevention and removal accomplished by executing an implementation. That is, testing.*“ definiert. Hierbei wird das Testobjekt mit festgelegten Eingaben in einer kontrollierten Umgebung ausgeführt und die Auswirkungen gegenüber der Spezifikation verifiziert. „*Ziel des [dynamischen] Testens ist der Nachweis der Erfüllung der festgelegten Anforderungen durch das implementierte Programmstück und die Aufdeckung von eventuellen Abweichungen und Fehlerwirkungen.*“ [Spillner u. Linz, 2012, S. 110]

Die Tests können hierbei wieder manuell durch Personen oder mittels Automatisierung erfolgen. Bei automatisierten Tests werden diese durch einen Testtreiber[2] gesteuert und müssen durch Tester oder Entwickler geschrieben werden.

[2] „*Programm bzw. Werkzeug, das es ermöglicht, ein Testobjekt ablaufen zu lassen, mit Testdaten zu versorgen und Ausgaben/Reaktionen des Testobjekts entgegenzunehmen.*“ [Spillner u. Linz, 2012, S. 267]

Dynamische Tests werden wiederum nach der Perspektive, aus der sie erstellt werden, eingeteilt. Whitebox-Verfahren, auch strukturelle Testverfahren genannt, involvieren die interne Struktur des Testobjekts bei der Testfallerstellung und verifizieren z.B. die Abdeckung aller Verzweigungen im Quelltext [Spillner u. Linz, 2012, S. 149]. Hierbei kann der innere Ablauf und die Struktur des Testobjektes betrachtet und bei Bedarf kontrolliert werden. Ein typischer Vertreter von Whitebox-Verfahren ist der Zweig- bzw. Entscheidungstest. Dabei wird überprüft, ob alle Zweige des Kontrollflussgraphen durch die vorhandenen Testfälle abgedeckt wurden.

Blackbox-Verfahren verifizieren das Testobjekt anhand vorhandener Spezifikationen und vermeiden die Berücksichtigung der internen Struktur. Das Testobjekt wird lediglich über die Eingabeparameter des Tests gesteuert. Dadurch können aber auch nur die Ausgaben beobachtet werden. Dadurch sind diese Tests widerstandsfähiger gegenüber Änderungen der internen Implementierung, geben gleichzeitig aber auch weniger Hinweise zur Lokalisierung des Fehlerzustandes[3].

2.2.3 Testobjekt

Die Analyse und Verbesserung der Testbarkeit kann auf allen Ebenen und Phasen der Softwareentwicklung geschehen. So ist es von Vorteil, wenn Anforderungsdokumente eine formale Struktur einhalten und so mittels Review relativ einfach auf Korrektheit und ausreichende Vollständigkeit untersucht werden können. In [Nazir u. a., 2010] wird z.B. ein Rahmenwerk zur Integration von Testbarkeit in der Entwurfsphase beschrieben. Binder geht in [Binder, 1999, S. 269ff.] auf das Testen von Entwürfen basierend auf der UML Notation ein und beschreibt Erweiterungen zur Testbarkeit von UML Modellen.

Im Rahmen dieser Thesis stehen einzelne Komponenten und die Beziehungen zwischen den Komponenten des Programms in automatisierten Tests im Fokus. Entwurfsdokumente oder andere Artefakte werden nicht behandelt.

[3] Als *Fehlerzustand* bezeichnet man den Defekt einer Komponente, die durch eine *Fehlhandlung*, z.B. eines Programmierers, entstanden ist und zu einer wahrgenommenen *Fehlerwirkung* führen kann [Spillner u. Linz, 2012, S. 251].

3 Objektorientierte, dynamisch typisierte Programmiersprachen

Dieses Kapitel beschreibt die Eigenschaften objektorientierter, dynamisch typisierter Programmiersprachen und deren Unterschiede zu statisch typisierten Sprachen. Anschließend wird ein Überblick über die Programmiersprache Ruby gegeben.

In heutigen objektorientierten Programmiersprachen gibt es hauptsächlich zwei Typ-Systeme, statische und dynamische, die sich dadurch unterscheiden, wann ein Typ-Fehler erkannt wird. Bei statisch typisierten Sprachen werden die Fehler beim Kompilieren entdeckt und verhindern die Programmausführung. In dynamisch typisierten Sprachen werden Fehler zur Laufzeit erkannt und führen zum Programmabbruch.

Statische Typisierung hilft damit bei der frühzeitigen Erkennung von bestimmten Fehlern während der Programmierung und erlaubt gleichzeitig dem Compiler Optimierungen im Maschinencode zu platzieren (→ Abschnitt 3.2.1). Dadurch verringert sich im Vergleich zu dynamischer Typisierung aber auch die Ausdrucksstärke des Quelltextes und damit einhergehend die Produktivität beim Programmieren (→ Abschnitt 3.2.3).

3.1 Objektorientierung

In der Informatik existieren unterschiedliche Programmierparadigmen, wozu u.a. die prozedurale, funktionale und objektorientierte Programmierung zählt. Programmiersprachen konzentrieren sich meist auf die Umsetzung eines dieser Paradigmen, erlauben aber auch teilweise die Verwendung von Konzepten aus anderen Paradigmen oder sind als hybride Sprachen entwickelt worden. Die Wahl des Typsystems einer Programmiersprache verhält sich dabei orthogonal zum Paradigma, es ist also nicht davon abhängig.

Da die Programmstruktur stark von der gewählten Programmiersprache abhängt und damit auch die verbundenen Analyse- und Testverfahren variieren, wurde für diese Thesis der Rahmen auf objektorientierte Sprachen festgelegt.

Die objektorientierte Programmierung stellt Objekte als zentrales Konzept in den Mittelpunkt. Ein Objekt vereint Daten und die darauf möglichen Operationen. Objekte kommunizieren über Nachrichten miteinander. Zu den Grundprinzipien der Objektorientierung gehören Polymorphie, Vererbung und Kapselung. Nach [Kirch-Prinz u. Prinz, 2002, S. 25] bestehen die wesentlichen Vorteile gegenüber prozeduraler Programmierung in geringerer Fehleranfälligkeit, besserer Wiederverwendbarkeit und geringerem Wartungsaufwand.

3.2 Typen und Typsysteme

Typen und Typsysteme stellen grundlegende Bausteine heute verwendeter Programmiersprachen dar. Typen dienen der formalen Deklaration und Klassifizierung von Werten und der Ermittlung der darauf möglichen Operationen. Tratt schreibt dazu: „*In programming languages, types are typically used to both classify values, and to determine the valid operations for a given type.*“ [Tratt, 2009]. Alle Objekte eines bestimmten Typs müssen die gleichen Typ-Eigenschaften aufweisen.

In [McDaniel, 1993, S. 712] wird ein Typ folgendermaßen definiert:

Definition (Typ): „*A class of objects. All objects of a specific type can be accessed through one or more of the same interfaces.*“

Das Institute of Electrical and Electronics Engineers (IEEE) definiert in ihrem Glossar zur Softwaretechnik [IEEE, 1990] den Begriff Typ wie folgt:

Definition (Typ): „*A class of data, characterized by the members of the class and the operations that can be applied to them.*“

Beide Definitionen beschreiben die Einordnung von Daten oder Objekten zu einer zusammengehörigen Gruppe und wie der Zugriff auf Elemente dieser Gruppe stattfinden kann.

Dementsprechend liefert [Pierce, 2002, S. 1] folgende Definition für ein Typsystem:

Definition (Typsystem): „*A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*“

Demnach dient ein Typsystem innerhalb eines Programms der Festlegung von Typen und der Umsetzung der damit verbundenen Restriktionen.

3.2.1 Verwendung von Typen

Neben der oben beschriebenen Klassifizierung von Werten ergeben sich aus der Deklaration von Typen in statisch typisierten Sprachen weitere Verwendungsmöglichkeiten.

In erster Linie wird das frühzeitige Erkennen von Fehlern genannt. Dabei sollen Typangaben auch bei der Lokalisierung von Fehlern genauere Angaben liefern. Weiterhin können Typangaben beim Verständnis großer Programme helfen und dienen dabei als im Quelltext eingebettete Dokumentation. Integrierte Entwicklungsumgebungen verwenden Typangaben für Funktionen wie Auto-Vervollständigung und Refaktorierung, wobei auch für dynamisch typisierte Sprachen ähnlich mächtige Werkzeuge zur Verfügung stehen[1]. Compiler verwenden statische Typangaben zur Optimierung des produzierten Maschinencodes [Tratt u. Wuyts, 2007; Tratt, 2009; Pierce, 2002, 4ff.]. Zusätzlich vereinfachen Typ-Deklarationen statische Analysen, wie sie in Kapitel 2.2.2 beschrieben werden.

3.2.2 Statische Typisierung

Bei einer statisch typisierten Programmiersprache muss der Typ von Variablen, Parametern und Rückgabewerten von Methoden im Quelltext angegeben werden oder durch Typinferenz ermittelbar sein. Die Typprüfung erfolgt hierbei zum Kompilierungszeitpunkt. In diesen Programmiersprachen ist der Typ eines Objekts meist durch seine Klasse und die implementierten Schnittstellen festgelegt.

3.2.3 Dynamische Typisierung und Duck-Typing

Bei dynamisch typisierten Programmiersprachen wird der Typ nicht vorab festgelegt, sondern erst zur Laufzeit überprüft. Der Typ eines Objekts wird hierbei nicht durch die Klasse, inkludierte Mixins oder implementierte Schnittstellen, sondern nur durch die vorhandenen Fähigkeiten definiert. Umgangssprachlich wird dieses Prinzip auch *Duck Typing* genannt und wie folgt beschrieben:

> *„Instead, the type of an object is defined more by what that object can do. In Ruby, we call this duck typing. If an object walks like a duck and talks like a duck, then the interpreter is happy to treat it as if it were a duck."* [Thomas u. a., 2009, S. 372]

[1] Für die dynamisch typisierte Programmiersprache Smalltalk existiert bereits seit vielen Jahren eine Programmierumgebung mit umfangreichen Funktionen inkl. der Unterstützung von Refaktorierung [Opdyke, 1992].

Beispielsweise können in Ruby Objekte auf Nachrichten antworten in dem dynamisch zur Laufzeit passende Methoden erzeugt werden. Es ist somit ausschließlich zur Laufzeit möglich, den vollständigen Typ eines Objektes zu erfahren.

Zunehmende Bedeutung dynamisch typisierter Programmiersprachen

Trotz der oben genannten Vorteile statisch typisierter Programmiersprachen gewinnen dynamisch typisierte Sprachen an Bedeutung. Nach [Tilkov, 2008] liegen die Gründe für das gesteigerte Interesse an der gestiegenen Rechenleistung und Verfügbarkeit von schnellem Speicher moderner Computer, höherer Entwicklungsproduktivität sowie Flexibilität zur Laufzeit.

Binder nennt in [Binder, 1999, S. 1085] weitere Vorteile:

- Typ-Deklarationen sind nicht notwendig. Der Übersetzer muss die Arbeit des Typisierens leisten.
- Der Typ eines Objektes kann zur Laufzeit geändert werden, was polymorphe Funktionen erlaubt.
- Übersetzung des Quelltextes kann schneller sein.

Binder nennt in [Binder, 1999, S. 1085] aber auch Nachteile dynamischer Typisierung:

- Das Debugging ist schwerer.
- Erfordert zusätzlichen Speicher für die Beschreibung der Typen zur Laufzeit.
- Prüfung der Typen zur Laufzeit erfordert substantiellen Zusatzaufwand.
- Es ist weniger verlässlich.
- Der resultierende Quelltext ist weniger selbstbeschreibend.

Die Autoren Thomas, Fowler und Hunt von Programming Ruby, einem der bekanntesten Bücher zu Ruby, sind in Bezug auf die Verlässlichkeit durch statische Typdeklarationen anderer Meinung: „*Static typing can be good for optimizing code, and it can help IDEs do clever things with tooltip help, but we haven't seen much evidence that it promotes more reliable code.*" [Thomas u. a., 2009, S. 370]

Weiterhin wird in [Tratt u. Wuyts, 2007] argumentiert, dass statische Typisierung die häufigsten Fehler nicht verhindern kann. Zudem würden dynamische Tests, die für qua-

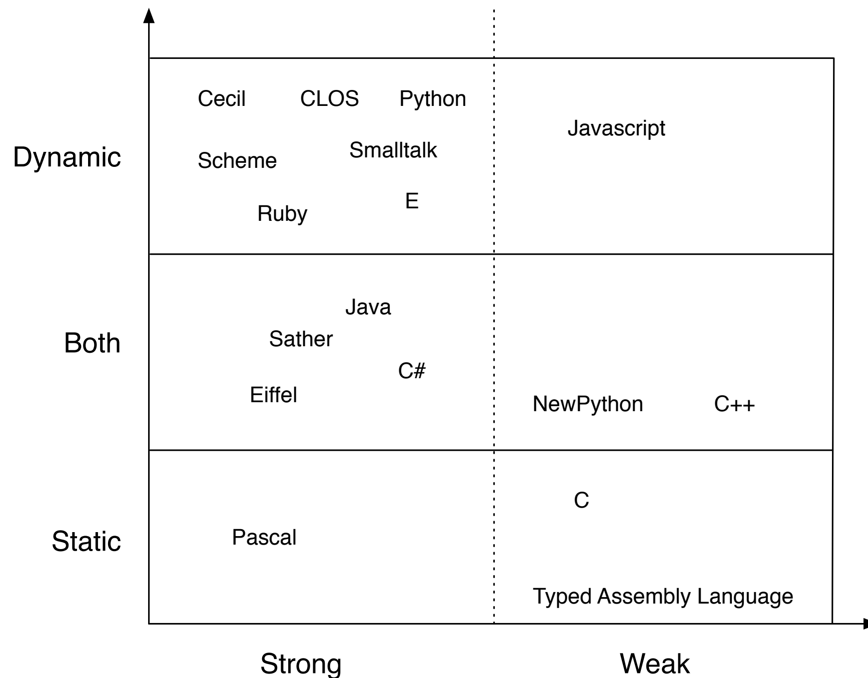

Abbildung 3.1: Programmiersprachen und die verwendeten Typsysteme nach [Ebraert u. Vandewoude, 2005]

litative Software unabdingbar sind, automatisch die meisten durch statische Typisierung erkennbaren Fehler offen legen.

Ein Überblick gängiger Programmiersprachen und ihrer Typsysteme aus [Ebraert u. Vandewoude, 2005] findet sich in Abbildung 3.1.

3.3 Introspektion und Metaprogrammierung

Die meisten dynamisch typisierten Sprachen bieten eine Möglichkeit für Introspektion und erlauben den Einsatz von Metaprogrammierung. Statische Typisierung behindert Metaprogrammierung, da hierbei zur Laufzeit alle Bestandteile des Programms, auch Typen, verändert werden können. „*Dynamic typing does not prevent the presence of a static type system. Nevertheless, static typing and reflection are two properties that do hinder each other.*“ [Ebraert u. Vandewoude, 2005]

Ruby bietet eine umfangreiche Sammlung an Möglichkeiten zur Introspektion von Codestrukturen zur Laufzeit und ebenso deren Manipulation durch Metaprogrammierung. Introspektion ist die Fähigkeit eines Programms, den internen Zustand sowie die vorhan-

denen Strukturen zu untersuchen. Metaprogrammierung erlaubt dem Programm, diese internen Strukturen selbst zur Laufzeit zu verändern, z.B. indem Methoden zu einem Objekt hinzugefügt werden oder die Klassenhierarchie verändert wird [Flanagan u. Matsumoto, 2008, S. 266]. Ein beliebtes Einsatzgebiet von Metaprogrammierung innerhalb von Ruby Programmen ist die Definition von internen, domänenspezifischen Sprachen (sogenannte DSL) [URL:Fowler, a].

Metaprogrammierung ist ein mächtiges Sprachkonstrukt zur Gestaltung flexibler Programme. Jedoch können damit auch sehr einfach wichtige Prinzipien der Objektorientierung umgangen werden. Zum Beispiel erlauben viele Programmiersprachen das Festlegen der Sichtbarkeit von Methoden oder Variablen, um die Kapselung von Objekten sicher zu stellen. Durch Metaprogrammierung können diese Einschränkungen leicht umgangen werden. So bietet Ruby für jedes Objekt die Methode `send` an, womit Methoden unabhängig von ihrer festgelegten Sichtbarkeit ausgeführt werden können. Ebenso ist es möglich, Instanzvariablen einzusehen und zu verändern sowie viele weitere Eingriffe in die Objekt- und Klassenstruktur durchzuführen. Wie bei vielen anderen mächtigen Konzepten ist es notwendig, Metaprogrammierung wohlüberlegt einzusetzen, da damit auch unerwartete Probleme entstehen können.

3.4 Überblick der Programmiersprache Ruby

„*Among its advantages are flexible syntax, powerful metaprogramming facilities, closures, pure object orientation.*“ [Nunes u. Schwabe, 2006]

Die dynamisch typisierte, objektorientierte, interpretierte Programmiersprache Ruby wurde ursprünglich von Yukihiro „Matz“ Matsumoto entwickelt. Die Sprache hat neben Einflüssen aus Lisp, Smalltalk und Perl auch Anleihen aus funktionalen Programmiersprachen wie z.B. den Closures[2] und wird deshalb als Multiparadigmen-Sprache bezeichnet.

Sie verfügt über eine Ausnahmeverwaltung, automatische Speicherbereinigung und ist vollständig objektorientiert, d.h. alle Datentypen, auch Zeichen, Zahlen und Klassen, sind Objekte. Introspektion und Metaprogrammierung ist ein vollwertiger Teil der Sprache mit umfangreichen Möglichkeiten zur Analyse und Manipulation von nahezu allen Bestandteilen eines Programms zur Laufzeit.

Ruby unterstützt neben der reinen Vererbung von Klassen auch Vererbung mittels Mixins, wodurch Code-Wiederverwendung und Polymorphie auf vielfältige Weise realisiert werden können. Module können wie Klassen Methoden, Konstanten und Variablen zusam-

[2] Ein Closure wird auf deutsch Funktionsabschluss oder auch Block genannt.

menfassen und werden ähnlich wie Klassen definiert[3]. Module werden sowohl für Mixins als auch für die Definition von Namensräumen benutzt.

Es gibt lokale Variablen, Instanzvariablen, Klassenvariablen, globale Variablen und Konstanten, die sich jeweils durch ihre Schreibweise unterscheiden, siehe Listing 3.1. Kommentare beginnen mit einer Raute und werden in den nachfolgenden Listings zur Verdeutlichung der Rückgabe von Ausdrücken verwendet.

Listing 3.1: Namensschema für Variablen in Ruby

```
1 locale_variable = 1
2 @instance_variable = 2
3 @@class_variable = 3
4 $global_variable = 4
5 Constant = 5
```

Im Folgenden wird teilweise eine verkürzte, unter Ruby Entwicklern gebräuchliche, Variante für die Angabe von Klassen bzw. Modulen und ihren Methoden verwendet. Bei der Trennung von Empfänger und Nachricht mit einer Raute, z.B. `Enumerable#each` ist eine Instanzmethode gemeint, bei der Trennung mit einem Punkt, z.B. `File.open` ist eine statische Methode gemeint.

[3] Die Realisierung von Klassen und Modulen innerhalb von Ruby weist nur geringe Unterschiede auf. Klassen sind Module, von denen Instanzen erstellt werden können, die jedoch nicht in andere Module bzw. Klassen als Mixins inkludiert werden können.

4 Testbarkeit von Software

Dieses Kapitel beschreibt die verschiedenen Aspekte der Testbarkeit. Dazu wird zunächst der Begriff Testbarkeit definiert. Daraufhin wird beschrieben, warum Testbarkeit wichtig ist und welche Vorteile aus der Verbesserung von Testbarkeit entstehen. Schließlich werden basierend auf einer Literaturrecherche allgemeine Methoden zur Analyse und Verbesserung der Testbarkeit vorgestellt.

4.1 Definition

Testbarkeit wurde in Kapitel 2.1 als Untercharakteristik der Wartbarkeit innerhalb des Qualitätsmodells der ISO eingeführt. Nun soll der Begriff genauer definiert werden.

Das IEEE liefert in [IEEE, 1990] zwei Definitionen:

Definition (Testbarkeit): „*The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.*"

Definition (Testbarkeit): „*The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.*"

Es wird eine Unterscheidung zwischen der Testbarkeit von Anforderungen und dem eigentlichen Softwaresystem getroffen, wobei das Aufstellen der Testkriterien und die Überprüfung dieser Kriterien durch Tests in den Vordergrund gestellt wird.

In [Binder, 1994] wird Testbarkeit in Bezug zum Aufwand gesetzt:

Definition (Testbarkeit): „*Testbarkeit ist die relative Bequemlichkeit und [benötigter] Aufwand für das Aufdecken von Software Fehlern.*"

Weiterhin werden in [Binder, 1994] Kontrollierbarkeit und Beobachtbarkeit als zwei Schlüsselfacetten von Testbarkeit angegeben. Um eine Komponente zu testen, sei es wichtig, die Eingaben zu kontrollieren und die Ausgaben beobachten zu können.

