Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Masterarbeit

Ein Framework für automatisierte Architekturtests

23. April 2015

Vorgelegt von:

Samuel Eickelberg

Trappenweg 10A

14612 Falkensee

Kontakt: samuel.eickelberg@gmail.com

Matrikelnummer: 4518699

Betreuerin:

Prof. Dr. Ina Schieferdecker, Arbeitsgruppe Software Engineering

Gutachterin:

Prof. Dr. Margarita Esponda-Argüero, Arbeitsgruppe Programmiersprachen

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Falkensee, im April 2015

_____

(Samuel Eickelberg)

Danksagung

Zunächst möchte ich mich bei meinem Arbeitgeber, der ANYBET GmbH, bedanken für die Bereitschaft, mich die Masterarbeit neben der Berufstätigkeit durchführen zu lassen. Insbesondere danke ich dabei für hilfreiche Diskussionen Thomas Feger sowie Thomas Schupp.

Weiterhin danke ich der Betreuerin dieser Arbeit, Prof. Dr. Ina Schieferdecker, für ihre Anregungen, insbesondere den Forschungsbeitrag der Arbeit noch klarer herauszustellen.

Ganz besonders danke ich meiner Frau, Roxana Eickelberg, für ihre moralische und liebevolle Unterstützung. Ohne sie hätte ich das Master-Studium der Informatik an der Freien Universität nicht durchführen können. Ihr widme ich daher diese Arbeit.

Kurzzusammenfassung

Automatisierte Tests konzentrieren sich auf die Funktionsweise von Softwarekomponenten und deren Zusammenspiel. Neu ist jedoch das automatisierte Testen von *Entwurfsmustern, Programmierparadigmen* und *Codemetriken*. Hier ist der Begriff des *Architekturtests* prägend als Oberbegriff für solche Arten von Tests. Diese Arbeit hat zum Ziel, unterschiedliche Dimensionen von Architekturtests aufzuzeigen, diese als dynamische Tests gegenüber statischer Codeanalyse abzugrenzen und als Framework zu implementieren.

Architekturtests sind Unit-Tests und kommen ohne Mock-Testdaten und ohne andere Komponenten, wie etwa Drittsysteme oder Datenbanken, aus. Das Reporting der Testergebnisse erfolgt in der Continuous-Integration-Umgebung. Das Konzept des Testframeworks sowie Arten der Tests werden erarbeitet; dabei wurden Anforderungen an die zunächst drei verschiedenen Testkategorien, Codemetriken, Programmierkonventionen und Entwurfsmuster, formuliert. Daraus entstand der Entwurf, geeignete Annotationen für die Kennzeichnung der Klassen zu verwenden, die zur Laufzeit vom Testframework den jeweiligen Tests zugeführt werden sollen.

In der Implementierungsphase des Frameworks stellten sich zwei wesentliche Ergänzungen heraus: Die vierte Testkategorie *Abhängigkeiten*, welche auch die meisten Tests enthält. Architektur*entscheidungen* beruhen oft auf bestimmten Abhängigkeiten von Klassen, Paketen und Modulen. Die zweite Ergänzung verleiht dem Framework seine eigentliche Mächtigkeit und Robustheit gegenüber Veränderungen am zu testenden Code, die *Rule-API*. Mit ihr wurde es möglich, *Regeln* zu definieren, welche die Durchsetzung von Architekturentscheidungen im Code unabhängig vom Gebrauch der Annotationen machen.

Im Rahmen der Bewertung wurde eine *Fallstudie* in zwei Teilen durchgeführt. Der erste Teil befasste sich mit dem Refactoring von bestehendem Code aufgrund der Architekturtests, der zweite Teil bestand aus der Neuimplementierung eines Features mit den Architekturtests als „Wächter“ über die Architekturentscheidungen. Die nachträgliche Einführung solcher Tests in Legacy-Software ist auf den Erhalt der bisherigen Codequalität zu beschränken. Die Neuentwicklung wird durch den Einsatz der Tests so unterstützt, dass entlang der in den Tests fixierten Architekturentscheidungen entwickelt wird. Abweichungen hiervon werden gemäß *Architecture-Test-first*-Prinzip unmittelbar gemeldet.

Das Framework verhindert nicht schleichenden Architekturverlust. Es unterstützt Entwickler dabei, Architektur- und Designentscheidungen einzuhalten durch die regelmäßige Ausführung der Tests, ebenso häufig wie Unit-Tests. Dafür sind keine Spezialkenntnisse über statische Codeanalysetools erforderlich; die Offenheit des Frameworks ermöglicht Entwicklern das eigenständige und Erweitern der Architekturtests.

Abstract

Automated tests focus on the functionality and interaction of software components. In a very early development stage, we find automated testing of *design patterns* in code, as well as *programming paradigms* and *software metrics*. This is where the term *architecture test* comes into play. This thesis aims at different types of architecture tests, distinguishes them from static code analysis tools, and contains their implementation as a framework.

Architecture tests are unit tests, without the need of creating mock test data, or attaching thirt-party systems, such as databases or other components. The reporting of test results takes place within the continuous integration platform. The concept of architecture tests as well as their requirementrs for the different types of tests are being derived and described. From these requirements, the tests including appropriate annotations to mark classes to be tested, were designed.

During implementation phase of the framework, two substantial extensions, compared to the initial design, came up: A fourth test category *dependencies*, which contains more tests than any other of the original types. Architecture *decisions* are often based on certain dependencies among classes, modules, and packages. The second addition makes the framework powerful and robust towards changes in the code to be tested, the *Rule API*. It allows to define *rules* which enforce architecture decisions on the code regardless of the use of the annotations.

Within the evaluation, a *case study* consisting of two parts had been conducted. First, the architecture tests were run on legacy code, resulting in a number of refactoring activities. The second part consisted of the implementation of a new feature using the architecture tests. They served as guards over architecture decisions. Late introduction of the tests into legacy code could only aim at conserving the given code quality. Early introduction from the start supported the development process in a way that implementation took place alongside the architecture decisions. Deviations from them resulted in immediate feedback.

Slow loss of architecture cannot be avoided by the use of this framework. However, it supports developers to stick to architecture and design decisions by running the tests frequently, just as often as unit tests. Specialized knowledge about static code analysis tools is not needed. The openness of the framework allows developers to quickly adapt and create new architecture tests.

Inhaltsverzeichnis

Eidesstattliche Erklärung....2
Danksagung....3
Kurzzusammenfassung....4
Abstract....5
Inhaltsverzeichnis....6
1 Einleitung....10
1.1 Allgemeine Hinweise....11
1.2 Problemstellung....11
1.3 Begriffe....12
1.3.1 Programmiersprache....12
1.3.2 Entwurfsmuster....13
1.3.3 Framework....13
1.3.4 Automatisiertes Testen....13
1.3.5 Codemetrik....13
1.3.6 Architekturtest....14
1.4 Zielskizze und Überleitung....14
2 Voraussetzungen....15
2.1 Begriffe....15
2.1.1 Architektur....15
2.1.2 Entwurfsmuster....15
2.1.3 Codemetriken....16
2.1.4 Programmierrichtlinien....17
2.1.5 (Dynamische) Tests....18
2.1.6 Statische Codeanalyse....23
2.1.7 Zwei Welten....24
2.2 Werkzeuge....25
2.2.1 Maven....25
2.2.2 IntelliJ IDEA....26
2.2.3 Jenkins....27
2.3 mybet-Software....27
2.3.1 mAS....28
2.3.2 mFES....29
2.3.3 BetShop....31

2.3.3.1 Architektur....31
2.3.3.2 Architekturtests....33
2.3.4 Test-Infrastruktur....34
2.4 Zusammenfassung und Überleitung....34
3 Konkretisierung / Abgrenzung....36
3.1 Analyse....36
3.1.1 mAS – Entitäten....36
3.1.2 mWS – Sports Bets Engine....37
3.1.3 Testanforderungen....38
3.1.3.1 Anforderungen an Codemetriken....38
3.1.3.2 Anforderungen an Programmierrichtlinien....39
3.1.3.3 Anforderungen an die Einhaltung von Entwurfsmustern....41
3.1.3.4 Anforderungen an Abhängigkeiten....41
3.2 Entwurf des Frameworks....43
3.3 Laufzeitumgebung und Hilfsmittel....45
3.4 Testen der Tests....45
3.5 Zusammenfassung und Überleitung....46
4 Umsetzung....47
4.1 Entwicklung des Frameworks....47
4.1.1 Test-Basisklassen....47
4.1.2 Annotationen....48
4.1.3 Regeln....49
4.1.4 Convention Tests....54
4.1.4.1 CamelCaseTest....55
4.1.4.2 OrderingTest....55
4.1.4.3 OverlyChainedMethodCallsTest....56
4.1.4.4 NoInnerClassesTest....56
4.1.4.5 Zwischenfazit zu Convention Tests....57
4.1.5 Metrics Tests....57
4.1.5.1 CyclomaticComplexityTest....57
4.1.5.2 HalsteadComplexityTest....57
4.1.5.3 MethodLengthTest....58
4.1.5.4 NumberOfMethodsPerClassTest....58
4.1.6 Pattern Tests....58

4.1.6.1 SingletonTest....59
4.1.6.2 AdapterTest....60
4.1.6.3 TemplateMethodTest....60
4.1.6.4 StateTest....60
4.1.6.5 StrategyTest....61
4.1.7 Dependency Tests....61
4.1.7.1 ExtendsTest....62
4.1.7.2 ImplementsTest....62
4.1.7.3 ThrowsTest....62
4.1.7.4 UsesTest....63
4.1.7.5 AcyclicDependenciesPrincipleTest....63
4.1.7.6 InstabilityTest....64
4.1.7.7 UncheckedCastsTest....64
4.1.7.8 DangerousCastsTest....65
4.1.8 Andere Klassen....65
4.1.8.1 ArchitectureTestRunListener....66
4.1.8.2 ArchitectureTestAbstractEvaluator....66
4.1.8.3 ArchitectureTestRuleEvaluator....67
4.1.8.4 ArchitectureTestAnnotationEvaluator....67
4.1.8.5 ArchitectureTestConstants....67
4.1.8.6 ArchitectureTestUtil....67
4.1.9 Verwendete Bibliotheken....68
4.1.10 Zusammenfassung....69
4.2 Integration des Frameworks....70
4.2.1 Regeln....71
4.2.2 Annotationen....71
4.2.3 Maven....73
4.2.4 Jenkins....74
4.2.5 Zusammenfassung....75
4.3 Konsequenzen für die weitere Entwicklung....75
5 Bewertung....77
5.1 Statische Codeanalyse in dynamischen Tests....77
5.1.1 Codeanalyse-Tools....77
5.1.2 Dynamische Tests....78

5.1.3 Zusammenfassung....79
5.2 Architekturtests im Einsatz....79
5.3 Gesammelte Erfahrungen....80
5.3.1 Fallstudie – Aufgaben....80
5.3.2 Fallstudie – Ergebnisse....81
5.3.2.1 Methodenlängen....82
5.3.2.2 Innere Klassen....83
5.3.2.3 Instabilität....83
5.3.2.4 Zyklenfreiheit....83
5.3.2.5 Zusammenfassung der Ergebnisse....83
5.4 Einschränkungen....83
5.5 Ansätze zur Weiterentwicklung der Architekturtests....84
5.5.1 Andere Sprachen....84
5.5.1.1 JavaScript....84
5.5.1.2 SQL und PL/SQL....85
5.5.2 Framework-spezifische Konventionen....85
5.5.2.1 Dateikonventionen....85
5.5.2.2 Annotationen....86
5.5.3 Generizität und Kapselung....86
5.5.4 Open Source-Projekt....87
5.5.5 Anbindung an eine Graphdatenbank....87
5.5.6 Architekturtests als Plugin für Entwicklungsumgebungen....90
5.5.7 Architekturtests als Kriterium für Gated Commits....90
5.5.8 Saubere Ressourcenbehandlung per Test fixieren....91
5.5.9 Sichere Programmierrichtlinien per Test fixieren....91
5.5.10 Erweiterungen bestehender Tests....93
5.6 Zusammenfassung....94
6 Fazit....95
Abbildungsverzeichnis....97
Literaturverzeichnis....98
Glossar....99
Anhang....101
Hinweise zur CD-ROM....104

1 Einleitung

Aktuelle Softwareprojekte zeichnen sich dadurch aus, dass sie zwar in einer ideellen Welt geplant und entworfen werden, das entstandene Endprodukt jedoch in seiner Komplexität und seinen Abhängigkeiten von anderen Produkten, Diensten und Entitäten häufig deutlich über der ursprünglichen Planung angesiedelt ist. Dies ist nicht erst ein mit der Informatik neu eingeführtes Phänomen, sondern es beschreibt eine ganz grundlegende Funktionsweise des menschlichen Geistes: Die abstrakte Modellbildung.

Schon im Altertum begannen Menschen, die Welt mittels Philosophie, Mathematik, Naturwissenschaft und auch Religion begreifbar und im Ganzen erfassbar zu machen, in dem von vielen Realitätsdetails abstrahiert und versucht wurde, den Kern, das Wesentliche, zu definieren. Dies ist die eine Seite der Modellbildung.

Die andere Seite befasst sich mit dem Erkennen und Beschreiben wiederkehrender Muster in unterschiedlichen Modellen, um einen bereits bestehenden Erkenntnissatz nicht immer wieder neu entdecken und beschreiben zu müssen. In den Anfängen der Informatik wurden Programme, im Vergleich zu heute, mehr oder weniger „im Blindflug“ geschrieben. Erst nach Fertigstellung einer kompletten Iteration, aufgrund der technischen Restriktionen der damaligen Zeit, konnte das geschriebene Programm getestet werden. Schlug der Test fehl, musste der Fehler korrigiert, das neue Programm in einen neuen Stapel Lochkarten gestanzt und – nicht selten über Nacht oder gar für mehrere Tage – einer erneuten Ausführung im Rechenzentrum zugeführt werden.

Ungeachtet der Tatsache, dass die Software dieser Tage weitaus weniger komplex war als aktuelle Produkte, ist heute das feingranulare Testen auf verschiedenen Ebenen, vom Unit- bis hin zum Regressions- und Systemtest, integraler Bestandteil aller großen Softwareprojekte. Schon auf Klassen- und Methodenebene kann so während des Entwicklungsprozesses der geschriebene Code automatisiert getestet werden. Dies dient dazu, den soeben implementierten Teil des abstrakten Modells direkt zu verifizieren gegen eine, wenn auch ebenfalls modellierte, Wirklichkeit.

Damit Softwareentwickler weltweit über das grundsätzlich gleiche Vokabular verfügen, um über wiederkehrende Vorgehensweisen zu sprechen, genügen die Mittel einer Programmiersprache nicht. Hier benötigt man einen gemeinsamen Satz sogenannter *Entwurfsmuster*, systematisch zusammengetragen in „Entwurfsmuster“ der '*Viererbande*'[1] um Erich Gamma.

Bisher wurde sich bei automatisierten Tests auf die Funktionsweise von Softwarekomponenten und

1 Als Viererbande (engl. „Gang of Four“, abgekürzt GoF) werden die vier Autoren von [GH96], Erich Gamma, Richard Helm, Ralph Johnson und John Vissides bezeichnet. Der Begriff GoF-Pattern ist daher eine bekannte Bezeichnung für Entwurfsmuster, die auf das vorgenannte Standardwerk dieser vier Autoren zurückgeht.

deren Zusammenspiel konzentriert. Bislang in den Kinderschuhen steckt jedoch das Testen von Entwurfsmustern im Code, die automatisierte Überprüfung der Einhaltung von Programmierparadigmen sowie Obergrenzen verschiedener Codemetriken. Hier ist der *Architekturtest* prägend als Oberbegriff für solche Arten von Tests. Diese Arbeit hat zum Ziel, unterschiedliche Dimensionen von Architekturtests aufzuzeigen und diese als dynamische Tests gegenüber statischer Codeanalyse abzugrenzen. Darüber hinaus werden ausgewählte Architekturtests in einzelnen Komponenten eines bestehenden, verteilten, produktiv eingesetzten Softwaresystems implementiert und ihr Nutzen für den Entwicklungsprozess und die Softwarequalität bewertet.

1.1 Allgemeine Hinweise

In dieser Arbeit wird zwar ausschließlich die männliche Form verwendet, z. B. der *Entwickler* oder der *Systemarchitekt*, gleichermaßen sind auch stets die weiblichen Formen gemeint, also z. B. die *Entwicklerin* oder die *Systemarchitektin*. Aus Gründen der Einfachheit für den Lesefluss wurde jedoch auf diese Unterscheidung verzichtet. Beim Leser werden für das Verständnis dieser Arbeit Kenntnisse der Programmiersprache Java sowie Grundbegriffe aus der Softwaretechnik und der Mathematik, auch zu Graphen, vorausgesetzt, wie sie in einem Bachelorstudiengang Informatik vermittelt werden.

Im Text werden Codebeispiele immer in *Monospaced* (hier Courier New) sonstiger Fließtext in *Serif* (hier Times New Roman) gesetzt.

1.2 Problemstellung

Die *mybet SE* ist eine europäische Aktiengesellschaft, die Sportwetten und Online-Glücksspiele im Internet anbietet. Darüber hinaus betreibt sie in mehreren Ländern weltweit mehrere hundert Annahmestellen für Sportwetten („Wettläden“).

Die ANYBET GmbH als IT-Dienstleister der mybet-Unternehmensgruppe entwickelt und betreibt die Softwareprodukte und Systeme, um Sportwetten in den Annahmestellen weltweit sowie im Internet anbieten zu können. Dazu gehören im Wesentlichen vier Bestandteile:

1. **Der Backend-Server („mAS“)**, der die Sportereignis- und Wettquotenverwaltung, Wettverrechnung, Benutzerverwaltung und lizenzrechtliche (regionsspezifische) Zugriffssteuerung der Angebote enthält.
2. **Der Frontend-Server („mFES“)**, der die Applikationslogik für die Website, das Content Management, die Darstellung der Events und deren Wettquoten sowie die Anbindung an die Online-Glücksspiele enthält.

3. **Die Frontend-Widget-Bibliothek („mWS“)**, welche einzelne, anpassbare Frontend-Komponenten, bspw. zur Darstellung von Events und Wettquoten auf der Website[2] oder auch zur Wettabgabe bereitstellt. Mittelfristig wird der mFES auf diese Komponenten zurückgreifen.

4. **Der „BetShop“**: Hierbei handelt es sich um die in den Wettannahmestellen eingesetzte Client-Software, welche das komplette Wettangebot inklusive Livewetten bereitstellt, die Wettannahme und Abrechnung enthält und mit dem mAS kommuniziert.

mAS, mWS und mFES sind vollständig in Java (Version 6) implementiert. mAS verwendet JBoss 5[3] als Application Server; mFES verwendet als Web-Framework Apache Tapestry 5[4]. Der BetShop ist hingegen in Microsoft's C# (.NET Framework 3.5.1[5]) geschrieben.

Architekturtests im Sinne dieser Arbeit sind bisher partiell im BetShop erfolgreich eingeführt worden. Der praktische Teil der Arbeit umfasst die exemplarische Einführung vergleichbarer Architekturtests in mAS und mFES bzw. mWS. Hierfür wird zunächst ermittelt, welche Arten von Architekturtests dafür sinnvoll sind und in welchen Teilkomponenten diese integriert werden. Es gilt also, in die bestehende Struktur ein weiteres Teilprojekt 'Architekturtests' einzuführen, das diese Tests auf ausgewählten Komponenten automatisch durchführt und zwar auf derselben Ebene wie bereits vorhandene Unit-Tests. Ziel ist, Fehler bei der Nichteinhaltung von Architekturvorgaben ebenso schnell zu entdecken wie bei der eigentlichen Funktionalität der Software.

1.3 Begriffe

Im Verlauf dieser Arbeit werden immer wieder Begriffe verwendet, für die an dieser Stelle ein einheitliches Verständnis geschaffen werden soll. Am Schluss dieser Arbeit gibt es auch ein Glossar, das dazu dient, die Sportwetten-bezogenen Begriffe zu erläutern.

1.3.1 Programmiersprache

Da mAS und mFES komplett in Java und der BetShop in C# geschrieben sind, beziehe ich mich im Verlauf dieser Arbeit nur auf diese beiden Sprachen. Dort, wo ich mich auf diese beziehe, sind stets

2 z. B. http://www.mybet.com, Zugriff am 04.02.2015.

3 JBoss ist ein Applikationsserver, bestehend aus verschiedenen Komponenten wie *Messaging*, *EJB* (*Enterprise Java Beans*), *JBoss Cache* und *JBoss Transaction* (vgl. http://www.redhat.com/en/technologies/jboss-middleware, Zugriff am 11.02.2015)

4 Siehe http://tapestry.apache.org/, Zugriff am 20.03.2015. Tapestry ist ein quelloffenes Framework zur Erstellung von Web-Applikationen in Java. Es baut auf der *Java Servlet*-Technologie auf und kann so in Anwendungscontainer wie Tomcat oder Jetty integriert werden.

5 Siehe https://technet.microsoft.com/de-de/library/cc772269.aspx, Zugriff am 20.03.2015. Das .NET Framework ist eine umfangreiche Sammlung von Bibliotheken, die zur Erstellung von Applikationen sowohl für Windows als auch für das Web verwendet werden. Sie umfasst u. a. Bibliotheken für GUI-Komponenten (Windows Forms, Windows Presentation Foundation *WPF*), Webseiten (*ASP.NET*), Kommunikation (Windows Communication Foundation *WCF*) und Persistenzschichten (Entity Framework EF).

Java 1.6 (Language Level 6) sowie C# 3.0 (.NET Framework 3.5.1) gemeint. Der Grund hierfür ist, dass die erwähnten Systeme in diesen Sprachversionen geschrieben sind.

1.3.2 Entwurfsmuster

„Christopher Alexander schreibt: 'Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.' [AIS+ +77, S. x].“ ([GH96], S. 3) Diese Definition spricht zwar von Städten und Gebäuden, lässt sich aber auf objektorientierte Entwurfsmuster übertragen (vgl. [GH96], S. 3). Es sind „beide Arten von Mustern als Problemlösungen für bestimmte Situationen zu verstehen.“ ([GH96], S. 3) Das Prüfen bestimmter Entwurfsmuster im Code bildet den ersten zentralen Aspekt bei der Einführung von Architekturtests. Welche Kategorien von Entwurfsmustern für diesen Zweck besonders in Frage kommen, wird in Kapitel 2 genauer behandelt und analysiert.

1.3.3 Framework

Ein Framework stellt eine Sammlung von Bibliotheken mit zu einer Anwendungsdomäne gehörenden, etablierten Funktionalitäten dar. Es kapselt diese gegenüber der verwendenden Anwendung.

1.3.4 Automatisiertes Testen

Automatisiertes Testen umfasst all jene Testaktivitäten, die nicht durch Menschenhand direkt ausgeführt werden. Alle solche Arten von Tests, von *Unit*- über *Integrations*- und *Systemtests* bis hin zu *GUI-basierten Regressionstests*, werden im Verlauf dieser Arbeit als automatisierte Tests verstanden. Hier erstmals verwendete Begriffe werden in Kapitel 2 ausführlich erläutert.

1.3.5 Codemetrik

Unter Codemetriken versteht man eine Reihe von Softwaremaßstäben, die ein Verständnis dafür vermitteln, welche Methoden und / oder Typen überarbeitet werden sollen. Mit ihrer Hilfe lassen sich potenzielle Risiken hinsichtlich der Wartbarkeit von Code früher erkennen und überdies die Qualität des geschriebenen Codes bis hinunter auf Typ- und Methodenebene sicher ermitteln. In dieser Arbeit stellt das Testen von Codemetriken den zweiten zentralen Aspekt bei der Einführung von Architekturtests dar. Welche Codemetriken hierbei im Einzelnen betrachtet werden, wird in Kapitel 2 behandelt.

1.3.6 Architekturtest

Der Begriff des *Architekturtests* im Sinne dieser Arbeit ist ein Oberbegriff für spezielle automatisierte *dynamische* Tests, welche die Einhaltung von Codemetriken, Entwurfsmustern sowie und Programmierkonventionen *statisch* testen.

1.4 Zielskizze und Überleitung

Das Ziel der Arbeit ist die exemplarische Einführung von Architekturtests in komplexe, Java-basierte Frontend- und Backendumgebungen, wobei die bereits gewonnenen Erkenntnisse aus einem dritten Produkt und der darin bereits eingeführten Architekturtests berücksichtigt werden. Insbesondere wird herausgearbeitet, inwiefern sich die Tests für diesen Zweck eignen. Es wird anhand zweier Module der mybet-Plattform als Fallstudie untersucht, welchen Nutzen die Architekturtests als dynamische Tests bringen.

Kapitel 2 stellt den Voraussetzungs- und Analyseteil der Arbeit dar, in welchem zunächst die eingeführten Begriffe präzisiert und danach die vorhandenen Systeme genauer betrachtet werden.

2 Voraussetzungen

Dieses Kapitel befasst sich zunächst mit der Präzisierung der bereits eingeführten Begriffe, hierbei strebe ich vor allem nach einer stärkeren Kontextualisierung in diese Arbeit. Insbesondere die verschiedenen Codemetriken werden erläutert und Entwurfsmusterkategorien auf ihre Relevanz für Architekturtests hin überprüft.

Des Weiteren werde ich hier als Voraussetzung für die Einführung von Architekturtests in die bestehenden Systeme mAS, mFES und mWS diese genauer beschreiben und die bisher eingeführten Architekturtests im BetShop analysieren.

2.1 Begriffe

2.1.1 Architektur

Aus dem Lateinischen übersetzt heißt *architectura* „Baukunst“ und befasst sich zunächst mit der Gestaltung von gebautem Raum. Dazu gehören das planvolle Entwerfen und Gestalten von Gebäuden. In der Softwareentwicklung wird der Begriff der Architektur ebenfalls für planvolles Entwerfen und Gestalten von Software verwendet, jedoch weniger für den Vorgang, als vielmehr für das Ergebnis. So wird unter Software-Architektur das Gesamtkonstrukt des entworfenen Softwareprodukts verstanden. Martin Fowler schreibt jedoch in [FM03], dass Architektur ein Begriff sei, „den zahlreiche Leute zu definieren versuchen, ohne letztlich einen Konsens zu erzielen. Es gibt jedoch zwei gemeinsame Elemente: Erstens: Es geht um die Zerlegung eines Systems in Komponenten auf höchster Ebene. Zweitens: Es geht um Entscheidungen, die sich nur schwer ändern lassen.“ ([FM03], Seite 14) Darüber hinaus befindet er, „dass es nicht nur eine Architektur eines Systems gibt, sondern dass ein System mehrere Architekturen enthalten kann“ ([FM03], Seite 14).

Für den Kontext dieser Arbeit ist es wichtig, Architektur so zu verstehen, dass sie bestimmte Merkmale aufweist, die für das Zusammenspiel der Komponenten von Bedeutung sind. Würden solche Architekturmerkmale zerstört, funktioniert die Softwarekomponente nicht mehr. Folglich muss herausgearbeitet werden, welche Merkmale von Architekturen für automatisierbare Tests herangezogen werden.

2.1.2 Entwurfsmuster

Gamma et al. schreiben in [GH96] über 3 Kategorien von Entwurfsmustern:

- Erzeugungsmuster,
- Strukturmuster und

- Verhaltensmuster.

Zu den Erzeugungsmustern zählen die abstrakte Fabrik, das Erbauermuster, die Fabrikmethode, der Prototyp und das Singletonmuster. Vor dem Hintergrund der automatischen Überprüfbarkeit eignen sich Erbauer und Singleton besonders, da sich deren Merkmale im Code leicht verifizieren lassen. Ein Test, der die Einhaltung der Fabrikmethode prüft, muss die beiden parallelen Klassenhierarchien Produkt und Erzeuger kennen und darin prüfen, ob zu jedem vorhandenen konkreten Produkt auch der richtige Erzeuger verwendet wird.

Den Strukturmustern sind der Adapter, die Brücke, der Dekorierer, die Fassade, das Fliegengewicht, das Kompositum, und der Proxy zuzuordnen. Besonders leicht umzusetzen ist ein Test auf Einhaltung des Kompositums, da seine feste Zuordnung von Objekten zu Baumstrukturen trivial überprüfbar ist. Für den Kontext dieser Arbeit wird jedoch das Fassadenmuster sehr viel relevanter, da dieses Muster in vielen Teilen des mAS eingesetzt wird. Ein Test, der die Einhaltung dieses Musters sicherstellt, muss prüfen, ob Zugriffe auf Objekte „hinter“ der Fassade auch nur über diese erfolgen.

Das Befehlsmuster, der Beobachter, der Besucher, der Interpreter, der Iterator, das Memento, die Schablonenmethode, die Strategie, der Mediator, der Zustand und die Zuständigkeitskette gehören zu den Verhaltensmustern. Insbesondere das Zustandsmuster wird an mehreren Stellen im mAS und im mFES verwendet. Ein Test hierfür würde überprüfen, ob alle definierten Zustände jeweils mindestens einmal im Code erreicht werden.

2.1.3 Codemetriken

In der Physik ist eine Metrik ein Maßzahlsystem, also ein Zahlenwert mit seiner zugehörigen Einheit als physikalische Größe. Hieraus geht schon hervor, dass es sich um etwas numerisch Mess- und Vergleichbares handelt. Grundsätzlich ist also zu sagen, dass Architekturtests, die Codemetriken prüfen sollen, für jede zu überprüfende Metrik bestimmte Grenzwerte benötigen, gegen die sie die ermittelten Ist-Werte vergleichen. Es gibt unterschiedliche Arten von Codemetriken, die in der Literatur auch Softwaremetriken genannt werden. Ich verwende bewusst den Begriff *Codemetrik*, um zu verdeutlichen, dass es sich hier um die codezentrische Betrachtung entsprechender Metriken handelt.

Robert C. Martin nennt zwei Codemetriken, das *Afferent Coupling (Ca)* und das *Efferent Coupling (Ce)*. Afferente Kopplung ist die Anzahl der Referenzierungen anderer Komponenten im Projekt in die Komponente hinein. Efferente Kopplung ist die Anzahl der Referenzierungen aus der Komponente heraus zu anderen Komponenten im Projekt. Beide Kennzahlen lassen sich trivial durch

das Zählen der Import-Statements (in Java) bzw. der Using-Direktiven (in C#) in jeder Klasse des betrachteten Pakets ermitteln. Voraussetzung hierfür ist, dass nicht genutzte Import-Statements bzw. Using-Direktiven vom Code entfernt werden, um die Verlässlichkeit eines solchen Architekturtests sicherzustellen.

Des Weiteren wird die *Zyklomatische Komplexität nach McCabe (MCC)* als Kriterium für die Anzahl möglicher Programmflusszweige herangezogen. Eine hohe McCabe-Zahl deutet auf schwieriger wart- und damit testbaren Code hin, bei dem es sich lohnen kann, ihn in mehrere (Methoden-)Fragmente zu zerlegen.[6]

Die *Halstead-Metrik* wurde 1977 von Maurice Howard Halstead vorgestellt und misst die Komplexität von Software basierend auf der Annahme, dass Programmteile aus Operationen und Operanden aufgebaut sind. Gerade für Java-Code lässt sich das Verfahren leicht berechnen und automatisieren. Es liefert u. a. eine Aussage darüber, wie viele Bugs in geschriebenem Code zu erwarten sind.[7]

2.1.4 Programmierrichtlinien

Wann immer mehrere Entwickler an einem Projekt arbeiten, treffen unweigerlich mehrere, unterschiedliche Programmierstile aufeinander. Formale Sprachen erlauben, zumindest theoretisch, eine unendlich große Zahl an Lösungswegen für jedes gegebene Problem. Dies führt dazu, dass bspw. Entwickler A das selbe Problem anders löst als Entwickler B. Solche Unterschiede schlagen sich im produzierten Code nieder. Hier wird es notwendig, sich auf gemeinsame Richtlinien festzulegen, wie sowohl syntaktisch, als auch (in gewissen Grenzen) semantisch Code zu schreiben ist. Wenn man Programmierrichtlinien vereinbart, wird sich zumeist auf die Lesbarkeit begrenzt. Dies erscheint zwar auf den ersten Blick profan, ist aber ein entscheidendes Merkmal für die Testbarkeit des Codes.

Eine oft verwendete Richtlinie ist, dass Variablen so lokal wie möglich und so global wie nötig deklariert werden sollen. Dies dient der *Schnittstellenminimierung* nach außen und stellt sicher, dass von einem Modul nur das sichtbar ist, was auch sichtbar sein soll. Weitere Richtlinien betreffen die Schreibweise von Klassen-, Feld-, Eigenschafts-, Methoden- und Variablennamen. Hier wird oft der sogenannte „*CamelCase*“ herangezogen, der sich allerdings bei Java und C# geringfügig unterscheidet.

Da eben schon die Lesbarkeit des Codes angesprochen wurde, ist eine Richtlinie, dass Methodenrümpfe eine bestimmte Anzahl von *LoC*'s (*Lines of Code*) und Klassen eine bestimmte

6 Vgl. [MT76].
7 Vgl. [HM77].

Anzahl von Methoden nicht überschreiten sollen. Dies hängt zumindest lose zusammen mit den im vorhergehenden Abschnitt vorgestellten Codemetriken nach Halstead und McCabe. Tendenziell weisen lange Methoden eine höhere zyklomatische Komplexität auf. Inwieweit hier Schnittmengen zu Codemetrik-Tests bestehen und sich Richtlinien-Tests damit zusammenlegen lassen, bleibt in dieser Arbeit zu untersuchen.

Der Umfang einzelner Klassen hinsichtlich ihrer Methodenanzahl ist ein Indiz dafür, ob man die betroffenen Klassen nicht besser modularisieren könnte. Damit ließen sich kleinere Module erreichen, die tendenziell wiederum besser testbar sind. Im folgenden Abschnitt wird darauf noch etwas näher eingegangen.