[Jungmayr, 2003] definiert Testbarkeit wie folgt:

Definition (Testbarkeit): „*The degree to which a software artifact facilitates testing in a given test context.*"

Der Fokus der Testbarkeit wird auf alle Artefakte ausgeweitet, die bei der Entwicklung anfallen. Software Artefakte umfassen hierbei nicht nur ausführbaren Quelltext, sondern alle Dokumente oder Produkte, die während der Entwicklung entstehen, wie z.B. Design- und Spezifikations-Dokumente. Die Unterstützung der Testbarkeit ist dabei stark vom Testkontext abhängig, worunter Test Ziele, Ressourcen, Techniken und Werkzeuge fallen.

In der aktuellen Version der ISO Norm SQuaRE 25010 [ISO, 2010] wird folgende Definition verwendet:

Definition (Testbarkeit): „*The ease with which test criteria can be established for a system or component and tests can be performed to determine whether those criteria have been met.*"

Diese Definition bezieht die Schwierigkeit für das Erstellen des Testorakels[1] sowie die Kontrollierbarkeit und Beobachtbarkeit mit ein.

4.2 Vorteile durch Testbarkeit

Die Verbesserung der Testbarkeit von Softwaresystemen ist aus mehreren Gründen von Bedeutung. Da das Testen von Software einen beachtlichen Teil der Kosten eines Softwareprojekts ausmacht [Spillner u. Linz, 2012, S. 15], ist es von Vorteil, den Aufwand dafür zu reduzieren. Spillner und Linz schreiben, dass „*Fehlerzustände, die nicht entdeckt werden, verursachen im Betrieb meist erheblich höhere Kosten*" [Spillner u. Linz, 2012, S. 31]. Durch gesteigerte Testbarkeit und dem somit frühzeitigen Erkennen von Fehlern können die angesprochenen Wartungskosten deutlich verringert werden.

Wird Testbarkeit als ein Ziel der Entwicklung zu Projektbeginn festgelegt, hilft dies beim Schätzen und Planen des Testaufwandes. Deskriptive Fehlermodelle, basierend auf internen Messungen, erlauben einfache Vorhersagen der Testbarkeit während der Entwicklung

[1] Ein Testorakel ist eine „*Informationsquelle zur Ermittlung der jeweiligen Sollergebnisse eines Testfalls (in der Regel Anforderungsdefinitionen oder Spezifikationen)*" [Spillner u. Linz, 2012, S. 265]

und nach Auslieferung der Software [Lopez u. a., 2005]. Wird Testbarkeit von Beginn eines Projekts an umgesetzt, erleichtert dies alle nachfolgenden Testaktivitäten [Pettichord, 2002].

Leicht testbare Software erfüllt meist auch etablierte Gestaltungsprinzipien für objektorientierte Programme wie lose Kupplung, hohe Kohäsion und geringe Redundanzen [Catlett, 2007, S. 73; Shalloway u. Trott, 2004, S. 87]. Dadurch profitieren wiederum andere Qualitätskriterien der Wartbarkeit wie Wiederverwendbarkeit, Modularität und Änderbarkeit.

Schließlich entstehen auch Vorteile für die Diagnose von Fehlern. Durch erhöhte Beobachtbarkeit im Zuge der Testbarkeit wird das Debugging während des Betriebs erleichtert [Chowdhary, 2009].

4.3 Eigenschaften testbarer Software

„*Testability has two key facets: controllability and observability. To test a component, you must be able to control its input (and internal state) and observe its output.*“ [Binder, 1994]

Eingangs wurde in den Definitionen bereits Beobachtbarkeit und Kontrollierbarkeit als wichtige Eigenschaften der Software und ihrer einzelnen Komponenten für erfolgreiches Testen herausgestellt. Ebenso hat die interne Struktur Auswirkungen auf die Testbarkeit. Im Folgenden werden diese Eigenschaften näher beschrieben, wobei hauptsächlich Komponenten- und Integrationstests berücksichtigt werden.

4.3.1 Kontrollierbarkeit

„*Control can be defined as the degree to which you can control the state of the component and system such that all states (code paths) in the component can be easily reached.*“ [Chowdhary, 2009]

Um eine Komponente auf eine bestimmte (Teil-) Funktionalität hin zu überprüfen, sollte es so einfach wie möglich sein, nur diese eine Funktion auszuführen und dabei die Parameter, die die Ausführung beeinflussen, zu kontrollieren. Liegt z.B. eine lange Methode vor, ist es kaum möglich, nur einen Teil davon auszuführen. Viele Abhängigkeiten reduzieren oder verhindern die Kontrollierbarkeit der zu testenden Komponente, da diese entweder durch Platzhalter oder manuell zu initialisierende Kollaborateure aufgelöst werden müs-

sen. Teilweise erfolgt die Erstellung abhängiger Objekte nicht änderbar im Code, so dass im Test keine Kontrolle darüber ausgeübt werden kann [Jungmayr, 2003, S. 70].

Komponententests beziehen sich jedoch immer auf eine einzelne Komponente (→ Abschnitt 2.2.1). Je weniger Abhängigkeiten zu anderen Komponenten bestehen, desto leichter kann Kontrolle darüber ausgeübt und die Testerstellung vereinfacht werden. Weitere Beispiele für die Kontrollierbarkeit von Testobjekten sind Methoden, die ein Attribut auf einen expliziten Wert oder das Objekt auf seinen Ursprungszustand zurücksetzten.

4.3.2 Beobachtbarkeit

„*Observability is the degree to which one can effectively analyze the execution results on both the feature component under test and the overall system in order to accurately determine whether the test succeeded or failed.*“ [Chowdhary, 2009]

Das Testorakel liefert die erwarteten Ergebnisse eines Testfalls. Zur Verifizierung gegenüber diesen Erwartungen muss das Testobjekt seinen Zustand und die Auswirkungen auf das Gesamtsystem leicht beobachtbar machen.

Die Komponente muss entsprechende Schnittstellen bereitstellen, die während des Tests verwendet werden können. Wenn z.B. die Auswirkungen einer Komponente nur durch Heranziehen der Präsentationsschicht oder der Persistenzschicht beobachtet werden können, ist der entsprechende Test mit hohem Aufwand verbunden. Auch andere Daten wie der Ressourcenverbrauch des Testobjekts oder ausgelöste Ereignisse müssen beobachtbar sein. Solche Daten sind meist nicht ohne Weiteres nachträglich erfassbar und müssen von Beginn an eingeplant werden. Auch hängt es von den definierten Testzielen ab, welche Information gesammelt und verifizierbar sein müssen.

4.3.3 Softwaredesign

Wie bei den vorherigen Eigenschaften testbarer Programme bereits deutlich geworden ist, wirkt sich die interne Struktur von Programmen deutlich auf die Testbarkeit aus. In [Shalloway u. Trott, 2004, S. 87] werden etablierte Charakteristika gut strukturierter Software genannt, die zur erhöhten Testbarkeit beitragen:

- Zusammenhängender (kohäsiver) Code
- Lose gekoppelter Code
- Keine Redundanzen

- Lesbarer Code
- Gekapselter Code

Diese Charakteristika wirken sich, wie eingangs erwähnt, auch positiv auf andere Qualitätsmerkmale der Wartbarkeit aus. Dies ist nicht selbstverständlich, da es ebenso vorkommt, dass sich Qualitätsmerkmale gegenseitig behindern und damit nicht gleichzeitig erfüllt werden können.

Binder beschreibt in [Binder, 1994] weitere Einwirkungen auf die Testbarkeit. Neben dem Testobjekt selbst, sind auch der Testprozess, die verwendeten Werkzeuge, vorhandene Spezifikation und Dokumentation usw. von Bedeutung. Abbildung 4.1 stellt diese Einwirkungen in Form von Fischgräten dar.

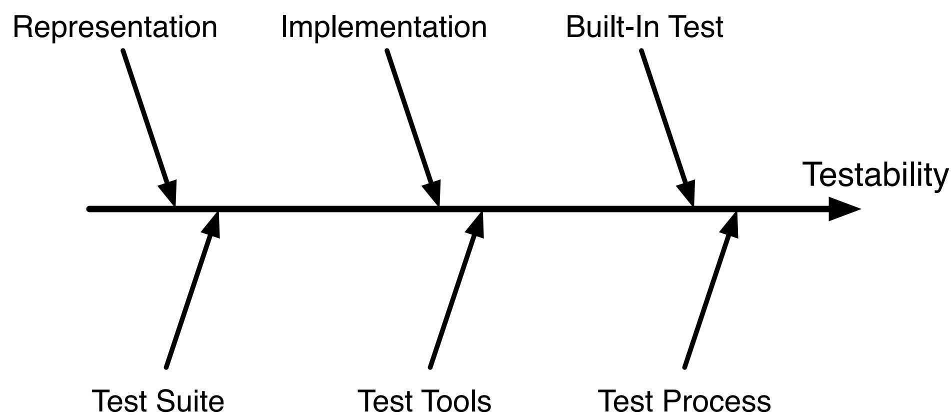

Abbildung 4.1: Hauptfacetten der Testbarkeit nach [Binder, 1994]

4.4 Analyse der Testbarkeit

Zur Verbesserung der Testbarkeit können unterschiedliche Herangehensweisen sinnvoll sein. Auf konstruktiver Ebene ist es hilfreich, bereits vor der Entwicklung der Software Regeln aufzustellen, die die spätere Testbarkeit verbessern. Hierauf wird in Abschnitt 4.5 näher eingegangen. Gleichzeitig ist es von Vorteil, wenn die Einhaltung dieser Regeln automatisiert überprüft werden kann [Spillner u. Linz, 2012, S. 102; du Bousquet, 2010], wozu meist statische Analysen herangezogen werden.

Aussagen über die Testbarkeit eines Programms werden durch Anwendung von Softwaremetriken, wie sie in Kapitel 2.2.2 vorgestellt werden, getroffen. Die Analyse des Programms auf die Erfüllung der oben genannten Kriterien gut strukturierter Software geben indirekt Aufschluss über den Grad der Testbarkeit.

In [Jungmayr, 2003] wird eine Methode zur Bestimmung der Testbarkeit mittels Abhängigkeitsanalyse beschrieben. Die Applikation wird dabei auf Integrationsebene betrachtet und die Anzahl und Art der Abhängigkeiten zwischen Komponenten bestimmen den Grad der Unterstützung von Tests.

Eine Komponente mit hoher Kohäsion weist eine klar definierte Verantwortung auf. In [Badri u. a., 2011] wird anhand einer empirischen Untersuchung eine Korrelation zwischen dem Fehlen von Kohäsion und geringer Testbarkeit in objektorientierten Programmen nachgewiesen. Softwaremetriken zur Bestimmung von Komponenten mit geringer Kohäsion sind etabliert und bewährt. Sie eignen sich damit gut zur Erfassung der Testbarkeit.

In [Bruntink u. van Deursen, 2004] wird die Eignung von einer Mehrzahl von objektorientierten Metriken zur Bestimmung der Testbarkeit von Klassen untersucht. Auch Metriken zur Komplexität wie z.B. die zyklomatische Komplexität nach McCabe[2] lassen Rückschlüsse zur Testbarkeit zu [Spillner u. Linz, 2012, S. 106].

Die vorgestellten Methoden zur Ermittlung der Testbarkeit wurden in nahezu allen Fällen mit statisch typisierten Programmiersprachen entwickelt und erprobt. Als Bestandteil dieser Ausarbeitung wird in Kapitel 6.1 auf die Bestimmung der Testbarkeit von dynamisch typisierten Programmiersprachen eingegangen.

4.5 Konstruktive Ansätze zur Verbesserung der Testbarkeit

„*Like other requirements and design faults, poor testability is very expensive to repair when detected late during software development.*“ [Gao u. Shih, 2005]

Durch frühes Planen der Testbarkeit innerhalb von Softwareprojekten kann der Aufwand für das Testen reduziert werden. Um diesem Umstand gerecht zu werden, wurden Methoden zur Berücksichtigung von Testbarkeitszielen innerhalb von Entwicklungsprozessen entwickelt.

In [Bass u. a., 2012, S. 164] werden Taktiken zur Steigerung von Beobachtbarkeit und Kontrollierbarkeit beschrieben sowie Wege zur Schaffung von Architekturen mit loser Kopplung, hoher Kohäsion und klar abgegrenzten Verantwortlichkeiten gegeben. Eine Checkliste hilft Testbarkeit während des Entwurfs von Softwaresystemen zu berücksichtigen.

[2] McCabe führte in [McCabe, 1976] eine Metrik zur Messung der Komplexität anhand der Anzahl von möglichen Pfaden im Quelltext ein.

Ähnlich dazu stellt David Catlett in [Catlett, 2007, S. 74ff.] das SOCK[3] - Modell inklusive Richtlinien zur Verbesserung der Testbarkeit vor. Die Richtlinien richten sich vor allem an Entwickler und Architekten von Software.

Die Autoren von [Nazir u. a., 2010] erläutern ein „prescriptive framework [...] in order to integrate testability within the development cycle." Darin beschreiben sie Tätigkeiten, die bereits in der Entwurfsphase zur Testbarkeit beitragen und bei der Aufwandsschätzung behilflich sind.

Schließlich wird durch die Extreme Programming Praktik der testgetriebenen Entwicklung nach [Beck u. Andres, 2004, S. 50] die Testbarkeit von Komponenten gesteigert, da die Tests vor dem Testobjekt erstellt werden: „*Code developed test-first is naturally designed for testability*" [Crispin u. Gregory, 2009, S. 111]. Laut [Turhan u. a., 2010, S. 212] zeichnen jedoch Experimente zu testgetriebener Entwicklung ein unschlüssiges Bild in Bezug auf die Komplexität von Programmen. Zwar wies der bei den Experimenten entstandene Code geringere Komplexität auf Methoden- und Klassenebene auf, war aber auf Paket- und Projektebene komplexer.

[3] SOCK steht für Simplicity, Observability, Control and Knowledge.

5 Fehleranalyse

Wie bereits in Abschnitt 2.2 erläutert, ist es beim Testen wichtig, die vorhandenen Ressourcen gezielt einzusetzen. Dazu ist es notwendig zu ermitteln, welche Bereiche der Software die meisten Fehler verursachen werden und daraufhin die Testbarkeit dieser Bereiche zu erhöhen. Unter diesem Aspekt wird ein Fehlermodell auf Basis festgestellter Fehlerwirkungen aufgestellt, welches die häufigsten Schwachstellen in objektorientierten, dynamisch typisierten Programmen herausstellt. Im Anschluss wird beschrieben, wie die ermittelten Schwachstellen im Quelltext identifiziert werden können. Der darauf basierende Prototyp wird im nächsten Kapitel vorgestellt.

5.1 Untersuchungsobjekte

Um realitätsnahe Daten über die Art und Verbreitung typischer Fehlerzustände in dynamisch typisierten, objektorientierten Programmen zu gewinnen, werden die aufgezeichneten Fehlerwirkungen zweier im Einsatz befindlicher Plattformen, die hauptsächlich in Ruby implementiert sind, untersucht. Um die Analyse zu vereinfachen, werden nur behobene Fehler betrachtet, die mit der entsprechenden Änderung im Quelltext assoziiert sind.

5.1.1 simfy.de

Die Musikstreaming-Plattform **simfy.de**[1] startete im Jahr 2010 und hat aktuell über 2 Millionen registrierte Benutzer. Das Produkt wird betriebsintern entwickelt. Neben der untersuchten Web-Applikation, die mit Hilfe des Rahmenwerks Ruby on Rails[2] realisiert wurde, werden Anwendungen für mobile- und desktopbasierte Plattformen angeboten, die über eine HTTP Schnittstelle mit der Web-Applikation kommunizieren. Die Applikation umfasst ca. 40.000 Zeilen Produktiv-Code und ca. 55.000 Zeilen Test-Code in Ruby.

[1] `http://simfy.de`

[2] Ruby on Rails ist ein Rahmenwerk zur Erstellung von Web-basierten Applikation, `http://rubyonrails.org`

Hinzu kommen weitere Bestandteile, die jedoch nicht in Ruby geschrieben sind und somit nicht näher untersucht werden. Insgesamt stehen hier von mehreren Hundert berichteten Fehlern 59 zur Verfügung, die sowohl bestätigt und behoben worden sind als auch eine Verknüpfung zu den Änderungen im Quelltext enthalten.

5.1.2 Spree

Die quelloffene Web-Shop Plattform **Spree**[3] ist ebenfalls seit mehreren Jahren im produktiven Einsatz und mit Hilfe von Ruby und Ruby on Rails umgesetzt. Auf der Open Source Plattform GitHub[4] zählt es zu den fünfzig populärsten Open Source Projekten[5] und verfügt über eine gut gepflegte Fehlerdatenbank mit zum Untersuchungszeitpunkt 99 bestätigten und behobenen Fehlern[6]. Die Anwendung umfasst ca. 14.000 Produktiv-Code Zeilen und ca. 15.000 Test-Code Zeilen.

Zusätzlich zu den Fehlerwirkungen der oben beschriebenen Plattformen werden die bei simfy.de aufgetretenen Ausnahmen als weitere Informationsquelle für das Fehlermodell verwendet.

5.2 Kategorisierung der Fehler

Eingangs wurde die Frage gestellt, welche speziellen Eigenschaften dynamisch typisierter Programmiersprachen Fehler begünstigen. Um der Antwort näher zu kommen, wurde eine Kategorisierung der Fehler durchgeführt. Damit deutlich wird, welche Arten bzw. Kategorien von Fehlerzuständen die meisten Fehlerwirkungen verursachen, wurde für jede Fehlerwirkung eine Ursachenanalyse durchgeführt und der gefundene Fehlerzustand dementsprechend kategorisiert. Die Dimensionen der Kategorisierung basieren auf der von Robert Binder in [Binder, 1999, S. 88ff.] erstellten Fehlertaxonomie. Die Taxonomie fokussiert auf Klassen von Fehlerzuständen in objektorientierten Programmen.

Die Taxonomie von Binder bietet im Vergleich zur internationalen Norm IEEE 1044 [IEEE, 2010] einen höheren Detailgrad und beschreibt Fehler bezogen auf den Quelltext von Programmen. Andere Methoden zur Fehlerkategorisierung wie z.B. die Orthogonal Defect Classification nach [Chillarege, 1996] sind zwar einfacher in der Anwendung, fokussieren jedoch auf Erkenntnisse zur Prozessverbesserung und sind somit nicht zielführend einsetzbar.

[3] `http://spreecommerce.com`
[4] `https://github.com`
[5] `https://github.com/popular/forked`, Stand 15.06.2013
[6] `https://github.com/spree/spree/issues?labels=verified&state=closed`, Stand 15.06.2013

Die Taxonomie unterscheidet die drei Ebenen Methode, Klasse sowie Cluster. Diese Ebenen werden wiederum nach dem zur Fehlhandlung gehörenden Prozessbereich aufgeteilt: Spezifikation, Design oder Implementierung. Hierunter werden dann die konkreten Fehlerkategorien aufgelistet.

5.2.1 Fehlerverteilung

Nicht alle betrachteten Fehlerwirkungen aus den Fehlerdatenbanken waren tatsächlich Fehler. Viele der eingetragenen Fehler waren eher als Änderungswunsch interpretierbar. Da die untersuchten Plattformen neben Ruby auch andere Programmiersprachen wie z.B. Javascript einsetzen, wurden nur Ruby bezogene Fehler beachtet. Von den insgesamt 158 zur Verfügung stehenden Fehlerbeschreibungen wurden 56 ausgefiltert, da die untersuchten Plattformen neben Ruby auch andere Programmiersprachen einsetzen. Tabelle 5.1 listet die Fehlerkategorien auf, zu denen Fehler gefunden wurden und gibt die Anzahl der Fehler je Kategorie an.

Die Einordnung der Fehler in die Taxonomie ist nicht immer eindeutig möglich und unterliegt stark der persönlichen Interpretation. Nichtsdestotrotz können Schlussfolgerungen aus den gewonnen Daten gezogen werden.

Bei der Betrachtung der Fehlerverteilung fällt auf, dass es sehr viele anforderungsbezogene Fehler gibt (ca. 41 Prozent). Diese Fehler sind auf eine falsche Implementierung der Geschäftslogik zurückzuführen, bei denen die Randbedingungen nicht bedacht wurden und dadurch zu einem Fehlverhalten des Programms führen. Nicht erfasste Anwendungsfälle oder implizite Anforderungen, die nicht umgesetzt wurden, führen zu unvollständigen Algorithmen und damit zu Fehlerwirkungen während der Programmnutzung. Teilweise wurden auch Spezifikationen nicht korrekt oder gar nicht umgesetzt. Die relativ hohe Häufigkeit von Fehlern in Bezug auf die Anforderungen spiegelt sich auch in der Umfrage von [Borner u. a., 2006] wieder.