2.1.5 (Dynamische) Tests

Im Kontext dieser Arbeit meine ich mit *Tests* stets *automatisierte* Tests. Dort, wo davon abgewichen wird, werden andere Tests explizit anders benannt. Auch *Regressionstests* fallen hier in die Kategorie automatisierter Tests. Als Voraussetzung für Tests gilt: „'[..] Improve the process and build quality into the product in the first place.'[8]“ ([HF11], S. 83). Grundsätzlich lässt sich sagen, je früher Tests für ein System in der Implementierungsphase geschrieben werden, um so testbarer ist es. Es gilt also, Qualität bereits im Entwicklungsprozess durch Verbesserung des Prozesses selbst zu etablieren. Weitere Faktoren sind Modularität und Wartbarkeit. Module lassen sich besser testen als das „große Ganze“ und wartbarer Code ist per se automatisch auch testbarer Code, aus einer Vielzahl von Gründen, die in der einschlägigen Literatur hinlänglich behandelt werden. Dies hier dezidiert darzulegen, würde den Rahmen der Arbeit sprengen. Dynamische Tests sind stets sog. *Black-Box*-Tests, d. h. der Quellcode wird bei der Testausführung nicht betrachtet, da das Testobjekt zur Laufzeit untersucht wird.

Die klassische Testeinteilung unterscheidet zwischen *Unit*- und *Integrations-Tests*. Unit-Tests befassen sich mit dem Testen genau einer *funktionalen Einheit* (= engl. *Unit*). Im Normalfall sind dies einzelne Methoden, so dass zwischen Testmethode und Testlingsmethode eine n:1-Beziehung besteht. Eine *vollständige Testabdeckung* einer Klasse durch Unit-Tests besteht dann, wenn jede Methode der Klasse in jeder Ablaufverzweigung durch Testmethoden getestet wird. Dies bedeutet oft, dass zu jeder Methode mehrere Testmethoden existieren. Ein Integrationstest prüft, ob die Kooperation zwischen mindestens zwei funktionalen Einheiten in allen erforderlichen Fällen der Geschäftslogik korrekt abläuft. Beispielsweise wird überprüft, ob eine Klasse A eine Klasse B richtig verwendet, d. h. die Schnittstelle zwischen beiden korrekt ist. Integrationstests arbeiten oft über mindestens eine Architekturschicht, z. B. Geschäftslogik → Datenabstraktion, hinweg. Z. B. testen

8 Zitat im Zitat aus [HF11], Kapitel 4, S. 83: „One of W. Edwards Deming's fourteen points is, '[..]'“

sie, ob Geschäftslogikentitäten korrekt in die darunterliegende Datenbank persistiert werden und umgekehrt, ob diese Entitäten aus der Datenbank gleichermaßen rekonstruiert werden können. Weitere Testarten, über deren Zuordnung sich die vorhandene Literatur aber oft uneins ist, sind Regressions- und Systemtests. Aufgrund dieser Uneinigkeit gebe ich hier mein Verständnis dieser Testarten wieder. Ein Systemtest behandelt eine Funktionalität und verwendet dabei das gesamte System, von der Benutzeroberfläche bis hin zur Datenbank / Dateisystem. Ein Regressionstest wird wiederholt ausgeführt, um auf oberster Ebene sicherzustellen, dass Modifikationen am System keine unerwünschten Effekte haben und immer die gleichen Ausgaben produzieren.

Jutta Eckstein schreibt in [EJ00] im fünften Kapitel, dass „Tests (vor allem Unit-Tests) direktes Feedback zu Design und Code liefern.“ ([EJ00], Seite 131) Dies ist das zentrale Anliegen und enthält schon mindestens zwei der drei Aspekte, die ich mit Architekturtests abdecken will: Entwurfsmuster (*Design*) und Programmierrichtlinien (*Code*). Wenn man '*Code*' noch geeignet erweitert um Codemetriken, so sind in dem Zitat alle drei Aspekte enthalten. Frau Eckstein bezieht sich mit ihrer Aussage auf Unit-Tests. Dies zeigt, dass Architekturtests auf mehr als einer Ebene angesiedelt sind: Betrachten wir Tests für Programmierrichtlinien und Codemetriken, so finden diese sicherlich auf der Ebene von Unit-Tests statt. Tests für Entwurfsmuster hingegen befinden sich auf der Ebene von Integrationstests, da sie das Zusammenspiel der zu testenden Einheiten untersuchen. Vor allem ist festzuhalten, dass Architekturtests nicht für genau eine Klasse oder eine Methode geschrieben sind (1:1- oder n:1-Relation), sondern für mehrere Klassen bzw. Methoden (1:n-Relation). Das wesentliche Unterscheidungsmerkmal ist, dass Architekturtests nicht die Korrektheit der implementierten Funktionalität überprüfen, sondern wie diese Funktionalität implementiert wurde. Folglich ersetzen Architekturtests auch keine Unit- oder Integrationstests, sondern ergänzen diese um das Bestreben, die Wartbarkeit des Codes zu erhalten. Im Idealfall also können Architekturtests neu geschriebene Klassen direkt mit prüfen, ohne, dass sie selbst dafür angepasst bzw. erweitert werden müssen.

Bei der Neuentwicklung eines Systems ist auch für Architekturtests das „Test first“-Prinzip sinnvoll. Man einigt sich im Vorfeld, oder zumindest schon in einer sehr frühen Phase der Entwicklung, auf bestimmte Schwellwerte für Metriken und Programmierrichtlinien, sowie auf die Überprüfung von Mustern und schreibt die entsprechenden Tests. Der damit überprüfte Code wird so bereits von Beginn an getestet, neben den eigentlichen Funktionalitätstests. Die gewünschte Qualität wird nicht nur am Anfang hoch gehalten, sondern über den Projektverlauf hinweg. Wenn ein solcher Test fehlschlägt, deutet dies darauf hin, dass an den getesteten Architekturmerkmalen noch nachgebessert werden muss. Diese Arbeit befasst sich jedoch mit der exemplarischen Einführung von Architekturtests in ein bestehendes, komplexes System, so dass dies zumindest Refactorings der

getesteten Bestandteile nach sich ziehen wird.

Lisa Crispin und Janet Gregory nehmen in „Agile Testing“ auf S. 274 ff eine Einordnung verschiedener Testarten in agilen Entwicklungsumfeldern vor, die „agilen Test-Quadranten“ (vgl. [CG09], S. 275). Abbildung 1 zeigt deren Anordnung im Verhältnis zu den empfohlenen Testmethoden.

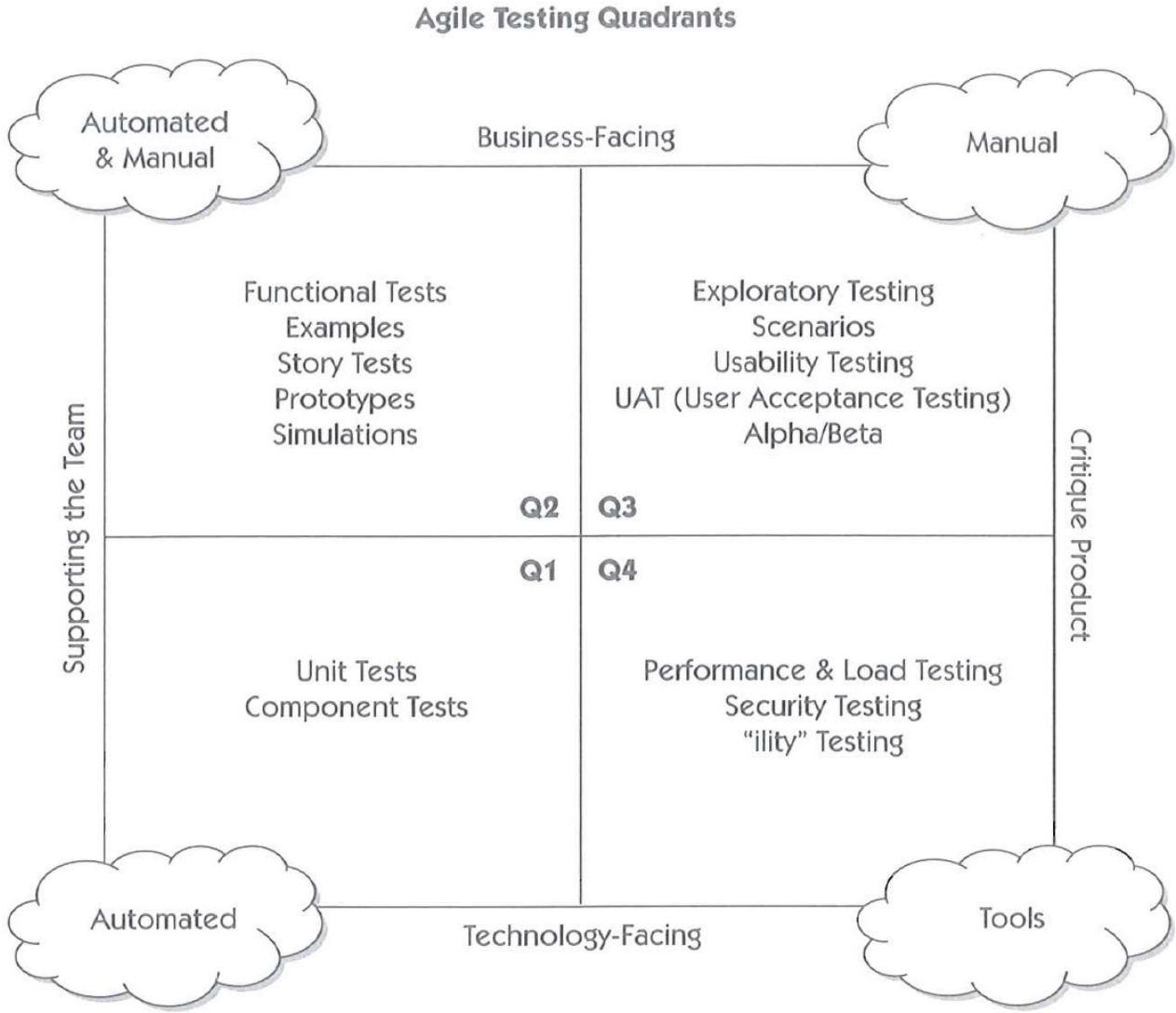


Abbildung 1: Testquadranten (Quelle: [CG09])

Architekturtests im Sinne dieser Arbeit sind zwischen Q1 und Q4 einzuordnen, da sie einerseits auf der Ebene von Unit-Tests ausgeführt werden. Andererseits werden in Q4 auch die „-ilities“ getestet, also auch die „Maintainability“ (Wartbarkeit) des Codes. Schließlich sind sauber eingehaltene Programmierrichtlinien, klar identifizierbare Entwurfsmuster und Werte für verschiedene Codemetriken, die sich innerhalb akzeptabler Grenzen befinden, sehr wesentliche Merkmale für wartbaren Code.

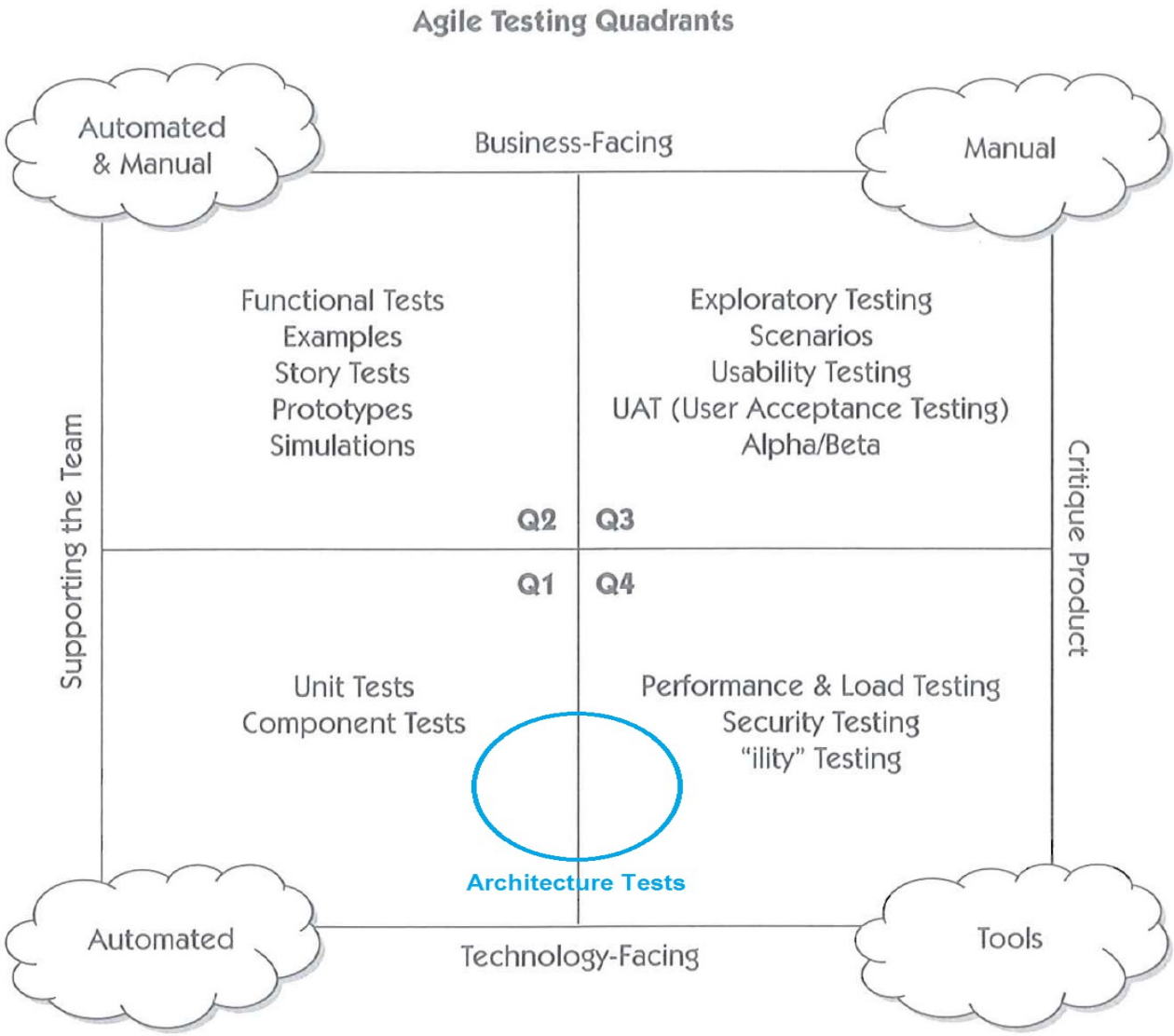


Abbildung 2: Testquadranten mit Architekturtests (Quelle: [CG09] und eigene Darstellung)

Auch an dieser Klassifikation wird deutlich, dass sich Architekturtests im Spannungsfeld zwischen Automation und Tools bewegen. Es muss daher sorgfältig abgewogen werden, was automatisiert getestet werden kann und was sich besser mit Tools zur statischen Codeanalyse durchführen lässt. Die Autorinnen von [CG09] schreiben bezugnehmend auf Quadrant 1, „how can it be coded test-first?“ ([CG09], S. 275), was hervorragend zu Architekturtests passt, da auch sie zuerst vorhanden sein können, um neuen Code direkt dagegen zu entwickeln. Zu Quadrant 4 beziehen sie sich hingegen (auch) auf Lasttests, was hier nicht Fokus der Betrachtungen ist.

Am Rand des Schemas stehen die hier interessanten Begriffe „Supporting the Team“, Technology-Facing“ und „Critique Product“. Alle drei bilden den Rahmen für die Architekturtests, da diese dem Team helfen, Technologie-orientiert sind und gleichzeitig (konstruktive) Kritik am Produkt (auf

Codeebene) üben. Weniger interessant hingegen ist „Business-Facing“, da Architekturtests sich nicht unmittelbar auf einen möglichen Geschäftserfolg auswirken. An dieser Stelle ließe sich zwar argumentieren, dass die durch Architekturtests zu fixierende Codequalität den Lebenszyklus des Produkts dahingehend beeinflusst, dass Code und Architektur langsamer „verrotten“, jedoch erfordert die Überprüfung dieser Vermutung Langzeituntersuchungen, die nicht Bestandteil dieser Arbeit sein können.

Eine weitere Klassifikation, vorgestellt auf S. 276 ff in [CG09], ist die „Test-Automations-Pyramide“ (siehe Abbildung 3). Sie zeigt, dass das Gros der Tests so nah wie möglich am Produkt liegt, also Unit- und Integrationstests, nach oben nimmt die Anzahl ab (Akteptanztests, GUI-Tests, manuelle Tests). Vielerorts sieht die Realität jedoch anders aus. Dies wird in [CG09] gern mit einem „V-Modell“ verglichen, so dass die Pyramide im Wesentlichen auf dem Kopf steht.

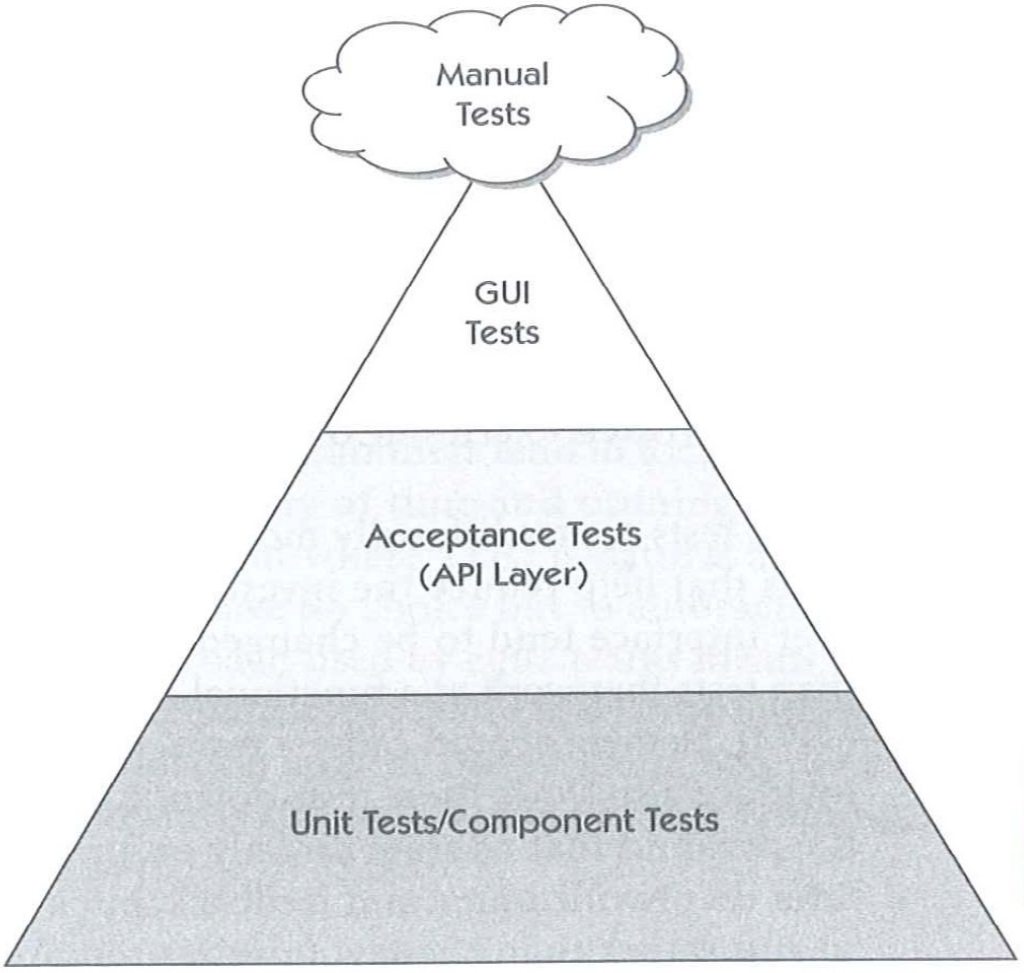


Abbildung 3: Testpyramide (Quelle: [CG09])

Architekturtests werden auf der Ebene von Unit-Tests ausgeführt, so dass sie zum unteren Teil der Pyramide gehören. Man kann sie auch bei den Akzeptanztests zuordnen, da die durch sie getestete Codequalität als Akzeptanzkriterium dient. In Relation zu Abbildung 2 muss hier jedoch klar

einschränkend bemerkt werden, dass der Zusatz „API Layer“ in Abbildung 2.3 entscheidend ist. In Abbildung 2 ist in Quadrant 3 von „User Acceptance Testing“ die Rede, was mit den hier zu entwickelnden Architekturtests nichts zu tun hat. Vielmehr geht es um die Akzeptanz der (Code-)Qualität (daher „API Layer“).

2.1.6 Statische Codeanalyse

In der statischen Codeanalyse wird der zu testende Code nicht (dynamisch) ausgeführt, sondern statisch analysiert. Dies hängt eng zusammen mit dem *Code Review*, in dem durch den / die Entwickler selbst der Code kritisch analysiert wird. Die eigentliche statische Analyse jedoch geschieht durch Softwarewerkzeuge, welche Konventionen und Metriken untersuchen.. Ein Code Review dient dazu, einem anderen Entwickler die eigene Arbeit zu zeigen und eine weitere Meinung hierzu einzuholen. Die eigene „Betriebsblindheit“ verschleiert nur allzu oft den kritischen Blick auf das Geschriebene. Dies belegen zahlreiche Studien über die Effektivität von Code Reviews, wenn diese klaren, messbaren Zielen folgen und der Prozessverbesserung dienen. Messbare Ziele sind z. B. Metriken, so weit möglich, vorher bestimmen, das Review durchführen und danach erneut die Metriken zum Vergleich erfassen. Ein weiteres Ziel effektiver Reviews ist, gefundene Defekte auch tatsächlich zu beheben.

Mittels statischer Analyse lassen sich *Code Smell*-Probleme finden, wie z. B. duplizierter Code, tiefe Verschachtelungen, lange Methoden oder auch große Klassen. Solche Merkmale lassen sich gut in Architekturtests überprüfen, da hierfür kein tieferes Verständnis des Codes erforderlich ist. Die Ermittlung von Metriken, wie der zyklomatischen Komplexität, gehört ebenfalls zur statischen Analyse und lässt sich ebenfalls trivial in Architekturtests implementieren.

Da statische Analyse ein sog. *White-Box*-Verfahren ist, wird der Quellcode benötigt – anders als bei dynamischen Black-Box-Tests, die nur das Ergebnis des Kompiliervorgangs benötigen, um dieses auszuführen. Viele moderne Entwicklungsumgebungen bieten bereits Funktionen zur statischen Codeanalyse an, die direkt beim Schreiben des Codes auf Probleme hinweisen, wie z. B. ignorierte Rückgabewerte oder nicht initialisierte Variablen. Diese ließen sich natürlich auch in Architekturtests realisieren, sind aber unnötig aus oben genanntem Grund. Weiterhin gibt es eine Vielzahl spezialisierter Werkzeuge, die statische Codeanalyse durchführen, als eigenständige Produkte. Einige von ihnen lassen sich sogar in bestehende Buildserver- oder Continuous-Integration-Umgebungen integrieren.

Kritisch in der statischen Codeanalyse sind „falsche Positive“, also Probleme, die keine sind. Da statische Analyseprobleme als nicht entscheidbar gelten, muss entweder das berechnete Modell oder die Analyse selbst approximiert werden (vgl. [BA10], S. 2). Dies kann über das Festlegen geeigneter

Schwellwerte für obere und / oder untere Schranken gelöst werden.

2.1.7 Zwei Welten

Die beiden vorangegangenen Abschnitte 2.1.5 und 2.1.6 betrachten die zwei Herangehensweisen von dynamischen Tests einerseits und statischer Codeanalyse andererseits. Der Beitrag dieser Arbeit besteht darin, beide miteinander zu „verheiraten“, indem statische Codeanalyse innerhalb dynamischer Tests realisiert wird, mit den einhergehenden Vorteilen beider 'Welten'. In beiden Abschnitten wurde festgestellt, dass sowohl dynamische Tests als auch statische Codeanalyse die Verbesserung des Entwicklungsprozesses zum Ziel haben. Beide wollen, dass die gefundenen Defekte und Fehler behoben werden. Dynamische Tests, die Architekturmerkmale, wie in 2.1.1 bis 2.1.4 herausgearbeitet, am Quellcode statisch prüfen sollen, benötigen Zugriff auf ebendiesen Quellcode. Das unterscheidet Architekturtests von anderen dynamischen Tests grundlegend.

Allerdings darf nicht unerwähnt bleiben, dass weder statische Codeanalyse noch Architekturtests Code Reviews ersetzen wollen oder können. Sie können diese jedoch unterstützen, bspw. durch Testausführung vor und nach dem Review zur Untersuchung der Verbesserungen. Spillner und Linz schreiben in [SA05] auf S. 94, dass statische Codeanalyse bereits vor einem Review hilft, die Menge der zu betrachtenden Aspekte im Review zu verringern.

„Grundsätzlich sollte die statische Analyse durchgeführt werden, auch wenn kein Review geplant ist. Jede entdeckte Unstimmigkeit erhöht die Qualität des analysierten Dokuments.“ ([SA05], S. 95) Es ist daher angeraten, Architekturtests, die eine statische Analyse durchführen, regelmäßig und so oft wie möglich ausführen zu lassen. „Mit der statischen Analyse lassen sich allerdings nicht alle Fehler und Unstimmigkeiten nachweisen. Es gibt Fehler oder präziser Fehlerzustände, die erst bei Ausführung, also zur Laufzeit des Programms, zur Wirkung kommen und vorher nicht ermittelbar sind.“ ([SA05], S. 95) Aus diesem Grund *ergänzen* Architekturtests stets nur andere Tests, vor allem andere dynamische Tests, die die Funktionalität und das Zusammenwirken von Komponenten untersuchen, sie *ersetzen* sie jedoch nicht.

Zusammengefasst bedeutet dies für Architekturtests, dass sie die folgenden Eigenschaften und Ziele haben:

1. Architekturtests betreiben statische Codeanalyse; sie sind *Codeprüfer*.
2. Architekturtests haben als statische Analysetools auch dieselben Ziele wie die statische Codeanalyse.
3. Architekturtests sind implementiert als spezielle dynamische Tests, die den Quellcode benötigen.

4. Architekturtests sind Unit-Tests, die jede zu testende Klasse separat untersuchen.
5. Architekturtests ersetzen kein Code-Review, sie unterstützen es, genau so wie dies statische Codeanalysewerkzeuge tun.
6. Architekturtests sind als Tests, z. B. auf der Grundlage von JUnit, in derselben Sprache (hier Java) geschrieben wie andere dynamische Tests – sie sind daher transparent allen Entwicklern zugänglich und durch sie auch wartbar.

Diese Eigenschaften und Ziele werden im weiteren Verlauf der Arbeit verfolgt, insbesondere, wenn es darum geht, ein Framework zu konzipieren und zu entwickeln, das solche Architekturtests zur Verfügung stellt. An dieser Stelle sei darauf hingewiesen, dass es keinen Codeprüfer gibt, der Entwicklern garantieren kann, dass ein Programm korrekt sei (vgl. [BA10], S. 2). „'In fact, no code checker is complete or sound. A complete code checker would find all errors, while a sound code checker would report only real errors and no false postitives.'“ ([BA10], S. 2) Die Erwartung an die Architekturtests geht daher nicht über die statische Codeanalyse hinaus – ist jedoch bereichert um die Vorteile, dass sie genauso oft ausgeführt werden wie andere Unit-Tests, keine zusätzlichen Werkzeuge nötig sind und sie allen Entwicklern transparent zur Verfügung stehen.

2.2 Werkzeuge

Bei mybet geschieht die Softwareentwicklung in Java primär mittels IntelliJ IDEA (ab Version 13.x) als IDE, Apache *Maven* (ab Version 3.x) als Buildmanager sowie *Jenkins* (ab Version 1.5xx) als Continuous-Integration-Server. Aus diesem Grund sind diese Werkzeuge hier kurz beschrieben. Wo sinnvoll, werden Querbezüge dieser Produkte untereinander sowie zu Aspekten dieser Arbeit hergestellt.

2.2.1 Maven

Apache Maven ist ein Buildmanagement-Tool, das unter der *Apache License 2.0*[9] frei verfügbar und quelloffen ist. Bei der Entwicklung von Maven wurde konsequent das Prinzip „*Convention over Configuration*“ (oft auch „*Coding by Convention*“ genannt) umgesetzt. Dies bedeutet, dass sich alle Beteiligten an gemeinsame Richtlinien halten, wie z. B. gleichartige Bezeichner, um diese nicht

9 Siehe http://www.apache.org/licenses/LICENSE-2.0.html, Zugriff am 01.03.2015. Die Apache License 2.0 ist eine durch die *Free Software Foundation* anerkannte Lizenz für freie Software.

konfigurieren zu müssen. Frameworks wie *Spring*[10], *Hibernate/JPA*[11] oder *Ejb3Unit*[12] verwenden diesen Ansatz ebenfalls. Hier wird sehr schnell auch die technische Erfordernis von Programmierrichtlinien deutlich, um Tools wie Maven effizient nutzen zu können.

Maven implementiert standardmäßig die klassischen Schritte im Software-Lebenszyklus, ohne dass diese in jedem Projekt immer durchlaufen werden müssen. Folgt man den durch Maven vorgegebenen Standards, so müssen vom Anlegen eines Projektes bis hin zum Verändern der Projektkonfiguration und Bereitstellen der Applikation nur wenige Konfigurationsarbeiten durchgeführt werden. Diese Konfiguration wird pro Projekt zentral in einer XML-Konfigurationsdatei (*pom.xml*) abgelegt. Maven erlaubt auch die Verwendung von Projekt*hierarchien*, indem einem Projekt auch ein Elternprojekt zugeordnet werden kann. Als Beispiel wird hier das zu erstellende Architekturtests-Projekt genannt, dass dem Projekt *test* zugeordnet ist, welches wiederum dem Stammprojekt *mybet-project* untergeordnet ist.

2.2.2 IntelliJ IDEA

Die integrierte Entwicklungsumgebung (engl. „Integrated Development Environment“, IDE) *IntelliJ IDEA* des Herstellers *JetBrains* wurde selbst in Java geschrieben und ist für die Softwareentwicklung in dieser Sprache vorgesehen. Es gibt sie seit Version 9.0 in zwei Varianten: Einer proprietär lizenzierten *Ultimate* sowie einer unter der Apache License 2.0 lizenzierten, quelloffenen *Community* Edition. Die Ultimate Edition enthält den Funktionsumfang der Community Edition und ist überdies in der Lage, auch andere Sprachen neben Java zu unterstützen. Eine hervorzuhebende und in frühen Versionen schon vorhandene Stärke von IntelliJ IDEA liegt in der Unterstützung des Entwicklers beim Refactoring von Code. Bereits 2001 schrieb Martin Fowler in [FM01] im Abschnitt „Crossing the Rubycon“, dass IDEA die „Extract Method“-Funktion unterstütze, womit er meint, dass ein markiertes Codefragment mit wenigen Mausklicks als eigene Methode extrahiert werden kann, unter Berücksichtigung von Parametern und Rückgabewerten.

Neben verschiedenen, verbreiteten Application-Frameworks wie Spring, *Struts*[13], Tapestry oder

10 Das *Spring*-Framework dient zur Entwicklung von Anwendungen auf Basis von Java EE, macht dieses jedoch deutlich einfacher und entkoppelt Anwendungskomponenten durch *Dependency Injection* / *Inversion of Control*. Es bietet Applikationsvorlagen, die das Verwenden der APIs von Java EE vereinfachen und z. B. Fehlersituationen, wie nicht geschlossene Ressourcen, automatisch bereinigen. Siehe http://spring.io/, Zugriff am 20.03.2015.

11 Siehe http://hibernate.org/, Zugriff am 20.03.2015. Die *Java Persistence API* mit *Hibernate* ist ein Framework zur standardisierten Persistierung von Java-Objekten in eine relationale Datenbank. Man spricht hierbei von *Object Relational Mapping (ORM)*.

12 EJB3Unit ist ein Unit-Testframework speziell für Enterprise Java Beans 3.0 und automatisiert das Testen von Entities und Session Beans. Siehe http://ejb3unit.sourceforge.net/, Zugriff am 20.03.2015.

13 Siehe https://struts.apache.org/, Zugriff am 20.03.2015. Apache Struts ist ein MVC-basiertes Framework zur Entwicklung von Web-Applikationen. Bei mybet ist das mAS-BackOffice mit Struts entwickelt worden.

Hibernate/JPA und Application-Servern wie *Tomcat*[14], *Jetty*[15] oder JBoss, unterstützt IntelliJ IDEA auch das Buildmanagment-Tool Apache Maven.

2.2.3 Jenkins

Jenkins ist ein Fork des *Hudson*-Projektes, das ursprünglich bei Sun Microsystems entwickelt worden war. Nach der Übernahme von Sun durch Oracle verließ der verantwortliche Entwickler das Unternehmen und hat seither das Projekt eigenständig vorangetrieben. Jenkins ist ein Continuous-Integration-Server, der ein Web-Interface zur Verfügung stellt, mit dem Build- und Testvorgänge konfiguriert und Build- und Testergebnisse überwacht werden können. Das Ziel ist, die Architekturtests genau so überwachen zu können wie alle anderen funktionalen Tests. In Abbildung 4 ist ein Screenshot von Jenkins mit ersten Architekturtests zu sehen; Details des Reportings der Architekturtestergebnisse folgen in den Kapiteln 4 und 5.

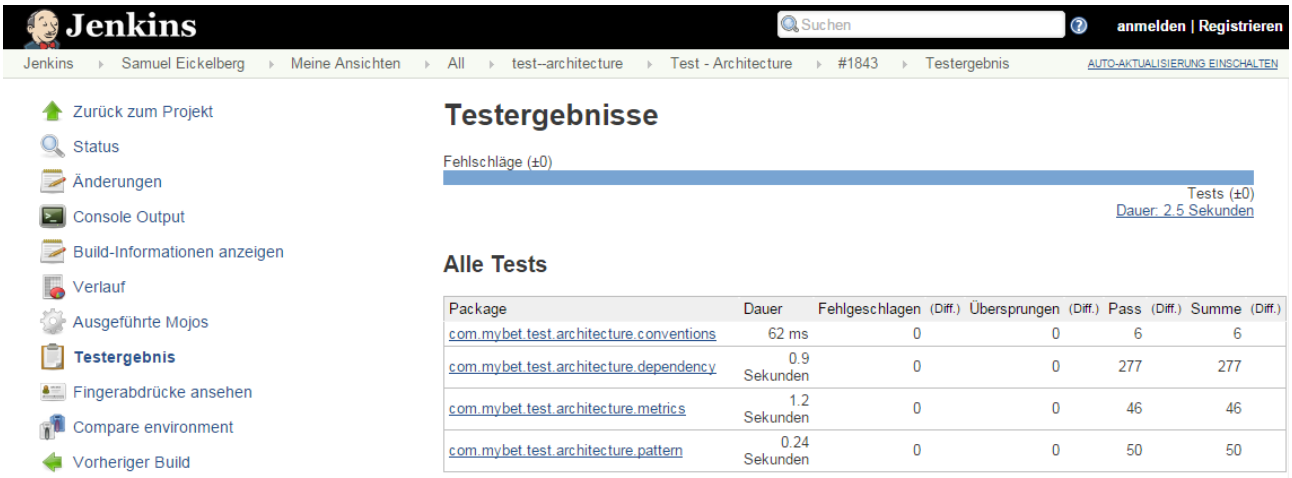


Abbildung 4: Jenkins, Ansicht 'Testergebnisse' (Quelle: Eigene Darstellung)

2.3 mybet-Software

Die durch die ANYBET GmbH entwickelte und betriebene Software ist in Abbildung 5 durch blaue Boxen dargestellt. Die Kunden von mybet greifen von außen lediglich auf den mFES zu; die Buchmacher verwenden ebenfalls einen Browser, um auf die Verwaltungsplattform im mAS zuzugreifen. Das mWS ist das „*mybet Widget System*“, dessen Komponenten schrittweise die monolithisch konzipierte GUI des mFES mittelfristig ablöst. Mittels *node.js*[16] werden die

14 Apache Tomcat ist ein Webserver und Applikationscontainer. Er ist eine quelloffene Implementierung der *Java Servlet*- und *Java Server Pages*-Tecnologien. Siehe http://tomcat.apache.org/, Zugriff am 20.03.2015.

15 Jetty ist, ebenso wie Apache Tomcat, ein Webserver und Applikationscontainer und nutzt sogar denselben JSP-Compiler wie Tomcat. Er ist jedoch kleiner und modularer aufgebaut, was ihn leicht integrierbar macht in andere Software. Siehe http://eclipse.org/jetty/, Zugriff am 20.03.2015.

16 *node.js* ist ein Framework zur serverseitigen Programmierung von Anwendungen mit JavaScript. Es basiert auf der Chrome V8 JavaScript-Runtime. Siehe https://nodejs.org/ (Zugriff am 19.03.2015).

Quotenupdates als kompakte Datenpakete dem Browser bereitgestellt. Diese müssen per JMS[17] vom mAS an den mFES geschickt werden, der diese wiederum mittels node.js an die Client-Browser ausliefert. Innerhalb des mAS sind die B2B-API und der Shop-Service angesiedelt. Die B2B-API ist eine Programmierschnittstelle zum Einen für den Abruf des Wettprogramms und zum Anderen für die Abgabe von Wetten. Der Shop-Service ist eine SOAP[18]-basierte Schnittstelle für den BetShop, der die Anmeldung und Wettabgabe für die Wettläden bereitstellt.

In den Abbildungen 5 und 10 ist das als .zip-Datei alle 15 Minuten erzeugte Wettprogramm noch als extra Block zu sehen. Dies liegt daran, dass es derzeit noch parallel mit der dynamischen Generierung aus der B2B-API vom BetShop, der in einem Teil der Wettläden noch nicht auf die B2B-API zugreift, und vom mFES verwendet wird. Mittelfristig wird das Wettprogramm jedoch abgelöst werden, wenn alle Wettläden und der mFES ausschließlich auf die B2B-API zugreifen.

Die mybet-Softwareinfrastruktur folgt, im Großen betrachtet, einer klassischen 3-Schicht-Architektur, in der die GUI-Ebene durch mFES (Web) bzw. BetShop (Wettläden), die Geschäftslogik-Ebene durch mAS und die Datenebene durch die DB repräsentiert werden.

2.3.1 mAS

Der mAS bildet den Kern der mybet-Softwareinfrastruktur. Unter Anderem verarbeitet er Wettabgaben, steuert Benutzerzugriffe, stellt Quotenupdates für Mainbook- und Livewetten bereit und verteilt diese. Er stellt das Wettprogramm bereit und verrechnet die Märkte, damit Kunden ihre Wetten entweder verloren oder gewonnen haben und im Falle der gewonnenen Wette ihren Gewinn ausgezahlt bekommen können.

Im mAS befindet sich auch das sog. *BackOffice*, die Web-Applikation für die Verwaltung der Plattform. Diese wird primär durch die Buchmacher genutzt, um das Wettangebot zu verwalten, d. h. Events zu veröffentlichen, Märkte für die Events festzulegen und in den Märkten Outcomes mit Quoten zu versehen. Daneben wird es auch durch die Marketingabteilung genutzt, um kundenorientierte Aktionen sowie Web-Inhalte zu verwalten, d. h. deren Sichtbarkeit auf der Website zu steuern und die Inhalte zu pflegen. Die Kundensupport-Abteilung verwendet das Backoffice ebenfalls, um Wettläden zu verwalten und beispielsweise bei Beschwerden Einblick in die abgegebenen Wetten zu erhalten.

---

17 Der *Java Messaging Service* (JMS) ist die Standard-API, die zur *Java Enterprise Edition* (JEE) gehört, für das Versenden und Empfangen von Nachrichten zwischen JEE-basierten Komponenten einer verteilten Anwendung. Er erlaubt die asynchrone, lose gekoppelte und verlässliche Kommunikation zwischen ihnen (vgl. http://www.oracle.com/technetwork/java/jms/index.html, Zugriff am 12.02.2015).

18 *Das Simple Object Access Protocol* (SOAP) setzt auf HTTP bzw. HTTPS auf und dient zum Austausch XML-basierter Nachrichten zwischen Systemen. Es hat seine Grundlage in XML-RPC und stellt Regeln für das Design von Nachrichten auf und wie diese zu interpretieren sind (vgl http://www.w3.org/TR/soap/, Zugriff am 12.02.2015).

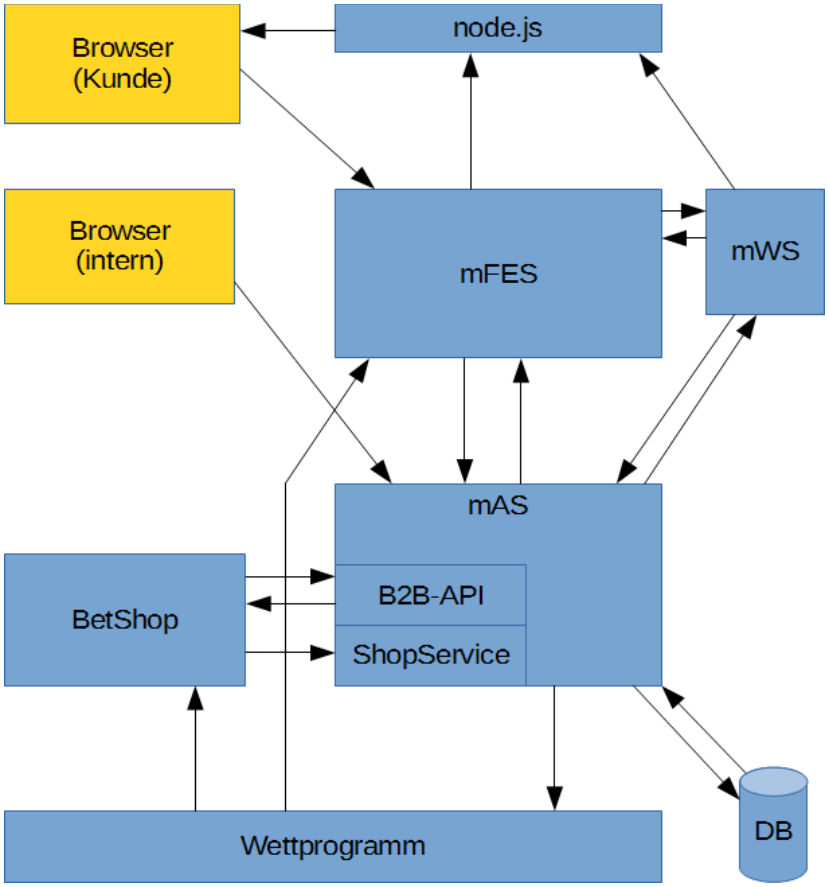


Abbildung 5: Softwareinfrastruktur bei mybet (Quelle: Eigene Darstellung)

Der mAS wurde auf Basis der Middleware JBoss Version 5 entwickelt. Er verwendet daraus unter anderem das Messaging, den Cache und EJB 3.0.

2.3.2 mFES

Der mFES bildet das Web-Frontend für die Kunden, welches über mybet.com[19] und mybet.de[20] im WWW erreichbar ist. Er wurde mit dem Ziel entwickelt, die Geschäftslogik von der GUI zu trennen und die Last von den mAS-Servern zu nehmen, indem er das Wettprogramm vom mAS sowie die Inhalte der Website in einem eigenen Cachingmechanismus vorhält. Mittels Push-Mechanismus hat er ausserdem die Aufgabe, Nachrichten vom mAS an node.js und damit an die Browser der Kunden auszuliefern. Die Produktsichtbarkeiten (z. B. für Mainbookwetten, Livewetten, Casino, Spiele) holt er sich vom mAS in Abhängigkeit des angemeldeten Kunden. Da gibt es regionale und / oder lizenzbedingte Unterschiede, so dürfen z. B. Kunden aus Deutschland unter mybet.de Poker nur im „free“-Modus spielen und daher keine „Poker-Dollar“ erwerben. Die Kunden verwalten ihr

19 https://www.mybet.com/ ist das Angebot für die internationalen Kunden, lizenziert durch die Malta Gaming Authority. Zugriff am 05.12.2014.

20 https://www.mybet.de/ ist das Angebot für die Kunden mit Wohnsitz in Deutschland, lizenziert durch das Innenministeruim des Landes Schleswig-Holstein. Zugriff am 05.12.2014.

Benutzerkonto inklusive ihrer persönlichen Daten oder auch einer beabsichtigten Spielerlimitierung (Sperrphase zum Selbstschutz vor möglicher Glücksspielsucht) im „Mein Konto“-Bereich in mFES. Die Benutzerkonten selbst sind jedoch auch im mAS steuerbar und werden in der Datenbank gespeichert.

mFES wurde als Tapestry[21]-IoC[22]-basierte Applikation entworfen und entwickelt. IoC bedeutet, dass die konkrete Implementierung immer erst zum Zeitpunkt der Initialisierung ausgewählt wird, so dass sämtliche Komponenten gegen definierte Interfaces des Tapestry-Frameworks entwickelt wurden.

Die Basisstruktur der Seitenkomponenten ist in Abbildung 6 zu sehen als Klassendiagramm. Die Seite ist hierarchisch aufgebaut aus „renderbaren“ Bestandteilen (Basisklasse `BaseRenderable`). In `BaseRenderable` befinden sich `@Inject`-Annotationen mit lediglich den Interfaces, zu denen erst zur Initialisierung die konkreten Implementierungen geladen werden. Der modernere (und laut Martin Fowler auch der präzisere) Begriff von IoC ist *Dependency Injection*, einem Prinzip, dem auch Frameworks wie Spring folgen.

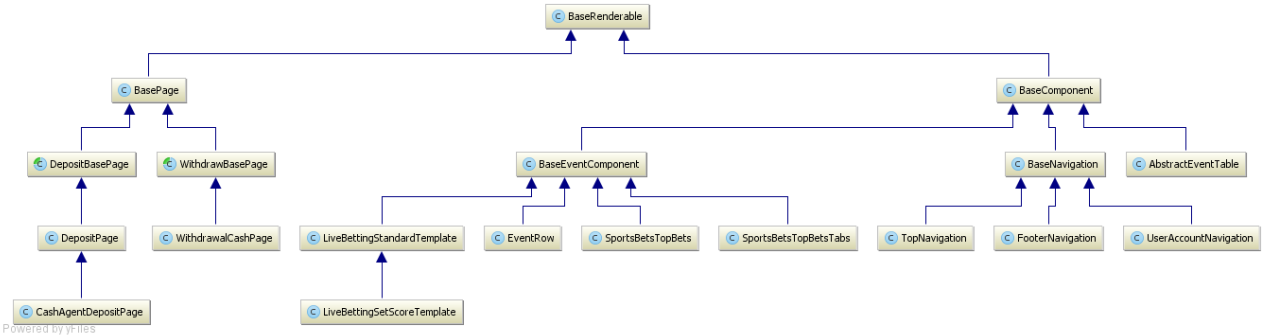


Abbildung 6: Vererbungsstruktur der Seitenelemente des mFES (Quelle: Eigene Darstellung)

Erbend von `BaseRenderable` gibt es `BasePage` und `BaseComponent`. In Abbildung 6 sind nur exemplarisch die `DepositBasePage` (für Einzahlungen) und `WithdrawalBasePage` (für Auszahlungen) dargestellt. Von `BaseComponent` erben bspw. `BaseEventComponent`, welche die Event-bezogenen Seitenbestandteile rendert und `BaseNavigation`, von der z. B. `TopNavigation`, `FooterNavigation` oder `UserAccountNavigation` erben. Abbildungen 7 und 8 zeigen die tatsächlichen Darstellungen einiger der genannten Komponenten.

21 *Tapestry* ist ein Open-Source-Framework zur Erstellung von Web-Applikationen in Java. Durch seine komponentenbasierte Architektur trennt es strikt zwischen Darstellung und Inhalt und bringt seine eigene IoC-Implementierung mit (vgl. http://tapestry.apache.org/index.html, Zugriff am 13.02.2015).

22 *Inversion of Control* (IoC), dt. „Steuerungsumkehr“, ist ein Entwurfsparadigma, das darauf beruht, dass Module einer Applikation bei einer Standardbibliothek registriert werden und erst zu einem späteren Zeitpunkt aufgerufen wird. Das *Beobachter*-Muster folgt diesem Ansatz beispielsweise.

Abbildung 7: Lage einzelner Seitenelemente im Bereich 'Mein Konto' auf mybet.com (Quelle: Eigene Darstellung)

Komponenten wie `TopNavigation` ändern ihr Verhalten und das, was sie anzeigen, kontextabhängig, wie in Abbildung 8 zu sehen ist.

Abbildung 8: Lage einzelner Seitenelemente im Bereich 'Sportwetten' auf mybet.com (Quelle: Eigene Darstellung)

2.3.3 BetShop

In diesem Abschnitt wird die Funktionalität des BetShops nur marginal beschrieben; wesentlich sind hier die Architektur sowie die bereits vorhandenen Architekturtests. Beiden sind hier eigene Unterabschnitte gewidmet.

2.3.3.1 Architektur

Der BetShop ist nach dem MVVM-Entwurfsmuster[23] aufgebaut. Dies ist die beste verfügbare

23 Das Model-View-ViewModel (MVVM) Entwurfsmuster wird vorwiegend in der Anwendungsentwicklung im .NET-Umfeld angewandt. Es wurde 2005 von John Gossman entwickelt und ist identisch mit Martin Fowler's

Herangehensweise, um darstellungsrelevanten Code von der Geschäftslogik zu trennen mit der Grundidee, dass die Präsentation nicht über die Geschäftslogik zu wissen braucht. So können Tests einerseits die GUI und ihr Verhalten testen, ohne dass eine konkrete Geschäftslogik dafür in Anspruch genommen wird. Andererseits können andere Tests die Geschäftslogik verifizieren ohne dafür eine GUI zu benötigen. Abbildung 9 zeigt die Module, aus denen der BetShop besteht.

BetShopTerminal enthält die Benutzeroberfläche und die Logik für die Terminalapplikation, die dem darunterliegenden BetShopProgram hinzugefügt werden. *BetShopProgram* ist der Ausgangspunkt für den BetShop und ist die normale Kassensoftware, wenn der nicht über BetShopTerminal gestartet wird. *Print* enthält alle Funktionen zur Ansteuerung von Druckern, u. a. Bondruckern. *PosLibrary* enthält alle Funktionen zur Ansteuerung von Münz- und Scheinprüfern sowie Kartenlesern. *Scan* enthält alle Funktionen zur Ansteuerung von Scannern zum Einlesen von Papierwettscheinen. *LiveViewerProgram* kann sowohl von BetShopProgram in einem eigenen Thread, als auch als unabhängige Applikation gestartet werden und zeigt die Quoten von Live-Events an, bspw. für die gut sichtbare Darstellung auf Videowänden im Wettladen. *ControlLibrary* enthält alle GUI-Elemente, die in BetShopTerminal, BetShopProgram oder in LiveViewerProgram verwendet werden. *DataTransfer* sind alle Funktionen zusammengefasst, die zur Kommunikation mit anderen Systemen im Netzwerk dienen, wie dies bei mehreren Stationen mit dem BetShop der Fall ist. *Global* enthält alle weiteren Funktionen, die von allen anderen Modulen benötigt werden, unter Anderem das POCO[24]-Datenmodell und die Berechnungslogik für Wettarten und -scheine.

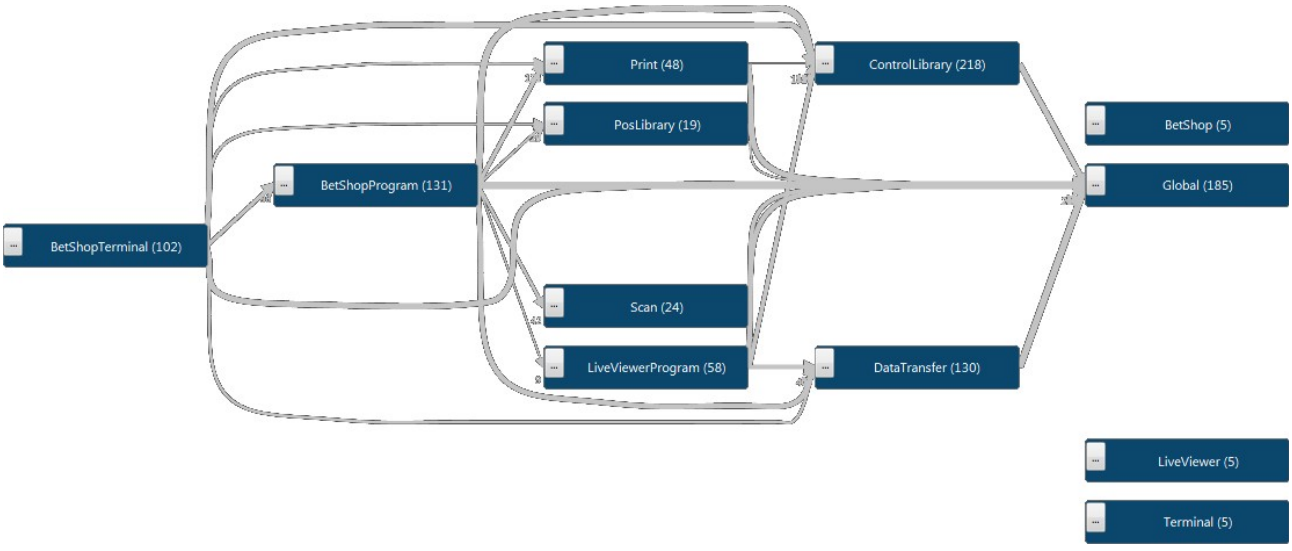


Abbildung 9: Blockdiagramm der Module des BetShop (Quelle: Interne Dokumentation)

Die hier beschriebene Modularisierung wird durch die speziellen Architekturtests, wie im folgenden

PresentationModel-Muster (vgl. [SJ09]). Die *Windows Presentation Foundation* (WPF) macht MVVM zur Verkehrssprache bei der Strukturierung einer Anwendung (vgl. [SJ09]).

24 POCO ist der analoge Begriff zu POJO: *Plain Old C# Object*.

Abschnitt beschrieben, fixiert. BetShop, LiveViewer und Terminal sind außerhalb der Modulstruktur zu betrachten, da sie die jeweiligen Launcher (.exe) sind, die BetShopProgram, LiveViewerProgram bzw. BetShopTerminalProgram starten. Diese ändern ihre Version und Signatur nie, so dass nicht nach jedem Update in Windows die Ausführung durch den Beuntzer bestätigt werden muss.

2.3.3.2 Architekturtests

Die im BetShop vorhandenen Architekturtests beziehen sich auf Codemetriken und Abhängigkeiten sowohl innerhalb der Geschäftslogik als auch der GUI. Dabei werden u. a. die zyklomatische Komplexität (nach McCabe), die Anzahl von Parametern in Methoden, Methodenlänge und die Anzahl von Methoden in einer Klasse gemessen. Die Abhängigkeiten der Komponenten untereinander werden spezifisch für den BetShop fixiert, d. h. Tests, die dies prüfen, enthalten explizit Referenzen auf erlaubte bzw. nicht erlaubte Komponenten. Weiterhin wird untersucht, ob zyklische Abhängigkeiten vorhanden sind. Die nachfolgende Tabelle enthält eine Übersicht der im BetShop vorhandenen Architekturtests.

Testname	Funktionen (auszugsweise)
`CodingConventionsTest`	Programmierrichtlinien, wie CamelCase, Sichtbarkeit (`public`, `protected`, `private`)
`MethodQualityTest`	Zyklomatische Komplexität in Methoden, maximale Schachtelungstiefe, maximale Parameterzahl
`FileOperationsTest`	Festgelegte Speicherorte für Konfigurationsdateien
`LackOfCohesionTest`	Starke Typkohäsion (bspw. keine Codeduplizierung), d. h. hohe Abhängigkeit zusammengehöriger Module untereinander
`FindUnreferencedElementsTest`	Finden ungenutzter Referenzen in Klassen
`BetShopForbiddenReferencesTest`	Festlegung nicht erlaubter Referenzen
`BetShopNamespaceHierarchyTest`	Erzwingung der vorgegebenen Typhierarchie
`ApprovedNamespacesTest`	Festlegung erlaubter Namespaces[25]
`DataTransferAccessTest`	Festlegung, welche Komponenten auf die `DataTransfer`-Bibliothek (enthält sämtliche Komponenten zur Datenübertragung zum Shop-Service und zur B2B-API) zugreifen dürfen
`TypeComplexityTest`	Zyklomatische Komplexität in Klassen, maximale Anzahl Felder, maximale Anzahl Methoden.
`ControlLibraryAccessTest`	Festlegung, welche Komponenten auf die `ControlLibrary`-Bibliothek (enthält sämtliche UI-Controls des BetShop) zugreifen dürfen

25 Siehe https://msdn.microsoft.com/de-de/library/bb979264.aspx, Zugriff am 20.03.2015: „Namespaces bieten eine übersichtliche, hierarchische Strukturierung funktional zusammengehöriger Klassen und Strukturen an. Ihre optionale Verwendung macht Code übersichtlicher und vereinfacht seine Wiederverwendung.“

Die Architekturtests im BetShop betrachten im Wesentlichen Abhängigkeiten, wie erlaubte und nicht erlaubte Referenzierungen sowie Programmierrichtlinien und Codemetriken. Tests auf die Einhaltung bestimmter Entwurfsmuster sind hierin nicht enthalten.

2.3.4 Test-Infrastruktur

Bei mybet besteht derzeit die in Abbildung 10 dargestellte Test-Infrastruktur. Ziel dieser Arbeit ist, die Architekturtests so in die Test-Infrastruktur zu integrieren, dass diese nicht beeinflusst wird. Links der gestrichelten Linie sind die Zusammenhänge der Softwareinfrastruktur zu sehen (blaue Boxen). Rechts der gestrichelten Linie in den grünen Boxen sind die Testarten. JMeter[26] und SoapUI[27] werden für Lasttests verwendet (orange Boxen) und werden hier nicht weiter betrachtet. Die Architekturtests werden, wie bereits klar gestellt wurde, innerhalb der Unit-Tests laufen.

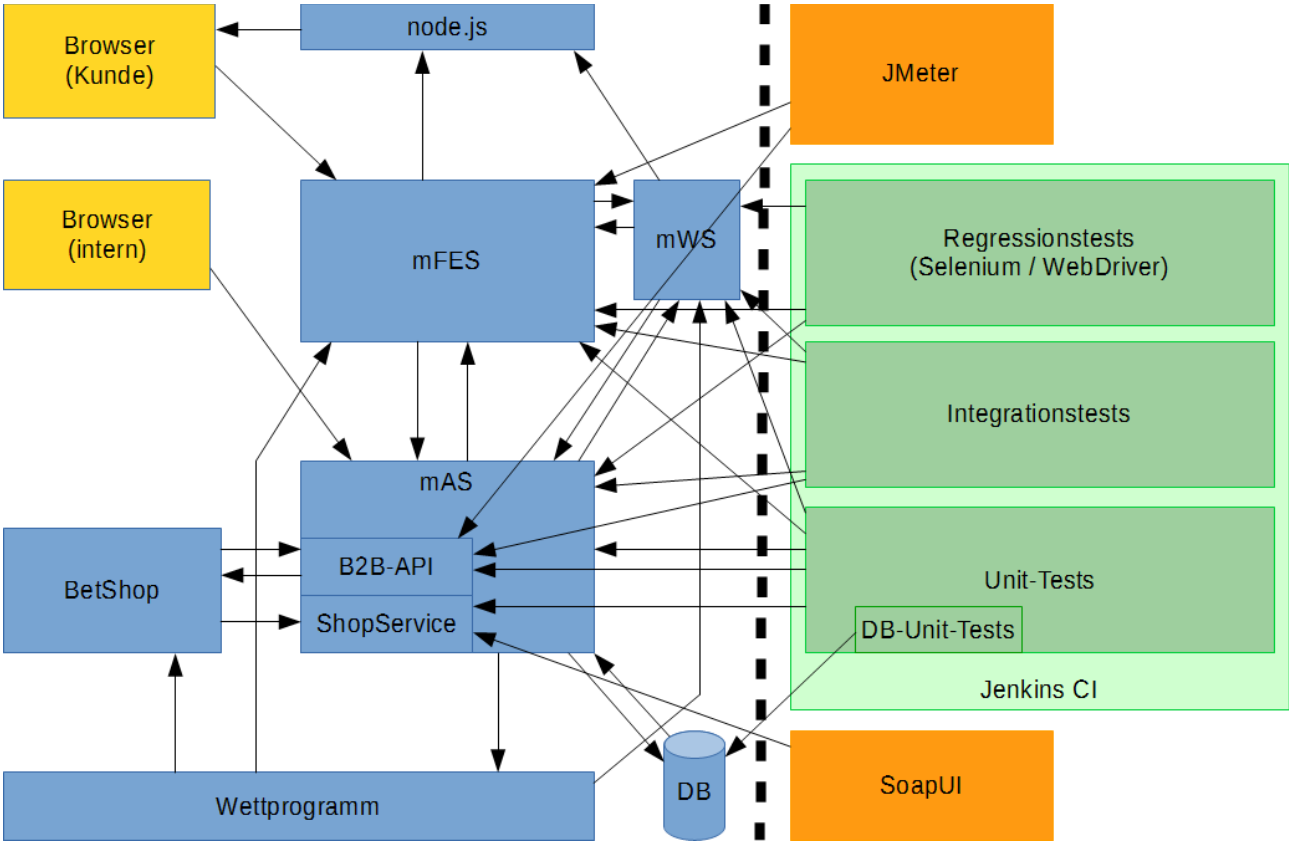


Abbildung 10: Testinfrastruktur bei mybet (Quelle: Eigene Darstellung)

2.4 Zusammenfassung und Überleitung

Bei der Betrachtung der Voraussetzungen wurde festgestellt, dass Architekturtests auf der Ebene von

26 „Apache JMeter may be used to test performance both on static and dynamic resources (Webservices (SOAP/REST), Web dynamic languages - PHP, Java, ASP.NET, Files, etc. -, Java Objects, Data Bases and Queries, FTP Servers and more)“ (http://jmeter.apache.org/, Zugriff am 20.03.2015).

27 Siehe http://www.soapui.org/, Zugriff am 20.03.2015. SoapUI ist ein Werkzeug zur Erstellung funktionaler Tests für SOA-basierte Schnittstellen, v. a. SOAP. Es ist vollständig in Java implementiert.

Unit-Tests laufen. Zu Beginn des Abschnitts 2.1.5 wurde herausgehoben, dass testbarer Code gleichzeitig wartbarer Code ist. *Architekturtests* im Sinne dieser Arbeit dienen dazu, Code in einer Weise zu betrachten, wie es, stark vereinfacht ausgedrückt, bei einem *Code Review* geschehen würde.[28] Natürlich nicht mit der intellektuellen Schärfe und Erfahrung, wie es ein Entwickler, der das Code Review durchführt, tun würde, aber die Aufgabe ist beiden gemein: Architekturtests finden Schwächen im *Code-Design*, während alle anderen Arten von Tests Schwächen in der *Funktionalität* finden. Je weiter die Entwicklung eines „*Mining for Bugs*“ getrieben wird, auch mithilfe von Mustererkennung und maschinellem Lernen, umso besser wird die Erkennung von Schwächen im Code-Design.

Es wurde auch betrachtet, wo die Architekturtests in die bestehende Test-Infrastruktur eingeordnet werden sollen. Die im BetShop bereits vorhandenen Architekturtests geben ein erstes Bild darüber wieder, welche Arten von Architekturtests prototypisch in dieser Arbeit für die mybet-Plattform entworfen und entwickelt werden können. Die Konkretisierung der Architekturtests ist Hauptaugenmerk des folgenden Kapitels.

28 Siehe „Mining for Bugs with Graph Database Queries“, Videovortrag auf der Chaos Computer Club Conference 2014: https://media.ccc.de/browse/congress/2014/31c3_-_6534_-_en_-_saal_g_-_201412282030_-_mining_for_bugs_with_graph_database_queries_-_fabs.html#video&t=196, Zugriff am 15.03.2015.

3 Konkretisierung / Abgrenzung

Bisher wurde der Versuch unternommen, ein klares Bild von Architekturtests zu zeichnen. Jedoch sind nicht alle Architektureigenschaften auch im konkreten Anwendungsfall durch automatisierte Tests sinnvoll testbar. In diesem Kapitel geht es also darum, die Module, in die Architekturtests eingeführt werden sollen, zu identifizieren und einen konkreten Ansatz für Architekturtests zu entwickeln. Dabei sollen vor allem die in Java verfügbaren Sprachmittel zum Einsatz kommen und, wenn erforderlich, frei verfügbare Bibliotheken zur Untersuchung von Softwaremetriken verwendet werden.

3.1 Analyse

Im Analyseteil der Konkretisierung geht es darum, heraus zu arbeiten, welche Module aufgrund ihrer Eigenschaften am besten für die exemplarische Einführung von Architekturtests geeignet sind und welche konkreten Testanforderungen sich daraus ergeben. Dabei soll es sich idealerweise um ein Modul aus mAS und eines aus mWS handeln, um die Einsatzmöglichkeiten der Tests zu demonstrieren.

3.1.1 mAS – Entitäten

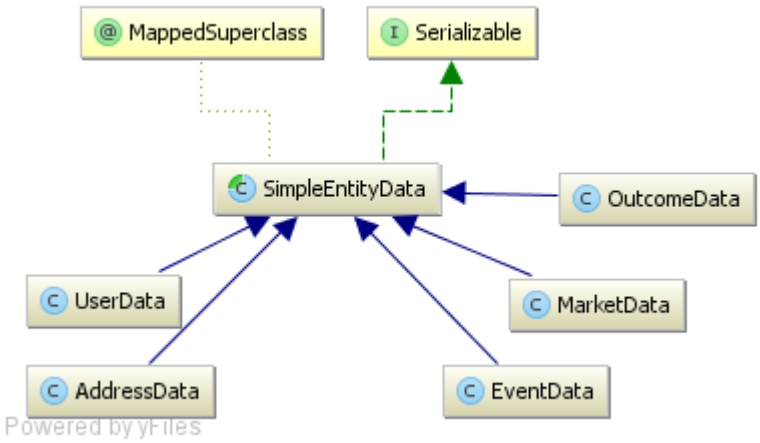


Abbildung 11: SimpleEntityData mit einigen erbenden Klassen (Quelle: Eigene Darstellung)

Die Entitäten im mAS sind die zentralen Klassen für die Geschäftslogik und Persistenz. Es handelt sich um rund 200 Klassen, die von `com.ibet.SimpleEntityData` erben. Diese Klassen müssen bestimmte Eigenschaften aufweisen. Sie sollen bspw. Die `clone()`-Methode nur dann überdecken, wenn sie `Cloneable` implementieren und sollen `Serializable` grundsätzlich nicht

implementieren, da `SimpleEntityData` dies bereits tut. Dies ist eine vor einigen Jahren getroffene Architekturentscheidung, die mithilfe der Architekturtests fixiert werden soll. Abbildung 11 zeigt `SimpleEntityData` und beispielhaft fünf der von ihr erbenden Klassen.

3.1.2 mWS – Sports Bets Engine

Das „mybet Widget System“ (mWS) ist eine Bibliothek von webbasierten Controls, die durch externe Partner auf deren Webseiten integriert werden können, um das Sportwettenangebot von mybet zu nutzen. Die *Sports Bets Engine* spielt eine vergleichbar zentrale Rolle wie die BetEngine im mAS. Statt einer Teilfunktionalität des mFES wurde dieses Teilprojekt ausgewählt, da es sich zum Einen um einen modularen Aufbau handelt und zum Anderen mWS die Frontend-Technologie der Zukunft bei mybet darstellt. Der mFES soll zukünftig durch mWS abgelöst werden.

Die Sports Bets Engine enthält rund 120 Klassen mit sehr unterschiedlichem Aufbau, an denen sich Konventionen, Abhängigkeiten und Metriken testen lassen. Abbildung 12 zeigt die Paketstruktur. Von besonderem Interesse sind die Pakete `services`, `cache` und `b2b`, da sich hierin die Geschäftslogik befindet.