Anforderungsfehler stehen in keinem direkten Zusammenhang zu objektorientierten, dynamisch typisierten Programmen und werden nicht weiter behandelt. Auch Fehlerwirkungen, die in Zusammenhang mit Effizienz und Persistenz stehen, werden nicht näher untersucht. Das unten erläuterte Fehlermodell bezieht sich damit auf die um diese Fehler bereinigten Schwachstellen und basiert somit auf 53 Fehlern. Ebenso beziehen sich die folgenden Prozentangaben auf die bereinigte Fehlermenge.

Die große Mehrheit der Fehler wurde dem *Method Scope* zugeordnet. Meist besteht hier das Problem im Nachrichtenaustausch zwischen den einzelnen Komponenten und der Referenzierung von falschen Objekten innerhalb einer Methode. Oft geben Methoden

unerwartete Werte zurück oder erhalten falsche Parameter. Hierzu zählen die Fehlerkategorien *Contract violation, Server contract violated* und *Message parameters incorrect or missing*. Weiterhin werden Methoden auf den falschen Objekten aufgerufen: *Message not implemented* und *Message sent to object without corresponding method*.

Diese Fehlerkategorien deuten darauf hin, dass die damit verbundenen Fehler durch statische Typisierung hätten verhindert werden können. Im Detail betrachtet wurde hier jedoch sehr oft der Nullwert referenziert oder als Parameter übergeben. Diesen Fehler kann auch statische Typisierung nicht aufdecken, da der Nullwert immer ein valider Wert ist. Die ausführliche Betrachtung der Ausnahmen in Abschnitt 5.2.2 bestätigt diese Aussage. Nur ein Fehler, *Incorrect type coercion*, konnte direkt mit Typisierung in Verbindung gebracht werden.

Bereich			Fehler	Anzahl
Method Scope	Requirements		Incorrect or missing transformation	3
			Requirement omission: missing use case, and so on	17
	Design	Abstraction	Responsibilities overlap	1
		Modularity	Method has low cohesion	3
		Responsibilities	Incorrect algorithm	5
	Implementation	General	Message sent to object without corresponding method	6
			Contract violation	5
		Algorithm	Incorrect output	4
			Message parameters incorrect or missing	5
			Persistence incorrect	6
			Syntax incorrect	2
			Message not implemented	1
			Inefficient, too slow	1
		Exceptions	Missing object	2
		Instance variable define/use	Server contract violated	1
			Incorrect serialization, resulting in corrupted state	4
			Reference to undefined or deleted object	7
			Incorrect type coercion	1
Class Scope	Requirements		Requirement omission	21
	Design	Abstraction	Association missing or incorrect	1
		Refinement	Hierarchy doe not conform to LSP	3
Cluster Scope	Requirements		Requirement omission	1
	Design	Modularity	Inconsistent component	1
		Responsibilities	Contract inconsistent	1

Tabelle 5.1: Fehlerkategorien und Anzahl zugeordneter Fehler

Nachdem die Ursachen der Fehlerwirkungen analysiert und einer Fehlerkategorie aus Binders Taxonomie zugeordnet wurden, erfolgte eine Gruppierung inhaltlich verwandter Kategorien. Abbildung 5.1 zeigt die prozentuale Verteilung der gruppierten Fehlerkategorien. Die Gruppierung ist als ein Schritt zum nachfolgenden Fehlermodell anzusehen und dient der Annäherung an die Eigenschaften dynamisch typisierter Sprachen, die für die meisten Fehler verantwortlich sind.

Die Gruppe **Fehlerhafte Objektreferenzen** fasst die Kategorien *Missing object, Reference to undefined or deleted object* und *Association missing or incorrect* zusammen und repräsentiert mit 29,4 Prozent den größten betrachteten Fehleranteil. Hierbei verweisen Variablen auf falsche Objekte. In den meisten Fällen handelt es sich um Referenzen auf Instanzen die mit dem Nullwert gefüllt sind, in selteneren Fällen auch um referenzierte Klassen, die nicht existieren.

Die Gruppe **Methoden nicht vorhanden** fasst die Kategorien *Message not implemented* und *Message sent to object without corresponding method* zusammen. Fehler in dieser Gruppe beziehen sich auf Nachrichten an Objekte, die die entsprechende Methode nicht implementiert haben. Diese Objekte sind entweder vom falschen Typ oder sind Nullwerte. Ebenso werden die falschen Nachrichten an die richtigen Objekten gesendet, etwa weil die Methode anders benannt ist.

Zu der Gruppe der **Vertragsverletzung** gehören die Kategorien *Contract violation, Server contract violated* und *Contract inconsistent*. Hierbei kommt es zu Fehlern, wenn Objekte in einen inkonsistenten Zustand geraten oder beim Aufruf einer Methode die Vorbedingungen nicht erfüllt sind, z.B. wenn sich ein Parameter im falschen Zustand befindet. Diese Fehler entstehen meist bei semantischer Inkompatibilität zwischen Objekten, die über Typinkompatibilität hinaus geht, d.h. trotz korrekter Typen werden bestimmte Bedingungen zur korrekten Ausführung eines Ablaufs nicht erfüllt.

Die restlichen Kategorien beziehen sich auf einzelne Fehler der Binder'schen Taxonomie. **Falsche oder fehlende Parameter** entspricht *Message parameters incorrect or missing* und betrifft Fehler, bei denen Methoden mit Parametern vom falschen Typ aufgerufen werden oder benötigte Parameter gänzlich fehlen. Die Kategorie **Methode hat geringe Kohäsion** entspricht *Method has low cohesion*-Fehlern und beschreibt Methoden, die aus Gestaltungssicht zu viele Verantwortlichkeiten aufweisen. Dadurch sind sie nur schwer verständlich und testbar. Auch die Kategorie **Inkonsistente Komponente** umfasst solche Fehler, wobei hier ganze Klassen mit zu vielen Verantwortlichkeiten gemeint sind. Fehler der Kategorie **Inkorrekte Typ-Überführung** treten auf, wenn Objekte explizit oder implizit von einem Typ in einen anderen überführt werden sollen und dies fehlschlägt.

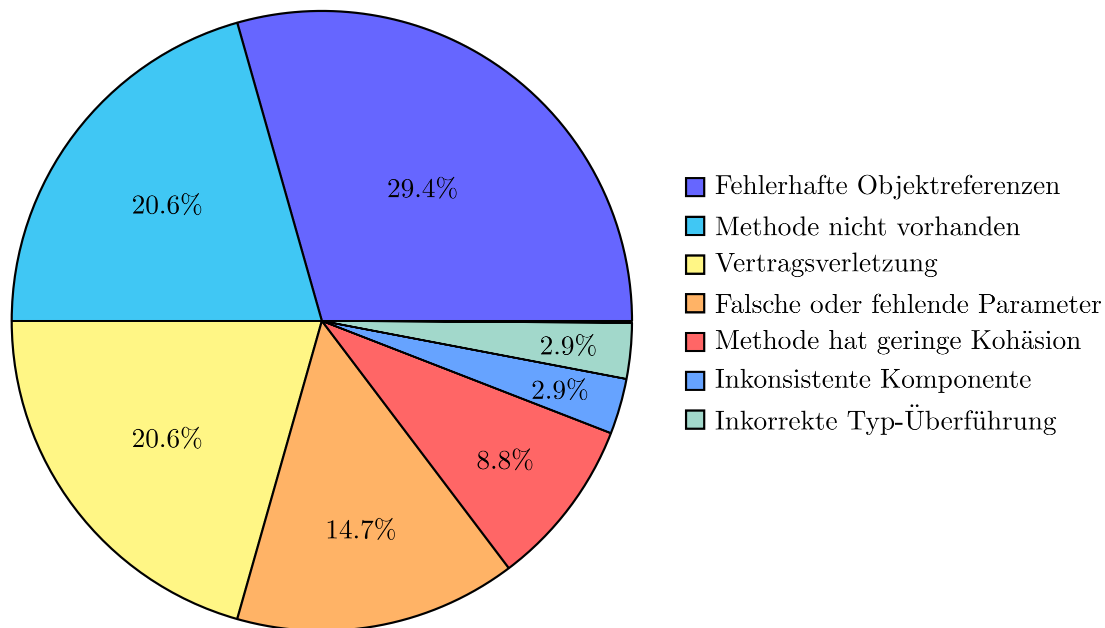

Abbildung 5.1: Verteilung der Fehler nach den relevanten und zusammengefassten Fehlermodellkategorien

Bei der Fehlerursachenanalyse und anschließenden Kategorisierung der Fehler konnte die Meinung, dass in dynamisch typisierten Programmiersprachen Typfehler vermehrt auftreten (→ Abschnitt 3.2.1), nicht bestätigt werden. Nur in wenigen Fällen hätte statische Typisierung geholfen, Fehler zu vermeiden. Bei vielen Fehlern war der Nullwert als Referenz einer Variable, als Parameter oder als Rückgabewert einer Methode beteiligt. Diese Fehlerart kann in statisch typisierten Sprachen ebenfalls erst zur Laufzeit festgestellt werden.

Eine weitere beachtenswerte Fehlerquelle stellen Schnittstellenfehler dar, bei denen zwar die vereinbarten Typen übereinstimmen, jedoch andere Bedingungen oder die Zustände von involvierten Objekten nicht korrekt sind. Dies stimmt mit den Aussagen in [Borner u. a., 2006] überein und scheint unabhängig vom Typsystem der Programmiersprache ein weit verbreitetes Problem darzustellen. Neben den verzeichneten Fehlern dieser Kategorie könnten auch Defekte anderer Kategorien durch den Einsatz von Verträgen minimiert werden (→ Abschnitt 8.1.1).

5.2.2 Ausnahmen während der Programmausführung

Bei der Plattform simfy.de wurden ausgelöste Ausnahmen[7] bei der Programmausführung aufgezeichnet. Ausnahmen führen nicht zwangsweise zu einer Fehlerwirkung und sind damit nicht so aussagefähig wie die erfassten Fehler. Eine Auswertung erscheint jedoch sinnvoll, um abschätzen zu können, welchen Anteil Typ-bezogene Defekte an der Gesamtzahl aller Defekte aufweisen. Im Zeitraum vom 23.01.2013 bis 14.07.2013 wurden an 297 Stellen im Programm Ausnahmen erfasst, wobei manche Ausnahmen nur wenige Male, manche auch tausende Male ausgelöst worden sind. Grafik 5.2 listet die Ausnahmen auf die an mehr als drei Stellen im Programm aufgetreten sind.

Den größten Anteil mit ca. 33 Prozent (insgesamt an 83 Stellen im Quelltext) stellen Ausnahmen vom Typ `NoMethodError` dar. Diese Ausnahmen werden ausgelöst, wenn ein Objekt eine Nachricht bekommt, für die es zur Laufzeit keine Methode implementiert hat. Bei näherer Betrachtung wurde festgestellt, dass bei 72 von den genannten 83 Stellen (87 Prozent) die Nachricht an den Nullwert gesendet wurde. Verglichen mit Java als Vertreter einer statisch typisierten Programmiersprache entsprechen diese Ausnahmen der *NullPointerException* und würden ebenfalls erst zur Laufzeit entdeckt werden können. Die restlichen elf Ausnahmen vom `NoMethodError`-Typ hätten durch statische Typisierung bereits während der Implementierung verhindert werden können.

Die zweithäufigste Ausnahme `ActionView::MissingTemplate` sagt aus, dass HTML Vorlagen, die als Basis für die Generierung von Webseiten dienen, fehlen, d.h. die entsprechende Datei nicht existiert. Hierbei handelt es sich in den meisten Fällen um fehlende Übersetzungen, was eher auf Mängel im Prozess als in der Implementierung hindeutet.

Weitere Typ-bezogene Ausnahmen sind `ArgumentError` und `TypeError`. Eine `ArgumentError` Ausnahme wird ausgelöst, wenn eine Methode mit der falschen Anzahl an Parametern aufgerufen wird oder der Parameter im Kontext der Methode nicht verarbeitet werden kann [URL:RDoc]. Beispielsweise wird die `ArgumentError` Ausnahme ausgelöst, wenn aus einem Datenfeld die ersten x Elemente gezogen werden sollen, die übergebene Zahl x jedoch negativ ist, siehe Listing 5.1. Das Beispiel verdeutlicht, dass hier eine Prüfung über den Typ des Parameters hinaus durchgeführt wird.

Listing 5.1: Beispiel für das Auslösen einer ArgumentError Ausnahme

```
1 [1,2,3].first(-3) # => ArgumentError: negative array size
```

[7] [McDaniel, 1993, S. 248] definiert eine Ausnahme (zu English *Exception*) als „*[...] an abnormal situation that may arise during execution, that may cause a deviation from normal execution sequence, and for which facilities exist in a programming language to to define, raise, recognize, ignore, and handle it.*“

Ein `TypeError` tritt auf, sobald eine implizite Typumwandlung oder eine manuelle Typprüfung fehlschlägt. Im Beispiel aus Listing 5.2 löst die Methode `first` eines Datenfeldes diese Ausnahme aus, falls der Parameter keine Zahl ist. Die Typprüfung findet hierbei nicht automatisch durch den Interpreter statt, sondern muss manuell implementiert werden.

Listing 5.2: Beispiel für das Auslösen einer TypeError Ausnahme

```
1 [1,2,3].first('two') # => TypeError: can't convert String into Integer
```

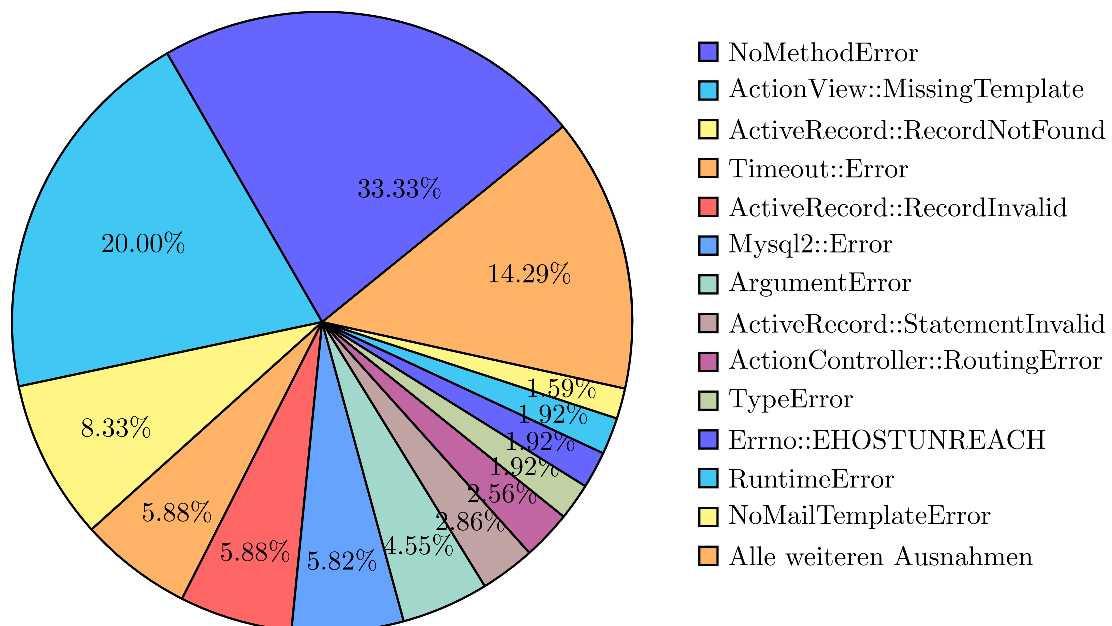

Abbildung 5.2: Verteilung der Ausnahmen, die an mehr als drei Stellen im Programm aufgetreten sind

5.3 Fehlermodell

In Abschnitt 2.2 wurde erläutert, dass nur kleine Programme komplett getestet werden können. Nach [Binder, 1999, S. 66] sollte somit jede Teststrategie mittels eines Fehlermodells geleitet werden.

„*Unter einem Fehlermodell verstehen wir ein quantitatives, empirisches Verfahren zur Bewertung der Qualität eines Softwareproduktes, das gleichzeitig ein Aufwandsschätzmodell für die dazu notwendigen Testaktivitäten ist*“ [Kropfitsch u. Vigenschow, 2003, S. 1]. Ein Fehlermodell basiert auf empirisch gesammelten Daten und gibt Aufschluss auf zukünftige Testaktivitäten und -aufwände. Zusätzlich zu empirisch gesammelten Daten

können Erfahrungen, Experimente oder auch der gesunde Menschenverstand als Basis für das Fehlermodell herangezogen werden.

In [Binder, 1999, S. 66] wird zwischen spezifischen und unspezifischen Fehlermodellen unterschieden. Beim unspezifischen Modell wird das Testen durch Konformität zur Spezifikation geleitet. Die Zulänglichkeit des Modells hängt damit von der Abbildung der Anforderungen ab. Ein spezifisches Fehlermodell legt die Tatsache zugrunde, dass Fehler trotz Konformität zur Spezifikation vorhanden sein können und fokussiert die Suche nach Fehlern in typischen Schwachstellen der Implementierung.

Das folgende, spezifische Fehlermodell basiert auf den in der vorangegangenen Analyse gesammelten Erkenntnissen über die Art und den Kontext der beobachteten Fehlerzustände. Das Modell beschreibt die Schwachstellen objektorientierter, dynamisch typisierter Programme. Die unten genannten Schwachstellen des Modells begünstigen sich teilweise gegenseitig, so dass z.B. inkorrekt definierte Methoden falsche Objekte zurückgeben, die dann wiederum zu inkorrekten Objektreferenzen führen können.

Die ersten drei Schwachstellen, *Fehlerhafte Objektreferenzen*, *Fehlerhafte Parameter* und *Falsche Rückgabewerte von Methoden*, beziehen sich auf ca. 65 Prozent der relevanten Fehler. Bei diesen Fehlerquellen operieren Methoden auf den falschen Objekten, weisen Fehler in Bezug auf die erhaltenen Parameter oder den Rückgabewert auf. Das herausragende Problem dieser Schwachstellen ist der Nullwert, der zu inkorrekten Objektzuständen führt sowie Vor- und Nachbedingungen verletzt.

Die Schwachstellen *Fehlender Aufruf einer Methode der Basisklasse*, *Mehrfachvererbung durch Verwendung von Mixins*, *Zugriff auf Instanzvariablen durch Module* und *Verletzung des liskovschen Substitutionsprinzips* begünstigen Fehler auf Klassen- oder Komponentenebene. Hierin sind die Fehlerkategorien *Vertragsverletzung* und *Inkonsistente Komponente* eingeflossen.

Die darauf folgenden Schwachstellen *Monkey Patching*, *Dynamische Auflösung von Konstanten und Methoden* sowie *Dynamische Generierung von Methoden* entsprechen nicht direkt einer Fehlerkategorie, waren jedoch für eine Reihe von Fehlern verantwortlich und sollen deshalb explizit beschrieben werden. Diese Schwachstellen sind auf die Verwendung von Metaprogrammierung zurück zu führen.

5.3.1 Fehlerhafte Objektreferenzen

Die hier beschriebene Schwachstelle bezieht sich auf die Fehlerkategorie *Fehlerhafte Objektreferenzen* aus der Fehlerkategorisierung in Abschnitt 5.2 und entspricht damit knapp

30 Prozent der relevanten Defekte. Diese Fehlerwirkungen manifestieren sich in den meisten Fällen dadurch, dass anstatt des erwarteten Objekts der Nullwert referenziert wird oder, in sehr wenigen Fällen, auch andere Objekte referenziert werden. Beispielsweise geben nicht initialisierte Instanzvariablen beim Zugriff darauf den Nullwert zurück. Auch wenn der Zugriff auf die Instanzvariable über eine Abfragemethode[8] erfolgt, muss der Sender der Nachricht die Rückgabe auf den Nullwert hin überprüfen. Diese Fehlerquelle entstammt aus der Eigenschaft der meisten objektorientierten Programmiersprachen Referenzen auf den Nullwert zu erlauben und Variablen initial mit dem Nullwert zu belegen. Dieses Verhalten wird auch bei der Programmiersprache Ruby und ihrer Standardbibliothek deutlich, die den Nullwert (bei Ruby `nil` genannt) in vielen Situationen zurück gibt:

- Rückgabewert beim Zugriff auf ein nicht vorhandenes Datenfeld-Element mittels `Hash#[]`
- Standard-Rückgabewert einer Methode ohne Methodenkörper
- Rückgabewert einer if-Verzweigung bei nicht-Erfüllung der if-Bedingung
- Rückgabewert einer case-Verzweigung bei nicht-Erfüllung einer der case-Bedingungen
- Initialer Wert einer Instanzvariable
- Standardrückgabewert vieler Methoden der Standardbibliothek als Hinweis auf einen Fehler oder eine leere Menge, z.B. gibt die Methode `Enumerable#detect` den Nullwert zurück, wenn die Suche nach einem Objekt in einer Kollektion nicht erfolgreich war.
- nil wird in boolschen Ausdrücken als unwahr interpretiert. Dies vereinfacht Codekonstrukte, fördert aber auch die Verwendung des nil-Objekts.