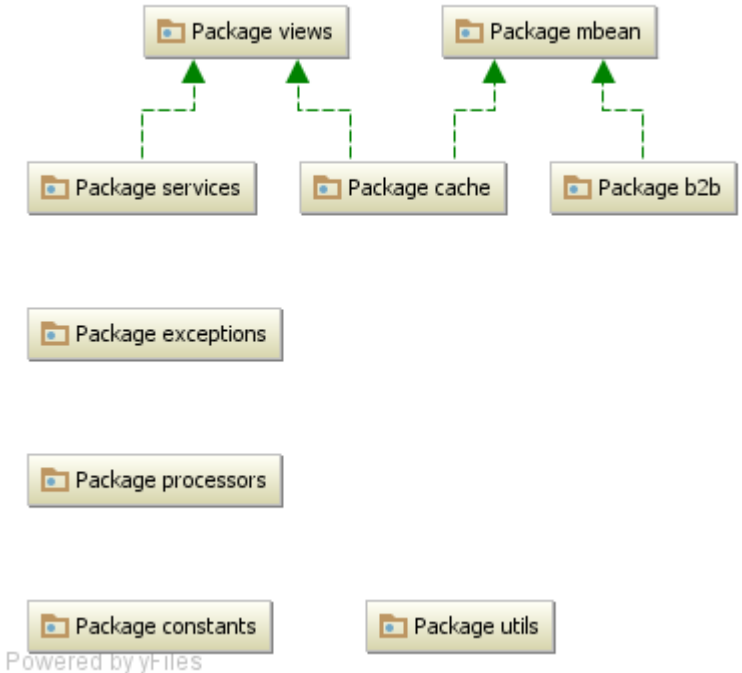


Abbildung 12: Paketstruktur der mWS Sports Bets Engine (Quelle: Eigene Darstellung)

Die erste Sichtanalyse des Codes dieser drei Pakete zeigt bereits einige Konventionen, die nicht eingehalten werden. Auch hinsichtlich Klassen- und Methodenlänge, sowie McCabe-Metrik gibt es

Brüche, die anhand von Architekturtests identifiziert werden können. Insbesondere bin ich interessiert an der Instabilität der Klassen.

3.1.3 Testanforderungen

3.1.3.1 Anforderungen an Codemetriken

Methodenlänge. In einigen Klassen der beiden ausgewählten Projekte finden sich Methoden, die aufgrund ihrer Länge schwer wartbar sind. Grundsätzlich kann man sagen, dass ein geübter Entwickler in der Lage ist, eine Bildschirmseite auf einmal zu überblicken. Daraus folgt, dass ein Schwellwert von 20 LoC für die Länge einer Methode als noch akzeptabel erachtet wird. In „Clean Code“ schreibt 'Uncle Bob', dass Methoden diese Länge nicht überschreiten sollten (vgl. [MR08], Seite 34, Abs. 4).

Für Konstruktoren sollte erfahrungsgemäß ein anderes Maß gelten. Da hier keine fachliche Funktionalität enthalten sein soll (über die Gründe diskutiert die einschlägige Literatur umfassend), sondern nur, was zur Erzeugung eines Objekts der Klasse vonnöten ist, ist es sinnvoll, das Maß für die Methodenlänge auf 10 LoC zu begrenzen.

(Öffentliche) Methodenzahl in einer Klasse. Dieses Maß ist schon schwerer eindeutig zu bestimmen. Grundsätzlich lässt sich aber sagen, dass die Anzahl (möglichst sinnvoller) Methodennamen ebenfalls bei 20 festzulegen ist. Die meisten gängigen IDEs bieten die Möglichkeit, alle Methodenrümpfe zusammenzuklappen, so dass nur die Methodensignaturen angezeigt werden. Da eine Methodensignatur dann einer LoC entspricht, ist es naheliegend, dieses Maß auch für die Anzahl der Methoden in einer Klasse anzunehmen (ohne private Hilfsmethoden).

Zyklomatische Komplexität. Diese Metrik hängt eng zusammen mit der Methodenlänge, da eine hohe zyklomatische Komplexität einen hohen Wert für LoC in einer Methode bedingt. Umgekehrt ist es nicht immer so. Ein hoher Wert für LoC bedeutet nicht immer eine hohe zyklomatische Komplexität, wenn in den meisten Zeilen nicht-verzweigende Statements angegeben sind. Aus diesem Grund muss die zyklomatische Komplexität auch getrennt von der Methodenlänge betrachtet werden.

In Java bedeutet zyklomatische Komplexität in einer Methode das Zählen der Verzweigungen. So wird zunächst von einem Wert 1 ausgegangen, der für jedes `if`- / `switch`- / `while`- / `for`-Statement um 1 erhöht wird. In der einschlägigen Literatur wird ein Wert von 10 als sinnvolle Obergrenze für den Wert der zyklomatischen Komplexität angenommen. Dieser Wert bedeutet auch, dass diese Anzahl von Tests nötig ist, um die Methode vollständig zu testen.

3.1.3.2 Anforderungen an Programmierrichtlinien

Bei mybet gibt es bereits Programmierrichtlinien, die zum Einen im firmeninternen Wiki auf der Basis von *Atlassian 'Confluence'* dokumentiert sind, zum Anderen durch das Verwenden von *Gated Commits* in Subversion bereits überwacht werden, wie *-Imports. Im Folgenden werden die bestehenden Richtlinien für Java aufgelistet. Dies ist wichtig, um entscheiden zu können, welche Richtlinientests in das Framework mit einfließen und auf welche davon der Produktivcode getestet werden soll.

- Konstanten werden in Großbuchstaben mit Unterstrichen geschrieben, z. B.
  `static String DEFAULT_DOCUMENT_TYPE = „xml“;`
- Objektfelder und lokale Variablen werden im CamelCase geschrieben, mit einem Kleinbuchstaben beginnend, z. B.
  `String documentType = DEFAULT_DOCUMENT_TYPE;`
- Klassen werden im CamelCase geschrieben, mit einem Großbuchstaben beginnend, z. B.
  `public class DocumentTypeIdentifier`
- Innerhalb einer Klasse gilt absteigende Sichtbarkeit, d. h. von oben nach unten gelesen: `public`, (friendly), `protected`, `private`.
- Innerhalb einer Klasse werden zuerst Klassen-, dann Objektmitglieder definiert, d. h. `static` zuerst.
- Innerhalb einer Klasse werden zuerst Felder, dann Methoden definiert.
- Innere Klassen haben `private` zu sein und folgen am Schluss.

Es bietet sich an, für die Erkennung der Schreibweisen von Klassen, Methoden, Feldern und Konstanten reguläre Ausdrücke zu verwenden. Code auf diese hin zu prüfen, ist trivial. Ein solcher Test schlägt dann fehl, wenn eine dieser Schreibweisen nicht eingehalten wird.

Für die Reihenfolge soll ein Test den zu untersuchenden Code von oben nach unten prüfen und die Modifizierer jedes Klassen- und Objektmitglieds vergleichen. Bei Nichteinhaltung der Reihenfolge soll der Test fehlschlagen.

Bisher wurden Programmierkonventionen im engeren Sinne betrachtet, die sich am Code direkt prüfen lassen. Eine weitere Konvention bei mybet ist die Einhaltung des Prinzips *Separation of Concerns (SoC)*. Dies ist aber semantischer Natur und durch automatisierte Tests nicht abbildbar. Hier ist ein Code Review der bessere Weg.

Weiterhin kann jedes einzelne Statement auf die in Java nicht selten vorzufindenden „Punktketten“ (mit Ausnahme des Builder-Patterns, wo genau das die „Parameterketten“ verhindern soll) untersucht werden. Ein Beispiel:

```
String[] aStringTokenArray = otherString
     .substring(startIndex, endIndex)
     .toUpperCase()
     .trim()
     .split(";");
```

Derartige Ausdrücke verstecken mögliche `RuntimeExceptions` auf schwer debugbare Weise. Dem lässt sich entgegenwirken, indem man zwischen einzelnen Schritten der Verarbeitung (wie hier von Zeichenketten) lokale Variablen einführt, welche die Zwischenergebnisse halten, also z. B. wie folgt:

```
String upperCaseSubString =
     otherString.substring(startIndex,
     endIndex).toUpperCase();
String trimmedString = upperCaseSubString.trim();
String[] aStringTokenArray = trimmedString.split(";");
```

So ersetzen die drei Statements zwar lediglich das obige, einzelne Statement. Dies kann jedoch, aus Gründen der Wartbarkeit, als besserer Code angesehen werden. In einem Test, der diese Eigenschaft prüft, kann trivial die Anzahl der Punkte (außerhalb von Zeichenketten) pro Ausdruck mit einem Schwellwert, z. B. 5, verglichen werden. Wird dieser überschritten, schlägt der Test fehl. Dieser Schwellwert muss so ermittelt werden, dass er einen sinnvollen Kompromiss zwischen Lesbarkeit des Codes und Anwendbarkeit von Objektmethoden bildet. Eine wichtige Ausnahme bildet das Erbauermuster: Wird dieses eingesetzt, ist die Verkettung von Methoden, die einem Objekt nach und nach Eigenschaften hinzufügen, gewollt. Ein solcher Test soll entweder selbst solche entwurfsmusterbedingten Verkettungen identifizieren oder der Entwickler, der den Test einsetzt, muss ihn hier explizit auslassen.

Weiter oben in diesem Abschnitt wurde bereits argumentiert, SoC sei eine Anforderung, die sich am besten mittels Code Review prüfen lässt. Seit Java 1.1 gibt es *innere Klassen*, also Klassen, die sich innerhalb anderer Klassen definieren lassen. Die JVM selbst kennt innere Klassen nicht, da der Compiler diese aus dem Quellcode herauszieht und als separate Bytecode-Dateien abspeichert. Dasselbe passiert mit anonymen Klassen. Innere Klassen verleiten dazu, weitere Aspekte einer Klasse hinzuzufügen, die mit ihr selbst nicht unbedingt etwas zu tun haben. Sie verleiten dazu, das

Prinzip SoC zu verwässern. Ein Test, der dies verhindern soll, schlägt dann fehl, wenn es innerhalb einer Klasse weitere Klassen gibt.

3.1.3.3 Anforderungen an die Einhaltung von Entwurfsmustern

Grundsätzlich ist zu sagen, dass Tests auf Entwurfsmuster sämtliche Möglichkeiten umfassen, die mittels eines *AST*[29]-basierten Java-Parsers möglich sind. D. h. es können unter anderem Merkmale wie die Modifizierer (`public`, `protected` oder `private`) und Namen von Feldern, Konstruktoren und Methoden, sowie Vererbungshierarchien untersucht werden. Weiterhin kann der Code auf bestimmte Statements untersucht werden, bspw. ob Zuweisungen von Variablen erfolgen und in welchem Kontext dies geschieht. Die Korrektheit der Implementierung kann nicht überprüft werden (das machen idealerweise die funktionalen Tests), da Syntaxbäume keine Aussage über die Reihenfolge der Lexeme liefern, sodass hier die Erwartungshaltung nur an die oben genannten Merkmale geknüpft werden kann.

Solche Tests haben, bei erster Betrachtung, einen Vor- und einen Nachteil. Zum Einen helfen sie dabei, bestimmte Konventionen einzuhalten (sie bilden daher auch eine nichtleere Schnittmenge mit den Richtlinientests), wie bspw. eine Methode, die eine Singleton-Instanz zurückliefern soll, zu benennen ist. Zum Anderen ist aber genau dies auch der Nachteil. Erfahrungsgemäß lassen sich Entwickler in Ihrer Benennungs-"Politik" nicht gern beeinflussen. Hier wird die Akzeptanz solcher Tests noch zu untersuchen sein. In der Implementierung des prototypischen Frameworks soll mindestens je ein Struktur-, Erzeugungs- und Verhaltensmuster in codeinspizierenden Tests realisiert werden.

3.1.3.4 Anforderungen an Abhängigkeiten

Die Abhängigkeiten zwischen Modulen, Untermodulen und Typen (Klassen) bilden den Kern von Softwarearchitekturen. Tests für Abhängigkeiten müssen prüfen, welche Abhängigkeiten bestehen können, sollen und welche nicht bestehen dürfen. Ferner gibt es „Fallstricke" bei Abhängigkeiten, die auch zur Laufzeit problematisch werden können. Einer dieser Fallstricke sind **zyklische Abhängigkeiten**. Martin Fowler definiert das „Acyclic Dependencies Principle" (ADP) so, dass der Abhängigkeitsgraph kreisfrei sein muss. Dies setzt voraus, dass die Menge der Module, Untermodule bzw. Typen (Klassen) vor der Untersuchung auf dieses Prinzip bekannt sein muss. Da es effiziente Algorithmen zum Finden von Kreisen in Graphen gibt, ist die Implementierung eines solchen Tests trivial.

29 Ein *Abstract Syntax Tree* (dt. abstrakter Syntaxbaum) ist eine Darstellung eines gegebenen Quellcodes als Baum. Jeder Knoten in diesem Baum entspricht einem Element der Programmiersprache (vgl. http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/, Zugriff am 12.03.2015). Dabei gehen jedoch chronologische sowie Datenflussinformationen, wie sie im Quellcode selbst ersichtlich sind, verloren.

Die Abhängigkeit eines Moduls von anderen Modulen wird als **afferente Kopplung** (engl. afferent Coupling, *Ca*) bezeichnet; der umgekehrte Weg, also die Abhängigkeit anderer Module vom betrachteten Modul hingegen als **efferente Kopplung** (engl. efferent Coupling, *Ce*). Die **Instabilität** *I* einer Klasse wird wie folgt berechnet:

$$I = \frac{Ce}{Ce + Ca}$$

Daraus folgt, dass *I* gegen 0 strebt, wenn eine Klasse überwiegend von anderen Klassen abhängt, von ihr hingegen nur wenige andere Klassen abhängen (geringe Stabilität). Dagegen strebt *I* gegen 1, wenn die Klasse kaum von anderen Klassen abhängt, von ihr jedoch sehr viele andere Klassen abhängen (hohe Stabilität). Ein Test, der auf Instabilität prüft, muss das ermittelte *I* mit einem Schwellwert knapp unter 1 vergleichen und fehlschlagen, wenn dieser Schwellwert unterschritten wird. Ein solcher Test lässt sich trivial umsetzen, da er, mittels einer geeigneten Datenstruktur, die vor dem eigentlichen Test aufzubauen ist, lediglich die direkten Verbindungen zählen muss.

Grundsätzlich dienen Architekturtests „im Großen“ dazu, die Softwarearchitektur „rein“ zu halten, sodass die Eigenschaft von Code, im Laufe des Lebenszyklus zu „verrotten“, verlangsamt wird. In der Mathematik wird unter **Methodenreinheit** (engl. 'Method purity') verstanden, dass für die Lösung eines Problems keine fremden Konzepte herangezogen werden sollen. Abgewandelt für die Softwareentwicklung bedeutet dies, dass eine Methode dann als **rein** gilt, wenn sie ihre Umgebung möglichst nicht oder in einem zu definierenden Rahmen minimal beeinflusst. So ist eine Methode dann **rein**, wenn sie:

- ein von ihr instanziiertes Objekt verändert,
- ihren Rückgabewert verändert *und*
- andere reine Methoden aufruft.

Als **unrein** gilt eine Methode dann, wenn sie:

- ein (statisches) Klassenmitglied verändert,
- Parameter entgegennimmt, die von ihr verändert werden *oder*
- andere unreine Methoden aufruft.

Ein Test, der diese Eigenschaft prüft, soll feststellen, ob eine Methode statische Member verändert, ihre Parameter verändert oder andere Methoden aufruft, die dies tun. Falls dies geschieht, schlägt der Test fehl. Gerade das Kriterium der Methodenreinheit ist sorgsam anzulegen bei der späteren Einführung der Tests. Oft ist es erforderlich, die hier gegebenen Postulate zu brechen, wenn z. B.

Klassenvariablen von Methoden geändert werden oder aufgrund der Viralität der Methodenreinheit Methoden aufgerufen werden, die unrein sind.

3.2 Entwurf des Frameworks

Da es sich um eine vollkommen neue Art von Tests handelt, die jedoch gemeinsame Funktionalität enthält, ist es sinnvoll, von vorn herein die Entwicklung als Testframework vorzusehen, das auf bekannten Testframeworks wie *JUnit*[30] aufbaut. Für jede der drei Architekturtestarten (Codemetriken, Programmierrichtlinien und Entwurfsmuster) soll es eine abstrakte Oberklasse geben, deren Benennungen schon intuitiv einleuchten:

- `CodeMetricsTestBase`,
- `ConventionsTestBase` und
- `PatternTestBase`.

Für darüberliegende, gemeinsame Funktionalität soll es eine abstrakte Oberklasse `ArchitectureTestBase` geben, von der die drei zuvor genannten Klassen erben.

Bereits diese Einhaltung kann intern in `ArchitectureTestBase` überprüft werden, in dem eine mit `@BeforeClass` annotierte Methode `checkHierarchy()` prüft, ob für einen Architekturtest die Vererbungshierarchie eingehalten wird. Dies führt zu einer weiteren, in den vorhergehenden Kapiteln nicht betrachteten Art von Architekturtest, dem *Abhängigkeitstest* (engl. *Dependency Test*). Folglich wird noch eine vierte abstrakte Klasse `DependencyTestBase`, die von `ArchitectureTestBase` erbt, eingeführt.

Das Testframework muss auch definiert bekommen, welche Klassen den Architekturtests unterzogen werden sollen. Hierzu wird eine Annotation `@ArchitectureTest` definiert, die wiederum vor der eigentlichen Testausführung vom Annotation Processor der JVM ausgewertet werden kann. Diese Annotation wird vor jede Klasse geschrieben, die durch die Architekturtests geprüft werden soll. Dazu muss `@ArchitectureTest` selbst mit `@Retention(RetentionPolicy.RUNTIME)` annotiert sein, um zur Laufzeit auswertbar zu sein.

Tests für Codemetriken beruhen auf festgelegten Schwellwertvorgaben und Programmierrichtlinien gelten als fix vereinbart. So genügt es, betreffende Klassen mit `@ArchitectureTest` zu annotieren, damit diese Tests durchlaufen werden. Für Entwurfsmuster jedoch muss angegeben werden, auf welche Muster die annotierte Klasse geprüft werden soll. Man könnte zwar eine Logik

30 Siehe http://junit.org/, Zugriff am 12.02.2015.

implementieren, die versucht, das Muster, das verwendet wurde, zu erkennen, dies würde aber klar über das gesteckte Ziel hinausschießen. Denn Entwickler wissen, welche Muster sie in welcher Situation einsetzen, daher kann das konkret verwendete Muster auch explizit über der Klasse annotiert werden. Dies soll so geschehen:

Code	Bedeutung
`@ArchitectureTest`	Attribut `patterns` ist leer, es wird **nicht** auf Entwurfsmuster geprüft.
`@ArchitectureTest(patterns = {singleton, composition})`	Attribut `patterns` enthält die Entwurfsmusternamen 'singleton', 'composition'. Daher wird auf die Muster **Singleton** und **Kompositum** geprüft.

Aufgrund von Implementierungsdetails, die ein bewusstes Brechen von Codemetrik-Schwellwert- oder Programmierrichtlinien-Vorgaben unvermeidbar machen, soll es möglich sein, diese in der Annotation `@ArchitectureTest` explizit abzuschalten:

Code	Bedeutung
`@ArchitectureTest`	Attribute `omitMetrics` und `omitConventions` sind `false`, es wird **auf Codemetriken und auf Programmierrichtlinien** geprüft.
`@ArchitectureTest(omitMetrics = true)`	Attribut `omitMetrics` ist `true`, d. h. es wird **nur auf Programmierrichtlinien** geprüft.
`@ArchitectureTest(omitConventions = true)`	Attribut `omitConventions` ist `true`, d. h. es wird **nur auf Codemetriken** geprüft.
`@ArchitectureTest(omitMetrics = true, omitConventions = true)`	Attribute `omitMetrics` und `omitConventions` sind `true`, d. h. es werden **keine Architekturtests** durchgeführt. *Anmerkung: Diese Annotation hat den gleichen Effekt wie das Weglassen der gesamten Annotation.*

Da Abhängigkeitstests auch solche Tests umfassen, die festlegen, welche Abhängigkeiten bestehen dürfen oder sollen und welche nicht bestehen dürfen, muss, vergleichbar mit Entwurfsmustertests, ein Attribut `dependencies` vorhanden sein, das es erlaubt, bestimmte Abhängigkeitstests durchführen zu lassen. Dies soll folgendermaßen geschehen:

Code	Bedeutung
`@ArchitectureTest`	Attribut `dependencies` ist leer, d. h. es werden **keine Abhängigkeitstests** durchgeführt.
`@ArchitectureTest(dependencies =`	Attribut `dependencies` enthält die

`{acyclicDependencyPrinciple, instability}`	Abhängigkeitsnamen 'acyclicDependenciesPrinciple' und 'instability'. Daher wird auf das ADP sowie auf Instabilität geprüft.

3.3 Laufzeitumgebung und Hilfsmittel

Für die Umsetzung der Tests selbst ist darauf zu achten, möglichst die „Bordmittel“ von Java einzusetzen, wie z. B. die Reflections API, die es ermöglicht, zur Laufzeit Java-Klassen auf ihre Eigenschaften, Felder, Methoden, Vererbung und Implementierung von Schnittstellen zu „befragen“. Nicht möglich ist damit jedoch die Analyse von Quellcode. Hierzu bedarf es eines geeigneten Parsers, der als Bibliothek mittels Maven-Dependency eingebunden wird. Als geeignet für das Framework erscheinen sämtliche Implementierungen, die mit dem *JavaCC*[31] erstellt wurden. Ein sehr schlankes Beispiel ist der „*Java 1.5 Parser and AST*“[32], der auch Generics und Annotationen, die es seit Java 1.5 (Language Level 5) gibt, unterstützt.

Damit das Testframework sowohl auf Bytecode (für Reflections) als auch auf Quellcode (für die Tiefenanalyse mittels JavaCC-basiertem Parser) operieren kann, müssen sämtliche Quellcode- sowie Bytecode-Dateien vor der eigentlichen Testausführung initial erfasst werden. In größeren Softwareprojekten sind dies nicht selten mehrere tausend Dateien Quellcode- und mindestens ebenso viele Bytecode-Dateien. Beide sollten jeweils in einer Datenstruktur vorgehalten werden, die es den Tests erlaubt, schnell und gezielt auf beide zu zu greifen. Wie bereits in Kapitel 2 erwähnt, spielt die Laufzeit eine herausgehobene Rolle, um den gesamten Buildzyklus, der dann auch die Ausführung der Architekturtests beinhalten soll, nicht weiter in die Länge zu ziehen. Für die Implementierung muss gelten, dass die Durchführung aller Architekturtests, einschließlich ihrer Vorbereitung, nicht mehr als einige Sekunden in Anspruch nehmen darf. Folglich müssen größere Erfassungsoperationen über den gesamten Quell- sowie Bytecode hinweg, welche die für die Tests nötigen Datenstrukturen im Speicher aufbauen, so beschaffen sein, dass sie zu Beginn der Architekturtest-Suite einmal ausgeführt werden und dann bis zum Ende des letzten Tests der Suite erhalten bleiben. Erst danach dürfen (und müssen) diese wieder abgeräumt werden.

3.4 Testen der Tests

Das Testframework selbst besteht in seinem wesentlichen, fachlichen Kern aus Tests, also aus Klassen, die anderen Code testen. Eine der interessanten Fragen im Verlauf dieser Arbeit ist für mich:

31 Der *Java Compiler Compiler* (https://javacc.java.net/, Zugriff am 28.02.2015) ist ein Parser-Generator für Java. Er erzeugt anhand einer gegebene Grammatik einen Parser, der Quellcode gegen die Grammatik matcht.

32 Siehe https://code.google.com/p/javaparser/, Zugriff am 28.02.2015

Wie teste ich die Tests?

Eine Möglichkeit ist, Testklassen zu schreiben, die nicht im `test`-Zweig, sondern im `main`-Zweig des Quellcodes zu finden sind. Für jeden Test kann so also mindestens eine Testklasse geschrieben werden, die dem Test entsprechend annotiert ist und bestimmte Eigenschaften aufweist, die das erwartete Verhalten des Tests provozieren soll. Solche Klassen würden vom Testframework genau so erfasst werden, wie im vorhergehenden Abschnitt 3.3 beschrieben. Diese Klassen haben jedoch eine ganz wesentliche Einschränkung: Sie können nur das Gutverhalten der Tests feststellen, müssen also so beschaffen sein, dass die Tests selbst nicht fehlschlagen. Den interessanten Teil, die Fehlerprovokation, kann man auf diese Weise gar nicht testen, da das Fehlschlagen der Tests bedeutet, dass der Buildzyklus insgesamt fehlschlägt. Entsprechend des Vorhabens, Architekturtests auf dieselbe Ebene wie Unit-Tests zu stellen, muss dies so sein.

Die andere Möglichkeit ist, die Tests schrittweise auf den Produktivcode, wie in den beiden Modulen in den Abschnitten 3.1.1 und 3.1.2 beschrieben, „loszulassen“, sich die Ergebnisse anzusehen und zu evaluieren, ob die monierten Probleme tatsächlich berechtigt sind, oder ob es sich um „False Positives“ handelt. Hier zeigt sich die Unentscheidbarkeitsproblematik in den Tests, wie bereits in Abschnitt 2.1.7 diskutiert. Dies lässt sich für jeden geschriebenen Test mit der entsprechenden Annotation (siehe Abschnitt 3.2) in den Klassen des Produktivcodes unmittelbar durchführen. Bei der Einführung der Tests in den Kapiteln 4 und 5 werde ich die zweite Möglichkeit aufgrund der Untersuchungsmöglichkeit des Fehlerfalls umsetzen.

3.5 Zusammenfassung und Überleitung

Am Schluss dieses Kapitels sind die Rahmenbedingungen klar, unter denen das Testframework umgesetzt werden soll. Es wurde zunächst analysiert, in welche Module der mybet-Plattform die Tests eingeführt werden sollen und welche Spezifika der Code dieser Module hat. Anschließend wurden Anforderungen an Codemetriken, Konventionen, Entwurfsmuster und Abhängigkeiten erarbeitet, die als Grundlage für die zu implementierenden Tests dienen. Besonders wichtig wird bei der Umsetzung die Frage sein, wie zu Beginn Quellcode und Bytecode aller zur mybet-Plattform gehörenden Klassen den Tests verfügbar gemacht werden können. Schließlich ist auch untersucht worden, wie die geschriebenen Tests getestet werden können, was im Rahmen der Implementierung direkt am Code der Module geschehen wird.

4 Umsetzung

Die bisherigen Ausführungen haben klargestellt, dass ich ein Framework entwickeln möchte, das die bequeme, möglichst generische Einführung von Architekturtests, unter Berücksichtigung der speziellen Anforderungen der mybet-Systemlandschaft ermöglicht. Dieses Kapitel unterteilt sich daher in zwei Abschnitte: Zum Einen in die Beschreibung der Entwicklung des Frameworks selbst und zum Anderen die in Integration des Frameworks und Realisierung von Architekturtests in mAS und mWS. Insbesondere wird hier auf Schwierigkeiten eingegangen, die bei der Entwicklung und Integration auftreten und es wird erörtert, wie diese angegangen und gelöst werden.

4.1 Entwicklung des Frameworks

Das Testframework dieser Arbeit verfolgt den Ansatz wie in Kapitel 2 und 3 beschrieben. Daher bestehen auch größtenteils nicht die gewöhnlichen Anforderungen, die ein Testframework mitbringen soll, wie z. B. das *Mocken*[33] von Testdaten oder das Berichten über Testergebnisse. Wohl aber setzt es auf bestehenden Testframework-Technologien auf und verwendet deren Funktionalitäten, wo erforderlich.

Als Basisframework wird *JUnit* verwendet. Das heißt, sämtliche zu entwickelnden Klassen verwenden Klassen und Annotationen aus JUnit. Der Grund für diese Wahl ist, dass JUnit das bekannteste Unit-Testframework für Java und in der verwendeten Version 4.12 bewährt, ausgereift und umfassend dokumentiert ist. Es erlaubt z. B. auch eine oder mehrere Listener für anpassbare Ergebnisausgabe zu implementieren. Dieses Feature ist nützlich, um die Meldungen der Architekturtests in geeigneter Form darzustellen und ermöglicht es so, die Testergebnisse in beliebige Zieldatenstrukturen zu schreiben.

4.1.1 Test-Basisklassen

Die Basisklasse für alle Architekturtests ist `ArchitectureTestBase`. Von ihr leiten alle weiteren Testklassen ab. Dabei gibt es für jede der vier Testkategorien (Konventionen, Metriken, Muster und Abhängigkeiten) spezifische, aber immer noch abstrakte Klassen, die jeweils gemeinsam benötigte Funktionalität für die Tests kapseln. Abbildung 13 zeigt die Vererbungsstruktur der Basisklassen für die Tests.

33 Als *Mock-Up* (aus dem Englischen) wird eine Imitation verstanden, die das Original nur in einigen seiner Eigenschaften (denen, die im aktuellen Kontext notwendigen) imitiert. Beim Testen verwendet man *Mock*-Datenstrukturen, um etwaige Schnittstellen bedienen zu können.

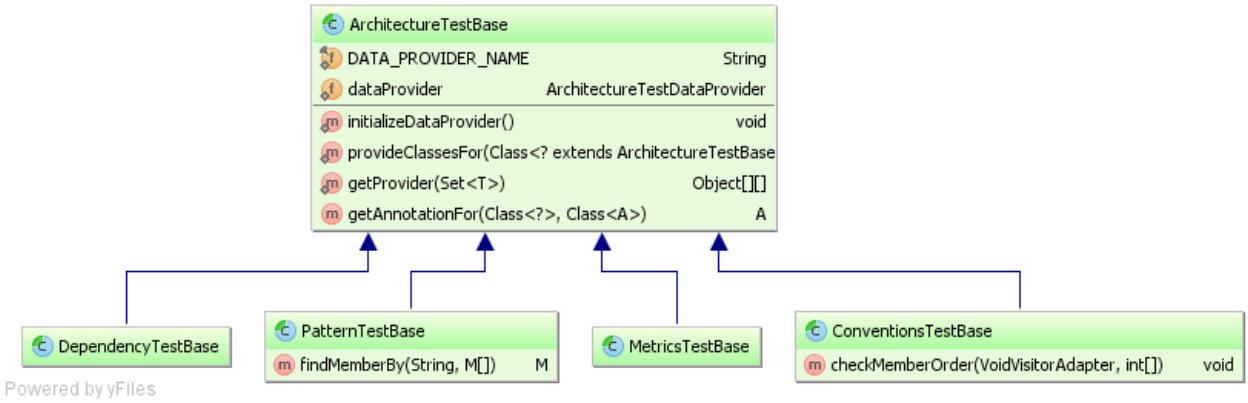


Abbildung 13: Vererbungsstruktur der Basisklassen (Quelle: Eigene Darstellung)

Es ist zu sehen, dass `MetricsTestBase` und `DependencyTestBase` leer sind, da die Tests derzeit keine gemeinsame Funktionalität, welche über `ArchitectureTestBase` hinaus geht, benötigen.

4.1.2 Annotationen

Die implementierten Annotationen dienen dazu, die Klassen oder Methoden, die designierten Architekturtests unterzogen werden sollen, diesen auch zuzuführen. Dies geschieht zur Laufzeit bei der Testausführung über den `ArchitectureTestDataProvider`. Dieser verwendet den `ArchitectureTestAnnotationEvaluator`, der die architekturtestbezogenen Annotationen erfasst und auswertet. Er stellt diese dann als *DataProvider*[34] für die Tests zur Verfügung. Diese DataProvider werden, neben der `@Test`-Annotation, in `@UseDataProvider`-Annotationen angegeben. Die wichtigste Annotation für Architekturtests ist, wie in Kapitel 3 bereits konzipiert, `@ArchitectureTest`. Sie erwartet vier Parameter:

- `boolean omitMetrics() default false;`
- `boolean omitConventions() default false;`
- `Pattern[] patterns() default {undefined};`
- `Dependency[] dependencies() default {none};`

Diese und ihre Funktionen sind aus Kapitel 3 bereits ebenfalls bekannt. `Pattern` und `Dependency` sind Enumerationen, wobei `Pattern` die möglichen Entwurfsmsuter, die getestet

34 Ein DataProvider ist ein Reihungsobjekt (typischerweise Object[][]), das dazu verwendet wird, Tests zu parametrisieren. So kann eine Testmethode mehrmals aufgerufen werden mit den Testdaten, die im DataProvider enthalten sind.

werden können, benennt und `Dependency` die durchzuführenden Abhängigkeitstests.

Neben `@ArchitectureTest` gibt es die folgenden weiteren Annotationen, die das jeweilige Testverhalten der Abhängigkeitstests genauer spezifizieren:

- `@MustUse`, `@MustNotUse`,
- `@MustImplement`, `@MustNotImplement`,
- `@MustExtend`, `@MustNotExtend`,
- `@MustThrow`, `@MustNotThrow`,
- `@MustBePure`,
- `@MustOverride` und `@MustNotOverride`.

Die meisten Annotationen erklären sich intuitiv von selbst. Der fehlenden Mehrfachvererbung in Java wird Rechnung getragen, indem `@MustExtend` und `@MustNotExtend` genau eine Klasse als Parameter `type` zulassen, außerdem wird in einem zweiten Parameter `entireHierarchy` mit `true` oder `false` angegeben, ob der Test nur die unmittelbare Superklasse betrachten soll oder die gesamte Hierarchie bis hin zu `java.lang.Object`. `@MustUse`, `@MustNotUse`, `@MustThrow`, `@MustNotThrow`, `@MustImplement` und `@MustNotImplement` erfordern jeweils ein Array von Klassen als Parameter. `@MustOverride` und `@MustNotOverride` erfordern jeweils ein Array von Zeichenketten der Methodennamen. `@MustBePure` erwartet keine Parameter. `@MustThrow`, `@MustNotThrow` und `@MustBePure` sind Methodenannotationen, alle anderen sind Klassenannotationen. Für alle gilt, wird

```
@ArchitectureTest(dependency = extendsDependency¦throwsDependency¦
usesDependency¦implementsDependency¦overridesDependency¦methodPurity)
```

nicht mit angegeben, werden sie nicht ausgewertet und bleiben wirkungslos.

4.1.3 Regeln

Die Pflege der Annotationen bei sehr vielen Klassen, die den Architekturtests unterzogen werden sollen, ist für sich genommen bereits ein gewisser Aufwand. Außerdem entstand während der Implementierung die immer offenkundigere Anforderung, Klassen nach bestimmten Regeln automatisch, also unabhängig davon, ob sie annotiert sind oder nicht, Architekturtests zuzuführen. Dies macht den Entwurf und die Implementierung einer API für diese Regeln, im Folgenden *Rule-API* genannt, erforderlich.