Diesem Prinzip folgend setzen auch von Ruby Entwicklern implementierte Programme das nil Objekt vielfältig ein. So sind bei simfy.de über 25 Prozent der im Betrieb aufgetretenen Ausnahmen der Aufruf einer Methode auf dem nil Objekt, welche in einem `NoMethodError` resultieren, siehe Abbildung 5.2. Die größte Schwierigkeit liegt vor allem darin, dass dadurch nur schwer festzustellen ist, an welcher Stelle in der Ausführungskette das nil Objekt aufgetreten ist. Freeman und Pryce gehen soweit, dass Sie die Vermeidung von Nullwerten in Nachrichten zwischen Objekten zur Gestaltungsregel erklären: „*Never Pass Null between Objects*“ [Freeman u. Pryce, 2009, S. 274].

[8]Eine Abfragemethode, auch Getter Methode genannt, erlaubt den Zugriff auf interne, von außen nicht sichtbare Objektattribute.

Dabei ist das Überprüfen auf das Vorhandensein des nil Objekts nicht immer eine geeignete Lösung. Dies würde zu unnötig vielen Nullwert Überprüfungen führen, die die Komplexität steigern würden. Weiterhin stellt ist Überprüfung auf den Nullwert hin praktisch gesehen eine Typprüfung dar und widerspricht damit sowohl gegen das Duck Typing als auch gegen das *Tell, Don't Ask* Prinzip der Objektorientierung [Freeman u. Pryce, 2009, S. 17]. Es müssen Wege gefunden werden, die Häufigkeit des Auftretens der Nullwerte zu reduzieren.

Die bei dieser Schwachstelle entstehenden Fehlerzustände können ebenfalls als eine Vertragsverletzung interpretiert werden. Aufgrund ihrer Häufigkeit aber werden sie als eigenständige Schwachstelle definiert. Durch die weite Verbreitung des Nullwerts als Standardrückgabewert und initialer Wert von Parametern und Variablen ist es schwierig, eindeutige Charakteristika zur Erkennung und Lokalisierung dieser Fehlerquelle zu definieren. Die obige Liste für Situationen, in denen der Nullwert der Standardrückgabewert ist, liefert jedoch einen guten Überblick für mögliche Quellen des Nullwerts innerhalb der Standardbibliothek von Ruby.

Ein Fehler aus simfy.de verdeutlicht diese Schwachstelle. Die Beschreibung des Fehlers lautet: „*Bei der Anzeige von Teasern kommt es zu Exceptions, wenn kein Benutzer eingeloggt ist. Dadurch erscheint kein Teaser in der jeweiligen App.*“ Im Quelltext wird keine Unterscheidung getroffen, ob ein Benutzer eingeloggt ist oder nicht, so dass die entsprechende Variable (`current_user`) den Nullwert enthalten kann. Es wird eine Ausnahme der Klasse `NoMethodError` geworfen, da die Methode `signup_country_code` auf `nil` anstatt auf einem `User` Objekt aufgerufen wird.

Das Listing 5.3 zeigt die Methode `country_code`, in der die Ausnahme geworfen wird.

Listing 5.3: Aufruf einer Methode auf dem Nullwert

```
1 def country_code
2   params[:country].presence || current_user.signup_country_code
3 end
```

Der vorhandene Test für die Methode `country`, in der `country_code` aufgerufen wird, erstellt einen Stub für die Rückgabe der Methode `country_code`, so dass es nicht zur Auslösung der Ausnahme kommen kann, siehe Listing 5.4. Die oben beschriebene Methode `country_code` ist privat und wurde nicht explizit getestet.

Listing 5.4: Testfall für die Methode country der Klasse TeaserRepository

```
1 describe "#country" do
2   it "should return germany as default if no teaser is available for requested country" do
3     params[:teaser_set_id] = stub(:teaser_set_id, :presence => false)
4     country_without_teasers = stub(:country, :teaser_available? => false)
5     teaser_repository.stub(:country_code)
6     Countries.stub(:by_country_code).and_return(country_without_teasers)
7     teaser_repository.send(:country).should be(Countries::Germany)
8   end
```

```
9 end
```

Das Sequenzdiagramm in Abbildung 5.3 veranschaulicht den Ablauf, der zur Ausnahme führt.

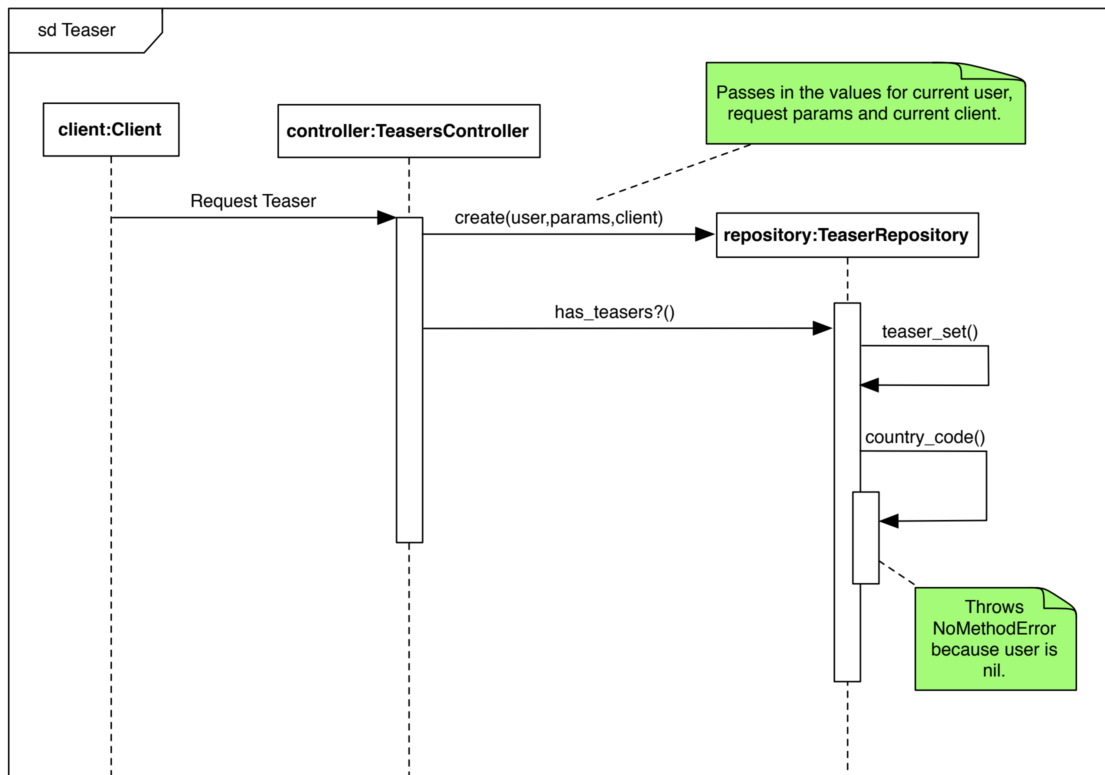

Abbildung 5.3: Ausschnitt der Sequenz zum Abrufen eines Teasers aus simfy.de

5.3.2 Fehlerhafte Parameter

Beim Aufruf einer Methode findet zur Laufzeit lediglich eine Überprüfung der Anzahl der Parameter durch den Interpreter statt. Es besteht also grundsätzlich die Gefahr, dass Parameter in der falschen Reihenfolge übergeben werden. Auch findet beim Aufruf keine Typprüfung statt, so dass Objekte vom unerwarteten Typ übergeben werden können.

Um das Problem der Parameterreihenfolge zu minimieren und die Lesbarkeit zu erhöhen, ist es bei mehr als einem Parameter üblich, diese als Werte eines assoziativen Datenfelds zu übergeben und die Schlüssel des Datenfeldes als Parameternamen zu verwenden.[9] Im Methodenkörper wird über die Schlüssel auf die Parameter aus dem Datenfeld zugegriffen.

[9]Das Verwenden von assoziativen Datenfeldern als benannte Parameter ist aufgrund seiner Popularität als Bestandteil der Ruby Spezifikation übernommen worden und seit Version 2.0 ein Teil der Sprache. Hierbei werden für jeden Schlüssel des Datenfeldes lokale Variablen innerhalb der Methode angelegt.

Hierbei kommt es zum Tragen, dass in Ruby beim Zugriff auf assoziative Datenfelder mittels der oft genutzten Methode `Hash#[]` ein Nullwert zurückgegeben wird, falls für den abgefragten Schlüssel kein Eintrag existiert. Dadurch kommt es zu Fehlern wie sie in Abschnitt 5.3.1 erläutert werden.

Diese Fehlerquelle bezieht sich auf die Fehlerkategorie *Falsche oder fehlende Parameter* und entspricht damit 14,7 Prozent aller betrachteten Fehler. Zusätzlich könnten die hierbei entstehenden Fehler auch als Vertragsverletzung nach Bertrand Meyer angesehen werden [Binder, 1999, S. 807]. Beim Entwurf nach Vertrag wird jedoch mehr als nur der Typ überprüft, womit die hier beschriebene Schwachstelle als Untermenge einer Vertragsverletzung angesehen werden kann. Falsche Parameter verletzten Vorbedingungen, die zur Ausführung der Methode notwendig sind.

Die Typprüfung der Parameter muss vom Entwickler implementiert werden, wobei dies allgemein hin als Antimuster verstanden wird, da in dynamisch typisierten Sprachen das Duck Typing eingesetzt wird (→ Abschnitt 3.2.3).

5.3.3 Falsche Rückgabewerte von Methoden

Die Fehler dieser Schwachstelle haben gemeinsam, dass Methoden falsche Werte zurückgeben. Meist handelt es sich dabei um Nullwerte, teilweise auch um Objekte vom falschen Typ. Die Fehlerwirkungen manifestieren sich dabei oft durch das Auslösen der `NoMethodError` Ausnahme, sobald auf dem Objekt eine Methode aufgerufen wird. Hierin fließen sowohl Fehler aus der Kategorie *Methode nicht vorhanden* als auch der Kategorie *Vertragsverletzung* ein.

5.3.4 Fehlender Aufruf einer Methode der Basisklasse

Werden in einer abgeleiteten Klasse Methoden der Basisklasse überschrieben, ist es in manchen Fällen notwendig, die Basisklassenimplementierung aufzurufen, da die abgeleitete Klasse nur Funktionalitäten ergänzt. Fehlt dieser Aufruf, entstehen meist Fehler im Kontrollfluss, z.B. wird die gewünschte Funktionalität nur in Teilen erfüllt, ohne dass eine Ausnahme ausgelöst wird. Die Schwachstelle bezieht sowohl fehlende Aufrufe von Initialisierungsmethoden als auch normalen Methoden, wie sie im folgenden Beispiel erläutert werden, mit ein.

Abbildung 5.4 veranschaulicht den Fehler an einem Beispiel. Die Implementierung der Methode `start` der Basisklasse `SomeSuperClass` ruft die Methode `important_task` auf.

Die überschriebene Methode `start` in der Subklasse `SomeSubClass` ruft weder die vererbte Methode `important_task` noch die Implementierung der Basisklasse auf.

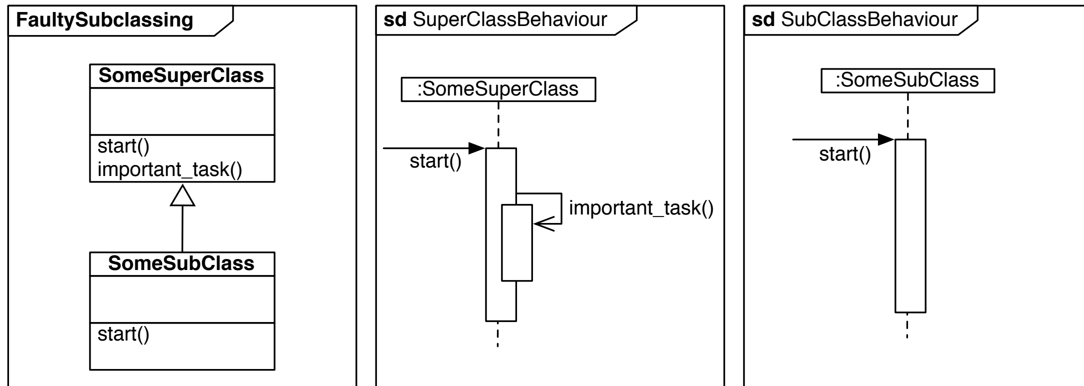

Abbildung 5.4: Klassen- und Sequenzdiagramm mit Überschreiben der `start` Methode beim Vererben und fehlendem Aufruf der `important_task` Methode

Diese Schwachstelle ist eine Spezialisierung der Verletzung des liskovschen Substitutionsprinzips. Die Subklasse erfüllt zwar die Schnittstellen, aber nicht den Vertrag der Basisklasse und kann nicht in allen Fällen als Substitut verwendet werden. Im Weiteren können diese Fehler auch in anderen Programmiersprachen auftreten und sind nicht direkt mit der dynamischen Typisierung in Verbindung zu bringen. Jedoch ist diese Schwachstelle mehrfach aufgetreten und wurde deswegen hier aufgeführt.

5.3.5 Mehrfachvererbung durch Verwendung von Mixins

Mixins sind ein viel genutztes Verfahren, um Querschnittsfunktionalitäten wie z.B. Authentifizierung oder Lokalisierung, in mehreren Klassen zur Verfügung zu stellen (→ Abschnitt 3.4). Ein Mixin vereint zusammengehörige Funktionalitäten und erlaubt es, diese in beliebig viele Klassen zu inkludieren, ohne dabei auf Mehrfachvererbung durch Klassen zurückgreifen zu müssen. In Ruby werden Mixins mit Hilfe von Modulen realisiert. Hierin können Methoden und Konstanten definiert werden, die beim Inkludieren des Moduls in eine Klasse dort zur Verfügung stehen.

Fehler können in Zusammenhang mit Mixins entstehen, wenn mehrere Module in eine Klasse inkludiert werden und die Module gleichnamige Methoden aufweisen, so dass diese unabsichtlich überschrieben werden. Weiterhin ist das Geheimnisprinzip innerhalb der Vererbungshierarchie aufgehoben, was auch für beteiligte Module gilt. Die Methoden von Modulen haben uneingeschränkten Zugriff auf alle Variablen und Methoden der Klasse, in die die Module inkludiert werden. Durch diesen Umstand können Fehler zwischen

vererbten und überschriebenen Methoden entstehen [Winter, 1997]. Beim Zugriff auf Instanzvariablen der Klasse durch inkludierte Module können unerwartete Seiteneffekte auftreten, die nur in bestimmten Kontexten auftreten. In [Binder, 1999, S. 75-76] werden weitere Fehlerquellen, die durch Mehrfachvererbung entstehen, aufgelistet.

5.3.6 Verletzung des liskovschen Substitutionsprinzips

Bei der Fehleranalyse wurden Fehler festgestellt, die mit Vererbung in Zusammenhang stehen. Sie entstammen der Fehlerkategorie *Vertragsverletzung*. Bedingt durch die Überschreibung von Methoden der Basisklasse entstehen Fehler beim Zusammenspiel einer vererbten und einer überschriebenen Methode [Binder, 1999, S. 104]. Hierbei wird das liskovsche Substitutionsprinzip verletzt [Liskov u. Wing, 1994], wonach Objekte einer Klasse oder eines Typs durch jeweilige Subklassen oder Subtypen ersetzbar sein müssen.

Einerseits wird diese Schwachstelle durch die späte Typprüfung begünstigt, da der Interpreter bei überschriebenen Methoden weder Parameteranzahl und -typ noch die Rückgabewerte auf Übereinstimmung mit der Methode der Basisklasse überprüft. Andererseits ist oft eine reine Typprüfung nicht ausreichend, da gegebenenfalls Inkompatibilitäten zur Basisklasse über die reine Typübereinstimmung hinaus entstehen können [Binder, 1999, S. 79-80]. Weiterhin ist eine Typprüfung anhand der Klasse oder der implementierten Schnittstellen eines Objekts nicht sinnvoll, da der Typ eines Objekts in einer dynamisch typisierten Sprache nicht von der Klasse, sondern nur von den Nachrichten, die es beantworten kann, abhängt (→ Abschnitt 3.2.3).

5.3.7 Monkey Patching

Monkey Patching bezeichnet umgangssprachlich das dynamische Überschreiben oder Hinzufügen von Funktionalität, beispielsweise zu Objekten der Programmiersprache oder der Standardbibliothek. Es wird hauptsächlich eingesetzt, um die Funktionalität von Objekten von nicht eigens verwalteten Bibliotheken zu erweitern oder Fehler darin zu beheben. Das Rahmenwerk Ruby on Rails weist mit ActiveSupport eine eigens zu diesem Zweck hergestellte Bibliothek auf, die zu großen Teilen daraus besteht, Basisobjekte von Ruby wie `String`, `Integer` oder `Date` um Methoden zu erweitern, die im Kontext der Webseitenerstellung von Nutzen sind. Wie bereits in Abschnitt 3.3 beschrieben, ist das einer der Vorteile, der aufgrund dynamischer Typisierung leicht eingesetzt werden kann.

Neben dem Hinzufügen neuer Funktionalität ist es ebenso möglich, vorhandene Methoden zu verändern. Dabei sind die Modifikationen nicht begrenzt, sondern wirken sich auf das

gesamte Programm aus. In manchen Fällen führt dies zu schwer testbaren Fehlern, da ein Monkey Patch schwer lokalisierbar und damit schwer kontrollierbar ist.

5.3.8 Dynamische Auflösung von Konstanten und Methoden

Bei dieser Art von Fehlern wird der Name einer Konstante dynamisch mittels einer Zeichenkette erstellt. Der Name basiert z.B. auf den Parametern der Methode, die den Namen erstellt. Die Ruby eigene Klassenmethode `const_get`, die über das Modul `Kernel` an jedes Objekt vererbt wird, gibt für eine Zeichenkette eine Konstante zurück, falls diese existiert. Andernfalls wird eine `NameError` Ausnahme geworfen. Bei der Suche der Konstante werden vorhandene Namensräume beachtet. Als Ausgangspunkt der Suche muss also der korrekte Namensraum gewählt werden.

Dementsprechend werden auch Methodennamen dynamisch als Zeichenketten erstellt und als Nachrichten an Objekte gesendet. Falls das Objekt keine Methode zur Nachricht implementiert hat, wird eine `NoMethodError` Ausnahme ausgelöst.

Ein Beispiel für solch einen Fehler bei der dynamischen Auflösung einer Konstante aus Spree wird in Listing 5.5 dargestellt. In Zeile 4 wird der Name der Konstanten, in diesem Fall ein Klassenname, in einer Zeichenkette erstellt. In der nächsten Zeile wird überprüft, ob eine Konstante mit dem entsprechenden Namen existiert. Da jedoch `Object` als Namensraum für die Auflösung gewählt wird, kann die Konstante nicht gefunden werden.

Listing 5.5: Falscher Ausgangspunkt für die Auflösung einer Konstante

```
1  Spree::Admin::BaseController.class_eval do
2    protected
3      def model_class
4        const_name = "Spree::#{controller_name.classify}"
5        if Object.const_defined?(const_name)
6          return const_name.constantize
7        end
8        nil
9      end
10 end
```

5.3.9 Dynamische Generierung von Methoden

Die dynamische Generierung von Methoden erlaubt vor allem die Erstellung von domänenspezifischen Sprachen, ist aber auch bei vielen anderen Programmiertätigkeiten hilfreich, bei denen es um die Erstellung von vielen ähnlichen Methoden geht. Die Generierung basiert in den meisten Fällen auf programmexternen Daten, wie z.B. den Spalten einer Datenbank-Tabelle oder Konfigurationsdateien. Eine häufige Fehlerquelle ist dabei

die Synchronisierung der Erstellung und dem Aufruf der Methoden. Es kann zu Fehlern kommen, wenn der Methodenaufruf erfolgt bevor die Methode erstellt worden ist.

Neben der Generierung von Methoden zum Programmstart können Objekte auch Nachrichten beantworten, zu denen keine Methoden existieren. Falls zu einer empfangenen Nachricht keine Methode in der Vererbungshierarchie eines Objekts vorliegt, wird die Methode `method_missing` aufgerufen. Die von Ruby gestellte Implementierung von `method_missing` löst die `NoMethodError` Ausnahme aus. Durch Überschreiben der Standardimplementierung kann die erhaltene Nachricht an ein anderes Objekt weitergeleitet oder durch den Aufruf einer anderen Methode beantwortet werden[10]. Im Regelfall wird anhand des Nachrichtennamens entschieden, ob eine Verarbeitung erfolgen soll oder die Standardimplementierung aufgerufen wird. Ein typischer Anwendungsfall ist der Zugriff auf Werte eines Datenobjekts, so dass nicht für jedes Attribut eine Zugriffsmethode erstellt werden muss.