Ausgehend von den bestehenden Annotationen besteht zwischen Annotation *A* und Regel *R* eine *1:n-Relation*, d. h. zu einer Annotation kann es *n* Regeln geben. Innerhalb dieser Regeln gibt es eine

Bedingung *C*, die bestimmt, wann die Regel zutreffen soll. Die Regeln werden vor den eigentlichen Architekturtests ausgeführt, also wenn bestimmt wird, welche Klassen welchen Tests zugeführt werden.

Die Regeln haben eine höhere Priorität als die Annotationen, so dass bei eventuellen Kollisionen die Regel gewinnt. Ein Beispiel: Definiert sei eine Regel, die für alle Klassen, die `Cloneable` implementieren, vorgibt, dass sie nicht `Serializable` implementieren dürfen. Eine der Klassen ist jedoch wie folgt annotiert:

```
@ArchitectureTest(dependency = {implementsDependency})
@MustImplement(interfaces = {Serializable}
public class SomeClassThatImplementsCloneable implements Cloneable,
Serializable {
      ...
}
```

Die Regel würde also bewirken, dass die `@MustImplement`-Annotation wirkungslos bleibt und in der Folge der Test fehlschlägt, da die zu testende Klasse im Bespiel `Cloneable` implementiert, obwohl sie auch `Serializable` implementiert und dies gemäß Regel nicht darf.

Das obige Beispiel kann direkt herangezogen werden, die Umsetzung der Rule-API zu erläutern. Es gibt 4 wesentliche abstrakte Klassen:

- `AnnotationBasedRule<C extends Condition<Class<?>>>`
- `ClassAnnotationBasedRule<C extends Condition<Class<?>>> extends AnnotationBasedRule<C>`
- `MethodAnnotationBasedRule<C extends Condition<Class<?>>> extends AnnotationBasedRule<C>`
- `Condition<T>`

Abbildungen 14 und 15 zeigen die Klassendiagramme für die abstrakten Klassen der Regeln bzw. die Klassen für die Bedingungen. Die Quellcodes dazu befinden sich auf der beiliegenden CD-ROM.

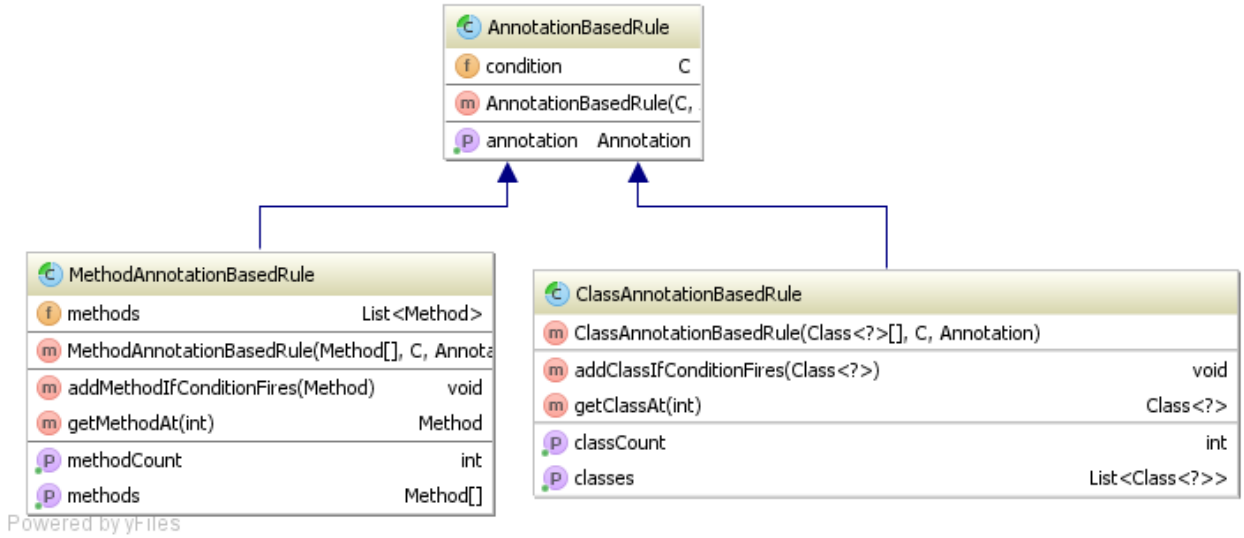


Abbildung 14: Vererbungsstruktur der Regel-Basisklassen (Quelle: Eigene Darstellung)

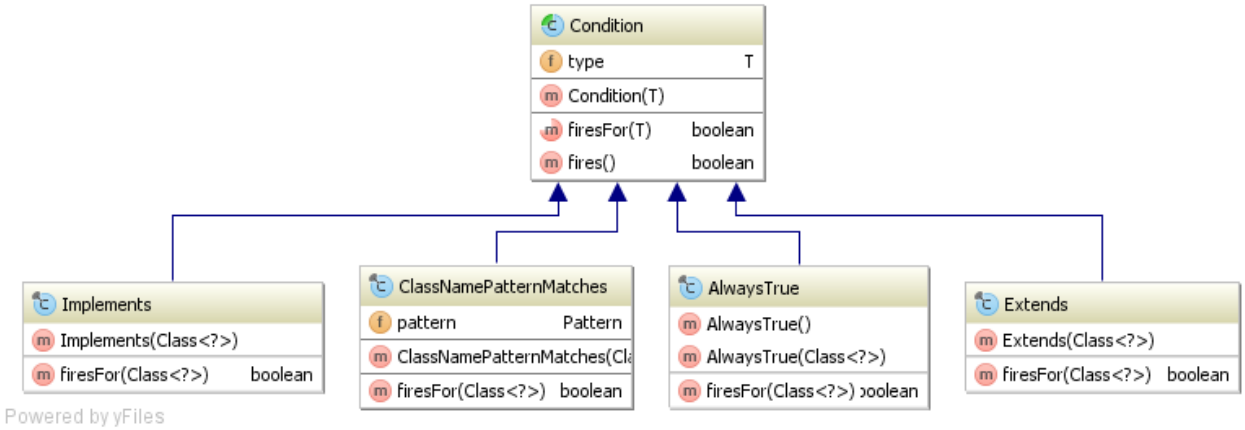


Abbildung 15: Vererbungsstruktur der Bedingungsklassen (Quelle: Eigene Darstellung)

Die Regel, die das Beispiel also erfüllt, muss wie folgt implementiert werden:

```
public class MustNotImplementSerializableIfCloneableIsImplementedRule extends
      ClassAnnotationBasedRule<Implements> {
      public MustNotImplementSerializableIfCloneableIsImplementedRule(Class<?>[]
            classes, MustNotImplement annotation) {
            super(classes, new Implements(Cloneable.class), annotation);
      }
      public Class<?>[] interfaces() {
            return new Class<?>[] {Serializable.class};
      }
}
```

Der Name der Regel mag zwar auf den ersten Blick lang erscheinen, doch nicht ohne Grund spricht

sich u. a. Robert C. Martin in [MR08] wiederholt dafür aus, sprechende Namen zu verwenden – auch wenn diese länger ausfallen, dank Camel-Case jedoch leserlich bleiben.

`@MustNotImplement` ist eine Klassenannotation, daher erbt die Regel von `Class-AnnotationBasedRule` mit dem Template `<Implements>`. `Implements` ist seinerseits eine Bedingung, die von `Condition` mit dem Template `<Class<?>>` erbt:

```
public class Implements extends Condition<Class<?>> {
      public Implements(Class<?> type) {
            super(type);
      }
      @Override
      public boolean firesFor(Class<?> clazz) {
            Class<?>[] interfaces = clazz.getInterfaces();
            for (Class<?> in : interfaces) {
                  if (in.equals(type)) {
                        return true;
                  }
            }
      return false;
      }
}
```

Am Code von `Implements` wird deutlich, wann die Bedingung die sie verwendende Regel dazu bringt, zu *feuern*: Genau dann, wenn ein gegebener Typ das im Konstruktor gegebene Interface implementiert. Die verwendende Regel fügt eine Klasse also erst dann den zu prüfenden Klassen hinzu, wenn die Bedingung die Regel dazu veranlasst. Es gibt auch eine Bedingung `AlwaysTrue`, die immer `true` zurückliefert, also jede Klasse, die der Regel gegeben wird, dieser auch hinzufügt. Der Konsistenz halber gibt es auch eine Bedingung `AlwaysFalse`, die genau das Gegenteil bewirkt, also keine Klasse der Regel hinzufügt. Diese wird jedoch produktiv bisher nicht verwendet, da eine Regel mit `AlwaysFalse` keinerlei Auswirkung hätte.

Regeln werden erst zur Laufzeit generiert, unmittelbar vor der Testausführung. Dies geschieht mittels *Regelgenerierer*, welche von `ArchitectureTestRuleGenerator` erben. Abbildung 16 zeigt die Vererbungsstruktur der Regelgenerierer.

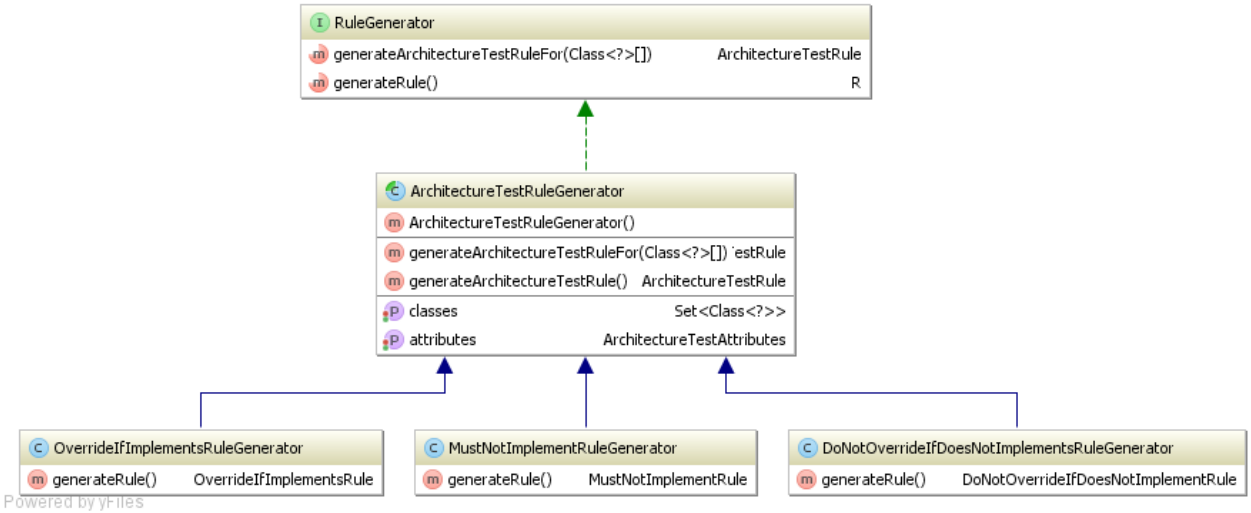


Abbildung 16: Vererbungsstruktur der Regelgenerierer (Quelle: Eigene Darstellung)

Die Regelgenerierer erzeugen Annotationen, die im Quellcode der Klassen, die durch die Regeln erfasst werden, nicht vorhanden sind. Der Grund für diese Designentscheidung ist, dass der Mechanismus, der die Klassen den Tests anhand von Annotationen zuordnet, auch für die Regeln verwendet werden kann. Am Beispiel des `OverrideIfImplementsRuleGenerator` sei die Funktionsweise der Regelgenerierer erklärt:

```
public class OverrideIfImplementsRuleGenerator extends
      ArchitectureTestRuleGenerator<MustOverrideMarker,
      OverrideIfImplementsRule> {

    @Nonnull
    @Override
    public OverrideIfImplementsRule generateRule() {
        setClasses(ArchitectureTestUtil.getSubTypesOf(SimpleEntityData.class));
        setAttributes(ArchitectureTestAttributes
            .createWithDependencyOnly(Dependency.overridesDependency));
        MustOverrideMarker mustOverrideMarker =
                ArchitectureTestUtil.createAnnotation(MustOverrideMarker.class,
                        "methodNames", Arrays.asList("clone"));
        return new OverrideIfImplementsRule(Cloneable.class, getClasses(),
            mustOverrideMarker);
    }
}
```

Der obige Regelgenerierer ist einer der Ersten, die spezifisch für die mybet-Plattform implementiert wurden. Er erstellt zwei Regeln:

1. Jede Klasse, die `SimpleEntityData` beerbt, erhält die Annotation

```
@ArchitectureTest(omitConventions = true, omitMetrics = true,
    dependencies = {implementsDependency})
```

2. Jede Klasse, die `SimpleEntityData` beerbt, erhält die Annotation
   `@MustNotImplement(interfaces = {Serializable.class})`

Im Klartext: jeder Subtyp von `SimpleEntityData` darf `Serializable` nicht implementieren. Dies ist eine der grundlegenden Architekturentscheidungen für die mybet-Plattform, die mit dieser Regel fixiert wird. Die beiden erstellten Regeln veranlassen `ImplementsTest` (siehe Abschnitt 4.1.7.2) dazu, für alle Subtypen von `SimpleEntityData` sicher zu stellen, dass sie `Serializable` nicht implementieren. Schlägt der Test bei mindestens einer Klasse fehl, ist die Regel gebrochen und der Build schlägt ebenfalls fehl.

Zusammenfassend ist zu sagen, dass die hier definierte API, im folgenden *Rule-API* genannt, welche zunächst nur den Bedenken Rechnung tragen sollte, dass man Annotationen auch mal falsch belegen oder gar vergessen könnte, sich als das eigentlich mächtige Werkzeug des ganzen Frameworks herausstellt. Erste Versionen dieser API waren programmatisch noch um einiges sperriger als jetzt. Regelgenerierer lassen sich nunmehr recht leicht hinzufügen, ebenso wie weitere Regeln und Bedingungen. Die Offenheit des Frameworks ist auf solche Erweiterungen ausgelegt.

4.1.4 Convention Tests

Stellvertretend für Programmierkonventionen sind hier vier Tests implementiert worden: Einer für die richtige Schreibweise von Feld-, Methoden- und Klassennamen, ein Weiterer für die richtige Reihenfolge der Sichtbarkeiten innerhalb von Klassen. Damit ist gemeint, in welcher Reihenfolge Konstanten, Felder und Methoden, abhängig von ihrer Sichtbarkeit in einer Klasse, angeordnet sein sollen. Die Festlegung der Anordnung im Code hilft, gerade bei größeren Klassen die Wartbarkeit des Codes zu erhalten. Ein Entwickler sucht z. B. `private` Hilfsmethoden unten in einer Klasse, Klassenkonstanten, die nach außen sichtbar sind, hingegen ganz oben. Der dritte Test ermittelt, ob innere Klassen vorhanden sind und der vierte, ob es lange Methoden-"Verkettungen" gibt, die ein eventuelles Debuggen bei *unchecked Exceptions*[35] sehr schwer machen können. Abbildung 17 zeigt die Vererbungsstruktur der Convention Tests.

35 *Ausnahmen* gibt es in Java in zwei Arten: *Checked* und *Unchecked*. Der Unterschied ist, dass *checked Exceptions* entweder im Code explizit gefangen werden oder in der Methodensignatur angegeben sein müssen. Auf *unchecked Exceptions* trifft dies nicht zu. Dazu zählen auch die schwerer zu lokalisierenden *NullPointerException* und *ClassCastException*.

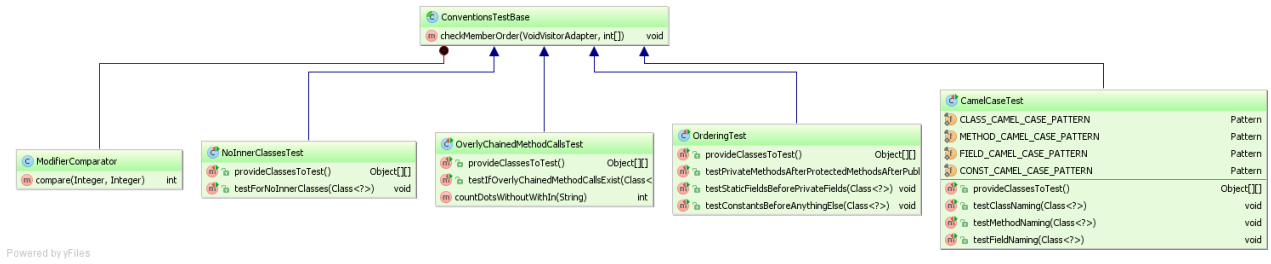


Abbildung 17: Vererbungsstruktur der Conventions Tests (Quelle: Eigene Darstellung)

4.1.4.1 CamelCaseTest

`CamelCaseTest` erstellt zunächst drei `Pattern`-Objekte, gegen die im Test die jeweiligen Namen mit Hilfe der Java-eigenen Bibliothek für reguläre Ausdrücke (siehe Paket `java.util.regex`) gematcht werden . Die vier regulären Ausdrücke sind:

- `([A-Z]+[a-z0-9]*)+` für Klassennamen sowie
- `[a-z]+([A-Z]*[a-z0-9]*)*` für Methoden- und Feldnamen und
- `[A-Z]+([A-Z0-9]*[_]?)*` für Konstanten.

Hier sind zwar nur drei Spiegelstriche angegeben, im Test selbst aber sind es tatsächlich vier Ausdrücke, da z. B. vielerorts Felder in Klassen mit _ (Unterstrich) als Präfix geschrieben werden. Um solchen Anforderungen Rechnung zu tragen, hat der Test vier Konstanten, von denen drei (hier bei mybet) gleich belegt sind.

Sofern in der zu testenden Klasse Namen vorkommen, die nicht gegen die jeweiligen regulären Ausdrücke matchen, schlägt der Test fehl. Wie viele andere Tests im Framework auch bedient sich dieser Test der Java Reflections API, um Felder und Methoden in der zu testenden Klasse zu ermitteln.

4.1.4.2 OrderingTest

`OrderingTest` prüft die zu testende Klasse auf die Einhaltung der Reihenfolgen von Sichtbarkeiten. Dafür gibt es drei Testmethoden, die sich der Java Reflections API bedienen, um Felder und Methoden in der Klasse zu ermitteln. Ferner wird die Enum `Modifier` für die Festlegung der Reihenfolge verwendet:

- Methodensichtbarkeit `public` → `protected` → `private`:
  ```
  int[] methodModifierOrder = {PUBLIC, PROTECTED, PRIVATE};
  ```
- Klassenvariablen vor Objektvariablen:

```
int[] fieldModifierOrder = {(PUBLIC & STATIC), STATIC, (PUBLIC | PROTECTED
      | PRIVATE)};
```

- Konstanten vor Objektvariablen:

```
int[] fieldModifierOrder = {(PUBLIC & STATIC & FINAL), (STATIC & FINAL),
      FINAL, (PUBLIC | PROTECTED | PRIVATE)};
```

Alle drei Tests verlassen sich auf die Reihenfolge, wie sie von `getMethods()` bzw. `getDeclaredFields()` geliefert werden. Die Eigenschaft von `Modifier`, die Sichtbarkeiten als `int`-Werte 1, 2, 4, 8, 16, … vorzuhalten, erlaubt die bitweise logische Verknüpfung mittels `+` oder `|`. Die folgende Tabelle verdeutlicht dies anhand einiger Beispiele.

Verknüpfung	Codebeispiel
`(PUBLIC & STATIC)`	`public static int classVariable;`
`(PUBLIC & STATIC & FINAL)`	`public static final int classConstant = 1;`
`(PUBLIC | PROTECTED | PRIVATE)`	`public int value1;` oder `protected int value2;` oder `private int value3;`

Wird an nur einer Stelle die gegebene Reihenfolge nicht eingehalten, sind also z. B. `private` Methoden oder Feldern oberhalb von `public` Methoden oder Feldern, schlägt der Test fehl.

4.1.4.3 OverlyChainedMethodCallsTest

`OverlyChainedMethodCallsTest` ermittelt in jeder Methode der zu prüfenden Klasse die Methodenaufrufe. Jeder Methodenaufruf wird als Statement betrachtet, das dann auf die Anzahl der Punkte nach schließenden runden Klammern (`").`") hin untersucht wird. Übersteigt die Anzahl einen Schwellwert (der hier experimentell mit fünf festgelegt wurde), schlägt der Test fehl. Die in Abschnitt 3.1.3.2 beschriebene Ausnahme ist die Anwendung des Erbauermusters, in dem genau diese Methodenverkettung erwünscht ist. Der Test nimmt daher Methoden, die mit `"with"` beginnen, explizit heraus aus der Zählung. Das ist gleichzeitig die Schwäche dieses Tests, denn es ist nirgends vorgeschrieben, dass im Erbauermuster die spezifischen Methoden so anfangen müssen. Es ist lediglich gängige Praxis.

4.1.4.4 NoInnerClassesTest

`NoInnerClassesTest` ermittelt für die zu testende Klasse, ob sie innere Klassen enthält. Dazu wird der Quellcode der Klasse auf weitere Klassendeklarationen hin untersucht. Werden welche gefunden, schlägt der Test fehl.

4.1.4.5 Zwischenfazit zu Convention Tests

Programmierkonventionen sind in unterschiedlichen Organisationen oft verschieden festgelegt. Selbst innerhalb einer Organisation besteht hierzu nicht immer Konsens. Daher sind solche Tests sinnvoll – sofern sich vorher über die einzuhaltenden Konventionen Gedanken gemacht wurden und diese in der Konfiguration der Tests festgehalten sind.

4.1.5 Metrics Tests

Die hier beschriebenen Tests setzen die zu testenden Metriken aus den Kapiteln 2 und 3 um. Abbildung 18 zeigt die Vererbungsstruktur dieser Tests.

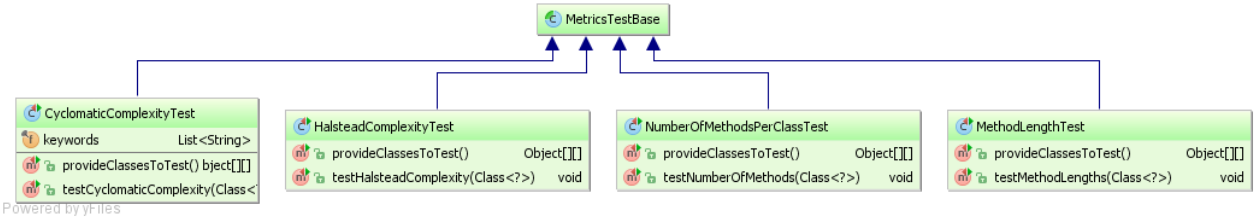


Abbildung 18: Vererbungsstruktur der Metrics Tests (Quelle: Eigene Darstellung)

4.1.5.1 CyclomaticComplexityTest

In Java lässt sich die Messung der zyklomatischen Komplexität nach McCabe trivial durch die Zählung der Ausführungsverzweigungen realisieren. Folgende Schlüsselwörter werden also im Quellcode der zu testenden Klasse pro Methode gezählt:

```
List<String> keywords = Arrays.asList("if", "for", "while", "case",
      "catch", "&&", "||", "?");
```

Die McCabe-Zahl beginnt bei 1, jedes Vorkommen der angegebenen Schlüsselwörter erhöht den Wert um 1. Der Schwellwert der McCabe-Zahl ist in `ArchitectureTestConstants` mit 20 festgelegt. Hier wird die Feldstudie zeigen, welcher Wert tatsächlich am sinnvollsten ist. In der einschlägigen Literatur finden sich Empfehlungen zwischen 10 und 20. Wird der Schwellwert für eine Methode überschritten, schlägt der Test fehl.

4.1.5.2 HalsteadComplexityTest

`HalsteadComplexityTest` bildet aus dem gesamten Quellcode der zu prüfenden Klasse (ohne Kommentare) die gesamte *Wortmenge* (einschließlich Wiederholungen) sowie aus dieser Wortmenge das *Vokabular* (ohne Wiederholungen). Das sog. *Halstead-Volumen HV* wird daraus wie folgt berechnet:

$$HV = \frac{N(Wortmenge) \cdot \ln(N(Vokabular))}{\ln(2)}$$

Die gängige Berechnung der *Halstead-Fehlerquote*[36] *B* (engl. „*Delivered bugs*“) ist:

$$B = \frac{HV}{3000}$$

Überschreitet *B* den gegebenen Schwellwert, schlägt der Test fehl. *B* gibt eine Schätzung der Fehler in der Implementierung allein aufgrund der Größe der Klasse an. Verzweigungen in der Ausführungslogik werden hier nicht betrachtet. Für große Klassen wird auch *B* größer, so dass der festzulegende Schwellwert mit der Größe der Klasse korreliert. Interessant sind die Ergebnisse dieses Tests für neu zu schreibende Klassen, da er eine Aussage darüber liefert, wie viele Bugs in ihnen zu erwarten sind. Für bereits bestehenden Code wird der Test aus genannten Gründen nur begrenzt aussagefähig sein.

4.1.5.3 MethodLengthTest

`MethodLengthTest` ermittelt die Anzahl der Statements pro Methode in der zu testenden Klasse. Dies geschieht trivial auf Quellcodeebene. Übersteigt die Anzahl der Statements den in `ArchitectureTestConstants` mit 20 angegebenen Schwellwert, schlägt der Test fehl. Unter anderen hält Robert C. Martin in [MR08] 20 Codezeilen für den maximal akzeptablen Wert für die Länge einer Methode, um diese noch als les- und damit wartbar zu betrachten.

4.1.5.4 NumberOfMethodsPerClassTest

`NumberOfMethodsPerClassTest` ist ähnlich trivial wie `MethodLengthTest` und 20 ist hier aus dem gleichen Grund der empfohlene Schwellwert für die Anzahl Methoden, die in der zu testenden Klasse vorkommen dürfen. Der Test bedient sich der Java Reflections API, um die Anzahl der Methoden zu bestimmen. Übersteigt der Wert den Schwellwert, schlägt der Test fehl.

4.1.6 Pattern Tests

Es gibt einen interessanten Ansatz, der Entwurfsmuster in bestehendem Code mittels Softwaremetriken und maschinellem Lernen untersucht: In [US14] schreiben Satoru Uchiyama und andere, dass viele automatisierte Ansätze, die auf statischer Codeanalyse beruhen, Probleme haben, Entwurfsmuster, deren Implementierung ähnlich aussieht, zu unterscheiden. Darüber hinaus ist es schwierig, Entwurfsmuster in ihrer Anwendungsvielfalt zu unterscheiden (vgl. [US14], S. 1). Der

36 Die Anzahl der ausgelieferten Bugs (B) ist laut http://www.verifysoft.com/de_halstead_metrics.html, Zugriff am 14.03.2015, die wichtigste Halstead-Metrik: „Beim Softwaretest sollte man so viele Fehler im Modul finden wie die Metrik B angibt.“

Ansatz teilt die Erkennung in eine Lern- und eine Detektionsphase. In der Lernphase lernt ein neuronales Netz den richtigen Kandidaten aus der Auswahl von durch statistische Metriken ermittelten Entwurfsmusterkandidaten mithilfe eines Entwicklers, der entscheidet, welches das richtige verwendete Muster ist, zu unterscheiden. Solche Ansätze sind in der künstlichen Intelligenz und ihren mannigfaltigen Anwendungsfeldern weit verbreitet und erfordern den Aufbau einer geeigneten Trainingsmenge vor dem eigentlichen Lernen. Nachdem die Trainingsmenge erlernt (und wenn möglich, geeignet persistiert) wurde, kann ein solches neuronales Netz weitaus zuverlässiger auch konkrete Anwendungen von Entwurfsmustern erkennen als statische Codeanalyse. Zu diesem Schluss kommen die Autoren von [US14] und beschreiben ferner, wie sie sowohl kleine „small-scale“ Programme als auch „large-scale“ Echtanwendungen untersuchten und die Ergebnisse mit denen statischer Codeanalyse-Tools verglichen.

So reizvoll die Umsetzung des skizzierten Lösungsweges ist, sprengt sie doch den Rahmen dieser Arbeit und ist eine der größeren, späteren Ausbaustufen des Frameworks. Es wird jedoch auf die Auswahl der fünf Entwurfsmuster aus [US14] Bezug genommen in der Implementierung: *Singleton*, *Adapter*, *Template Method*, *State* und *Strategy* (vgl. [US14], S. 5). Für diese 5 Muster werden hier stark vereinfachte Architekturtests implementiert, die lediglich auf bestimmte Merkmale im Code achten und dabei sicher in vielen konkreten Anwendungsfällen falsche Positive produzieren. Abbildung 19 zeigt die Vererbungsstruktur der Pattern Tests.

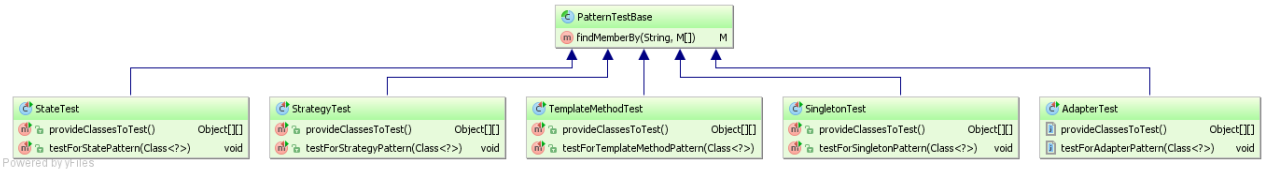


Abbildung 19: Vererbungsstruktur der Pattern Tests (Quelle: Eigene Darstellung)

4.1.6.1 SingletonTest

SingletonTest prüft die zu testende Klasse auf die folgenden Merkmale:

- Es gibt nur einen Konstruktor und dieser ist `private`, nach außen also nicht sichtbar.
- Es gibt eine Methode `getInstance()`, die `public static` ist, also für alle sichtbar und statisch, und sie liefert eine Instanz der Klasse zurück.

Ist eines der Kriterien nicht erfüllt, schlägt der Test fehl.

Weitergehende Tests, ob also die Klasse stets nur eine Instanz erzeugt und bei Existenz einer solchen keine weiteren Instanzen erzeugt und nur die bestehende von `getInstance()` zurückgeliefert

wird, macht der Test nicht. Folglich würde er nicht fehlschlagen, wenn beispielsweise immer nur der Konstruktor der Klasse aufgerufen und stets die neu erzeugte Instanz zurückgeliefert würde.

4.1.6.2 AdapterTest

Vom Adaptermuster gibt es zwei Varianten: Den Klassenadapter und den Objektadapter. Da in Java aber aufgrund der fehlenden Mehrfachvererbung nur der Objektadapter überhaupt zur Geltung kommt, prüft `AdapterTest` eine zu testende Klasse auch nur auf diesen. Der Test prüft, ob die in den Interfaces der Klasse deklarierten Methoden (der eigentliche Adapter) sich in der zu testenden Klasse mit gleichem Namen und Typ des Rückgabewerts wiederfinden. Ist dies nicht der Fall, schlägt der Test fehl.

Nun setzt der Java-Compiler per se voraus, dass ein implementiertes Interface in einer konkreten Klasse alle seine Methoden implementiert hat. Dies erweckt den Eindruck, dass dieser Test unsinnig sei, da er immer erfolgreich ist. Ein Adapter soll jedoch immer in einer Klasse die Implementierungen enthalten. Eine zwischen Interface und zu testender Klasse bestehende (ggf. abstrakte) Klasse etwa, die bereits Teile des Interfaces implementiert, würde dazu führen, dass die zu testende Klasse nicht alle Methoden aus ihren Interfaces implementiert. Ist dies der Fall, schlägt der Test fehl.

Grundsätzlich wird der Einsatz und Nutzen dieses Tests im Feld, wie in der Fallstudie (siehe Abschnitt 5.3) beschrieben, zu untersuchen sein.

4.1.6.3 TemplateMethodTest

`TemplateMethodTest` prüft die zu testende Klasse zunächst daraufhin, ob ihre Superklasse abstrakt ist. Enthält die Superklasse abstrakte Methoden, wird die zu testende Klasse daraufhin untersucht, ob sie diese abstrakten Methoden konkretisiert und diese konkreten Methoden `public` sind. Ist dies nicht der Fall, schlägt der Test fehl. Ferner wird noch überprüft, ob Rückgabetypen und Parameterlisten zwischen Schablone und konkreter Methode übereinstimmen. Ist dies nicht für jede der Methoden der Fall, schlägt der Test fehl.

Dieser Test arbeitet analog zum `AdapterTest`, nur dass das Schablonenmethodenmuster lediglich Methoden und ihre abstrakten Schablonen betrachtet, wohingegen der Objektadapter ein ganzes Interface ist.

4.1.6.4 StateTest

Die Implementierung des `StateTest` ist, wie der `SingletonTest`, relativ starr. Es gibt viele verschiedene Ansätze, Zustandsautomaten zu implementieren – das schließlich ist der Kern des

Zustandsmusters. Für den Test habe ich angenommen, dass die zulässigen Zustände in einer inneren Enumeration definiert sind und jeder dieser Zustände im Automaten mindestens einmal gesetzt, also erreicht wird. Sind die vorgenannten Kriterien nicht erfüllt, schlägt der Test fehl.

4.1.6.5 StrategyTest

Das Strategiemuster entkoppelt spezifisches Verhalten in eine oder mehrere vererbte Methoden, deren konkreter Inhalt zum Zeitpunkt des Aufrufs nicht bekannt ist. Strategien sind semantisch den Adaptern und Schablonenmethoden nahe.

`StrategyTest` ermittelt in der Superklasse der zu testenden Klasse zunächst die darin definierten Felder. Danach prüft der Test, ob alle gefundenen Felder in der zu testenden Klasse je mindestens einmal verwendet werden, sei es lesend oder schreibend. Ist dies nicht der Fall, schlägt der Test fehl. Dies zu überprüfen genügt, da die konkrete Strategie zum Zeitpunkt ihres Aufrufs (über eine oder mehrere abstrakte Klassen) nicht bekannt ist. Jedoch müssen alle Felder, die zu einer konkreten Strategie gehören, auch verwendet werden.

Der Test parst den Quellcode der zu testenden Klasse und untersucht alle Methodenrümpfe, ob die ermittelten Feldnamen der Superklasse darin vorkommen.