Durch den Einsatz der `method_missing` Funktion können mehrere Probleme entstehen. Soll die Nachricht nicht vom Objekt beantwortet werden, muss die Standardimplementierung von `method_missing` aufgerufen werden. Fehlt dieser Aufruf, kann es zu Endlosschleifen kommen. Auch kann es zur Annahme von nicht beabsichtigten Nachrichten kommen, die dadurch „verschluckt" werden. Wird bei einer Komponente diese Funktionalität eingesetzt, sinkt die Lesbarkeit und die Fehlersuche wird zusätzlich erschwert.

Ähnlich dazu können auch Konstanten, also z.B. Klassen oder Module, dynamisch erstellt werden. Hierzu bietet Ruby die Methode `const_missing` an, die aufgerufen wird, sobald ein Objekt die aufgerufene Referenz nicht auflösen kann. Obwohl in Fehleranalyse keine Fehler in Zusammenhang mit `const_missing` vorgekommen sind, kann die Verwendung davon die gleichen Auswirkungen wie `method_missing` auslösen.

[10]Diese Eigenschaft von Ruby ist einer der Gründe, warum der Typ eines Objekts nur durch sein Verhalten bestimmt werden kann.

6 Identifikation der Schwachstellen

Das vorangegangene Fehlermodell beschreibt die Schwachstellen objektorientierter, dynamisch typisierter Programme. Auf Basis der dabei gewonnenen Erkenntnisse wird der Einfluss dynamischer Typisierung auf die analytische Qualitätssicherung diskutiert. Dabei werden Methoden und Techniken aufgezeigt, um die durch die dynamische Typisierung resultierenden Nachteile zu kompensieren. Anschließend werden die Schwachstellen formal beschrieben und Methoden zur Lokalisierung und Identifikation aufgezeigt.

6.1 Einfluss dynamischer Typisierung auf die analytische Qualitätssicherung

Wie in Kapitel 2.2 beschrieben bedient sich die analytische Qualitätssicherung statischer und dynamischer Tests. Dementsprechend werden nun die Auswirkungen dynamischer Typisierung auf beide Testformen diskutiert.

6.1.1 Statische Tests

Wie schon der Name andeutet, ist eine statische Analyse von dynamisch typisierten Programmen nur bedingt möglich, da die Typen der Objekte erst zur Laufzeit ermittelt werden. Aus Sicht der statischen Testbarkeit ist damit die Beobachtbarkeit dynamisch typisierter Programme geringer als die von statisch typisierten Programmen. Dynamischen Sprachen fehlt die notwendige Formalisierung. Die Identifikation bestimmter Schwachstellen, wie z.B. **falsche Rückgabewerte von Methoden** oder **fehlerhafte Parameter** wird dadurch erschwert.

Die Fehleranalyse hat jedoch gezeigt, dass viele Fehler nicht durch statische Typisierung hätten verhindert können und somit auch die Typangaben für eine statische Analyse nur eine begrenzte Aussagekraft haben. Ebenso gibt es Schwachstellen, die zwar auf dynamische Typisierung zurück zu führen sind, aber mittels statischer Analyse erkannt werden können.

Neben der Analyse, basierend auf Typangaben im Quelltext, gibt es weitere Methoden zur statischen Informationsgewinnung. In Kapitel 2.2.2 wurde bereits die Datenflussanalyse vorgestellt, die auch in dynamisch typisierten Programmiersprachen durchgeführt werden kann. In [Madsen u. a., 2007] beschreiben die Autoren die statische Inferenz von Typen in Ruby mit Hilfe des kartesischen Produktalgorithmus. Hierbei wird aufgrund von Variablenzuweisungen und Operationen auf den Variablen deren Typ bestimmt. Auch für andere dynamisch typisierte Programmiersprachen existieren ähnliche Ansätze, wie z.B. für Javascript von [Jensen u. a., 2009].

Eine teilweise automatische Typinferenz wird von [Furr u. a., 2009] beschreiben. Die Autoren stellen eine Erweiterung zu Ruby vor, die Typen mit Hilfe von Typinferenz ableitet und dem Entwickler Möglichkeiten zur Annotation von nicht inferierbaren Variablen und Rückgabewerten von Methoden bietet. In [Plevyak u. Chien, 1994] wird eine Methode zur Typinferenz basierend auf Einschränkungen beschrieben: „...incremental constraint-based type inference which produces precise concrete type information for a much larger class of programs at lower cost.“

Metaprogrammierung

Durch Metaprogrammierung erzeugte oder veränderte Komponenten können zwangsläufig nur sehr eingeschränkt oder gar nicht durch statische Analysen erfasst werden. [Holkner u. Harland, 2009] untersuchten den Einsatz von Metaprogrammierung und stellten fest, dass die meisten Funktionen zur dynamischen Anpassung und Änderung von Komponenten zur Startzeit des Programms statt finden. Diese Änderungen könnten somit durch die Initialisierung des Programms in die Analyse mit einfließen, ohne das Programm komplett, d.h. mit allen Verzweigungen, ausführen zu müssen.

In [Furr, 2009] wird eine Technik vorgestellt, bei der die dynamischen Tests einer Applikation verwendet werden, um mittels zusätzlicher Annotationen die Identifikation von Typen zu erlauben, so dass dadurch eine statische Analyse erfolgen kann. Hierbei würden auch per Metaprogrammierung hinzugefügte Bestandteile analysiert werden können.

Eine rein statische Analyse aller Bereiche einer Applikation, die Metaprogrammierung einsetzt, ist somit nicht ohne zusätzlichen Aufwand möglich.

6.1.2 Dynamische Tests

Dynamische Testverfahren führen das Testobjekt aus. Dadurch hat das Typsystem hierbei nur geringe Auswirkungen auf die Testbarkeit. Die geringere Formalisierung dyna-

misch typisierter Programme erlaubt in stärkerem Maße Sprachfunktionen wie Metaprogrammierung und Introspektion (→ Abschnitt 3.3), die wiederum Kontrollierbarkeit und Beobachtbarkeit vereinfachen können. Es ist damit sehr einfach, die Sichtbarkeit von Attributen und Methoden zu umgehen und so die, durch die Kapselung bedingt geringere Beobachtbarkeit, zu verbessern. Auch die Kontrollierbarkeit kann damit erhöht werden. Mit Metaprogrammierung ist es z.B. sehr einfach private Methoden aufzurufen, den Zustand von Instanzvariablen zu verändern oder die Funktionsweise von nicht auflösbaren Abhängigkeiten anzupassen. Dies erhöht indirekt die Testbarkeit durch Vermeidung von Komplexität, die zur Durchführung mancher Tests erbracht werden müsste.

Zum Beispiel erfordert das Testen zeitabhängiger Geschäftslogik das Ersetzen dieser Abhängigkeit (Zeit), durch ein im Test kontrollierten Platzhalter. Damit dieser Platzhalter injiziert werden kann, müsste in einer statisch typisierten Sprache Dependency Injection[1] verwendet werden. Bei dynamischer Typisierung kann die Abhängigkeit im Test mittels Metaprogrammierung zur Laufzeit durch den Platzhalter ersetzt werden, ohne dass dafür der Produktivcode angepasst werden muss. In den meisten Fällen ist jedoch die Anwendung solcher Techniken in Tests nicht sinnvoll. Vor allem bei Blackbox-Verfahren entstünde dadurch eine nicht erwünschte Abhängigkeit zwischen Testfällen und dem Testobjekt.

Ein weiterer Vorteil der dynamischen Typisierung ist gerade die geringere Abhängigkeit zwischen Testfällen und Testobjekt auf Komponenten- und Integrationstestebene. Tests verifizieren Komponenten im Optimalfall mittels ihrer öffentlichen Schnittstellen, d.h. Test und Komponenten sind abhängig von der gemeinsam genutzten Schnittstellendefinition. Änderungen daran müssen damit sowohl im Test als auch der zu testenden Komponente ausgeführt werden. Im Gegensatz zu dynamisch typisierten sind in statisch typisierten Sprachen neben den möglichen Operation und deren Parametern auch die Typen ein Teil der Schnittstellendefinition. Die Typen selbst sind also eine Abhängigkeit, die die Testbarkeit negativ beeinflussen.

Aus dem vorherigen Punkt ergibt sich auch die Tatsache, dass die Substitution von Stellvertreterobjekten, ein wichtiges Testwerkzeug zur Kontrolle von Abhängigkeiten, in dynamisch typisierten Sprachen einfacher umsetzbar ist. Es genügt die für den Testfall benötigten Methoden zu simulieren, ohne die gesamte Schnittstelle des ersetzten Objekts implementieren zu müssen.

In der Fehleranalyse wurde festgestellt, dass viele Fehlerwirkung durch das unerwartete Auftreten des Nullwerts ausgelöst wurden. Ebenso konnte eine hohe Anzahl an Fehlern

[1] Mit Hilfe von Dependency Injection können die Abhängigkeiten für das Konstruieren von Objekten außerhalb der Klassen dieser Objekte konfiguriert werden [URL:Fowler, b].

auf die Anforderungsermittlung und -spezifikation zurück geführt werden. Beides kann damit zusammenhängen, dass ein unspezifisches Fehlermodell bei der Entwicklung der Testfälle verwendet wird (→ Abschnitt 5.3) und die Testobjekte dadurch nur auf vorher spezifiziertes Verhalten hin getestet werden. Es wird also lediglich die Konformität zur vorher festgelegten Anforderungen oder Spezifikationen getestet. Daraus ergibt sich die Empfehlung, zusätzlich ein spezifisches Fehlermodell, wie es in Kapitel 5.3 beschrieben wird, einzusetzen. Daraus ergibt sich z.B. die Anforderung, bei der Bildung von Äquivalenzklassen[2] für Eingabewerte von Methoden, den Nullwert zu als Testwert zu inkludieren.

6.2 Identifikation der Schwachstellen

Nachdem die allgemeinen Zusammenhänge zwischen dynamischen und statischen Testverfahren in Bezug auf die Typisierung betrachtet wurden, sollen im Folgenden die in Kapitel 5.3 ermittelten Schwachstellen formal beschrieben werden, was wiederum als Basis für den Prototyp verwendet wird.

6.2.1 Fehlerhafte Objektreferenzen

Die am häufigsten aufgetretene Schwachstelle bezieht sich auf fehlerhafte Objektreferenzen. Variablen die auf Objekte verweisen sollten, referenzieren stattdessen den Nullwert. Häufig resultiert dieser Zustand aus der fehlerhaften Initialisierung von Objekten. Attribute, die zum fehlerfreien Funktionieren des Objekts notwendig sind, werden nicht initialisiert.

Eine Möglichkeit zur Aufdeckung dieser Schwachstelle ist die Datenflussanalyse, die in Kapitel 2.2.2 beschrieben wird. Dabei sollte vor allem der Konstruktor einer Klasse auf die Initialisierung aller Instanzvariablen mit validen Werten überprüft werden. Weiterhin ist vor allem Design By Contract, wie es in Kapitel 8.1.1 beschrieben wird, sehr effizient bei der Aufdeckung von Fehlern dieser Schwachstelle.

6.2.2 Fehlerhafte Parameter

Die Charakteristik dieser Schwachstelle liegt in der Übergabe falscher Parameter an eine Methode. Dazu zählen Parameter vom falschen Typ, dem Nullwert als Parameter, falsche

[2] Eine Äquivalenzklasse definiert den „*Teil des Wertebereichs von Ein- oder Ausgaben, in dem ein gleichartiges Verhalten der Komponente oder des Systems angenommen wird, basierend auf der zugrunde liegenden Spezifikation*“ [Spillner u. Linz, 2012, S. 245].

Anzahl sowie falsch benannte Schlüssel in assoziativen Datenfeldern als Parameter (→ Abschnitt 5.3.2). Durch statische Analysen lässt sich diese Schwachstelle nicht ohne Weiteres bestimmen, da bei dynamisch typisierten Sprachen der Typ einer Variable bzw. des Wertes der Variable erst zur Laufzeit fest steht.

Zur Verifikation der Übereinstimmung der verwendeten Typen kann die oben erläuterte Typinferenz herangezogen werden. Der Nullwert als Parameter kann nur sehr eingeschränkt durch statische Analysen erkannt werden. Hinweise darauf kann die Datenflussanalyse geben. Ebenso ist bei der Übergabe von Parametern innerhalb eines assoziativen Datenfeldes eine statische Aufdeckung von Fehlern kaum möglich und auch meist nicht sinnvoll, da diese Parameter meist optional sind.

Die Übereinstimmung der Anzahl von Parametern einer Methode ist in vielen Fällen leicht statisch überprüfbar: Die Signatur der Methode muss mit der Anzahl der Parameter, mit denen die Methode aufgerufen wird, verglichen werden. Im folgenden wird dies mit Hilfe der Unified Modeling Language (UML) und der Object Constraint Language (OCL) formal beschrieben.

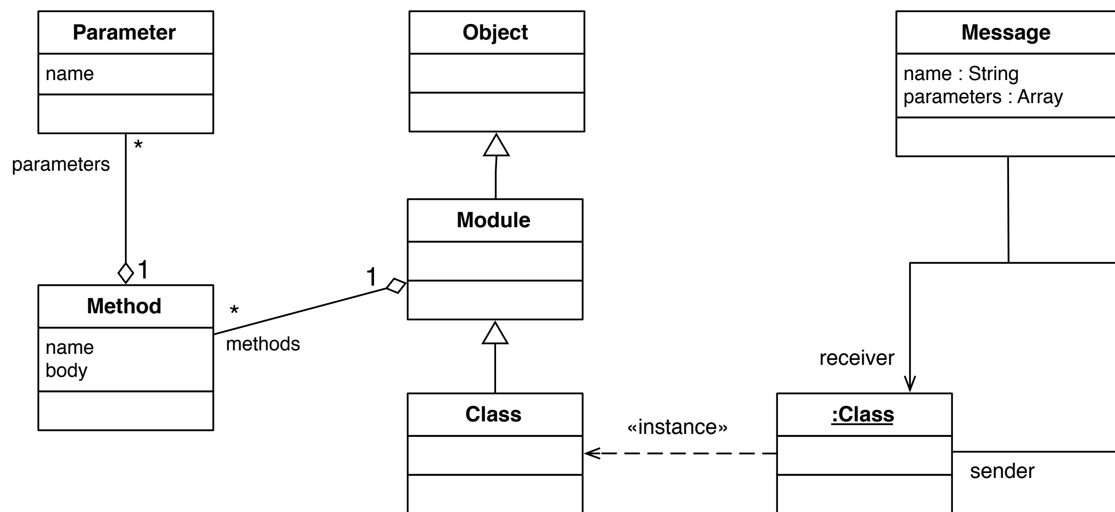

Abbildung 6.1: Auszug aus dem Meta-Modell der Programmiersprache Ruby

Die Abbildung 6.1 zeigt einen Ausschnitt des Meta Modells der Programmiersprache Ruby. Jede Klasse in Ruby ist eine Instanz der Klasse `Class`. `Class` ist eine Subklasse der Klasse `Module`, die wiederum von der Klasse `Object` erbt. Jede Klasse kann beliebig viele Methoden besitzen. Methoden bestehen aus einem Namen, möglichen Parameter und einem Methodenkörper. Von Klassen gebildete Instanzen können Nachrichten an sich selbst oder andere Objekte senden. Eine Nachricht besteht aus einem Namen und mitgeführten Parametern bzw. Objektreferenzen. Eine Nachricht an ein Objekt ist nicht

valide, wenn der Name der Methode mit dem Namen der Nachricht übereinstimmt, die Anzahl der Parameter jedoch unterschiedlich ist.

Wird die folgende, in OCL verfasste Invariante verletzt, erfolgt ein Nachrichtenaufruf mit der falschen Anzahl an Parametern.

```
context Message inv:
    self.receiver.methods -> forAll( m |
      m.name = self.name and
      m.parameters.size = self.parameters.size
    )
```

Diese Darstellung ist stark vereinfacht, da Ruby eine Vielzahl an weiteren Möglichkeiten für die Definition von Parametern einer Methode unterstützt. Es können Standardwerte für Parameter festgelegt werden. Ebenso erlaubt Ruby eine variable Anzahl an Parametern, wodurch sich die Erkennung dieser Schwachstelle in der Praxis als etwas komplexer herausstellt. Diese Gegebenheiten können aber immer noch statisch erkannt werden, sollen jedoch hier nicht weiter diskutiert werden.

Auf Seite der dynamischen Testverfahren bietet sich Design By Contract als sehr effiziente Methode zur Aufdeckung der Fehler dieser Schwachstelle an (→ Abschnitt 8.1.1).

6.2.3 Falsche Rückgabewerte von Methoden

Durch die fehlende Definition von möglichen Rückgabewerten einer Methode ist eine statische Überprüfung auf die Rückgabe von falschen Parametern nur sehr eingeschränkt möglich, wodurch auch eine formale Definition dieser Schwachstelle nicht ohne Weiteres gegeben werden kann.

Die Betrachtung der Verwendung des Rückgabewerts einer Methode kann Hinweise auf das Vorliegen dieser Schwachstelle geben. Wird beispielsweise auf dem Rückgabewert, ohne vorherige Überprüfung auf den Nullwert, direkt eine Methode aufgerufen, so sollte die Methode nie den Nullwert zurück geben. Auf diese Weise könnte bereits eine Vielzahl von Fehlern identifiziert werden. Die Datenflussanalyse kann hierbei mögliche Anomalien aufdecken.

Dieses Verfahren ist jedoch sehr anfällig für die Identifizierung von falschen positiven Testergebnissen, so dass anschließend eine aufwändige, manuelle Inspektion notwendig wird. Die Festlegung von validen Nachbedingungen, als Teil von dynamischen, eingebauten Tests, wäre hierbei sinnvoller.

6.2.4 Fehlender Aufruf einer Methode der Basisklasse

Vererbungshierarchien können zuverlässig mittels statischer Analyse ermittelt werden, solange die Hierarchie nicht zur Laufzeit durch Metaprogrammierung erweitert wird. Die Untersuchung erfolgt hierbei durch die Umwandlung des Quelltexts in eine logische Baumstruktur, wie dem abstrakten Syntaxbaum. Dieser muss dabei den gesamten Code abdecken, so dass alle Abhängigkeiten zwischen den Klassen und Modulen abgebildet werden. Anhand dieser Repräsentation kann schließlich erkannt werden, welche Methoden in Subklassen überschrieben wurden.

Der Aufruf der Methode der Basisklasse wird in Ruby durch das Schlüsselwort `super` realisiert. Es muss also für jede überschriebene Methode der Basisklasse überprüft werden, ob in der Subklasse die Implementierung der Basisklasse mittels `super` aufgerufen wird. Das Fehlen des Aufrufs ist somit ein notwendiger Hinweis für diese Schwachstelle. Der Aufruf der Methode oder des Konstruktors der Basisklasse ist jedoch funktional nicht immer erforderlich, wodurch ein hinreichender Hinweise für das Bestehen dieser Schwachstelle nicht geliefert werden kann.

6.2.5 Mehrfachvererbung durch Verwendung von Mixins

Wie in Kapitel 5.3.5 beschrieben, kann der Einsatz von Modulen zur Mehrfachvererbung zu einer Vielzahl von Fehlerquellen führen. Im Folgenden soll die Schwachstelle beschrieben, bei der inkludierte Methoden vorhandene Methoden überschreiben. Dies ist in so fern sinnvoll, als dass in Ruby Module zur Realisierung von Querschnittsfunktionalität verwendet werden. Dabei erfolgt die Vererbung von Methoden aus den Modulen zum Zwecke der der Code-Wiederverwendung und nicht zur Spezialisierung. Somit würde das Überschreiben einer Methode auf einen Fehler hindeuten. Da Module auch andere Module inkludieren können, kann dabei die gleich Schwachstelle auftreten. Der Einfachheit halber werden in diesem Abschnitt aber nur Klassen als Empfänger von Modulen behandelt.

Diese Schwachstelle kann durch Untersuchung des abstrakten Syntaxbaums statisch gefunden werden. Abbildung 6.2 stellt wieder einen Ausschnitt aus dem Meta Modell von Ruby dar. Jedes `Module` kann beliebig viele Methoden definieren und andere Module inkludieren, was in der Abbildung durch den Stereotyp «module»modelliert wird. Die Methode `included_modules` gibt alle inkludierten Module einer Klasse zurück. Die Methode `instance_methods` gibt alle Methoden einer Klasse oder eines Moduls zurück.

Die Schwachstelle liegt vor, falls die folgende Invariante verletzt wird:
context Class **inv**:

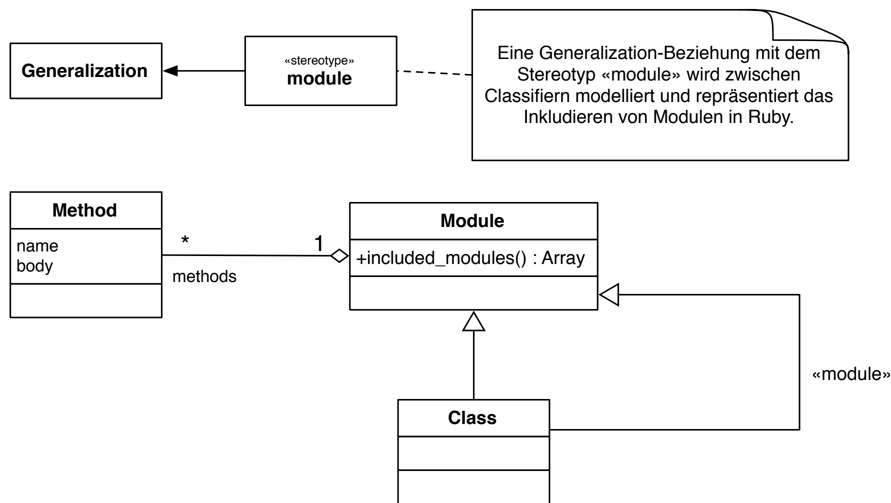

Abbildung 6.2: Auszug aus dem Meta-Modell der Programmiersprache Ruby mit relevanten Methoden von `Module`

```
self.included_modules -> select( module |
  module.instance_methods.select( meth |
    self.instance_methods -> exists ( self_method | self_method = meth )
  )
) -> isEmpty()
```

6.2.6 Verletzung des liskovschen Substitutionsprinzips

Das liskovsche Substitutionsprinzip ist in der Literatur ausführlich besprochen worden [Binder, 1999, S. 79-80], [Liskov u. Wing, 1994] und soll, in Bezug auf klassische Subtypen, an dieser Stelle nicht weiter diskutiert werden.

Jedoch ist hervorzuheben, dass bei Verwendung von Duck Typing (→ Abschnitt 3.2.3) der Typ eines Objekts vor allem durch seine Fähigkeiten bestimmt wird. Erfolgt nun ein Ersatz des Objekts durch ein anderes gleichwertiges Objekt, muss das neue Objekt ebenfalls das liskovsche Substitutionsprinzip einhalten. Bei der dynamischen Typisierung kommt es somit auf substituierbares Verhalten von Objekten an.

Binder beschreibt mit *Polymorphic Message Test* in [Binder, 1999, S. 438ff.] ein dynamisches Testverfahren für Methoden, die Nachrichten an Objekte einer Klassenhierarchie senden. Ebenso werden Testmuster für Klassenhierarchien selbst beschrieben [Binder, 1999, S. 513ff.].

Diese Testverfahren sind jedoch beim Einsatz von Duck Typing nur bedingt geeignet, da sie von einer Subtyp-Beziehung zwischen den beteiligten Klassen ausgehen, die bei der dynamischen Typisierung nicht immer gegeben ist. Ebenso kann, die für solche Testverfahren notwendige Beziehung zwischen Absenderobjekt und der betroffenen Klassenhierarchie, ohne Typangaben nicht ermittelt werden. Hier sei wieder auf Design By Contract (→ Abschnitt 8.1.1) als effiziente Methode zur Aufdeckung von Fehlern dieser Schwachstelle verwiesen. Auch Binder rät in einem ähnlichen Szenario zum Einsatz von zusicherungsbasierten Testverfahren, vgl. [Binder, 1999, S. 443].

6.2.7 Monkey Patching

Das Überschreiben einer bereits definierten Methode ist in Ruby sehr einfach durchführbar. Dazu erfolgt eine reguläre Klassendefinition mit der entsprechenden Methodendefinition. Falls eine gleichnamige Klasse bereits existiert wird die Methode zu dieser Klasse hinzugefügt. Falls in dieser Klasse eine gleichnamige Methode existiert, wird diese überschrieben. Listing 6.1 verdeutlicht dies durch neu definieren der Methode `to_s` auf der `Integer` Klasse.

Listing 6.1: Monkey Patching einer Methode

```
1 class Integer
2   def to_s
3     "zweiundvierzig"
4   end
5 end
```

Eine weitere Methode des Monkey Patching besteht im Aliasing. Hierbei wird die alte, zu überschreibende Methode unter einem anderen Namen referenziert, so dass sie durch die neue definierte Methode immer noch aufgerufen werden kann. Beide Verfahren können zu Fehlern führen und sind für den Aufrufer von Methoden nicht zu erkennen.

Da die Klassen und Methoden der Standardbibliothek von Ruby bekannt sind, kann ein Monkey Patch dafür, durch Überprüfung auf das Öffnen dieser Klassen, erkannt werden. Das Monkey Patching von allen anderen Klassen kann jedoch nicht statisch erkannt werden.

6.2.8 Dynamische Auflösung von Konstanten und Methoden

Bei dieser Schwachstelle werden Namen von Konstanten oder Nachrichten dynamisch in einer Zeichenkette erstellt und daraufhin die Konstante aufgelöst oder die Nachricht an ein Objekt gesendet.

Zur Auflösung einer Konstante muss die Ruby eigene Methode `const_get` verwendet werden. Da beide untersuchten Softwaresystem auf dem Rahmenwerk Rails aufbauen, empfiehlt es sich auch die Methode `String#constantize` ebenfalls zu beachten. Diese Methode gibt die Konstante, die dem Inhalt der Zeichenkette entspricht zurück oder löst die `NameError` Ausnahme aus.

Diese Schwachstelle kann damit mittels eines regulären Ausdrucks formuliert werden:

```
weakpoint := [ const_get | constantize ]
```

Zum Versenden der Nachricht wird die Methode `send` benutzt, die auf jedem Objekt aufgerufen werden kann und den Namen der Nachricht als Parameter akzeptiert. Die verwandte Methode `public_send` ruft nur öffentlich sichtbare Methoden auf.

Der entsprechende reguläre Ausdrucks dazu lautet wie folgt:

```
weakpoint := [ send | public_send ]
```

6.2.9 Dynamische Generierung von Methoden

Die dynamische Generierung von Methoden findet entweder zur Initialisierung oder während der Ausführung des Programms statt. In beiden Fällen werden dazu von Ruby zur Verfügung gestellte Methoden zur Metaprogrammierung verwendet. Dazu zählen `define_method`, `eval`, `instance_eval`, `class_eval` und `module_eval`. Zusätlich können zur Laufzeit die Methoden `method_missing` und `const_missing` benutzt werden, um dynamisch auf Nachrichten zu antworten.

Diese Schwachstelle kann damit mittels eines regulären Ausdrucks formal beschrieben werden:

```
weakpoint := [ define_method | eval | instance_eval | class_eval |
               module_eval | method_missing | const_missing ]
```

7 Prototypische Implementierung

Zur Evaluation der vorgestellten Methoden zur statischen Identifikation von Schwachstellen wird das prototypische Werkzeug *Inspector Juve*[1] implementiert. Als Eingabe dient der Quelltext der beschriebenen Untersuchungsobjekte. Mittels statischer Analyse werden Schwachstellen lokalisiert und mit Verweisen auf die entsprechende Stelle im Quelltext ausgegeben.

7.1 Funktionalität des Prototyps

Syntaktische Analyse, wie die Identifikation der meisten Schwachstellen bezogen auf Metaprogrammierung, ist mit regulären Ausdrücken relativ einfach umzusetzen und ist derweil nicht Teil des Prototyps. Die größere Schwierigkeit liegt in der Erstellung und Auswertung des abstrakten Syntaxbaumes sowie der statischen Inferenz von Typen. Dadurch werden Aussagen zu komplexeren Schwachstellen möglich.

Zur Zeit kann Inspector Juve die Schwachstellen **Fehlender Aufruf einer Methode der Basisklasse**, **Mehrfachvererbung durch Verwendung von Mixins** sowie **Dynamische Generierung von Methoden** mittels Syntaxbaumanalyse identifizieren. Bei der Schwachstelle Mehrfachvererbung durch Verwendung von Mixins wird auf den Zugriff von Instanzvariablen durch die Methoden des Moduls hin untersucht. Die Inferenz von Typen ist als nächste Ausbaustufe geplant.

7.2 Architektur

Der Prototyp basiert auf YARD[2], einem Werkzeug zur Dokumentation von Ruby Quelltext. YARD ist in die Teilprozesse Parsen, Persistieren des Codes in einer Baumstruktur und Generierung der Dokumentation aufgeteilt. Jeder dieser Einzelschritte kann unabhän-

[1] Inspektor Juve ist ein Charakter aus Fantômas, einer Serie von Kriminalromanen der Autoren Pierre Souvestre und Marcel Allain.

[2] `http://yardoc.org`

gig angesteuert werden und die jeweiligen Ergebnisse sind auch außerhalb des Werkzeugs verfügbar. Der Schritt Parsen erstellt einen abstrakten Syntaxbaum, die sogenannte `Registry`, welche r -> bezieht sich doch auf den Syntaxbaum, oder? alle gefundenen `CodeObjects`, also Klassen, Module, Methoden usw. beherbergt. Mit Hilfe der Registry ist es möglich, gezielte Informationen über die -> Oder wenn "die" wegbleibt, dann Code-Konstellationen Code-Konstellation zu sammeln, wovon der Prototyp zwecks Identifikation der Schwachstellen Gebrauch macht.

Inspector Juve erstellt bei Bedarf die Registry mit Hilfe von YARD und sucht daraufhin nach den Schwachstellen.

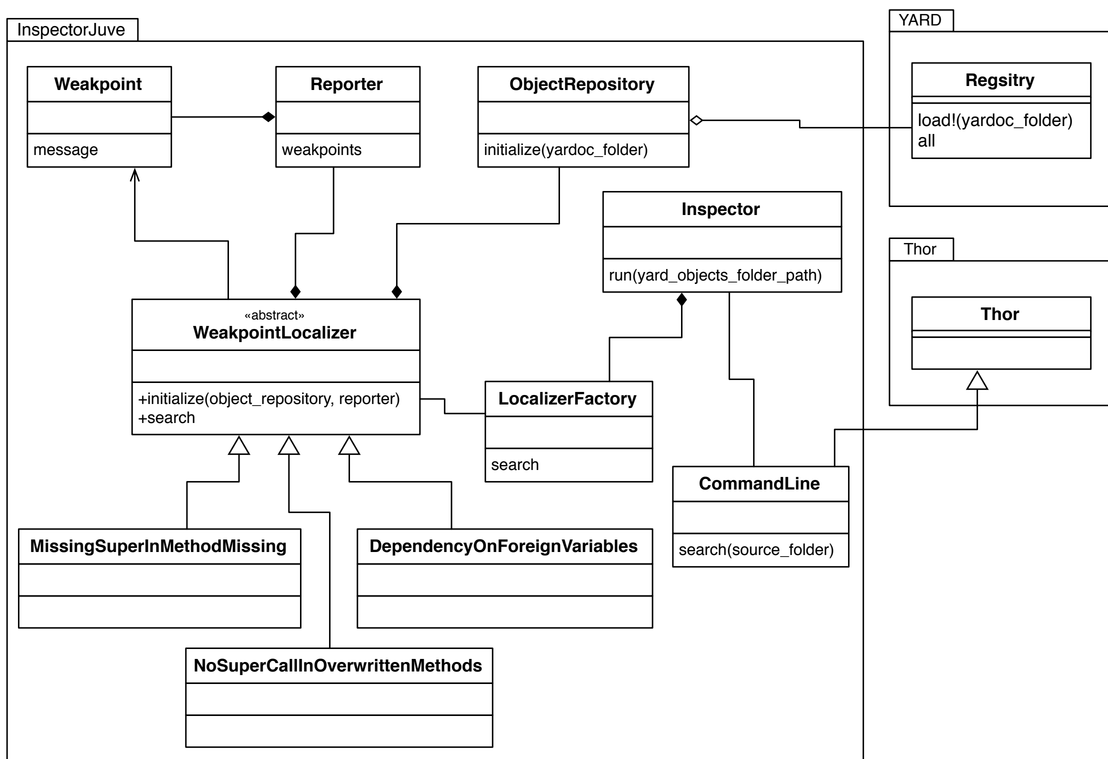

Abbildung 7.1: Klassendiagram des Prototyps

Der Prototyp ist in Ruby implementiert, dessen Aufbau in Abbildung 7.1 dargestellt wird.

7.3 Evaluierungsmethode

Zur Validierung der Identifikation der beschriebenen Schwachstellen wird der Quelltext von simfy.de und Spree mit dem Werkzeug analysiert.

Frage: mit welchem Werkzeug?

Die damit aufgezeigten Schwachstellen werden daraufhin auf vorhandene Fehler untersucht. Ebenso wird betrachtet, ob die bereits bekannten Fehler auf die identifizierten Schwachstellen zurückzuführen sind.

7.4 Evaluation der Analyseergebnisse

Bevor die Ergebnisse der Analyse durch Inspector Juve im Detail besprochen werden, soll hier eine Zusammenfassung gegeben werden. Durch die Analyse konnte im simfy.de Code ein Fehler gefunden werden. Dieser wurde stark maskiert und hätte, ohne die erfolgte Analyse, nur schwer erkannt werden können, da die Fehlerwirkung nur in Kombination mit anderen Faktoren auftritt. Dies unterstreicht den Wert eines spezifischen Fehlermodells und der statische n Suche nach typischen Fehlerquellen.

In der Masse lieferte die Analyse zwar viele Schwachstellen, jedoch waren auch viele falsch e positive Treffer darunter.

Frage: Besser fände ich: In der Masse lieferte die Analyse zwar viele Schwachstellen, es waren jedoch auch viele falsche positive Treffer darunter.

Die sehr flexible Syntax von Ruby und die vielfältigen Möglichkeiten zur Metaprogrammierung, die auch genutzt werden, erschweren die verlässliche Identifikation.

Frage: besser fänd ich: Die sehr flexible Syntax von Ruby und die vielfältigen Möglichkeiten zur Metaprogrammierung, welche bei simfy.de auch genutzt werden, erschweren die verlässliche Identifikation.

Die meisten falsch positiv erkannten Schwachstellen können durch Verbesserung des Erkennungsalgorithmus beseitigt werden, jedoch müsste sehr viel Aufwand in die Beachtung dieser Gegebenheiten einfließen.

Frage: besser fänd ich: ..., jedoch muss dafür sehr viel Aufwand in die Beachtung dieser Gegebenheiten einfließen.

7.4.1 simfy.de

Bei der Identifikation der Schwachstellen des simfy.de Quelltextes wurden folgende Ergebnisse erzielt:

- Drei Stellen, an denen die Methode `method_missing` die Implementierung der Basisklasse aufruft oder eine `NoMethodError` Ausnahme auslöst.
- Acht Stellen, an denen Module auf Instanzvariablen der Klasse, in die sie inkludiert wurden, zugreifen.
- 81 Stellen, an denen überschriebene Methoden die Implementierung der Basisklasse nicht aufrufen.

Der erste oben aufgelistete Punkt bezieht sich auf die Schwachstelle Dynamische Generierung von Methoden.

Frage: bisher hast du die Schwachstellen immer fett oder kursiv oder irgendwie hervorgehoben. Hier nicht mehr. Mach das einheitlich

Von den drei identifizierten Schwachstellen wurden zwei falsch erkannt, da diese die empfangenen Nachrichten mittels der Methode `send` an andere Objekte weiterleiten. Die dritte erkannte Stelle entspricht jedoch der Schwachstellendefinition und „verschluckt“ Nachrichten, die nicht durch die vorhandene Implementierung berücksichtigt werden. Dies konnte nicht mit einem bekannten Fehler in Verbindung gebracht werden, ist aber eine sehr fragile und fehleranfällige Konstruktion, die refaktoriert werden sollte.

Die identifizierten Zugriffe auf Instanzvariablen durch Methoden von inkludierten Modulen stellen zur Hälfte valide Schwachstellen dar. Die invaliden Treffer zeichnen sich dadurch aus, dass sie durch Metaprogrammierung in den Klassen Methoden erzeugen und somit nicht dieser Schwachstelle entsprechen. Drei der validen Schwachstellen verwenden zwar Instanzvariablen aus den Klassen in die sie inkludiert wurden, aber ohne den Zustand dieser zu verändern.

In einer der acht identifizierten Schwachstellen wurde ein Fehler gefunden. Die in Listing 7.1 dargestellte Methode `set_defaults` überschreibt die Instanzvariable `@user`, so dass die vorher getätigten Zustandsänderungen am User-Objekt verloren gehen[3]. In den meisten Fällen wird diese Methode nach der Verwendung der `@user` Variable aufgerufen, so dass dieser Fehlerzustand nicht zu einer Fehlerwirkung führt. Ein Aufrufer

[3] Der Inhalt der Variable ist ein Dekorierer-Objekt, das ein User-Objekt für den Emailversand erweitert. Die Methode des Moduls setzt die Variable wieder auf das ursprüngliche User-Objekt.

(PaymentMailer) verwendet die Variable jedoch nach dem Aufruf der fehlerhaften Methode, was dazu führt, dass Emails mit der falschen Sprache versendet werden. Die vorhandenen dynamischen Tests überprüfen zwar den mehrsprachigen Versand von Emails im allgemeinen, aber nicht für jede vorhandene Mail die verschickt wird, so dass der Fehler bisher nicht aufgefallen ist.

Listing 7.1: Fehler durch Überschreiben einer fremden Instanzvariable im simfy.de Code

```
module MailerBase
  private
  def set_defaults(user)
    @user = user
    set_template(user)
    # ...
  end
end

class PaymentMailer < ActionMailer::Base
  include MailerBase

  def first_payment_failed_trial(invoice, sent_on = Time.now)
    @invoice = invoice
    @user = decorate_for_mail(invoice.user)
    set_defaults(invoice.user)
    mail(
      :to      => @user.email,
      :subject => build_subject('mailer.payment_failed_trial', @user),
      :from    => SUPPORT_MAIL_ADDRESS,
      :bcc     => Simfy::Application.config.invoices_bcc_mail_address
    )
    # ...
  end
end
```

Die Analyse zur Schwachstelle Fehlender Aufruf einer Methode der Basisklasse lieferte unverhältnismäßig viele Treffer.

Frage: bisher hast du die Schwachstellen immer fett oder kursiv oder irgendwie hervorgehoben. Hier nicht mehr. Mach das einheitlich

Wie in Kapitel 5.3.4 erläutert, hängt es stark von der jeweiligen Implementierung ab, ob der Aufruf der Basisklassenimplementierung funktional notwendig ist. Nach der Betrachtung von mehreren berichteten Stellen, konnte in keiner davon ein Fehler gefunden werden. Eine statische Analyse dieser Schwachstelle kann somit nur begrenzt spezifische Informationen liefern und ist demnach nicht sinnvoll. Die Angabe von Vor- oder Nachbedingung für Methoden, die auch von den Implementierungen der Subklassen aufgerufen werden sollen, ist die effektivere Methode, Fehler dieser Art zu erkennen.

7.4.2 Spree

Bei der Identifikation der Schwachstellen des Spree Quelltextes wurden folgende Ergebnisse erzielt:

- Ein Modul, das auf fremde Instanzvariablen zugreift.
- Eine überschriebene Methode, die die Implementierung der Basisklasse nicht aufruft.

Da Spree nur ein knapp ein Drittel der Codezeilen im Vergleich zu simfy.de aufweist war anzunehmen, dass hierbei weniger Schwachstellen erkannt werden. Im Vergleich zu den Ergebnissen von simfy.de ist jedoch auffällig, dass nur eine Schwachstelle der Kategorie Fehlender Aufruf einer Methode der Basisklasse erkannt wurde.

Frage: bisher hast du die Schwachstellen immer fett oder kursiv oder irgendwie hervorgehoben. Hier nicht mehr. Mach das einheitlich

Eine mögliche Erklärung dafür ist, dass Spree als Basis für einen Webshop dient und deshalb keine umfangreichen Klassenhierarchien darin bestehen. Ebenso wäre es möglich, dass alle Subklassen die Basisklassenimplementierung aufrufen und somit nicht als Schwachstelle erkannt werden. Die einzige erkannte Stelle ist ein Stellvertreterobjekt in einem Testfall, welches keine Auswirkungen auf den Produktivcode haben kann.

Frage: "...haben kann." hört sich so unsicher han. Kanns du nicht schreiben: ...welches keine Auswirkungen auf den Produktivcode hat

Die zweite erkannte Schwachstelle betrifft den Zugriff auf Instanzvariablen durch Module. Hierbei wird die Variable nur dann überschrieben, wenn sie vorher den Nullwert oder `false` referenziert hat, siehe Listing 7.2. Durch dieses Vorgehen wird die Wahrscheinlichkeit für das Auftreten von Fehlern eingegrenzt, jedoch nicht ausgeschlossen.

Listing 7.2: Erkannte Schwachstelle im Modul Order aus dem Spree Quelltext

```
1 def associate_user
2   @order ||= current_order
3   if try_spree_current_user && @order
4     if @order.user.blank? || @order.email.blank?
5       @order.associate_user!(try_spree_current_user)
6     end
7   end
8   # ...
9 end
```

8 Verbesserung der Testbarkeit

Die vorherigen Kapitel beschreiben die typischen Schwachstellen dynamisch typisierter Programme und analytische Verfahren, um diese Fehlerquellen und deren Fehler aufzudecken. Im Folgenden werden Entwurfsmuster und -konzepte beschrieben, die die Testbarkeit bereits beim Entwurf und bei der Implementierung von Software verbessern bzw. etablieren. Ergänzend wird eine Übersicht von Refaktorierungen gegeben, um bereits vorhandene Schwachstellen zu beheben und Komponenten testbarer zu gestalten. Abschließend erfolgt eine Beurteilung der dynamischen Typisierung im Hinblick auf die konstruktive Qualitätssicherung.