4.1.7 Dependency Tests

Erweiternd zur Darstellung in Kapitel 3 ergeben sich bei der Implementierung weitere, sehr unterschiedliche Tests für *Abhängigkeiten*, von denen in diesem Testframework nur einige exemplarisch umgesetzt wurden. Folglich gibt es eine vierte Kategorie von Architekturtests mit der Oberklasse `DependencyTestBase`. Die Abbildungen 20 und 21 zeigen die Vererbungsstruktur der Dependency Tests. Aufgrund der Anzahl der Tests wurden diese auf zwei Diagramme verteilt.

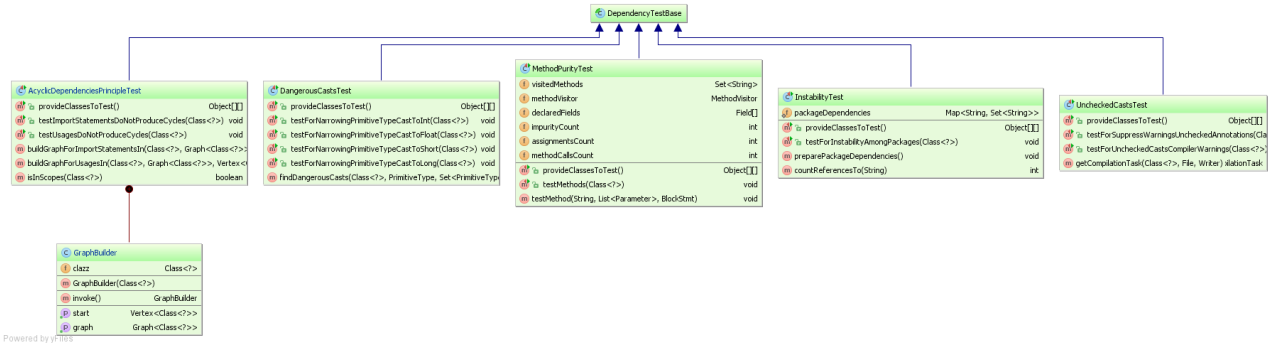


Abbildung 20: Vererbungsstruktur der Dependency Tests, Teil 1 (Quelle: Eigene Darstellung)

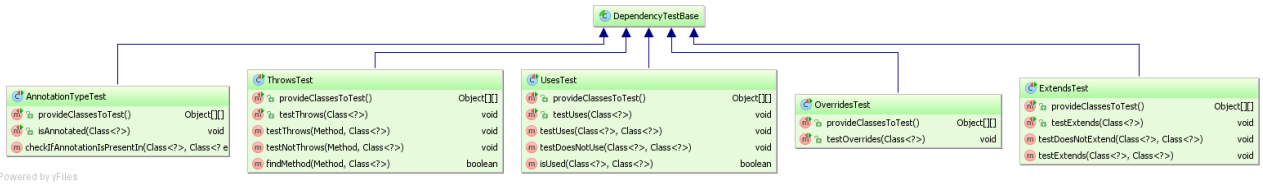


Abbildung 21: Vererbungsstruktur der Dependency Tests, Teil 2 (Quelle: Eigene Darstellung)

4.1.7.1 ExtendsTest

`ExtendsTest` prüft die zu testende Klasse, ob sie von der in `@MustExtend` angegebenen Klasse erbt. Der Test prüft, ob die zu testende Klasse von der in `@MustNotExtend` angegebenen Klasse nicht erbt. Daraus folgt, dass entweder `@MustExtend` oder `@MustNotExtend` für eine Klasse angegeben werden muss, sonst macht der Test nichts. Die Prüfung der Klasse erfolgt mittels der Java Reflections API:

```
Class<?> superClass = clazz.getSuperclass();
```

Ist in `@MustExtend` bzw. in `@MustNotExtend` `entireHierarchy` mit `true` angegeben, prüft der Test rekursiv, ob die zu beerbende Klasse in der Hierarchie steckt. Als Rekursionsabbruch gilt das Antreffen von `java.lang.Object`.

4.1.7.2 ImplementsTest

`ImplementsTest` prüft die zu testende Klasse, ob sie die in `@MustImplement` angegebenen Interfaces implementiert. Der Test prüft, ob die zu testende Klasse die in `@MustNotImplement` angegebenen Interfaces nicht implementiert. Daraus folgt, dass `@MustImplement` oder `@MustNotImplement` für eine Klasse angegeben werden muss, sonst macht der Test nichts. Die Prüfung der Klasse erfolgt mittels der Java Reflections API:

```
Class<?>[] interfaces = clazz.getInterfaces();
```

Sind zu einer zu testenden Klasse sowohl `@MustImplement` als auch `@MustNotImplement` und in beiden dasselbe Interface angegeben, wirft der Test eine `IllegalArgumentException`, da diese Angaben sich gegenseitig ausschließen.

4.1.7.3 ThrowsTest

`ThrowsTest` prüft die zu testende Klasse, ob die mit `@MustThrow` annotierten Methoden in ihr auch tatsächlich die in `@MustThrow` angegebenen `Throwables` werfen. Dementsprechend gilt für mit `@MustNotThrow` annotierte Methoden, dass sie die in `@MustNotThrow` angegebenen

`Throwables` nicht werfen dürfen. Die Prüfung der Klasse erfolgt mittels der Java Reflections API:

```
Class<? extends Throwable>[] throwables =
      method.getExceptionTypes();
```

Analog zum `ImplementsTest` (siehe Abschnitt 4.1.7.2) gilt, ist eine Methode mit `@MustThrow` und mit `@MustNotThrow` annotiert und ist in beiden dasselbe `Throwable` angegeben, wirft der Test eine `IllegalArgumentException`.

4.1.7.4 UsesTest

`UsesTest` prüft die zu testende Klasse, ob sie die in `@MustUse` angegebenen Typen als Felder verwendet. Der Test prüft, ob die zu testende Klasse die in `@MustNotUse` angegebenen Typen nicht verwendet. Daraus folgt, dass `@MustUse` oder `@MustNotUse` für eine Klasse angegeben werden muss, sonst macht der Test nichts. Die Prüfung der Klasse erfolgt mittels der Java Reflections API:

```
Field[] fields = clazz.getDeclaredFields();
for (Field field : fields) {
      Class<?> usedClass = field.getType();
      ...
}
```

Sind zu einer zu testenden Klasse sowohl `@MustUse` als auch `@MustNotUse` und in beiden der selbe Typ angegeben, wirft der Test eine `IllegalArgumentException`, da diese Angaben sich gegenseitig ausschließen.

4.1.7.5 AcyclicDependenciesPrincipleTest

`AcyclicDependenciesPrincipleTest` besteht aus drei Teilen, die nach zyklischen Abhängigkeiten auf drei Ebenen suchen: In `import`-Statements, in Feldern und in Paketen. Im letzten Fall „sammelt“ der Test erst alle Pakete der Klassen, die der Test prüft und führt den eigentlichen Test erst am Schluss, mit `@AfterClass` annotiert, aus.

Das Vorgehen in allen drei Teilen ist immer gleich. Zyklische Abhängigkeiten lassen sich effizient finden, indem man einen Graphen $G = (V, E)$ aus den zu untersuchenden Elementen (als Knotenmenge V) und Abhängigkeiten (als Kantenmenge E) auf seine Kreisfreiheit hin untersucht. Dies macht der Test auch, indem er einen Graphen aus den Klassen in den `import`-Statements bzw. Feldern zur zu testenden Klasse aufbaut und anschließend die Zyklen zählt. Ist die Anzahl der Zyklen größer 0, schlägt der Test fehl. Analog behandelt der Test Pakete: Er baut zum Schluss einen Graphen aus den Paketen aller getesteten Klassen auf und zählt die Zyklen darin.

Für die Implementierung dieses Tests wurde auf die sehr kompakte Graphenbibliothek von Scott Stark zurückgegriffen.[37] Sie besteht aus lediglich sechs Klassen, ist leicht zu integrieren und bietet genau die Funktionalität, die in diesem Test von Nöten ist.

4.1.7.6 InstabilityTest

`InstabilityTest` erstellt zunächst einen Abhängigkeitsgraph der importierten Pakete. Vom Paket der Klasse ausgehend werden die efferente Kopplung *Ce* durch die Anzahl der Referenzierungen des aktuellen Paketes zu anderen Paketen und die afferente Kopplung *Ca* durch die Anzahl der Referenzierungen von anderen Paketen auf Klassen im aktuellen Paket bestimmt. Entsprechend der in Abschnitt 3.1.3.4 vorgestellten Formel wird die Instabilität *I* berechnet. Für den Test existieren zwei Schwellwerte: Ein strikter Schwellwert für *I* von 0,4 und ein gelockerter Schwellwert von 0,8. Bei einer Instabilität von 0,5 spricht man von einer *ausgeglichenen Stabilität*. Wird der strikte Schwellwert überschritten, wird eine Warnung mit Hinweis auf die Klasse ausgegeben, der Test schlägt jedoch nicht fehl. Wird jedoch der gelockerte Wert überschritten, schlägt der Test fehl.

4.1.7.7 UncheckedCastsTest

UncheckedCastsTest prüft eine zu testende Klasse auf zwei Arten, ob Cast-Operationen vorhanden sind, die „*unchecked*“ sind, wie das folgende Beispiel aus dem `MybetRuleGenerator` zeigt:

```
MustNotImplementRule rule = new MustNotImplementRule(classes,
      createAnnotation(MustNotImplement.class, "interfaces",
            Arrays.asList(Serializable.class)));
```

Die Methode

```
static <A extends Annotation> A createAnnotation(Class<A>, String, List<?>)
```

erwartet als dritten Parameter eine generische List beliebigen Typs. Die Methode

```
static <T> List<T> asList(T...)
```

liefert jedoch bei der Hereingabe von `Serializable.class` eine getypte `List<Serializable>` zurück. Dies ist eine unchecked (wenngleich hier ungefährliche) implizite Cast-Operation, die der Compiler bemerkt und vor dieser warnt.

Der erste Test prüft den Quellcode der zu testenden Klasse, ob darin die `@SuppressWarnings`-Annotation auftritt mit dem Attribut `unchecked`:

```
@SuppressWarnings("unchecked")
```

37 Siehe http://www.java2s.com/Code/Java/Collections-Data-Structure/Adirectedgraphdatastructure.htm, Zugriff am 28.10.2014).

Ist dies der Fall, schlägt der Test fehl. Der zweite Test ruft aus dem JVM-eigenen `ToolProvider` den `SystemCompiler` mit der Option "-Xlint:unchecked" und der zu testenden Klasse auf. Die Option bewirkt, dass der Compiler die Warnungen zu unchecked Casts ausgibt. Die Compilerausgabe wird anschließend dahingehend untersucht, ob die Ausgabe "warning: [unchecked] unchecked conversion" enthalten ist. Ist dies der Fall, schlägt der Test fehl.

Beide Tests sind erforderlich, da die Annotation im ersten Test bewirkt, dass der Compiler solche unchecked Casts zwar bemerkt, deren zugehörige Warnungen jedoch unterdrückt.

4.1.7.8 DangerousCastsTest

Dieser Test ist wohl einer der Kandidaten, denen ich am meisten Potenzial zur Erweiterung beimesse. „Gefährliche“ Cast-Operationen gibt es, mathematisch gesehen, so viele, wie es in einem *vollständigen bipartiten* Graphen $G = (V, E)$ aller Typen (als Knotenmenge V mit Bipartion $\{A, B\}$) in Java Wege (als Kantenmenge E, in der jeder Knoten aus A mit jedem Knoten aus B verbunden ist) gibt. Stets läuft man als Entwickler Gefahr, durch Cast-Operationen Daten zu verlieren, zu verändern oder unvorhergesehene Seiteneffekte zur Laufzeit zu provozieren.

`DangerousCastsTest` beschränkt sich auf numerische primitive Datentypen und mahnt stets an, wenn eine Cast-Operation von einem höheren auf einen niederen Datentyp derselben Ko-Domäne (engl. „Codomain“) stattfindet, also z. B. von `int` auf `short`, `long` auf `int` oder `short` oder von `double` auf `float`. Darüber hinaus wird auch geprüft, ob so von einer anderen Ko-Domäne gecastet wird, dass zwangsläufig Informationen verloren gehen, also Nachkommastellen verworfen werden. Dies geschieht z. B. beim Casten von `float` auf `int` oder von `double` auf `long`. Zwar sind sämtliche dieser Cast-Operationen in der *Java Language Specification*[38] eindeutig geregelt und führen nicht zu Problemen zur Laufzeit, jedoch die „unüberlegten“ Seiteneffekte sind es, die zu zunächst nicht erklärbarem Verhalten führen, insbesondere dann, wenn die Wertegrenzen der jeweiligen Datentypen erreicht werden.

Der Test parsiert jede Methode der zu testenden Klasse auf Cast-Operationen und ermittelt für jede dieser Operationen zunächst den Zieldatentyp. Ist der Quelldatentyp des Ausdrucks in den „gefährlichen“ Quelldatentypen für den Zieldatentyp enthalten, schlägt der Test fehl.

4.1.8 Andere Klassen

Die Tests kommen nicht ohne weitere Klassen aus, die ihnen die benötigten Datenstrukturen und die Schnittstelle des Architekturtest-Moduls nach außen bereitstellen. Die Tests selbst sind von außen nicht sichtbar. Abbildung 22 zeigt die für die Testausführung wichtigsten Klassen.

38 http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html, Abschnitt 5.5, Zugriff am 28.02.2015

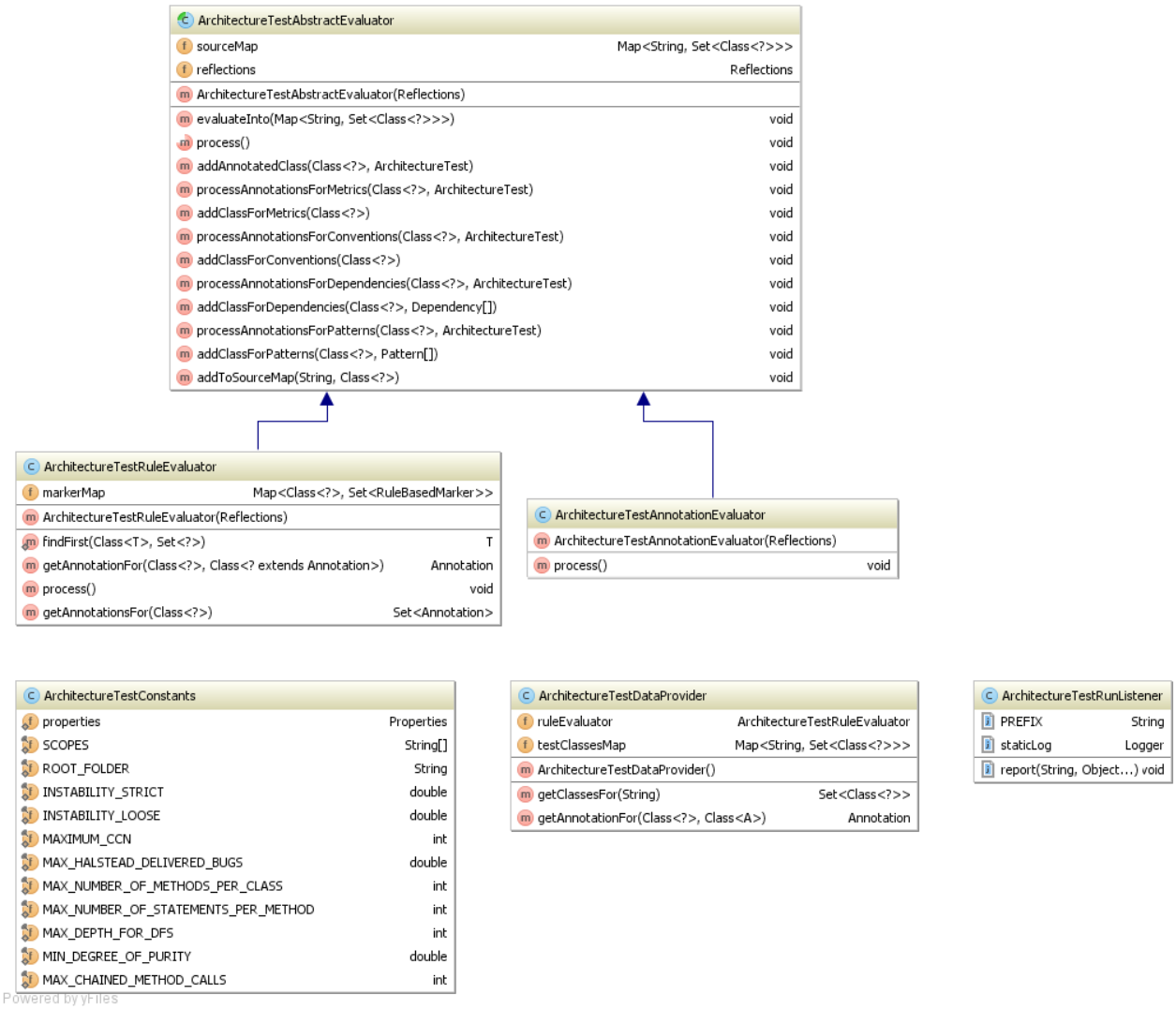


Abbildung 22: Klassendiagramm der Hilfsklassen (Quelle: Eigene Darstellung)

4.1.8.1 ArchitectureTestRunListener

`ArchitectureTestRunListener` ist der *Listener* für JUnit, der verwendet wird, um die Testausgaben (hier vorerst) auf der Konsole auszugeben. Er enthält eine Klassenmethode `report`, die mindestens die auszugebende Nachricht erhält und optional weitere Objekte, die zur Meldung gehören, wie z. B. Ausnahmen, die gefangen wurden.

4.1.8.2 ArchitectureTestAbstractEvaluator

`ArchitectureTestAbstractEvaluator` ist die Basisklasse für die beiden konkreten Klassen `ArchitectureTestRuleEvaluator` sowie `ArchitectureTestAnnotation-Evaluator` und enthält die gemeinsame Funktionalität für die Auswertungen der Annotationen. Sie ordnet die zu testenden Klassen anhand ihrer Annotationen, unabhängig davon, ob es sich um

„echte“ Annotationen aus dem Quellcode handelt oder um zur Laufzeit durch Regeln generierte Annotationen, den jeweiligen Tests zu. Diese Zuordnung wird in einer `HashMap` statisch vorgehalten und steht sodann dem `ArchitectureTestDataProvider` während der gesamten Ausführungszeit der Tests zur Verfügung. Die abstrakte Methode `process()` muss von jeder erbenden konkreten Klasse implementiert werden.

4.1.8.3 ArchitectureTestRuleEvaluator

In `AbstractArchitectureTestRuleEvaluator` werden in der vererbten und hier implementierten Objektmethode `process()` zunächst alle konkreten Regelgenerierer (die von `ArchitectureTestRuleGenerator` erben, siehe Abschnitt 4.1.3) ermittelt und nacheinander instanziiert. Jede dieser Instanzen erzeugt die Regeln, die jeweils so ausgewertet werden, dass aus ihnen anhand der Annotationen, die zur Laufzeit erzeugt werden, die Zuordnung der zu testenden Klassen (durch `ArchitectureTestAbstractEvaluator`) erfolgt.

4.1.8.4 ArchitectureTestAnnotationEvaluator

In `AbstractArchitectureTestAnnotationEvaluator` werden in der vererbten und hier implementierten Objektmethode `process()` zunächst alle Klassen ermittelt, welche mit `@ArchitectureTest` annotiert sind. Anhand dieser erfolgt die Zuordnung der zu testenden Klassen (durch `ArchitectureTestAbstractEvaluator`).

4.1.8.5 ArchitectureTestConstants

`ArchitectureTestConstants` enthält die *Constraints* für die Tests als öffentliche Klassenkonstanten, die aus der *architecture-tests.properties*-Datei ausgelesen werden. Diese Datei wird zur Compilezeit von Maven, z. B. im Buildziel *install*, erzeugt. Die Constraints sind ursächlich in der *pom.xml*-Datei des Moduls *test/architecture* in `<properties>` vermerkt. `ArchitectureTestConstants` stellt also eine *Convenience*-Klasse für das Abrufen der Constraints dar.

4.1.8.6 ArchitectureTestUtil

`ArchitectureTestUtil` stellt alle weiteren Hilfsmethoden als Klassenmethoden bereit, die von allen anderen Klassen im Framework, einschließlich der Tests, benötigt werden. Besonders hervorzuheben sind dabei folgende Funktionalitäten:

- Aufbau statischer Datenstrukturen für das schnelle Finden aller Quelldateien und Bytecodedateien,

- Convenience-Methoden für das Erzeugen von Annotationen zur Laufzeit und
- Convenience-Methoden für das Aufrufen des Parsers auf unterschiedlichen Quellen (`Class`, `BlockStmt`, `AssignExpr`) mit einem *Visitor*.

Insbesondere für den zweiten Punkt musste recht viel implementiert werden. Spezielle Hilfsklassen, sogenannte *Marker*, stellen als *Wrapper*klassen die Möglichkeit dar, alle Attribute ihrer zugehörigen Annotationen zu enthalten und erst bei Bedarf aus ihnen die eigentlichen Annotationen mittels *Inline*-Definitionen zu erzeugen. Diese sind für die Regeln erforderlich, um so erzeugte Annotationen, die nicht im Quellcode stehen, zu erhalten und mit ihnen genauso arbeiten zu können, als wären sie „echte“ Annotationen. So gibt es z. B. für `@ArchitectureTest` die Klasse `ArchitectureTestMarker`, die von `RuleBasedMarker<ArchitectureTest>` erbt.

4.1.9 Verwendete Bibliotheken

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tngtech.java</groupId>
        <artifactId>junit-dataprovider</artifactId>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
    </dependency>
    <dependency>
        <groupId>org.reflections</groupId>
        <artifactId>reflections</artifactId>
    </dependency>
    <dependency>
        <groupId>com.google.code.findbugs</groupId>
        <artifactId>jsr305</artifactId>
    </dependency>
    <dependency>
        <groupId>com.google.code.javaparser</groupId>
        <artifactId>javaparser</artifactId>
    </dependency>
</dependencies>
```

Abbildung 23: Auszug aus der pom.xml des Moduls test/architecture (Quelle: Eigene Darstellung)

Im Testframework kommen die Bibliotheken zum Einsatz, die in der pom.xml des Moduls test/architecture als Abhängigkeit angegeben sind (siehe Abbildung 23). Besonders wichtig sind dabei JUnit, JUnit-Dataprovider und der Java-Parser. Im Verlauf der Entwicklung wurden auch andere Bibliotheken verwendet, unter anderem JavaNCSS, es stellte sich jedoch heraus, dass die hier konzipierten Aufgaben allesamt mit dem Java-Parser, Version 1.0.8 lösbar sind und zwar so, dass größtmögliche Transparenz im Code gegeben ist, etwa durch das Schreiben eigener, angepasster Visitor. Von der Java SE, Version 1.6 werden die Reflections API sowie Annotations verwendet.

SLF4J ist eine verbreitete Bibliothek, die komfortable Logging-Funktionalität bereitstellt. Diese wird für die Ausgaben der Tests verwendet. Da Java 1.6 keine Annotationen für die Definition von Kontrakten enthält, wie z. B. `@Nonnull` oder `@Nullable`, wird hier das Artefakt *JSR305* aus *FindBugs* eingebunden. Ab Java 1.7 steht dies auch ohne diese Bibliothek zur Verfügung.

Besonders hervorzuheben ist, dass keine Module der mybet-Plattform eingehängt werden. Dies ist Absicht, um das Framework vom zu testenden Code unabhängig zu halten. Näher erläutert wird dies in Abschnitt 4.2.3 und in Kapitel 5.

4.1.10 Zusammenfassung

Während der Implementierungsphase wurden die in der folgenden Tabelle angegebenen Tests erstellt. Die Paketstruktur der Tests spiegelt die in Kapitel 3 konzipierte Einteilung in Konventionen, Metriken, Abhängigkeiten und Muster wider.

Testname	Funktion (auszugsweise)
`CamelCaseTest`	Groß- und Kleinschreibungsregeln entsprechend CamelCase für Java
`NoInnerClassesTest`	Keine inneren Klassen
`OrderingTest`	Reihenfolge der Sichtbarkeit von Feldern und Methoden in Klassen
`OverlyChainedMethodCallsTest`	Keine zu langen „Punkteketten“ (verkettete Methodenaufrufe)
`AcyclicDependenciesPrincipleTest`	Abhängigkeiten sollen zyklenfrei sein
`AnnotationTypeTest`	Bestimmte, vorgegebene Annotationen müssen vorhanden / dürfen nicht vorhanden sein
`DangerousCastsTest`	Castoperationen von größeren zu kleineren Datentypen (die zu Datenverlusten führen können) sollen nicht vorkommen
`ExtendsTest`	Eine bestimmte, vorgegebene Klasse muss beerbt / darf nicht beerbt werden
`ImplementsTest`	Bestimmte, vorgegebene Interfaces müssen implementiert / dürfen nicht implementiert werden

Testname	Funktion (auszugsweise)
`InstabilityTest`	Die Instabilität der Klasse, berechnet aus afferenter und efferenter Kopplung, darf den gelockerten Schwellwert nicht überschreiten
`MethodPurityTest`	Methodenreinheit soll gegeben sein
`OverridesTest`	Bestimmte, vorgegebene Methoden (mittels Methodenname) müssen überdeckt / dürfen nicht überdeckt werden
`ThrowsTest`	Bestimmte, vorgegebene Fehler oder Ausnahmen müssen geworfen / dürfen nicht geworfen werden (pro Methode)
`UncheckedCastsTest`	Keine ungeprüften Castoperationen
`UsesTest`	Bestimmte, vorgegebene Typen müssen verwendet / dürfen nicht verwendet werden
`CyclomaticComplexityTest`	Die zyklomatische Komplexität darf den Schwellwert nicht überschreiten
`HalsteadComplexityTest`	Die Halstead-Komplexität darf den Schwellwert nicht überschreiten (überarbeitungsbedürftig, da der Wert mit der Länge der Klasse korreliert)
`MethodLengthTest`	Die Länge einer Methode darf den Schwellwert nicht überschreiten
`NumberOfMethodsPerClassTest`	Die Anzahl von Methoden in einer Klasse darf den Schwellwert nicht überschreiten
`AdapterTest`	Das Adaptermuster ist richtig implementiert
`SingletonTest`	Das Singletonmuster ist richtig implementiert
`StateTest`	Das Zustandsmuster ist richtig implementiert
`StrategyTest`	Das Strategiemuster ist richtig implementiert
`TemplateMethodTest`	Das Schablonenmethode-Muster ist richtig implementiert

Sämtliche Komponenten des Frameworks wurden mit der Zielstellung entwickelt, vorrangig die *Idee* hinter dynamischen, aber dennoch generischen Architekturtests, die auf einem bekannten Unit-Test-Framework aufbauen, zu skizzieren. So sind sämtliche der hier entworfenen und implementierten Testarten um weitere Tests erweiterbar. Jeder Entwickler, der Tests mit JUnit im Speziellen und Unit-Tests im Allgemeinen implementieren kann, ist nach kurzem Studium der API des Frameworks in der Lage, selbst Architekturtests zu schreiben, bestehende anzupassen oder (zusätzliche) Regelgenerierer zu erstellen. Diese Eigenschaft ist die Grundvoraussetzung für eine erfolgreiche Integration in die mybet-Softwareinfrastruktur., die im folgenden Abschnitt beschrieben wird. Nicht zuletzt ist sie auch Voraussetzung für die Akzeptanz derart implementierter Architekturtests.

4.2 Integration des Frameworks

Die Integration des Frameworks erfolgte während der Implementierungsphase und betrifft alle

verwendeten Deployment- und Continuous-Integration-Werkzeuge. Zunächst wird aber darauf eingegangen, wie das Framework mit seinen Tests per Regeln und per Annotationen exemplarisch Teile des Quellcodes testet. Danach folgt die Betrachtung der Integration in Maven und Jenkins – einschließlich der Fallstricke, die sich in den Weg stellten.

4.2.1 Regeln

Obwohl zwar die Annotationen im Konzept in Kapitel 3 zuerst erdacht wurden, erweist sich die Einführung der Architekturtests mittels erster Regeln als mächtiger. Es wurden zunächst drei Regeln aufgestellt und zu jeder dieser Regeln ein Regelgenerierer implementiert. Jeder Regelgenerierer hat mindestens einen Unit-Test, der sicherstellt, dass er auch die richtigen Regeln erstellt.

Die drei initialen Regeln sind für jede Klasse *C*, die Subtyp von `SimpleEntityData` ist:

- Wird `Cloneable` von *C* implementiert, muss `clone()` in *C* überdeckt werden.
- Wird `Cloneable` von *C* nicht implementiert, darf `clone()` in *C* nicht überdeckt werden.
- `Serializable` darf von *C* nicht implementiert werden.

Alle drei Regeln sind grundlegende Architekturentscheidungen für die mybet-Plattform. Sie erfassen aktuell rund 200 Klassen, und damit rund 400 Testausführungen: Die dritte Regel betrifft alle Klassen, die ersten beiden Regeln schließen sich gegenseitig aus, die Menge der Klassen für die erste Regel ist also disjunkt von der Menge der Klassen für die zweite Regel. Der Vorteil ist, dass die Regeln immer alle Klassen, auf die zur Laufzeit die Eigenschaften, hier Subtypen von `SimpleEntityData`) und Bedingungen, hier `Cloneable` wird oder wird nicht implementiert, zutreffen. Diese Klassen werden durch die Architekturtests erfasst, ohne dass Annotationen an die Klassen selbst geschrieben müssen.

Die erste Ausführung der Tests anhand der Regeln ergab, dass `ImplementsTest` keinerlei Regelbrüche für die dritte Regel festgestellt hat. `OverridesTest` hat für Regel eins ebenfalls keine Brüche festgestellt. Bei Regel zwei stellte er jedoch fest, dass zwei Klassen die Regel brachen: Es gab darin jeweils eine `clone()`-Methode. Nachfrage bei den zuständigen Entwicklern ergab, dass diese Methoden gefahrlos gelöscht werden konnten. Nach Refactoring der beiden Klassen waren alle Ausführungen von `ImplementsTest` und `OverridesTest` grün. Der Quellcode für die Regelgenerierer sowie für die verwendeten Regeln befindet sich auf der beiliegenden CD-ROM.

4.2.2 Annotationen

Am Beispiel der *mWS-Sportsbets-Engine* (siehe Abschnitt 3.1.2) wird hier erläutert, wie die

Integration des Frameworks über die Annotationen erfolgt. Zunächst muss in der pom.xml des Moduls, das mit Architekturtests versehen werden soll, die Abhängigkeit eingefügt werden:

```
<dependency>
      <groupId>com.mybet.test</groupId>
      <artifactId>architecture</artifactId>
</dependency>
```

Nun können die Annotationen des Frameworks den Klassen sukzessive hinzugefügt werden. Wichtig ist, zunächst eine Klasse zu untersuchen und zu annotieren, dann zu testen, danach ggf. zu refactoren und erneut zu testen. Erst, wenn zu einer Klasse alle ihr zugewiesenen Tests grün sind, geht man weiter zur nächsten Klasse. Begonnen wird mit der Klasse `B2BRestClientServiceImpl` im Paket `com.mybet.mws.betengine.services` – einem der Pakete, die in Abschnitt 3.1.2 identifiziert wurden, die Geschäftslogik zu enthalten.