8.1 Entwurfsmuster und -konzepte zur Steigerung der Testbarkeit

Entwurfsmuster nach [Gamma u. a., 1995] beschreiben erprobte, generische Lösungen für wiederkehrende Probleme der Softwaremodellierung. Daneben existieren etablierte Entwurfskonzepte für den strukturierten Aufbau von objektorientierter Software. Die folgenden Muster und Konzepte beziehen sich auf die vorangegangenen Schwachstellen, so dass die entsprechenden Fehler bereits im Vorfeld verhindert oder minimiert werden können. Oft werden durch die Anwendung dieser Muster und Konzepte mehrere Schwachstellen auf einmal adressiert. Da eine Mehrzahl der untersuchten Fehler in Zusammenhang mit dem Nullwert aufgetreten sind, wird der Fokus auf die Behebung dieser Problematik gelegt.

8.1.1 Design By Contract

Mit Hilfe von Design By Contract können vor allem die Schwachstellen *Fehlerhafte Parameter*, *Falsche Rückgabewerte von Methoden* und *Referenzierung des Nullwerts* verbessert werden. Die Fehler dieser Schwachstellen entstehen durch Verletzung von Vor- und Nachbedingungen oder durch Bestehen von invaliden Objektzuständen.

Design By Contract, auf deutsch Entwurf nach Kontrakten, ist ein von Betrand Meyer entwickeltes Konzept zur Steigerung der Verlässlichkeit von objektorientierten Programmen [Meyer, 1992]. Realisiert wird es durch direkt im Code implementierte Kontrakte bzw. Zusicherungen, die festlegen, wie Komponenten und einzelne Methoden miteinander interagieren dürfen und welche Zustände der beteiligten Objekte valide sind. Dieses Konzept wird in [Binder, 1999, S. 821] als eine Ausprägung von in Klassen eingebauten Tests vorgestellt. Da es aber vor allem eine Entwurfstechnik für objektorientierte Software darstellt, wird es innerhalb dieses Kapitels erläutert.

Diese formalen „Verträge“ definieren Bedingungen von Komponenten, die bei deren Verwendung erfüllt werden müssen. Gleichzeitig sichern die Komponenten ihrerseits durch Verträge bestimmte Funktionalitäten zu. Die formulierten Bedingungen gehen dabei über die Vorgabe von Typen bei einer statisch typisierten Sprache hinaus.

Die Zusicherungen sind als Untermenge von direkt im Code eingebauten Tests anzusehen und gelten als effektives Mittel zur Erhöhung der Testbarkeit:

> *„If used consistently and systematically, this [assertions] can provide verification that would be difficult, time-consuming, and expensive to achieve with externally applied and evaluated test cases.“* [Binder, 1999, S. 820]

Weiterhin hebt du Bousquet die Vorteile dieser Technik bei Modifikationen von Programmen hervor: „*Assertions descrease the test suite maintenance effort in case of modifications, because the test case oracle are embedded (and thus centralized) in code.*“ [du Bousquet, 2010]

Dadurch, dass Zusicherungen direkt als Teil der Geschäftslogik und nicht als externe Tests implementiert werden, wird der Code um ausführbare Kommentare erweitert. Die direkte Einbettung steigert gleichzeitig die Testbarkeit, da die Zusicherungen die größtmögliche Beobachtbarkeit und Kontrollierbarkeit ihrer direkten Umgebung erlangen. In [Binder, 1999, S. 810 ff.] und [Hunt u. Thomas, 1999, S. 109] werden viele weitere Vorteile, die durch den Einsatz von Zusicherungen direkt im Code entstehen, erläutert. Dazu zählt beispielsweise das frühe Abbrechen durch Nichterfüllung von Bedingungen, so dass Fehler im Programm leicht lokalisiert werden können.

Kontrakte bestehen aus drei Komponenten, den Vor- und Nachbedingungen sowie sogenannten Invarianten. Auf Methodenebene können Vorbedingungen formuliert werden, die vor der Ausführung der Methode erfüllt sein müssen und durch die Methode in Form von Zusicherungen überprüft werden. Der Aufrufer der Methode ist hierbei für die korrekte Erfüllung der Bedingungen verantwortlich. Im Gegenzug setzt die Methode ihrerseits den Kontrakt durch Erfüllen der Nachbedingungen um. Invarianten legen auf Klassenebene

fest, in welchen Zuständen sich Objekte dieser Klasse befinden dürfen. Während der Ausführung von Methoden darf die Invariante verletzt werden, muss jedoch nach Abschluss wieder erfüllt sein. Weiterhin können mögliche Ausnahmen bei der Ausführung Teil des Vertrages sein. Verträge einer Basisklasse müssen auch von Subklassen erfüllt werden, womit das liskovsche Substitutionsprinzip Beachtung findet [Binder, 1999, S. 821].

In [Binder, 1999, S. 882] wird das Percolation-Muster vorgestellt, dass die Vererbung der Verträge für Programmiersprachen, bei denen Design By Contract nicht auf Sprachebene implementiert ist, beschreibt. Dadurch kann das liskovsche Substitutionsprinzip für Subklassen verifiziert werden: „*Percolation is effective at revealing bugs that result from inheritance and dynamic binding.*“ [Binder, 1999, S. 882]

Mittlerweile gibt es für Ruby und viele andere Programmiersprachen Bibliotheken oder Spracherweiterungen zur Umsetzung von Design By Contract[1]. Die Refaktorierung *Introduce Assertion* aus [Fields u. a., 2009, S. 293] beschreibt eine einfache Methode zur Einführung von Vorbedingungen auf Methodenebene.

Wie bereits erwähnt, reduziert der Einsatz von Design By Contract die Auswirkungen mehrerer Schwachstellen sehr wirksam. Durch Aufstellen und Überprüfen der Vorbedingungen für Methoden können Fehler der Schwachstelle **Fehlerhafte Parameter** sehr früh erkannt und bei jeder Ausführung des Codes daraufhin getestet werden. Dabei sollte nicht versucht werden, ein statisches Typsystem nachzubauen, sondern auf einer höheren Ebene die Korrektheit der Eingabeparameter überprüft werden. Beispielsweise wäre es bei einer Methode `überweise_geld(betrag)` sinnvoll, den Parameter nicht auf den Typ, sondern auf einen Wert größer Null hin zu überprüfen.

Die Referenzierung des Nullwerts wird ebenfalls verringert, da die nicht erwarteten Nullwerte durch die eingebauten Zusicherungen während der Ausführung erkannt werden. Ebenso verhindern zugesicherte Nachbedingungen **falsche Rückgabewerte von Methoden**. Durch Zusicherungen kann auch der **Aufruf einer Methode der Basisklasse** für überschriebene Methoden in Subklassen formuliert werden. Dadurch braucht diese Bedingung, anstatt in jedem Test einer Subklasse, nur ein Mal getestet werden.

8.1.2 Nullobjekt/Special Case Entwurfsmuster

Die Einführung des Nullobjektes ist eine Refaktorierung zur Vermeidung von Nullwert-Überprüfungen im Code [Fields u. a., 2009, S. 284]. Martin Fowler beschreibt in [Fowler,

[1] Die englische Wikipedia Seite zu Design By Contract listet unter [URL:Wikipedia] über 15 Programmiersprachen auf, die das Konzept nativ umsetzen und verlinkt auf Erweiterungen für die meisten modernen Programmiersprachen

2002, S. 496] mit *Special Case* eine generalisierte Form der Nullwert Refaktorierung als Entwurfsmuster. In diesem Abschnitt werden diese Muster zur Vereinfachung gleich gestellt. Die Refaktorierung macht vor allem dann Sinn, wenn an mehreren Stellen im Falle eines speziellen Wertes (vor allem des Nullwerts) das gleiche Verhalten ausgelöst werden soll:

> *„Use Special Case whenever you have multiple places in the system that have the same behavior after a conditional check for a particular class instance, or the same behavior after a null check.“* [Fowler, 2002, S. 497]

Das Nullobjekt verhält sich dabei polymorph zu dem im Regelfall erwarteten Objekt, d.h. es antwortet auf alle Nachrichten, die auch an das erwartete Objekt gesendet werden können. Die Antwort kann wiederum ein Nullobjekt oder ein definierter Wert sein. Der Einsatz des Nullobjekt dient oft dazu, irreguläre Fälle im Code abzudecken, die nur selten vorkommen, aber trotzdem abgefangen werden müssen. Da es polymorph zum erwarteten Objekt ist, braucht der Empfänger keine Kenntnis über die irregulären Fälle haben. Die Verantwortung und das Wissen zur Behandlung dieses speziellen Falls wird im Nullobjekt gekapselt.

Ein anschauliches Beispiel aus dem simfy.de Code ist der Fall, wenn ein Benutzer nicht eingeloggt ist. Der größere Teil der Geschäftslogik ist auf bekannte und eingeloggte Benutzer ausgelegt. Nur in wenigen Fällen, wie dem Anmeldeprozess oder der Login-Maske, wird explizit zwischen einem eingeloggten und einem nicht eingeloggten Benutzer unterschieden. In allen anderen Bereichen der Applikation ist ein Benutzer zum korrekten Ablauf der Geschäftslogik zwingend notwendig. Tritt nun der Fall ein, dass ein nicht eingeloggter Benutzer in solch einen Bereich gelangt, wird das Nullobjekt `NotLoggedInUser` anstatt eines echten Benutzerobjektes verwendet. Das Nullobjekt antwortet auf alle Nachrichten mit Rückgabewerten, die für einen nicht eingeloggten Benutzer gelten.

Durch diese Refaktorierung wird die Gefahr für die Referenzierung des Nullwerts verringert. Das Verhalten für spezielle Fälle wird an einer Stelle gekapselt anstatt redundant im Code verteilt. Das vereinfacht und bündelt die Tests zur Abdeckung des speziellen Falls und verringert die Notwendigkeit, das Auftreten des Nullwerts an vielen Stellen im Code zu testen.

8.1.3 Bedingte Ausführung durch Blöcke

Das Konzept der Blöcke, auch Closures genannt, stammt aus funktionalen Sprachen und bezeichnet anonyme Funktionen, die Zugriff auf den Kontext haben, in dem sie erstellt wurden. Mittlerweile weisen auch viele objektorientierte Programmiersprachen Blöcke als

Sprachelement auf. Sie können dazu benutzt werden, die Entscheidung zur Ausführung von darin enthaltenem Code an die aufgerufene Methode zu verlagern, anstatt anhand des Rückgabewertes die Entscheidung dem Aufrufer zu überlassen.

Listing 8.1 zeigt exemplarisch die Verwendung dieses Verfahrens. Die Methode `find_by_name` der Klasse `UserStore` evaluiert den übergebenen Block nur wenn ein Eintrag gefunden werden konnte. In der `update` Methode der Klasse `UsersController` wird der auszuführende Code als Block übergeben. Die Überprüfung auf den Nullwert hin entfällt hierbei, wodurch die Referenzierung des Nullwertes verringert und somit die Schwachstelle **Fehlerhafte Objektreferenzen** adressiert wird.

Listing 8.1: Codebeispiel für die bedingte Ausführung durch Blöcke

```
1  class UserStore
2    def find_by_name(name, &block)
3      if user = Database.find_record(:type => :user, :name => name)
4        block.call(user)
5      end
6    end
7  end
8
9  class UsersController
10   def update
11     UserStore.find_by_name(params[:username]) do |user|
12       user.updated_at = Time.now
13       # ...
14     end
15   end
16 end
```

Zusätzlich zur Verlagerung der Nullwert Überprüfung kann durch die Verwendung von Blöcken auch die Trennung von Methoden mit und ohne Nebeneffekt, wie in Abschnitt 8.2.1 beschrieben, leicht realisiert werden. Dabei wird der Rückgabewert einer Methode mit Nebeneffekt an den Block übergeben.

8.2 Refaktorierungen zur Steigerung der Testbarkeit

Der vorherige Abschnitt beschreibt Wege zur Modellierung von testbarer Software. Dieser Abschnitt geht auf die Verbesserung von existierendem Quellcode ein. Nachdem die Schwachstellen identifiziert worden sind, sollen diese refaktoriert und damit testbarer gemacht werden. Teilweise können die Schwachstellen so umgestaltet werden, dass die daraus entstehenden Fehlerzustände nicht mehr auftreten können. Oft sind diese Fehlerquellen, wie z.B. die Typprüfung zur Laufzeit, ein wichtiger Bestandteil der Sprache und stellen aus anderen Perspektiven der Softwareentwicklung große Vorteile dar (→ Abschnitt 3.2.3). Eine komplette Vermeidung der Schwachstellen würde somit die Verwendung der Sprache selbst ad absurdum führen.

8.2.1 Separate Query from Modifier - Refaktorierung

Methoden sollten möglichst so gestaltet werden, dass sie entweder beobachtbare Nebeneffekte aufweisen *oder* einen Wert zurückgeben [Fields u. a., 2009, S. 304]. Durch die Trennung reduziert sich die Verantwortlichkeit der Methode. Entweder der Zustand eines Attributs des Objektes wird zurück gegeben oder der Zustand wird verändert. Bei der Reduktion der Verantwortlichkeit einer Methode wird zumeist auch die Abhängigkeit zu anderen Komponenten verringert, wodurch die Kontrollierbarkeit gesteigert wird.

Mittels der Erstellung von Abfrage-Methoden (Query), die den Zustand der zu testenden Komponente nicht verändern, kann die Beobachtbarkeit erhöht werden. Diese Methoden können damit problemlos in Zusicherungen, sowohl bei eingebauten als auch bei externen Tests, verwendet werden, da diese nicht Teil der Geschäftslogik sind und damit den Zustand der Applikation nicht verändern dürfen (→ Abschnitt 8.1.1). Die Nebeneffekt-Methode (Modifier) verändert den Zustand des Objekts, ohne etwas zurückzugeben[2].

Ohne diese Aufteilung müsste beim Test der Rückgabe einer Methode jeder Nebeneffekt auf einer abhängigen Komponente gestubbt werden, wodurch die Lesbarkeit und die Testbarkeit sinkt.

8.2.2 Extract Method, Class und Module - Refaktorierungen

Die Refaktorierungen *Extract Method*, *Extract Class* und *Extract Module* dienen der Steigerung der Kohäsion der entsprechenden Codefragmente und helfen damit bei der Beseitigung von Fehlern der Kategorie **Methode hat geringe Kohäsion**. In der Regel führen diese Refaktorierungen auch zur erhöhten Testbarkeit.

Beim Extrahieren von z.B. einer Methode, wird Funktionalität, die vorher nur implizit durch Aufruf einer anderen Methode getestet werden konnte, nun direkt aufrufbar und damit besser kontrollierbar. Das Extrahieren einer neuen Klasse oder eines Moduls wird ausgeführt, wenn eine vorhandene Klasse zu viele Verantwortlichkeiten trägt [Fields u. a., 2009, S. 167]. Durch die Refaktorierung entstehen zwei Komponenten, die zwangsläufig geringere Verantwortlichkeiten aufweisen und damit kohärenter sind. Dies sind beides Eigenschaften testbarer Software (→ Abschnitt 4.3.3).

[2] In Ruby gibt jeder Ausdruck einen Wert zurück. Methoden geben z.B. die letzte Anweisung oder nil zurück. Dadurch ist es nicht möglich, dass eine Methode keinen Rückgabewert hat. Es kommt somit auf die Verwendung bzw. Nichtverwendung des Rückgabewerts an.

8.2.3 Reduktion von Metaprogrammierung

Wie in Kapitel 3.3 erläutert, bietet Metaprogrammierung viele Vorteile. Gleichzeitig können dadurch aber auch Schwachstellen entstehen (→ Abschnitt 5.3.9). Die Refaktorierung *Replace Dynamic Method Receptor with Dynamic Method* aus [Fields u. a., 2009, S. 158] reduziert die Anzahl möglicher Fehler durch die Verschiebung des Zeitpunkts, an dem die Methoden generiert werden.

Anstatt die Nachrichten per `method_missing` zur Laufzeit zu beantworten, können die Methoden bei der Initialisierung des Programms erstellt werden. Die Refaktorierung ist damit nur möglich, wenn vorab bekannt ist, welche Nachrichten das Objekt beantworten soll. Ist dies der Fall, können dadurch die meisten möglichen Fehler der Schwachstelle **Dynamische Generierung von Methoden** verhindert werden.

Kann die Erstellung der Methoden nicht zum Initialisierungszeitpunkt verschoben werden, so bietet sich die *Isolate Dynamic Receptor* Refaktorierung an [Fields u. a., 2009, S. 160]. Hierbei wird ein eigenständiges Objekt für die Behandlung der unbekannten Nachrichten mittels `method_missing` erstellt. Dadurch wird die Komplexität und Fehleranfälligkeit besser gekapselt.

8.3 Dynamische Typisierung und konstruktive Qualitätssicherung

Im Allgemeinen ist durch dynamische Typisierung größere Flexibilität gegeben. Änderungen am Quelltext können schneller umgesetzt werden, da keine Typanpassungen notwendig sind. Dies gilt dementsprechend auch für die zu erstellenden Testfälle.

Typangaben, deren Prüfung zum Kompilierungszeitpunkt und die daraus folgenden notwendigen Aufgaben für den Entwickler, wie z.B. die explizite Konvertierung von Typen, senken die Ausdrucksstärke des Quelltextes und können dessen Komplexität erhöhen. Ebenso weisen statisch typisierte Programme in der Regel mehr Codezeilen auf als dynamisch typisierte, vgl. [Prechelt, 2000], wodurch die Lesbarkeit und das Verständnis davon leiden können.

In dynamisch typisierten Programmen kann Polymorphismus leichter umgesetzt werden, da Objekte leichter ausgetauscht werden können. Die Objekte müssen lediglich die gleichen Nachrichten beantworten können: „*Ruby's duck typing makes it easy to introduce polymorphism.*“ [Fields u. a., 2009, S. 281] Durch diesen geringeren Formalismus werden manche Refaktorierungen vereinfacht. Dazu gehören beispielsweise *Replace Type Code*

with Polymorphism [Fields u. a., 2009, S. 226] und *Replace Conditional with Polymorphism* [Fields u. a., 2009, S. 280].

Statische Typisierung hingegen hat den Vorteil, Typfehler bereits während der Programmierung aufzuzeigen. In dynamisch typisierten Sprachen kann dies in der Regel nur durch dynamische Tests erkannt werden (→ Abschnitt 6.1).

Auch ist bei statischer Typisierung die bessere Unterstützung durch eine Integrated Development Environment (IDE) von Vorteil. Einfache Refaktorierungen, wie z.B. die Umbenennung einer Methode, können mit Hilfe einer modernen IDE sehr schnell und effizient umgesetzt werden. Mit der zunehmenden Verbreitung von dynamisch typisierten Programmiersprachen, sind aber auch auf diesem Gebiet Fortschritte zu verzeichnen. Beispielsweise hat die Firma Jetbrains die Refaktorierungsmöglichkeiten von Ruby, Python, Javascript und PHP in ihren IDE - Produkten in den letzten Jahren stark erweitert[3].

[3] http://www.jetbrains.com/ruby/features/ruby_ide.html#Intelligent_Code_Improvement

9 Fazit und Ausblick

Das Ziel dieser Arbeit war die Identifizierung von Faktoren, die die Testbarkeit dynamisch typisierter Programme beeinflussen. Dazu wurden, neben den Grundlagen des Testens von Software, die Typisierung von objektorientierten Programmen erläutert. Darauf folgte eine ausführliche Behandlung des aktuellen Forschungsstands auf dem Gebiet der Testbarkeit. Hierbei wurde die Bedeutung der Kontrollierbarkeit und der Beobachtbarkeit von Software deutlich gemacht , sowie der Zusammenhang zwischen Testbarkeit und gutem Softwaredesign hergestellt. Die gefundenen Abhandlungen zu diesem Thema wurden auf Basis von statisch typisierten Programmiersprachen durchgeführt, so dass mit dieser Arbeit Neuland betreten wurde.

Frage: statisch? ich dachte dynamisch typisierten Programmiersprachen...

In der Fehleranalyse wurden zwei Programme untersucht und deren Fehler kategorisiert. Die Wahl für Binder's Fehlertaxonomie als Grundstruktur der Kategorisierung stellte sich im Nachhinein als nicht optimal dar. Aufgrund der Komplexität dieser Aufgabe wäre eine weniger umfangreiche Fehlerkategoriestruktur, wie z.B. die aus [Borner u. a., 2008] angemessener gewesen. Ebenso ist auffällig, dass die meisten Fehler auf Methodenebene eingeordnet wurden, wodurch möglicherweise darüber liegende, strukturelle Fehlerzusammenhänge übersehen wurden. Teilweise war starke persönliche Interpretation bei der Zuordnung der Fehler zu den Kategorien notwendig. An dieser Stelle wäre eine Einordnung durch weitere Experten sinnvoll. Weiterhin sollten die zu Grunde gelegten Fehlerkategorien um weitere Fehler aus anderen Projekten ergänzt werden. Nach dem Filtern irrelevanter Fehlerbeschreibungen, wie z.B. den Fehlern , die auf falsche Umsetzung der Anforderungen zurückzuführen sind, blieben nur wenige Fehler für eine aussagekräftige Schlussfolgerung übrig. Nichtsdestotrotz ist ein umfangreiches, spezifisches Fehlermodell für die Programmiersprache Ruby entstanden, welches eine geeignete Grundlage für die effektive Fehlersuche darstellt.

Viele der Fehler stehen im Zusammenhang mit dem Nullwert. Eine vergleichende Untersuchung mit einer statisch typisierten Sprache wäre eine angebrachte Vertiefung, um zu

ermitteln, an wie vielen Fehlern der Nullwert beteiligt ist und ob dies durch das Typsystem beeinflusst wird.

Von den betrachteten Fehlerzuständen hätten sehr wenige durch statische Typisierung verhindert werden können, was durch die hohe Testabdeckung beider Plattformen begründet sein könnte. Auch die Wahl des Entwicklungsprozesses hat, wie in Kapitel 2.1 erläutert, Einfluss auf die interne Produktqualität. Testgetriebene Entwicklung, wie sie bei simfy.de ausgeübt wird, erfasst sehr früh einfache Fehler und dient damit als geeignete Kompensation für das Fehlen der Prüfung durch den Compiler statisch typisierter Sprachen.

Frage: macht Spree keine testgetriebene Entwicklung oder weißt du das nur nicht? Wenn doch, dann nimm beide hier auf oder wenn du es nicht genau weißt, würd ich auch simfy.de hier weglassen.

Es wäre interessant zu untersuchen, in welchem Maß die Wahl der Programmiersprache und des Entwicklungsprozesses zusammenhängen.

Frage: besser fände ich: Interessant wäre es nun zu untersuchen, in welchem Maß die Wahl der Programmiersprache und des Entwicklungsprozesses zusammenhängen.

Die Vermutung liegt nahe, dass beim testgetriebenen Vorgehen die Typisierung nur eine untergeordnete Rolle für die Fehlerhäufigkeit spielt.

Frage: Warum liegt die Vermutung nahe?

Bei der detaillierten Beschreibung der Schwachstellen in Kapitel 6 wurden die Nachteile der fehlenden Formalisierung dynamisch typisierter Programme für die statischen Tests herausgestellt. Ebenso behindert die oft verwendete Metaprogrammierung die Erstellung von Softwaremetriken. Allerdings gibt es Techniken, wie die Typinferenz oder die Datenflussanalyse, um diese Nachteile, zumindest teilweise, zu mindern.

Dementsprechend sind die Ergebnisse des Prototyps zur statischen Ermittlung der Schwachstellen nicht zufriedenstellend. Zwar wurde ein Fehler aufgrund der Analyse durch den Prototyp entdeckt, es wurden aber auch sehr viele falsche positive Schwachstellen gemeldet. Dies verdeutlicht die Relevanz von statischen Tests, da damit sonst nur aufwändig lokalisierbare Fehler viel effizienter entdeckt werden können.

Frage: Den Satz hab ich erst nach dem 3. mal lesen verstanden. Besser würde mir gefallen: Dies verdeutlicht die Relevanz von statischen Tests, denn damit können sonst nur sehr aufwändig lokalisierbare Fehler viel effizienter entdeckt werden.

Gleichzeitig wird durch den Prototyp deutlich, dass die Erstellung von nützlichen Aussagen über dynamisch typisierte Programme mit erheblichem Aufwand verbunden ist. Eine Erweiterung des Prototyps um Typinferenz und Datenflussanalyse wäre eine sinnvolle Vertiefung. Ebenso sollte die Identifikation der restlichen, statisch erkennbaren Schwachstellen im Prototyp implementiert werden, um das Fehlermodell mit seinen typischen Fehlerquellen weiter zu validieren. Durch das Hinzuziehen weiterer Untersuchungsobjekte könnte die Validität der Schwachstellen auf eine breitere Basis gestellt werden.

Anschließend wurden Programmierkonzepte und Entwurfsmuster vorgestellt, um Testbarkeit bereits beim Entwurf und der Implementierung zu etablieren. Im Code eingebettete Tests, wie es das Design By Contract Konzept beschreibt, erweisen sich bei dynamisch typisierten Sprachen als besonders sinnvoll, da sie selbst bei sehr großen Programmen für verlässlich interagierende Komponenten sorgen[1]. Ergänzend wurden mehrere Refaktorierungen vorgestellt, um Programme im Nachhinein testbarer zu gestalten.

Im Verlauf der Arbeit konnten somit alle eingangs gestellten Fragen beantwortet werden und es wurden Vertiefungsmöglichkeiten für nachfolgende Arbeiten identifiziert.

Abschließend ist festzuhalten, dass dynamisch typisierte Programmiersprachen zwar spezielle Herausforderung an die Erstellung testbarer Software stellen, diese jedoch mit den aufgezeigten Methoden zufriedenstellen d gelöst werden können.

Frage: Yippi yippi yeah, fertiiiiiiiiig! HERZLICHEN GLÜCKWUNSCH!!!!! Allerdings ist dein Fazit so negativ, wie du deine Arbeit immer einschätzt... Es hört sich ganz leicht so an, als wenn du nun alles anders machen würdest. Generell find ich das Fazit gut, weil du alle Fragen nochmal aufzeigst, die Ergebnisse davon, aber das eben alles etwas negativ.

Frage: In der Kapitelanzeige von Skim hat Fazit und Ausblick die selbe Seitenzahl wie das Literaturverzeichnis. Also beide fangen bei S. 68 an und wenn man auf Literaturverzeichnis klickt, kommt man zum Fazit. Ist das bei dir auch so? Und Vergiss nicht, die Sachen im Anhang noch einzufügen.

[1] Ein häufiges Argument gegen die Verwendung von dynamisch typisierten Programmen ist, dass diese sehr unzuverlässig werden, wenn die Anzahl von Codezeilen in die Millionen geht.

Literaturverzeichnis

[Badri u. a. 2011] Badri, L. ; Badri, M. ; Toure, F.: An empirical analysis of lack of cohesion metrics for predicting testability of classes. In: *International Journal of Software Engineering and Its Applications* 5 (2011), Nr. 2, S. 69–85

[Basili u. a. 1996] Basili, V.R. ; Briand, L.C. ; Melo, W.L.: A validation of object-oriented design metrics as quality indicators. In: *Software Engineering, IEEE Transactions on* 22 (1996), oct, Nr. 10, S. 751 –761. http://dx.doi.org/10.1109/32.544352. – DOI 10.1109/32.544352. – ISSN 0098–5589

[Bass u. a. 2012] Bass, L. ; Clements, P. ; Kazman, R.: *Software architecture in practice.* 3rd. Addison-Wesley Professional, 2012. – ISBN 978–0321815736

[Beck u. Andres 2004] Beck, Kent ; Andres, Cynthia: *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004. – ISBN 0321278658

[Binder 1994] Binder, Robert V.: Design for testability in object-oriented systems. In: *Commun. ACM* 37 (1994), September, S. 87–101. http://dx.doi.org/10.1145/182987.184077. – DOI 10.1145/182987.184077. – ISSN 0001–0782

[Binder 1999] Binder, Robert V.: *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Reading, Mass. : Addison-Wesley, 1999 (Object Technology). http://books.google.de/books?id=P3UkDhLHP4YC

[Borner u. a. 2008] Borner, L ; Ebrecht, L ; Jungmayr, S ; Hamburg, M ; Winter, M: Suchen Sie die richtigen Fehler? Warum es sich lohnt, Fehler zu kategorisieren. In: *Objektspektrum* 1 (2008), S. 73–80

[Borner u. a. 2006] Borner, Lars ; Fraikin, Falk ; Hamburg, Matthias ; Jungmayr, Stefan ; Schönknecht, Andreas: Fehlerhäufigkeiten in objektorientierten Systemen: Basisauswertung einer Online-Umfrage. In: *Softwaretechnik-Trends* 1 (2006), Februar, 43-45. http://www.ub.uni-heidelberg.de/archiv/8459. – ISSN 0-720-8928

[du Bousquet 2010] Bousquet, Lydie du: A New Approach for Software Testability. In: *Testing - Practice and Research Techniques, 5th International Academic and Industrial Conference (TAIC PART)* Bd. 6303. Windsor, UK : Springer, 2010 (Lecture Notes in Computer Science), S. 207–210

[Bruntink u. van Deursen 2004] Bruntink, M. ; Deursen, A. van: Predicting class testability using object-oriented metrics. In: *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, 2004, S. 136 – 145

[Catlett 2007] Catlett, David: Improving the Testability of Software. In: McDonald, M. (Hrsg.) ; Musson, R. (Hrsg.) ; Smith, R. (Hrsg.): *The Practical Guide to Defect Prevention.* Microsoft Press, 2007 (Best Practices). – ISBN 9780735622531, S. 73–81

[Chidamber u. Kemerer 1994] Chidamber, Shyam R. ; Kemerer, Chris F.: A metrics suite for object oriented design. In: *Software Engineering, IEEE Transactions on* 20 (1994), Nr. 6, S. 476–493

[Chillarege 1996] Chillarege, Ram: Handbook of software reliability engineering. Version: 1996. http://dl.acm.org/citation.cfm?id=239425.239453. Hightstown, NJ, USA : McGraw-Hill, Inc., 1996. – ISBN 0–07–039400–8, Kapitel Orthogonal defect classification, 359–400

[Chowdhary 2009] Chowdhary, V.: Practicing Testability in the Real World. In: *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, 2009, S. 260 –268

[Crispin u. Gregory 2009] Crispin, Lisa ; Gregory, Janet: *Agile Testing: A Practical Guide for Testers and Agile Teams.* 1. Addison-Wesley Professional, 2009. – ISBN 0321534468, 9780321534460

[Dijkstra 1970] Dijkstra, Edsger W.: *Notes on Structured Programming.* http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF. Version: April 1970. – circulated privately

[Ebraert u. Vandewoude 2005] Ebraert, P. ; Vandewoude, Y.: Influence of type systems on dynamic software evolution. In: *the electronic proceedings of the International Conference on Software Maintenance (ICSM'05) Badapest Hungary* Citeseer, 2005

[Fagan 1976] Fagan, M. E.: Design and code inspections to reduce errors in program development. In: *IBM Systems Journal* 15 (1976), Nr. 3, S. 182 –211. http://dx.doi.org/10.1147/sj.153.0182. – DOI 10.1147/sj.153.0182. – ISSN 0018–8670

[Fields u. a. 2009] Fields, Jay ; Harvie, Shane ; Fowler, Martin ; Beck, Kent: *Refactoring: Ruby Edition.* 1st. Addison-Wesley Professional, 2009. – ISBN 0321603508, 9780321603500

[Flanagan u. Matsumoto 2008] Flanagan, D. ; Matsumoto, Y.: *The Ruby Programming Language.* O'Reilly Media, 2008 http://books.google.de/books?id=jcUbTcr5XWwC. – ISBN 9780596554651

[Fowler 2002] Fowler, Martin: *Patterns of Enterprise Application Architecture.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0321127420

[Freeman u. Pryce 2009] Freeman, Steve ; Pryce, Nat: *Growing Object-Oriented Software, Guided by Tests.* 1st. Addison-Wesley Professional, 2009. – ISBN 0321503627, 9780321503626

[Furr 2009] FURR, Michael: *Combining Static and Dynamic Typing in Ruby*, Digital Repository at the University of Maryland University of Maryland (College Park, Md.), Diss., 2009. http://dx.doi.org/http://hdl.handle.net/1903/9958. – DOI http://hdl.handle.net/1903/9958

[Furr u. a. 2009] FURR, Michael ; AN, Jong-hoon (. ; FOSTER, Jeffrey S. ; HICKS, Michael: Static type inference for Ruby. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA : ACM, 2009 (SAC '09). – ISBN 978–1–60558–166–8, 1859–1866

[Gamma u. a. 1995] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2

[Gao u. Shih 2005] GAO, J. ; SHIH, M.-C.: A component testability model for verification and measurement. In: *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International* Bd. 2, 2005. – ISSN 0730–3157, S. 211 – 218 Vol. 1

[Holkner u. Harland 2009] HOLKNER, Alex ; HARLAND, James: Evaluating the dynamic behaviour of Python applications. In: *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2009 (ACSC '09). – ISBN 978–1–920682–72–9, 19–28

[Hunt u. Thomas 1999] HUNT, Andrew ; THOMAS, David: *The pragmatic programmer: from journeyman to master*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0–201–61622–X

[IEEE 1990] Norm IEEE Std 610.12-1990 1990. *IEEE Standard Glossary of Software Engineering Terminology*

[IEEE 2010] IEEE: IEEE Standard Classification for Software Anomalies. In: *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), 7, S. C1 –15. http://dx.doi.org/10.1109/IEEESTD.2010.5399061. – DOI 10.1109/IEEESTD.2010.5399061

[ISO 2010] Norm ISO/IEC 25010 01 2010. *ISO/IEC 25010: System and software product Quality Requirements and Evaluation (SQuaRE) – System and software and software quality models*. – International Organization for Standardization

[Jensen u. a. 2009] JENSEN, Simon ; MØLLER, Anders ; THIEMANN, Peter: Type Analysis for JavaScript. Version: 2009. http://dx.doi.org/10.1007/978-3-642-03237-0_17. In: PALSBERG, Jens (Hrsg.) ; SU, Zhendong (Hrsg.): *Static Analysis* Bd. 5673. Springer Berlin / Heidelberg, 2009. – ISBN 978–3–642–03236–3, 238-255

[Jungmayr 2003] JUNGMAYR, Stefan: *Improving testability of object-oriented systems*, FernUniversität in Hagen, Diss., 12 2003. http://www.testability.de

[Kirch-Prinz u. Prinz 2002] Kirch-Prinz, U. ; Prinz, P. ; Bombien, Volker (Hrsg.): *C++ - Lernen und professionell anwenden*. Bd. 2. Auflage. mitp, 2002. – ISBN 3826608240

[Kropfitsch u. Vigenschow 2003] Kropfitsch, Dietmar ; Vigenschow, Uwe: Das Fehlermodell: Aufwandsschätzung und Planung von Tests. In: *Objekt-Spektrum,(6)* (2003), S. 30–35

[Lanza u. a. 2005] Lanza, Michele ; Marinescu, Radu ; Ducasse, Stéphane: *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2005. – ISBN 3540244298

[Liskov u. Wing 1994] Liskov, Barbara H. ; Wing, Jeannette M.: A behavioral notion of subtyping. In: *ACM Trans. Program. Lang. Syst.* 16 (1994), November, Nr. 6, 1811–1841. http://dx.doi.org/10.1145/197320.197383. – DOI 10.1145/197320.197383. – ISSN 0164–0925

[Lopez u. a. 2005] Lopez, Miguel ; Habra, Naji ; Seront, Grégory: On the Applicability of Predictive Maintainability Models onto dynamic Languages. In: *none* - (2005), S. 5

[Madsen u. a. 2007] Madsen, M. ; Sørensen, P. ; Kristensen, K.: *Ecstatic–Type Inference for Ruby Using the Cartesian Product Algorithm*, Aalborg University, Diplomarbeit, 2007

[McCabe 1976] McCabe, T.J.: A Complexity Measure. In: *Software Engineering, IEEE Transactions on* SE-2 (1976), Nr. 4, S. 308–320. http://dx.doi.org/10.1109/TSE.1976.233837. – DOI 10.1109/TSE.1976.233837. – ISSN 0098–5589

[McDaniel 1993] McDaniel, G.: *IBM Dictionary of Computing*. Bd. 10. IBM Corp., 1993. – 758 S. – ISBN 0-07-031488-8

[Meyer 1992] Meyer, Bertrand: Applying "Design by Contract". In: *Computer* 25 (1992), Oktober, Nr. 10, 40–51. http://dx.doi.org/10.1109/2.161279. – DOI 10.1109/2.161279. – ISSN 0018–9162

[Nazir u. a. 2010] Nazir, M. ; Khan, R.A. ; Mustafa, K.: Testability Estimation Framework. In: *International Journal of Computer Applications IJCA* 2 (2010), Nr. 5, S. 9–14

[Nunes u. Schwabe 2006] Nunes, Demetrius A. ; Schwabe, Daniel: Rapid prototyping of web applications combining domain specific languages and model driven design. In: *Proceedings of the 6th international conference on Web engineering* ACM, 2006, S. 153–160

[Opdyke 1992] Opdyke, William F.: *Refactoring object-oriented frameworks*. Champaign, IL, USA, University of Illinois at Urbana-Champaign, Diss., 1992. – UMI Order No. GAX93-05645

[Pettichord 2002] Pettichord, Bret: Design for testability. In: *In Pacific Northwest Software Quality Conference*, 2002

[Pierce 2002] Pierce, B.C.: *Types and programming languages.* MIT press, 2002. – ISBN: 0-262-16209-1

[Plevyak u. Chien 1994] Plevyak, John ; Chien, Andrew A.: Precise concrete type inference for object-oriented languages. In: *SIGPLAN Not.* 29 (1994), Oktober, Nr. 10, 324–340. http://dx.doi.org/10.1145/191081.191130. – DOI 10.1145/191081.191130. – ISSN 0362–1340

[Prechelt 2000] Prechelt, L.: An empirical comparison of seven programming languages. In: *Computer* 33 (2000), Nr. 10, S. 23–29. http://dx.doi.org/10.1109/2.876288. – DOI 10.1109/2.876288. – ISSN 0018–9162

[Shalloway u. Trott 2004] Shalloway, A. ; Trott, J.: *Design patterns explained: a new perspective on object-oriented design, Second Edition.* 2nd. Addison-Wesley Professional, 2004. – Print ISBN-13: 978-0-321-24714-8

[Spillner u. Linz 2012] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest.* 5. Auflage. Heidelberg : dpunkt.verlag GmbH, 2012

[Thomas u. a. 2009] Thomas, D. ; Fowler, C. ; Hunt, A.: *Programming Ruby 1.9: The Pragmatic Programmers' Guide.* Pragmatic Programmers, LLC, 2009 (Pragmatic Bookshelf Series). http://books.google.de/books?id=f89GPgAACAAJ. – ISBN 9781934356081

[Tilkov 2008] Tilkov, Stefan: Dynamische Sprachen im Aufwind. In: *OBJEKTspektrum* 04 (2008), S. 28–29

[Tratt 2009] Tratt, Laurence: Dynamically Typed Languages. In: *Advances in Computers* 77 (2009), Juli, S. 149–184

[Tratt u. Wuyts 2007] Tratt, Laurence ; Wuyts, Roel: Guest Editors' Introduction: Dynamically Typed Languages. In: *Software, IEEE* 24 (2007), sept.-oct., Nr. 5, S. 28 –30. http://dx.doi.org/10.1109/MS.2007.140. – DOI 10.1109/MS.2007.140. – ISSN 0740–7459

[Turhan u. a. 2010] Turhan, Burak ; Layman, Lucas ; Diep, Madeline ; Erdogmus, Hakan ; Shull, Forrest: How Effective Is Test-Driven Development? In: Oram, A. (Hrsg.) ; Wilson, G. (Hrsg.): *Making Software: What Really Works, and Why We Believe It.* O'Reilly Media, 2010. – ISBN 9781449397760, S. 399–412

[URL:Fowler a] URL:Fowler, Martin: *Domain Specific Languages.* Webseite. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html. – Abgerufen am 28.07.2013

[URL:Fowler b] URL:Fowler, Martin: *Inversion of Control Containers and the Dependency Injection patterns.* Webseite. http://martinfowler.com/articles/injection.html. – Abgerufen am 11.08.2013

[URL:RDoc] URL:RDoc: *Documentation for the Ruby programming language.* Webseite. http://www.ruby-doc.org. – Abgerufen am 03.08.2013

[URL:Wikipedia] URL:Wikipedia: *Design By Contract.* Webseite. http://en.wikipedia.org/wiki/Design_by_contract#Language_support. – Abgerufen am 13.08.2013

[Winter 1997] Winter, Mario: Testbarkeit objekt-orientierter Programme. In: *GI-Fachgruppe Test, Analyse und Verifikation von Software, Arbeitskreis Testen objekt-orientierter Programme* (1997). ftp://ftp.fernuni-hagen.de/pub/fachb/inf/pri3/papers/winter/OOTestbarkeit.ps.gz

Anhang

Fehlerkatalog der Plattform simfy.de

TODO: Tabelle über mehrere Seiten umbrechen, siehe http://projekte.dante.de/DanteFAQ/TabellenSeitenumbruch

TODO: Fehlerkatalog für simfy aufbereiten

Fehlerkatalog der Plattform Spree

TODO: Fehlerkatalog für Spree aufbereiten

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Eduard Litau – Köln, den 19. August 2013