```
@ArchitectureTest(dependencies = {
        Dependency.implementsDependency,
        Dependency.dangerousCasts,
        Dependency.instabilityDependency,
        Dependency.acyclicDependenciesPrinciple,
        Dependency.uncheckedCasts
})
@MustImplement(interfaces = B2BRestClientService.class)
public class B2BRestClientServiceImpl implements B2BRestClientService {
    …
}
```

Die Annotationen für `B2BRestClientServiceImpl` bewirken, dass sämtliche Metriken, sämtliche Konventionen sowie die folgenden Abhängigkeiten getestet werden: `ImplementsTest`, `DangerousCastsTest`, `InstabilityTest`, `Acyclic-DependenciesPrincipleTest` und `UncheckedCastsTest`. Das Ergebnis der Tests für die Klasse zeigt, dass `HalsteadComplexityTest` fehlschlägt. In Abschnitt 4.1.5.2 wurde bereits diskutiert, warum dies bei bereits existierendem Code auftreten kann. Da die Klasse mit rund 200 Zeilen auch recht groß ist, verwundert dies umso weniger. In der Folge muss der Schwellwert für die erwartete Fehlerquote erhöht werden, bis der Test grün wird.

Als Nächstes wird die Klasse `BetEngineModule`, ebenfalls im Paket `com.mybet.mws.betengine.services` enthalten, den Architekturtests unterzogen. Dabei werden ihr die gleichen Tests zugewiesen wie `B2BRestClientServiceImpl`. Lediglich ImplementsTest wird hier nicht herangezogen, da die Klasse kein Interface implementieren muss:

```
@ArchitectureTest(dependencies = {
        Dependency.dangerousCasts,
        Dependency.instabilityDependency,
        Dependency.acyclicDependenciesPrinciple,
        Dependency.uncheckedCasts
})
public class BetEngineModule {
    …
}
```

Im Ergebnis zeigen sich zwei Probleme: `OrderingTest` schlägt fehl, da die Reihenfolge der Methodensichtbarkeit (`public` → `protected` → `private`) nicht eingehalten wurde. Das Refactoring der Klasse, bei dem die Methoden in die richtige Reihenfolge gebracht wurden, ließ den Test grün werden. Das zweite Problem in dieser Klasse wurde von `UncheckedCastsTest` gefunden: Es befand sich eine `@SuppressWarnings(„unchecked")`-Annotation im Quellcode. Nach Entfernen dieser Annotation und Refactoren der Klasse wurde auch dieser Test grün.

Im Verlauf der Einführung des Frameworks per Annotationen wurden alle konkreten Klassen, die verwendet werden (nicht gebrauchter Code muss auch nicht getestet werden), annotiert in der gleichen Weise, wie eben beschrieben. Dabei kamen rund 900 Einzeltestausführungen bei rund 45 annotierten Klassen hinzu, die nun regelmäßig ausgeführt werden. Die häufigsten initialen Probleme wurden durch `OrderingTest`, `NumberOfMethodsTest`, `InstabilityTest` und `UnckeckedCastsTest` gefunden, in rund 35 der 45 Klassen. Wird also am Code der mWS-Sportsbets-Engine etwas geändert, so werden die Architekturtests an den Stellen fehlschlagen, die durch die Änderungen dazu führen, dass Architekturvorgaben nicht mehr eingehalten werden. Der Vorteil der Anwendung des Frameworks auf diese Weise ist, dass durch die Annotationen sukzessive und feingranular getestet werden kann. Der Nachteil ist allerdings, dass die Annotationen der Klassen selbst gepflegt werden müssen.

4.2.3 Maven

In Maven sind zwei Module innerhalb der mybet-Plattform entstanden:

- *test/architecture* und
- *share/architecture-rules*.

In test/architecture befindet sich das eigentliche Framework einschließlich seiner Hilfsklassen. Das Modul wird per Abhängigkeit von anderen Modulen referenziert (z. B. um die Annotationen in

anderen Modulen der mybet-Plattform verwenden zu können), es gibt jedoch keinerlei Referenzen von test/architecture auf andere Module der mybet-Plattform. Der Grund liegt zum Einen in der Vermeidung zyklischer Abhängigkeiten auf Modulebene, zum Anderen in dem Anspruch, test/architecture unabhängig von jeglichem Produktivcode, den es ggf. testen soll, zu halten. Das wäre nicht zuletzt eine Voraussetzung, das Framework bspw. als *Open Source*-Projekt ausgliedern zu können.

Die Regelgenerierer hingegen sollen Teil des Produktivcodes der mybet-Plattform sein. Daher sind sie innerhalb des *share*-Moduls als eigenes Modul *share/architecture-rules* angesiedelt. Es stellt Abhängigkeiten zu anderen Modulen hier, wie z. B. *mas/share*, in dem sich Klassen befinden, die von `SimpleEntityData` erben, genau so wie es zwangsläufig eine Abhängigkeit zu test/architecture hat. Die *pom.xml*-Dateien zu beiden Modulen befinden sich auf der beiliegenden CD-ROM.

4.2.4 Jenkins

Das Modul test/architecture wurde in Jenkins genau so konfiguriert, wie alle anderen Module auch. Zusätzlich wurde eingestellt, dass vor jedem Build des Moduls Folgendes innerhalb des Modulordners in Jenkins auf der Kommandozeile ausgeführt wird:

1. `svn up`
2. `mvn compile`

Punkt 1 hat zur Folge, dass im Modulordner der komplette Codebaum aktualisiert wird (inkrementell). Punkt 2 sorgt dafür, dass der Baum, ebenfalls nur inkrementell, kompiliert wird. Das Buildziel `compile` genügt hier, da hierfür keine Ressourcen verarbeitet oder Application-Server-spezifische Pakete (z. B. *ear*- oder *war*-Archive) oder sonstige Artefakte gebaut werden müssen. Das Übersetzen des Quell- in Bytecode genügt für die Architekturtests völlig. Das erstmalige Abrufen und Bauen benötigte rund 10 Minuten. Alle weiteren Ausführungen dieser beiden Befehle benötigen (derzeit) deutlich unter einer Minute. Dabei ist zu berücksichtigen, dass Jenkins bei mybet auf drei sogenannten *Slaves* sowie mehreren *Teamservern* (jedes Entwicklerteam hat seinen eigenen Teamserver) die Continuous-Integration vornimmt. Zum Vergleich sei herangezogen, dass bspw. Unit-Tests für JavaScript, die mittels *Jasmine* und der Java-eigenen *Rhino-Engine* zum Ausführen von JavaScript innerhalb der JVM gebracht werden, derzeit etwa die doppelte Zeit benötigen – das ist nur die Testausführung selbst. Die Architekturtests hingegen sind innerhalb von fünf Sekunden durchgelaufen.

4.2.5 Zusammenfassung

Etwa 1.350 regelmäßige Einzeltestausführungen auf insgesamt rund 240 Klassen sind das Ergebnis der Einführung der Architekturtests. Bei einem Gesamtvolumen von rund 6.600 Klassen ist dies natürlich nur ein Anfang bei einer „Testabdeckung“ von 3,6% und auch nur einer auf bestehendem Code. Etwa 40 Klassen waren in der Folge von Refactoring-Maßnahmen betroffen. Mit Testabdeckung ist hier gemeint, dass diese sich auf die *Eigenschaften* des Codes bezieht und nicht auf die Funktionalität. Die funktionale Testabdeckung (im eigentliche Sinne des Begriffes) ist hier nicht untersucht worden. Auch muss man sagen, dass nicht alle Tests auf allen genannten Klassen ausgeführt wurden, so dass man von weniger als 3,6% Architekturtestabdeckung ausgehen muss. Dieser Abschnitt 4.2 soll lediglich aufzeigen, welche Schritte für die Einführung erforderlich waren und was mit dieser Einführung erreicht wurde. Weitere Erkenntnisse wird die Fallstudie (siehe Kapitel 5) liefern. Insbesondere zeigt sich hier, dass sich die Integration von Architekturtests als dynamische Tests nicht allein durch ihre Existenz bewerkstelligen lässt. Es ist unmittelbares Arbeiten mit dem zu testenden Code erforderlich, wie bei anderen dynamischen Tests auch. Der Unterschied ist, dass nicht die Funktionalität des Codes im Fokus steht, sondern seine Merkmale.

4.3 Konsequenzen für die weitere Entwicklung

Das in diesem Kapitel entwickelte und eingeführte Testframework birgt noch einiges an Verbesserungspotenzial. Wo dies auf einzelne Tests direkt zutrifft, sind die inhärenten Schwächen bereits eruiert worden, wie z. B. bei den Entwurfsmustertests oder bei `HalsteadComplexityTest`. Hier geht es nun darum, zu identifizieren, was bei der Verwendbarkeit des Frameworks noch verbessert werden kann, was also eine Konsequenz für die weitere Entwicklung des Frameworks ist.

So wirkt der Einsatz der Rule-API noch etwas „sperrig“. Man muss als Entwickler erst sehr genau verstehen, wie die Regeln aufgebaut sind und wie man daraufhin Regelgenerierer schreibt. Hier sollten Regeln selbst auch vom Framework entkoppelbar sein, nicht nur die Regelgenerierer.

Der Einsatz der Annotationen ist verbesserungswürdig dahingehend, dass viele der möglichen Werte für `dependencies` in `@ArchitectureTest` für die explizite Erwähnung eigentlich nicht nötig sind, da es weitere Annotationen gibt, wie `@MustExtend`, @MustImplement, usw., die derzeit noch nur dann berücksichtigt werden, wenn `extendsDependency` oder `implementsDependency` angegeben sind. Beide sind jedenfalls redundant, so dass diese `dependencies` aus der Enum `Dependency` weggelassen werden könnten. Dieser Verzicht würde auch die Pflege bestehender Annotationen an zu testenden Klassen erleichtern.

Ebenfalls zu verbessern ist noch die Tatsache, dass die Annotationen für die Regeln durch einen recht umständlichen Mechanismus mittels Marker-Klassen zur Laufzeit erzeugt werden müssen. Dies ist der Eigenschaft von Annotationen geschuldet, eigentlich keine „vollwertigen“ Objekte zu sein. Das kann, zumindest mit den in Java 1.6 zur Verfügung stehenden Sprachmitteln, nicht gelöst werden. Hier kann noch ein Ansatz gefunden werden, andere Datenstrukturen zu verwenden, um auf die Marker-Klassen zu verzichten und erst bei Auswertung der Regeln die Klassen den Tests so zuzuordnen, dass das Erzeugen von Annotationen zur Laufzeit durch Regeln unnötig wird.

Insgesamt gesehen ist das Testframework ein Prototyp, der bereits im Einsatz ist. Er soll zeigen, dass dynamische Tests statische Codeanalyse in einer Weise betreiben können, dass sie „von außen“ konfigurierbar sind und sich in bestehende Continuous-Integration-Systeme einfügen lassen wie andere Tests. Diesen Zweck erfüllt das Framework.

5 Bewertung

Die zentrale Argumentation in diesem Kapitel ist, warum in dieser Arbeit Tests entwickelt wurden, die sich auch durch gut etablierte, bekannte statische Codeanalyse-Werkzeuge durchführen ließen. Hierfür werden einerseits unterschiedliche Codeanalyse-Werkzeuge und ihre Funktionen betrachtet und andererseits Argumente herangezogen, die sich aus der Literatur sowie aus der Praxis ergeben.

Anschließend wird evaluiert, wie die entstandenen Architekturtests im Einsatz sind und welchen Mehrwert sie bieten, um die zuvor aufgestellte Argumentation praktisch zu stützen. Abschließend wird das in dieser Arbeit entwickelte Framework auf Möglichkeiten der Anpassbar- und Erweiterbarkeit hin untersucht und es werden Vorschläge dafür erarbeitet.

5.1 Statische Codeanalyse in dynamischen Tests

5.1.1 Codeanalyse-Tools

Eines der Pakete, die ursprünglich bei der Implementierung der Architekturtests zum Einsatz kamen, ist *JavaNCSS*. Es kann in Maven zur Erstellung von Reports eingebunden werden, die unter Anderem die LoC und die zyklomatische Komplexität aufbereitet auf einer Webseite anzeigen können. Hierfür muss Maven jedoch konfiguriert werden mit einem eigens dafür einzurichtenden Buildziel. Dieses Buildziel wird beispielsweise in Continuous-Integration-Umgebungen jedoch nicht immer aufgerufen, anders als Unit-Tests, die zu jedem Commit ausgeführt werden. Daher wurde eine Implementierung der betreffenden Codemetrik-Tests bereits früh in der Implementierungsphase wieder verworfen und stattdessen auf Eigenentwicklung, wie in Kapitel 4 beschrieben, zurückgegriffen.

Weiterhin sind mit *PMD* und *Checkstyle*, zwei weitere, recht verbreitete Werkzeuge zur statischen Codeanalyse, untersucht worden. Für beide gibt es Plugins für IntelliJ IDEA wie auch für andere Entwicklungsumgebungen. Darüber hinaus existieren kommerzielle Produkte wie *SonarJ* (heißt mittlerweile *Sonargraph for Java*) und *JArchitect*, die weitaus leistungsfähiger sind hinsichtlich der Breite und Tiefe der Codeanalyse. Alle vier Tools aber benötigen Installation, Konfiguration und Einarbeitung, um damit greifbare Ergebnisse zu erzielen. Auch diese werden nicht automatisiert regelmäßig ausgeführt.

Die erwähnten Gründe sprechen gegen den Einsatz dieser Tools im hier gewünschten Kontext der steten Ausführung zur Unit-Testzeit innerhalb der bestehenden CI-Umgebung. Dabei muss noch einmal hervorgehoben werden, dass nicht die Ergebnisqualität der Tools an sich gegen ihre Verwendung spricht.

5.1.2 Dynamische Tests

Dynamische Tests dienen dazu, Code funktional zur Laufzeit auf Korrektheit hin zu testen. Je früher die Tests im Lebenszyklus ansetzen, so wie hier bereits zur Unit-Testzeit, umso schneller werden Defekte gefunden und umso wartbarer werden der defekte Code und die zugehörigen Tests. Das Ziel, die im Rahmen dieser Arbeit implementierten Architekturtests auf der Ebene von Unit-Tests zu realisieren, ist gelungen. Die Ausführungszeit mehrerer hundert Einzeltests beschränkt sich auf rund 2 Sekunden pro ca. 1000 Testausführungen. Statische Codeanalyse-Tools, wie die in Abschnitt 5.1.1 genannten, würden dafür mehrere Minuten benötigen – vorausgesetzt, sie sind bereits fertig eingerichtet und konfiguriert. Für die Tests müssen keine Testdaten gemockt und keine Testumgebung erzeugt werden. Was jedoch konfiguriert und bereitgestellt sein muss, sind die Annotationen in den zu testenden Klassen im gesamten Projekt. Hierfür muss die Sichtbarkeit des Architekturtest-Moduls für Maven in den jeweiligen pom.xml-Dateien hergestellt sein. Für Jenkins mussten alle Klassen und Quellcodedateien in ein Verzeichnis im Dateisystem geladen werden vor der Ausführung, damit die Tests alle Quellcode- und Klassendateien in derselben Weise finden, wie in der lokalen Entwicklungsumgebung. Dies beansprucht nur bei der ersten Ausführung mehrere Minuten, da zunächst der Code mittels Subversion ausgecheckt und dann gebaut wird. Für alle weiteren Ausführungen werden '*svn up*' und '*mvn compile*' nacheinander aufgerufen, was nur inkrementelle Änderungen berücksichtigt und deutlich schneller vonstatten geht. Für derzeit rund 6.600 Quellcode- und mindestens ebenso viele Klassendateien beträgt die Dauer für die Vorbereitung der Tests etwa 3 Sekunden in der CI-Umgebung. Insgesamt sind es also etwa 5 Sekunden Testausführungszeit der Architekturtests und das Ziel, die Buildzeit der mybet-Plattform nicht wesentlich zu verlängern, wurde erreicht.

Weiterhin ist das Ziel, Architekturtests verständlich für Entwickler bereitzustellen und die zugehörigen APIs so offen zu gestalten, dass Erweiterbarkeit ohne aufwändige Einarbeitung gegeben ist, gelungen. Zum Einen sind die bestehenden Tests konfigurierbar, bspw. die Schwellwerte für die Metriken, oder das gezielte Ein- und Ausschalten einzelner Testfeatures bis hin zur Log-Ausgaben der Testergebnisse. Zum Anderen können weitere Architekturtests hinzugefügt und diese entweder per Annotation oder mittels Erweiterung des Regelwerkes auf spezifizierte Klassen hin zur Ausführung gebracht werden. Dies hat zur Folge, dass neue Erkenntnisse im Bereich der Softwarearchitekturen und zugehöriger Paradigmen flexibel in das hier präsentierte Framework eingearbeitet werden können.

Statische Codeanalyse innerhalb einer Entwicklungsumgebung wie IntelliJ IDEA ist nur dann organisationsweit sinnvoll einsetzbar, wenn alle Entwickler dieselbe Konfiguration hinsichtlich Codekonventionen verwenden. Dies ist jedoch nicht garantiert und meistens nicht der Fall. In

Abschnitt 4.1.4.3 wird deutlich, dass Codekonventionen, die in dynamischen Tests fixiert sind, stets und unabhängig von der verwendeten Entwicklungsumgebung greifen.

5.1.3 Zusammenfassung

Entscheidend ist, dass die Automatisierung statischer Codeanalyse innerhalb dynamischer Tests keine separat einzurichtenden Werkzeuge erfordert, sondern direkt in bestehenden Continuous-Integration-Umgebungen auf der Ebene von Unit-Tests und mit den bekannten Logging- und Monitoring-Mechanismen überwacht werden kann. Das löst die statische Codeanalyse heraus aus dem Spezialwissen einiger weniger Mitarbeiter im Unternehmen und trägt es hinein in den Kreis der Entwickler. So wird auch der Fokus in gleichem Maße wie auf funktionale Tests auf Tests mit statischer Codeanalyse gelenkt. Die erhöhte Aufmerksamkeit aller an der Entwicklung Beteiligten trägt so nachhaltig dazu bei, Softwarequalität nicht nur zu etablieren, sondern vor allem über den gesamten Lebenszyklus der Software zu erhalten.

Die betrachteten Tools zur Codeanalyse sind in ihrer Funktionalität größtenteils mächtiger und präziser als das hier entwickelte Framework. Die beiden entscheidenden Vorteile sind jedoch, dass die Codeanalyse selbst offen zugänglich in derselben Sprache geschrieben und daher beliebig erweiterbar ist, und andererseits nach jedem Commit in der CI-Umgebung ausgeführt wird – einschließlich der Anzeige aller Resultate und zugehöriger Build-Ampel für das Modul „*Test - Architecture*“.

5.2 Architekturtests im Einsatz

Zu Beginn der Arbeit am Framework und an dieser Ausarbeitung wurde davon ausgegangen, dass es am Sinnvollsten sei, sich auf einzelne Teilprojekte innerhalb des mAS und des mFES zu konzentrieren. Im Einsatz der Architekturtests hat sich allerdings gezeigt, dass, vor allem durch die Rule-API, die Möglichkeiten des schrittweisen Einsatzes vielmehr auf bestimmte Arten von Klassen, die spezifische Eigenschaften aufweisen oder von besonders zentraler Bedeutung sind, zu beziehen sind. So waren die ersten Regeln auf zentrale Entitäten, deren Interfaces und sauber zu überdeckende Methoden bezogen, um eine „Reinheit“ dieser Klassen, die immerhin einige hundert sind, zu gewährleisten. Hier erwies sich die umgebungsspezifische Implementierung eines Regelgenerierers mithilfe der Rule-API als hilfreich und nützlich, da dieser die händische Annotation jeder einzelnen Klasse überflüssig macht. Zudem hätte das händische Annotieren auch zur Folge, dass neu hinzukommende Entitäten vergessen werden können. Eventuelle Brüche in den Architekturvorgaben wären somit nicht sichtbar.

5.3 Gesammelte Erfahrungen

Um erste Erfahrungen mit den Architekturtests strukturiert zu erhalten, wurde im Anschluss an die Implementierung und das Testen des Frameworks eine *Fallstudie* durchgeführt, die sich in zwei Teile gliedert. Als Erstes wurde, wie in Abschnitt 4.2 beschrieben, das Framework exemplarisch in je ein Teilprojekt aus mAS und mWS integriert, sowohl unter Verwendung der Annotationen, als auch mit Hilfe der Rule-API. Der zweite Teil bestand in der Aufgabe, ein neues (Beispiel-)Feature in je einem anderen Teilprojekt aus mAS und mWS unter Zuhilfenahme des Frameworks zu implementieren entsprechend dem „Test-First“[39]-Prinzip.

5.3.1 Fallstudie – Aufgaben

Die erste Aufgabe besteht darin, die Annotationen des Frameworks zu verwenden, um ausschließlich bestehenden Code so zu annotieren, dass möglichst viele der Architekturtests für jede der annotierten Klassen ausgeführt werden. Dabei soll darauf geachtet werden, nach der Testausführung die Klassen mit fehlgeschlagenen Architekturtests dahingehend zu beheben, dass die Tests ausschließlich durch Refactoring grün werden.

Die zweite Aufgabe soll die Implementierung eines klar abgegrenzten neuen Features (innerhalb des etablierten *Scrum*[40]-Prozesses in einer *User-Story*[41]) so vollzogen werden, dass an jede neue Klasse als Erstes die vollständige `@ArchitectureTest`-Annotation geschrieben wird:

```
@ArchitectureTest(dependencies = Dependency.all)
```

Hierbei werden noch keine Entwurfsmustertests ausgeführt. Sobald bekannt ist, dass einzelne Klassen mindestens eines der fünf Entwurfsmuster (vgl. Abschnitt 4.1.6) des Frameworks verwenden, sollen diese mittels des `patterns`-Attributs der Annotation hinzugefügt werden. Gleichermaßen gilt, sobald die Vererbungshierarchie klar ist, soll diese durch die Annotationen `@MustExtend` und `@MustImplement` fixiert werden. Zu den *UACs*[42] der User-Story gehört, dass kein Architekturtest fehlschlägt. Die umgesetzte User-Story befindet sich im Anhang.

Festgehalten werden alle Auffälligkeiten, sowohl Hilfestellungen durch das Framework als auch Hindernisse. Bei beiden Aufgaben ist jeweils die Gesamtzahl der annotierten Klassen festzuhalten.

---

39 Als *Test-First*-Ansatz bezeichnet man ein Vorgehen bei der Implementierung, zuerst die Tests zu schreiben und danach den zu testenden Code. Dies erfordert, dass die Tests die Spezifikation des zu implementierenden Moduls vollständig abbilden. Besonders rigoros wird dieses Vorgehen im *Test-driven Development* (TDD) verfolgt.

40 „*Scrum* […] ist ein agiles Managementframework zur Entwicklung von Software, das aus wenigen klaren Regeln besteht“ ([PR08], S. 1). Es verkörpert die Werte des agilen Manifests, das Menschen in den Mittelpunkt der Softwareentwicklung stellt. (vgl. [PR08], S. 1). In den kurzen Arbeitszyklen, Sprints genannt, werden Anforderungen aus dem Product Backlog in ein auslieferbares Produktinkrement umgewandelt (vgl. [PR08], S. 7).

41 Eine *User-Story* ist eine Anforderung oder eine klar abgrenzbare Menge von Anforderungen aus dem Product Backlog innerhalb des Scrum-Prozesses.

42 *User Acceptance Criteria* sind die Akzeptanzkriterien für eine User-Story, die für dessen Abnahme mindestens erfüllt sein müssen.

Für die erste Aufgabe sind überdies die Anzahl, Namen und angegebenen Ursachen der fehlgeschlagenen Tests (nach dem ersten Testlauf, ohne Refactoring) sowie die Anzahl der Testzyklen, bis alle Tests grün sind, zu protokollieren. Ferner ist für beide Aufgaben festzuhalten, welche Tests nicht (mehr) ausgeführt werden sollen, mit Begründungen.

Das Ziel ist, herauszufinden, welche Tests den größten Nutzen haben, und welche Probleme am häufigsten durch die Tests beanstandet werden. Dabei ist insbesondere auf die Rückmeldungen der an der Fallstudie beteiligten Entwickler einzugehen.

5.3.2 Fallstudie – Ergebnisse

Ein Teil der Ergebnisse der Fallstudie wurde in Abschnitt 4.2.2 mit der Einführung der Tests in die mWS-Sportsbets-Engine bereits vorweggenommen und in Zahlen gefasst. Dies betrifft die erste Aufgabe der Fallstudie. Das durchgeführte Refactoring der Klassen beschränkte sich dabei auf richtige Schreibweisen und Anordnung von Methoden und Feldern in den annotierten Klassen. Weitergehende Umbauten wurden nicht vorgenommen, da hierfür fachliches Verständnis des Codes erforderlich wäre. So wurden die Tests entlang des Codes so aktiv bzw. inaktiv geschaltet, dass keiner der laufenden Tests rot wurde – die Tests dienen hier also nur dazu, die Qualität dieses *Legacy-Code* zu fixieren und weiteren „Verfall“ zu verhindern.

Da der Einsatz der Tests im Rahmen von neu zu implementierenden Komponenten aus organisatorischen Gründen bei mybet zur Zeit nicht möglich ist, habe ich mich für den zweiten Teil der Fallstudie dazu entschieden, die Neuentwicklung anhand der Architekturtests, also wie in Aufgabe 2 beschrieben, in einem bereits realisierten Softwareprojekt, das ich im Rahmen des *Softwareprojekts 'Anwendungen effizienter Algorithmen'* bei Prof. Dr. Rote realisiert habe, vorzunehmen. Da die Fachlogik des Projekts bekannt ist und die ursprüngliche Implementierung vorliegt, kann ich mich bei der (Neu-)Implementierung voll und ganz auf den Entwicklungsprozess selbst konzentrieren. Im Anhang befindet sich daher statt der geplanten User-Story ein Auszug des Abschlussberichts des Projekts, der die Anforderungen enthält. Der Nachteil dieses Schrittes ist, dass ich selbst diese Neuimplementierung vornehmen musste. So fällt an dieser Stelle das geplante Feedback anderer Entwickler weg, das möglicherweise wertvolle Hinweise zur Verbesserung des Frameworks enthalten hätte. Insgesamt habe ich nicht alles komplett neu geschrieben, sondern insbesondere die Berechnungslogik der Kreisbögen schrittweise aus dem Altprojekt übernommen. Nach jeder Iteration, die zu einem kompilierfähigen Codestand führte, wurden die Architekturtests ausgeführt.

Das Projekt umfasst 34 Klassen (ohne generierte Klassen, die für *JAXB*[43] erforderlich sind), deren

43 Die *Java API for XML Binding* (JAXB) ermöglicht die Abbildung eines XML-Dokuments anhand seines Schemas

Eigenschaften durch die Architekturtests im Test-First-Prinzip fixiert werden sollen. Die in der Aufgabenstellung festgelegte Annotation bewirkt, dass alle Tests, bis auf Entwurfsmuster, für jede Klasse durchgeführt werden sollen – auch die Methodenreinheit wird sichergestellt. Tests für Entwurfsmuster werden aktiviert für solche Klassen, die diese verwenden. Im Projekt wurden Unit- und Integrationstests geschrieben, die unverändert erhalten blieben. Sie dienen dazu, sicherzustellen, dass das Neuschreiben der Applikation entlang der Architekturtests die durch die funktionalen Tests fixierte Spezifikation nicht bricht. Außer Acht gelassen wurde hierbei die Testabdeckung dieser Tests, da es hier nur darum geht, zu untersuchen, was beim Neuschreiben mit den Architekturtests am Code-Design anders wird.

Bei der Integration des Frameworks in dieses aus mybet-Sicht externe Projekt traten zwei Probleme auf. Zum Einen waren die Testklassen nicht im eingebundenen Artefakt enthalten, sodass diese per Hand in das Projekt übernommen werden mussten. Dies birgt das Risiko, dass die Tests nicht mehr zum Artefakt, dass die Hilfsklassen enthält, passen bei Weiterentwicklung. Zum Anderen mussten die Abhängigkeiten, die in den Testklassen direkt benötigt werden, also `junit-dataprovider` und `javaparser`, in die *pom.xml* des Projekts übernommen werden. Dies ist aber ein Folgeproblem aus dem zuerst Genannten. Für eine spätere Bereitstellung als Open-Source-Framework muss also darauf geachtet werden, dass auch die Testklassen im Artefakt enthalten sind.

Die vier häufigsten Fehlerursachen beim Implementieren mittels *Architecture-Test-First*-Ansatz waren zu lange Methoden, innere Klassen, Instabilität und zyklische Abhängigkeiten. Zu lange Methoden traten vor allem beim Neuschreiben der komplexen Berechnungen auf, innere Klassen werden häufig in Swing-GUI-Klassen verwendet, insbesondere für die Implementierung von Subkomponenten der Benutzeroberfläche. Instabilität und zyklische Abhängigkeiten traten hingegen an mehreren, unterschiedlichen Stellen auf.

5.3.2.1 Methodenlängen

Drei Methoden innerhalb der Pakete `de.fuberlin.inf.swp.geom` und `de.fuberlin.inf.swp.core` wurden ursprünglich zu lang geschrieben. Diese wurden in weitere Methoden und Methodenobjekte untergliedert, sodass sich die Lesbarkeit des Codes klar erhöht hat. Jedes Methodenobjekt wird ebenfalls allen Architekturtests unterzogen. Insgesamt fünf solcher Methodenobjekte sind hinzugekommen. Voraussetzung ist natürlich, dass sinnvolle, funktional zusammengehörige Teile in eigene Methoden extrahiert und diese präzise benannt wurden.

auf eine Java-Objekthierarchie, die in beide Richtungen verwendet werden kann, sowohl lesend als auch schreibend.

5.3.2.2 Innere Klassen

Im Vergleich zum Originalprojekt sind sieben Klassen hinzu gekommen, die vorher innere Klassen waren, vier GUI-Komponenten, zwei Listener und ein Comparator. Jede dieser Klassen wird ebenfalls allen Architekturtests unterzogen.

5.3.2.3 Instabilität

Die Instabilität für sich genommen lässt sich innerhalb dieses Projekts nicht endgültig lösen. Das liegt daran, dass die implementierten Komponenten von außen keine Referenzierungen haben. Hier ist `InstabilityTest` noch nicht sinnvoll einsetzbar. Eine Erweiterung für diesen Test, die ihn auch hier nutzbar macht, wird in Abschnitt 5.5.10 erörtert.

5.3.2.4 Zyklenfreiheit

Während der Neuimplementierung des Projekts fielen an AcyclicDependenciesPrincipleTest zwei Fehler auf, die behoben wurden. Der Test baute im Falle von Selbst-Referenzierungen, wie es z. B. Beim Singletonmuster erforderlich ist, einen Zyklus im Graphen auf. Außerdem wurde das Erzeugen von Knoten anhand von Klassen so eingeschränkt, dass nur Klassen, die auch zum Projekt gehören überhaupt als Knoten dem Abhängigkeitsgraph hinzugefügt wurden. So wurde verhindert, dass Klassen außerhalb des Projekts zyklische Abhängigkeiten hervorrufen, die außerhalb meines Einflusses liegen.

5.3.2.5 Zusammenfassung der Ergebnisse

Insgesamt hatten die Architekturtests beim Neuschreiben des Projekts recht wenig zu monieren. Rund 1200 Einzeltestausführungen auf 46 Klassen hatten in maximal 50 Einzeltestausführungen, also 4% der Tests, Beanstandungen gemeldet, die während des Schreibens direkt behoben werden konnten. Es fiel auf, dass die Tests bei einer Neuimplementierung sehr viel weniger Codestellen monierten als bei der Einführung der Tests in bestehenden Code. Hinzu kommt, dass eine deutlich höhere Testabdeckung besteht, da alle verfügbaren Architekturtests auf allen Klassen (mit Ausnahme von Entwurfsmustertests, die selektiv hinzugenommen wurden, wo zutreffend) ausgeführt werden. Erstaunt hat mich, dass mir die Neuimplementierung des Projekts ohne ein einziges Monieren von `MethodPurityTest` gelang.

5.4 Einschränkungen

Hinsichtlich der Verwendung von Regeln ist zu beachten, dass die in der Regel erzeugten Annotationen eventuell vorhandene Annotationen an Klassen, für welche die Regel zutrifft, zur Laufzeit vollständig überschreiben. So ist ein Attribut `omitConventions = true` in der

Annotation wirkungslos, wenn die Regel dieses mit `omitConventions = false` überschreibt, auch wenn durch die Regel z. B. nur ein bestimmter Abhängigkeitstest getrigggert werden soll. Dies ist implementierungsbedingt, da `@ArchitectureTest` für `omitConventions` und `omitMetrics` nur jeweils einen `boolean`-Wert annimmt.

Hinsichtlich der Verlässlichkeit der Pattern-Tests wurde bereits in Kapitel 4 angemerkt, dass die Zuverlässigkeit der Erkennung der Entwurfsmuster stark eingeschränkt ist, da lediglich auf bestimmte Merkmale im Code geachtet wird, wie z. B. die exakte Benennung der `getInstance()`-Methode im `SingletonTest`.

5.5 Ansätze zur Weiterentwicklung der Architekturtests

Dieser Abschnitt zeigt, welche Erweiterungen sinnvoll sind, um das Framework, das im Rahmen dieser Arbeit recht spezifisch auf die bei mybet im Einsatz befindlichen Systeme zugeschnitten ist, auch in anderen Umgebungen einzusetzen.

5.5.1 Andere Sprachen

In der vorgelegten prototypischen Implementierung wurde nur die Untersuchung von Quellcode betrachtet, der in Java geschrieben wurde. Aufgrund der Testmechanismen in denjenigen Tests, die nicht auf Java Reflections beruhen, ist jedoch auch die Untersuchung von Quellcode in anderen Sprachen möglich. Bei mybet kommen neben Java und C# (für den BetShop, der selbst schon Architekturtests hat, wie in Kapitel 2 beschrieben) auch JavaScript (für Frontend-Aufgaben, hauptsächlich mFES und mWS) sowie SQL mit PL/SQL (für die relationale Datenbank Oracle) zum Einsatz.

Dabei muss jedoch beachtet werden, dass die mit dem vorliegenden Framework eingeführten Annotationen, die nur für Java-Klassen vorgesehen sind, nicht einsetzbar sind. Hierfür muss auf die Rule-API zurückgegriffen werden, um Regeln aufzustellen, was. z. B. mit bestimmten .js- und .sql-Dateien in den Tests geschehen soll. Auch muss der Quellcodefilter, der bisher nur .java-Dateien einliest, für solche Tests entsprechend erweitert werden.

5.5.1.1 JavaScript

Für JavaScript-Quellcode sollten dieselben Qualitätskriterien hinsichtlich Codemetriken und Konventionen gelten, wie für anderen Code. Im Framework verlassen sich die Tests für die Konventionen jedoch auf die Java Reflections API, sodass diese Tests für JavaScript neu geschrieben werden müssen. Codemetriken lassen sich auch in JavaScript auf Quellcodeebene ermitteln. Hierfür muss ein geeigneter JavaScript-Parser analog zum verwendeten Java-Parser (siehe Abschnitt 4.)

herangezogen werden. Es bietet sich an, das Framework generell vom verwendeten Parser zu entkoppeln, um Sprachneutralität herzustellen, die bislang noch nicht gegeben ist.

Weiterhin werden auch in JavaScript Standardbibliotheken wie jQuery[44] und spezielle, eigene Bibliotheken für die Berechnungen auf dem Wettschein im mFES verwendet, für die Abhängigkeitstests zu deren korrekter Verwendung geschrieben werden können.

5.5.1.2 SQL und PL/SQL

Das Selektieren (mittels `SELECT`) von Datensätzen aus Tabellen mittels Spalten, die nicht indiziert sind, kann zu erheblichen Leistungseinbußen auf der Datenbank führen. Um dies frühzeitig zu erkennen, können Queries mittels eines Tests dahingehend untersucht werden, ob die in der `WHERE`-Klausel angegebenen Spaltennamen auf den Tabellen, die in der `FROM`-Klausel angegeben sind, indiziert sind. Ist dies nicht der Fall, schlägt der Test fehl. Dasselbe gilt für `DELETE`- und `UPDATE`-Statements auf der Datenbank, also alle Datenmodifizierer, die eine `WHERE`-Klausel enthalten können.

Sog. *Stored-Procedures* sind für Oracle in PL/SQL geschrieben. Diese Sprache weist Konstrukte auf, die denen von Java oder auch JavaScript ähneln. So lassen sich bspw. Codemetriken, wie die Länge von Prozeduren oder auch die zyklomatische Komplexität darin messen. Schließlich gehört der Code in den Stored-Procedures zur Codebasis, die möglichst denselben Kriterien hinsichtlich ihrer Wartbarkeit genügen soll.

5.5.2 Framework-spezifische Konventionen

Viele Frameworks verlangen vom Entwickler eine gewisse Disziplin, was die Einhaltung bestimmter Vorgaben betrifft, beispielsweise wie Dateien benannt und wo sie gespeichert werden sollen, damit die Bibliotheken des eingesetzten Frameworks sie finden. Ebenso erfordern viele Frameworks, dass man im Code, der das Framework verwendet, bestimmte Annotationen setzt. Beides lässt sich durch geeignete Architekturtests fixieren, wie am Beispiel des Tapestry-Frameworks verdeutlicht wird.

5.5.2.1 Dateikonventionen

Tapestry erfordert für Komponenten, die `BaseRenderable` beerben, dass sie eine gleich benannte Vorlage im selben Verzeichnis haben, lediglich mit anderer Dateiendung. So gibt es zur in Abschnitt 2.3.2 erwähnten `TopNavigation` (TopNavigation.java) auch eine TopNavigation.tml-

44 jQuery ist eine JavaScript-Bibliothek, die unter Anderem das Traversieren und Manipulieren von HTML-Dokumenten, Ereignisbehandlung, Animationen sowie AJAX wesentlich vereinfacht (vgl. http://jquery.com/, Zugriff am 13.02.2015).

Vorlagendatei. `TopNavigation` hat hier die Eigenschaft des *Controllers* (siehe MVC[45]-Muster), der sich seine *View* (aus TopNavigation.tml) rendert und ausliefert. Ein hierfür geeigneter Test soll also prüfen, ob zu jeder Klasse, die `BaseRenderable` beerbt und keine weiteren Erben hat, auch eine gleichbenannte .tml-Datei im selben Verzeichnis existiert. Ist dies nicht der Fall, schlägt der Test fehl. Für die Implementierung ist innerhalb des vorgelegten Frameworks nur der Teil notwendig, der die Existenz der .tml-Datei für eine gegebene Klasse prüft. Die genannten Vorbedingungen können durch den Einsatz der Rule-API geprüft werden.

5.5.2.2 Annotationen

Tapestry verlässt sich darauf, dass Klassen, die Controller für Komponenten sind, die auf eine Seite gerendert werden sollen, geeignet annotiert sind. Beispielsweise ist die `@SetupRender`-Annotation erforderlich, was mit einem Test `AnnotationTest`, der die Klasse und die auf ihre Existenz zu prüfende Annotation enthält, prüfbar ist. Ist die Annotation nicht vorhanden, schlägt der Test fehl.

Da dieser Test für alle Kontexte, die bestimmte Annotationen erfordern, einsetzbar ist und nicht nur für Tapestry, heißt dieser Test auch nur `AnnotationTest` und nicht etwa `TapestryAnnotationTest`.

5.5.3 Generizität und Kapselung

Viele Komponenten des (prototypischen) Frameworks sind noch speziell auf die Umgebung bei mybet angepasst und das Framework selbst ist offen integriert, nicht gekapselt, wie z. B. JUnit gekapselt und in seiner Codebasis unabhängig von der mybet-Plattform ist. Dies trifft auf das hier entwickelte Framework noch nicht vollumfänglich zu. Dies wäre die vorrangige Aufgabe für eine Erweiterung über diese Arbeit hinaus.

Der betrachtete Scope für Pakete sollten konfigurierbar gemacht werden. Die konkreten Regelgenerierer sind bereits in ein eigenes Modul „*Share – Architecture Test Rules*“ ausgelagert. Dies führt dazu, dass das Modul „*Test – Architecture*“ bereits jetzt auf keine anderen Module der mybet-Plattform angewiesen ist. Die Voraussetzung für das Finden der speziellen Regelgenerierer ist durch das Interface `RuleGenerator` und seine abstrakte Klasse `ArchitectureTestRuleGenerator` als Bestandteil des Frameworks ja bereits gegeben, indem mittels Reflections nach Subtypen von `ArchitectureTestRuleGenerator` gesucht wird.

45 Das *Model-View-Controller*-Entwurfsmuster (MVC) und seine Ableger MVVM und PM, die ähnlich aufgebaut sind, werden zur Konstruktion von Benutzerschnittstellen verwendet mit dem Ziel, die Steuerung („Controller“), von der Darstellung („View“) und von den verwendeten Daten („Model“) zu entkoppeln, um diese wiederverwendbar und austauschbar zu machen (vgl. [GH96], S. 6ff)

Die Konfigurierbarkeit erfordert so ihren letzten „Feinschliff“ durch eine geeignete Konfigurationsschnittstelle, auch für die Integration in verschiedene Frameworks, wie Spring, die mittels *Dependency Injection*[46] arbeiten. Diese Themen sind aber nicht Bestandteil dieser Arbeit und werden zukünftig evaluiert und ggf. realisiert.

5.5.4 Open Source-Projekt

Die in Abschnitt 5.5.3 angesprochene Generizität ist Voraussetzung dafür, das Framework als Open Source-Projekt im WWW unter einer der gängigen Lizenzen, wie z. B. der Apache License oder der *GNU (Lesser) General Public License*[47] der interessierten Öffentlichkeit zur Verfügung zu stellen. Hierzu sind noch rechtliche Rahmenbedingungen zu klären, inwieweit Code, der bei der ANYBET GmbH entwickelt wurde, als Open Source freigegeben sein darf. Auch ist eine Paketbenennung, die nicht mit `com.mybet` beginnt, zu wählen, wenn das Unternehmen auf die Verwendung des Namens *mybet* in einem Open Source-Projekt verzichten möchte. Das entstandene Artefakt soll zunächst in ein externes, von der mybet-Plattform unabhängiges Projekt integriert werden, um dort noch eventuell auftretende Probleme zu entdecken und zu lösen. Ein gutes Beispiel für ein solches externes Open Source-Projekt, in das die Architekturtests eingeführt werden können, ist *JFreeChart*[48], mit dem ich bereits im Rahmen einiger Bugfixes in der Veranstaltung *Softwareprozesse* bei Prof. Dr. Prechelt zu tun hatte. Es hat mehrere Schichten, ist modular aufgebaut und hat einige für die Tests sehr interessante Codeteile. Insbesondere die Abhängigkeiten innerhalb des Projekts sowie Codemetriken bergen Verbesserungspotenzial. Dieses kann auch in den Tests selbst noch gefunden werden, da bisher nur innerhalb der Codebasis der mybet-Plattform getestet wurde.

5.5.5 Anbindung an eine Graphdatenbank

Am Ende des zweiten Kapitels wurde der Architekturtestbegriff dahingehend präzisiert, dass Architekturtests Schwächen im Code-Design finden. Im Zuge dessen wurde auf einen Videovortrag

46 Siehe http://martinfowler.com/articles/injection.html, Zugriff am 15.02.2015. Als *Dependency Injection*, kurz *DI*, bezeichnet man ein Entwurfsmuster, das erst zur Laufzeit die Abhängigkeiten von Komponenten untereinander regelt. Der ältere Begriff, *Inversion of Control*, kurz *IoC*, war Martin Fowler nicht präzise genug.

47 Siehe http://www.gnu.org/licenses/gpl-3.0.html, Zugriff am 01.03.2015. Die GNU General Public License Version 3 (GNU GPLv3) ist ebenfalls eine Lizenz für freie Software. Sie ist die am weitesten verbreitete Lizenz für freie Software; die Apache License 2.0 ist zu ihr kompatibel (nicht jedoch zu den Versionen 1 und 2). Beide Lizenzen sind sogenannte *virale* Lizenzen, d. h. der Urheber der Software (vor der Modifikation) muss weiterhin genannt werden, die Lizenz selbst muss dem Paket mit der Modifikation beiliegen und alle Modifikationen an der Software müssen deutlich sichtbar sein, z. B. durch geeignete Kommentarstellen im Quellcode.

48 JFreeChart ist eine in Java geschriebene Open Source-Bibliothek zur Erzeugung von Diagrammen jeglicher Art und ihrer Darstellung innerhalb der eigenen Java-Applikation. Es wird von einer aktiven Entwicklergemeinde mit drei *Core*-Entwicklern, einem Projektleiter und etwa 100 aktiven Beitragenden („*Contributors*“) weiter entwickelt, mit derzeit rund 300 ungelösten und seit Version 1.0 etwa 850 geschlossenen Bugs. Siehe http://www.jfree.org/jfreechart/ (Zugriff am 19.03.2015).

verwiesen, der die Verwendung einer Graphdatenbank zum *Data-Mining* von Code vorstellt. Diese Idee wird hier nun näher verfolgt.

Die in dieser Arbeit implementierten Architekturtests verlassen sich auf einen AST-basierten Parser, so dass sie auch nur die Strukturinformationen von Code analysieren können, die ein AST hergibt. In einem AST befinden sich alle Sprachelemente, die im zu untersuchenden Quellcode verwendet werden, hierarchisch strukturiert. Was darin nicht erkennbar ist, sind *Datenflüsse*. Diese mussten, wo sie für die Tests relevant sind, selbst modelliert und implementiert werden in `AcyclicDependencyPrincipleTest`, `InstabilityTest` und `MethodPurityTest`. Alle drei Tests sind *Abhängigkeitstests*, so dass deutlich wird, dass es um *Beziehungen* zwischen Codeteilen geht. Diese Beziehungen wurden überwiegend als Graphen oder zumindest graphenähnliche Datenstrukturen modelliert. Es wurde sogar eine Implementierung einer Graphdatenstruktur in die Hilfsklassen des Frameworks integriert. Die Idee ist, den zu untersuchenden Code in einer Graphdatenbank abzubilden, um die Beziehungen zu untersuchen.

Eine Graphdatenbank wie *Neo4j*[49] speichert Informationen als *Eigenschaftsgraphen*. Es existieren Abhängigkeitstests mit Namen `UsesTest`, `ExtendsTest`, `OverridesTest` oder `ImplementsTest`. Die Kanten in einem Eigenschaftsgraph, der für das hier angedachte Anwendungsszenario *Codeeigenschaftsgraph* genannt wird, sind beschriftet. In der Graphentheorie nennt man dies einen *kantengefärbten* Graph, wobei die möglichen Farben der Kanten <<*uses*>>, <<*extends*>>, <<*overrides*>> oder <<*implements*>> lauten können. Dies ist eine mögliche Ebene des Codeeigenschaftsgraphen. Die zweite Ebene ist die der Beziehungen innerhalb einzelner Klassen, innerhalb von Methoden, zwischen einzelnen Statements. Der AST-basierte Parser wird überwiegend durch spezifische `Visitor`-Klassen verwendet. Es gibt im Framework bspw. `MethodCallVisitor` und `DeclarationVisitor`, sodass als weitere Kantenfarben <<*calls*>>, <<*returns*>> und <<*declares*>> hinzukommen.

Wie könnte nun der Codeeigenschaftsgraph aussehen? Angenommen, jede Klasse ist zunächst einmal ein Knoten, der gleichzeitig Wurzel für den AST der Klasse ist. Die Klassen untereinander haben die vorgenannten Beziehungen wie <<*extends*>> oder <<*uses*>>, sind also untereinander mit entsprechend gefärbten Kanten verbunden. Dann gibt es in den ASTs Methodenaufrufe, die innerhalb desselben ASTs oder auch zu anderen ASTs im Codeeigenschaftsgraph auf Methodendeklarationen zeigen (mit <<*calls*>> eingefärbt). Methoden mit Rückgabewerten zeigen mittels der Farbe <<*returns*>> auf die Klassen im Codeeigenschaftsgraph, die von ihnen zurückgeliefert werden. Schnell lässt sich so vorstellen, dass, bei einer Codebasis von rund 6.600 Klassen ein äußerst

49 Siehe http://neo4j.com/developer/graph-database/, Zugriff am 12.03.2015.

komplexer Codeeigenschaftsgraph aufgebaut würde mit 6.600 ASTs und mehreren hunderttausend Kanten untereinander. Es entsteht also ein sehr komplexes Beziehungsgeflecht der gesamten Codebasis, die sich mithilfe leistungsfähiger Graphenalgorithmen untersuchen lässt.

Auch für die anderen Tests kann der Codeeigenschaftsgraph herangezogen werden. Die Tests für Codemetriken zählen allesamt Knoten innerhalb des AST der Klasse, die sie untersuchen. `NumberOfMethodsPerClassTest` zählt die Anzahl der Knoten, die Methodendeklarationen sind; `MethodLengthTest` zählt die Anzahl der direkt verbundenen Knoten innerhalb jeder Methodendeklaration, die nicht-verzeigende Statements sind und im Falle von verzweigenden Statements rekursiv die Anzahl der direkt darin verbundenen Knoten. `CyclomaticComplexityTest` zählt die Knoten, die verzweigende Statements sind, innerhalb jeder Methodendeklaration. `HalsteadComplexityTest` zählt sämtliche Knoten im AST, zum Einen absolut (Halstead-Länge), zum Anderen ohne Doppelungen (Halstead-Vokabular) und berechnet dann die erwartete Fehlerquote, wie in Abschnitt 4.1.5.2 beschrieben.

Die Tests für Konventionen untersuchen Knoten hinsichtlich ihrer Eigenschaften. `CamelCaseTest` überprüft für alle Knoten, die AST-Wurzel, Methoden-, Feld- oder Konstantendeklaration sind, auf die Eigenschaft Name, ob sie den jeweiligen regulären Ausdrücken entspricht. `OrderingTest` untersucht ebenfalls die genannten Knotentypen auf ihre jeweiligen Modifikatoren und auf die Reihenfolge im Code anhand der Zeilennummer, die dafür als Eigenschaft mit zu den Knoten gehören muss. `NoInnerClassesTest` schließlich prüft im AST, ob es Klassendeklarationen darin gibt. `OverlyChainedMethodCallsTest` ist schwierig in der Realisierung auf dem Codeeigenschaftsgraph. Er muss untersuchen, ob die Knoten, die Methodenaufrufe sind, Rückgabewerte haben und ob auf diesen direkt weitere Methodenaufrufe stattfinden. Dies ist nur insoweit umsetzbar, so lange es Methodenaufrufe sind, die zum Codeeigenschaftsgraph gehören. Anhand der Perspektive aus Sicht des Codeeigenschaftsgraphen ergibt sich die Überlegung, `OverlyChainedMethodCallsTest` zu den Abhängigkeitstests zu verschieben.

Für die Entwurfsmustertests wird der in Abschnitt 4.1.6 skizzierte Ansatz interessant, der mittels Mustererkennung und maschinellem Lernen, hier auf einem Graphen, Entwurfsmuster zunächst erlernt und dann erkennt. Das Konzept hierfür kann der Kern einer weiteren Arbeit ähnlichen Umfangs wie dieser hier sein und würde eine der weiterführenden Erweiterungen des Architekturtest-Frameworks darstellen. Hierfür müssen zunächst die Entwurfs*muster* jeweils in einen *Satz* von Knoteneigenschaften, Knotenanzahlen und Kantenfärbungen übersetzt werden. Diese Sätze müssen dann effizient, d. h. in *Polynomialzeit*, im Codeeigenschaftsgraph gefunden werden können.

Algorithmisch wird noch zu untersuchen sein, welche Kosten die stetige Aktualisierung des

Codeeigenschaftsgraphen hat, um diesen konsistent mit dem Code zu halten. Die vier Operationen hierfür sind Löschen und Einfügen von Knoten und Kanten. Es ergeben sich für die Weiterentwicklung mit einer Graphdatenbank als persistente Schicht drei Hauptherausforderungen:

1. Den Codeeigenschaftsgraph mit der Codebasis fortlaufend und effizient konsistent zu halten,
2. die Entwurfsmuster zu 'übersetzen' und
3. die übersetzten Entwurfsmuster effizient im Codeeigenschaftsgraph zu finden.

Zusammenfassend lässt sich sagen, dass die Graph-Perspektive über die Kategorisierung der Tests erneut nachdenken ließ und im Falle des `OverlyChainedMethodCallsTest` eine Neueinordnung sinnvoll macht. Alle anderen Tests lassen sich klar einordnen durch das, was sie im Codeeigenschaftsgraph untersuchen.

5.5.6 Architekturtests als Plugin für Entwicklungsumgebungen

Die hohe Ausführungsgeschwindigkeit der Architekturtests bietet an, gerade geschriebenen Code, sobald dieser richtig übersetzt wurde, den Architekturtests zuzuführen. Hier „wildern“ die Tests zwar in den Gefilden von *CheckStyle* und weiteren Tools, allerdings würden dieselben Regeln angesetzt werden, die eh schon für den bestehenden Code in den Regelgenerierern fixiert wurden. Die jeweiligen Plugin-APIs *für* bspw. IntelliJ IDEA oder auch Eclipse sind ausgereift und wohl dokumentiert, so dass sich ein Test-Runner aus JUnit mit der `ArchitectureTestSuite` immer dann ansteuern lässt, wenn ein „On-the-fly“-Kompiliervorgang erfolgreich war.

Auf diese Weise würden die Architekturtests, deutlich näher an der eigentlichen Entwicklung, bereits zur **Kompilierzeit** (*Compile Time*) ausgeführt.

5.5.7 Architekturtests als Kriterium für Gated Commits

Als *Gated Commit* wird eine Prüfung bezeichnet, die gerade eingecheckten Code erst dann der Codebasis in der Versionsverwaltung zuführt, wenn dieser bestimmten Anforderungen genügt. So werden oft *-`import`-Statements mittels Gated Commits mit dem Hinweis zurückgewiesen, diese durch die jeweiligen, tatsächlich benötigten Klassen zu ersetzen.

Vergleichbar mit Abschnitt 5.5.6 können nun die einzucheckenden Klassen einem Test-Runner für JUnit in der Continuous-Integration-Umgebung zugeführt werden. Als Bedingung für den Gated Commit gelte dann, dass dieser den Code erst dann durch lässt, wenn alle Tests grün sind.

Auf diese Weise würden die Architekturtests zwar nicht ganz so nahe an der Entwicklung sein wie mit einem Plugin für Entwicklungsumgebungen, allerdings bereits zur **Eincheckzeit** (*Check-in Time*)

ausgeführt.

5.5.8 Saubere Ressourcenbehandlung per Test fixieren

Sämtliche Objekte von Klassen, die `Closeable` implementieren, sind potenzielle Verursacher von *Memory Leaks*[50] und / oder zu hohem *GC-Count*[51]. Aus diesem Grund ist es sinnvoll, zu prüfen, ob `close()` in mindestens einem `finally`-Block aufgerufen wird für alle Verwendungen von Klassen, die `Closeable` implementieren. Dies erfordert einen neuen Test `ResourceHandlingTest` im Paket `conventions`, eine Regel `MustCallCloseInFinallyIfImplementsCloseableRule` sowie einen Regelgenerierer `MustCallCloseInFinallyIfImplementsCloseableRuleGenerator`.

5.5.9 Sichere Programmierrichtlinien per Test fixieren

Am *CERT Software Engineering Institute* der *Carnegie Mellon University* sind Regeln und Empfehlungen für einen sicheren Programmierstil in Java (siehe [MJ15]), wie auch für weitere Sprachen, entwickelt worden. Diese stellen derzeit den Stand dar, der aktuell erarbeitet wird. Mit sicherem Programmierstil ist gemeint, Code so zu schreiben, dass er möglichst wenige Angriffsvektoren zur Laufzeit bietet. Hier wird exemplarisch auf zwei der über 100 Regeln eigegangen, die in [MJ15] beschrieben sind, die sich leicht in das Framework, und ggf. auch geeignete Regelgenerierer, integrieren lassen.

Ein Beispiel für eine Regel ist „*IDS00-J*“, *SQL Injections*[52] zu vermeiden, in dem `java.sql.PreparedStatement` so verwendet wird, dass Benutzername und Passwort nicht im Klartext im SQL-Statement vorkommen:

```
String sqlString =
   "select * from db_user where username=? and password=?";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, username);
```

50 Ein *Speicherleck* ist ein Fehler in einer Software, der auftritt, wenn Arbeitsspeicher während der Ausführung zwar belegt, aber bei Nichtverwendung nicht mehr freigegeben wird. Das kann dazu führen, dass Teile des Speichers ausgelagert werden müssen, was die Ausführung erheblich verlangsamen kann oder dass Speicherbereiche, die vom Betriebssystem benötigt werden, überschrieben werden. Speicherlecks stellen auch, ob gewollt provoziert oder unabsichtlich, ein Sicherheitsrisiko dar.

51 *Garbage Collection* (GC, dt. sinngemäß „*Müllsammlung*“, tatsächlich *automatische Speicherbereinigung*) ist ein Mechanismus der JVM, nicht mehr genutzte Objekte im Hintergrund automatisch abzuräumen und den von ihnen belegten Speicher wieder frei zu geben. Damit Entwickler sich um das Speichermanagement nicht mehr kümmern müssen, wird dies in Java durch die JVM übernommen. Ein hoher GC-Count bedeutet, dass der *Garbage Collector* übermäßig häufig aufgerufen wird, um Speicher, der bspw. durch nicht geschlossene Ressourcen belegt wird, freizugeben.

52 Als *SQL Injection* wird ein Angriff bezeichnet, der mittels Eingabe spezieller SQL-Fragmente auf bspw. einer Login-Seite einer datengetriebenen Web-Applikation dazu führen kann, dass ein Angreifer Zugriffsrechte auf die Datenbank hinter der Web-Applikation erhält, mit denen unerwünschte Aktionen ausgeführt werden können (vgl. https://www.owasp.org/images/5/57/OWASP-AppSecEU08-Janot.pdf, Zugriff am 23.03.2015).

```
stmt.setString(2, pwd);
ResultSet rs = stmt.executeQuery();
```

(vgl. [MJ15], 'IDS00-J. Prevent SQL Injection')

Eine Regel, die dies überprüft, muss sicherstellen, dass für betroffene Klassen `UsesTest` prüft, ob `java.sql.PreparedStatement` verwendet wird und ob die Zeichenkette, die der `prepareStatement`-Methode aus `java.sql.SQLConnection` übergeben wird, „`username=?`“ und „`password=?`“ enthält. Danach muss die `setString`-Methode aus `PreparedStatement` zweimal aufgerufen werden, um das erste '?' durch den Benutzernamen und das zweite '?' durch das Passwort zu ersetzen. Für diesen Anwendungsfall muss das Framework jedoch noch dahingehend erweitert werden, dass es einen `StatementTest` erhält, der prüft, ob ein bestimmtes Statement, ggf. auch durch einen regulären Ausdruck definiert, vorkommt. Die Rule-API erlaubt bereits das Zuordnen von Klassen zu mehr als einem Test, so dass hier `UsesTest` und (der noch nicht vorhandene) `StatementTest` ausgeführt würden.

Ein weiteres Beispiel ist „MET01-J“. Diese Regel verbietet die Benutzung von *Assertions*, um Methodenargumente zu validieren. Der Hintergrund ist, dass Methodenargumente üblicherweise entsprechend dem Kontrakt der Methode überprüft werden sollen. Sind Assertions deaktiviert, kann der Kontrakt nicht länger überprüft werden, obwohl er weiterhin gilt. Mittels `StatementTest` lässt sich sicherstellen, dass in Methoden *keine* `assert`-Statements vorkommen, die auf den Methodenargumenten operieren:

```
public static int getAbsAdd(int x, int y) {
  assert x != Integer.MIN_VALUE;
  assert y != Integer.MIN_VALUE;
  int absX = Math.abs(x);
  int absY = Math.abs(y);
  assert (absX <= Integer.MAX_VALUE - absY);
  return absX + absY;
}
```

(vgl. [MJ15], 'MET01-J. Never use assertions to validate method arguments')

Für `StatementTest` soll es dann noch zwei Annotationen `@MustContain(statements = {`*`regex...`*`})` und `@MustNotContain(statements = {`*`regex...`*`})` geben. Es erscheint sinnvoll, *Security*-Tests als weitere, fünfte *Kategorie* von Tests einzuführen, in der spezifische Tests nach sicherheitsrelevanten Merkmalen im Code suchen. Dies ist aber ein anhand der Regeln und Empfehlungen in [MJ15] zu untersuchender Aspekt. Eine Suche in Google Scholar[53] nach „*static*

53 Siehe https://scholar.google.de/, Zugriff am 23.03.2015.

code analysis“ zeigt, dass sich etwa die Hälfte der Treffer mit statischer Codeanalyse speziell für das Finden potenzieller Sicherheitslücken in Software befasst. Da die Architekturtests statische Codeanalyse betreiben, ergibt sich hier eine Erweiterung, an deren Grundlagen derzeit aktiv geforscht wird.

5.5.10 Erweiterungen bestehender Tests

Im entwickelten Framework gibt es zwei Tests, deren Nutzen sich bei der Einführung der Tests als eingeschränkt herausgestellt hat: `HalsteadComplexityTest` und `InstabilityTest`. Der Erstgenannte ermittelt die erwartete Anzahl ausgelieferter Bugs, wobei diese mit der Größe der Klasse korreliert, wie bereits in Kapitel 4 dargestellt. Daher sollte dieser Test entweder so umgeschrieben werden, dass er die Anzahl zu erwartender Bugs pro Methode ermittelt. Es gibt einen weiteren Test, der die Anzahl der Statements in einer Methode prüft, so dass die Anzahl erwarteter Bugs einen bestimmten Wert, der abhängig von der maximal zulässigen Methodenlänge zu bestimmen ist, nicht überschreiten darf.

Die Instabilität *I* ist dann ausgeglichen, wenn das Verhältnis zwischen efferenter Kopplung und der Summe aus afferenter und efferenter Kopplung eines Moduls nahe 0,5 liegt. Werte für *I*, die gegen 1 (geringe afferente Kopplung, hohe efferente Kopplung, also hohe Abhängigkeit von anderen Modulen) streben, deuten darauf hin, das betroffene Modul zu überarbeiten und diese Abhängigkeit zu reduzieren. Ein weiteres, ähnliches Maß ist die *Abstraktheit A* eines Moduls, ausgedrückt durch das Verhältnis zwischen der Anzahl aus abstrakten Klassen / Interfaces und der Summe der Anzahlen aus konkreten und abstrakten Klassen / Interfaces. Auch hier ergibt sich ein Wert zwischen 0 und 1, wobei 1 bedeutet, dass ein Modul nur aus abstrakten Klassen und / oder Interfaces besteht. Ein Wert für *A* nahe 0,5 bedeutet ausgeglichene Abstraktheit. Beide Größen lassen sich, wie in Abbildung 24 gezeigt, in einem *Instabilitäts-Abstraktheits-Diagramm* darstellen. Darin gibt es zwei Bereiche, die *Zone of Pain* („Schmerzbereich“) und die *Zone of Uselessness* („Bereich der Nutzlosigkeit“). Die „Schmerzen“ entstehen dann, wenn ein Modul sehr stabil ist und dabei eine geringe Abstraktheit aufweist, es also so schwer änderbar ist, dass es „weh tut“. Andererseits wird ein Modul, von dem wenige andere Module abhängen, das sehr abstrakt, also leicht erweiterbar ist, als nutzlos betrachtet.[54] Der bisherige `InstabilityTest` berücksichtigt die Abstraktheit nicht, so dass ein Wert von 0,4 (strenger Schwellwert) nur dann außerhalb der Zone of Pain liegt, wenn das betreffende Modul mindestens eine Abstraktheit von etwa 0,4 hat. Der gelockerte Schwellwert von 0,8 ist nur dann außerhalb der Zone of Uselessness, wenn die Abstraktheit etwa 0,6 nicht überschreitet. Daher ist es sinnvoll, `InstabilityTest` zu teilen in zwei neue Tests `ZoneOfPainTest` und

54 Vgl. http://www.hanselman.com/blog/ExitingTheZoneOfPainStaticAnalysisWithNDepend.aspx, Zugriff am 02.04.2015.

`ZoneOfUselessnessTest`, die prüfen, ob ein gegebenes Modul in der Zone of Pain bzw. in der Zone of Uselessness liegt. Ist dies der Fall, schlagen die Tests fehl.

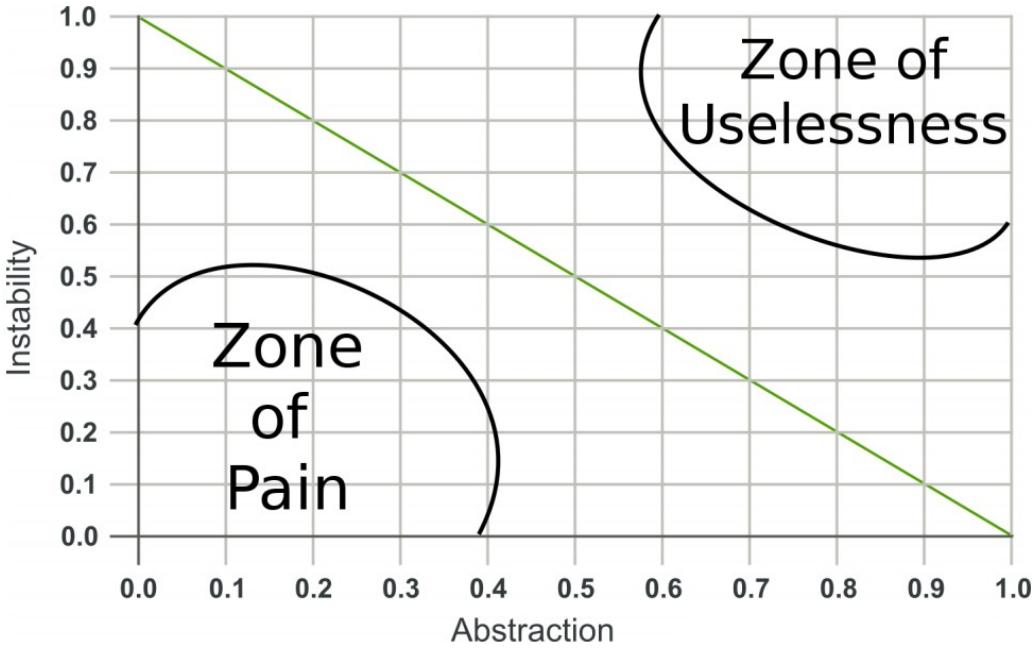


Abbildung 24: Instabilitäts-Abstraktheits-Diagramm (Quelle: http://team.fh-kl.de/uploads/media/Codequalitaet_Codemetrik.pdf, Zugriff am 02.04.2015, Folie 57)

5.6 Zusammenfassung

Die Bewertung der hier erstellten Architekturtests zeigt, dass die Einsatzmöglichkeiten klar über statische Codeanalyse hinausgehen. Sie dienen dazu, spezifische Architektur*entscheidungen* zu fixieren und dabei die Einhaltung flankierender Eigenschaften zu erzwingen. Die Rule-API erweist sich, obwohl sie ursprünglich nur als „fixe Idee“ recht spät in der Implementierungsphase Einzug fand, als mächtiges Werkzeug, um Architektur- und Designentscheidungen direkt (und ebenfalls) in Java-Code zu verfassen. Allein diese Eigenschaft macht die bisher dokumentierten Entscheidungen (sofern sie überhaupt dokumentiert wurden) für alle Entwickler greif- und überprüfbar. Weiterhin hat die Fallstudie demonstriert, wie sowohl Refactoring auf Grund der Architekturtests, als auch Neuimplementierungen mittels „Architecture-Test-First“-Prinzip die Softwarequalität positiv beeinflussen. Es wurde gezeigt, welche Erfahrungen mit dem Framework im konkreten Einsatz gemacht wurden und welche Einschränkungen der gegenwärtige Stand der Implementierung mit sich bringt. Schließlich wurden mögliche Erweiterungsszenarien entworfen, die über eine einzige Sprache, hier Java, hinaus gehen und weitere Sprachen in die Betrachtung durch die Tests rücken. Zudem wurde diskutiert, wie das Framework als Open Source-Projekt verfügbar gemacht werden kann und welche Möglichkeiten sich durch den Einsatz einer Graphdatenbank bieten.

6 Fazit

„Huch, wo ist meine Architektur hin?“

So heißt ein Vortrag von Oliver Gierke, den ich im Rahmen der *Berlin Expert Days* 2013 an der Freien Universität Berlin besucht hatte. Er handelte von der Problematik, dass Modularität zum Kernaspekt für Wartbarkeit wird, wenn Applikationen im Lauf der Zeit über eine bestimmte Größe hinaus wachsen. Auch finden sich getroffene Design- und Architekturentscheidungen kaum noch wieder.[55]

Das in dieser Arbeit entwickelte Framework wirkt dem im Zitat so plakativ und doch so treffend formulierten Effekt nachweisbar entgegen. Es resultiert aus den in den Kapiteln 1 bis 3 mit zunehmender Schärfe und Genauigkeit formulierten Ideen, die in Kapitel 4 ihre Umsetzung und in Kapitel 5 ihre Bewertung und Messung an der Wirklichkeit fanden.

In Kapitel zwei wurde eine Einordnung der Architekturtests auf die Ebene von Unit-Tests erarbeitet. Dabei ist insbesondere von Bedeutung, dass die Tests ohne Mock-Testdaten und ohne andere Komponenten, wie etwa Drittsysteme oder Datenbanken, auskommen. Der Softwareinfrastruktur- und Testkontext bei mybet wurde erläutert; das Reporting der Testergebnisse erfolgt in der bereits vorhandenen, zentralen Continuous-Integration-Umgebung.

Die Erarbeitung des Konzepts des Testframeworks sowie der Arten der zu realisierenden Tests erfolgte im dritten Kapitel. Dabei wurden Anforderungen an die zunächst drei verschiedenen Testkategorien, Codemetriken, Programmierkonventionen und Entwurfsmuster, formuliert. Daraus entstand der Entwurf, geeignete Annotationen für die Kennzeichnung der Klassen zu verwenden, die zur Laufzeit vom Testframework den jeweiligen Tests zugeführt werden sollen.

Im vierten Kapitel wurde detailliert beschrieben, wie das Framework entwickelt wurde. Erst hier stellten sich zwei wesentliche Ergänzungen zum vorhergehenden Kapitel heraus, während der Implementierungsphase: Die vierte Testkategorie 'Abhängigkeiten', welche auch die meisten Tests enthält. Architekturentscheidungen beruhen oft auf bestimmten Abhängigkeiten von Klassen, Paketen und Modulen. Die zweite Ergänzung verleiht dem Framework seine eigentliche Mächtigkeit und Robustheit gegenüber Veränderungen am zu testenden Code, die Rule-API. Mit ihr wurde es möglich, Regeln zu definieren, welche die Durchsetzung von Architekturentscheidungen im Code unabhängig vom Gebrauch der Annotationen machen.

Im Rahmen der Bewertung im fünften Kapitel wurde eine Fallstudie in zwei Teilen durchgeführt. Der erste Teil befasste sich mit dem Refactoring von bestehendem Code aufgrund der

55 vgl. http://lanyrd.com/2013/bedcon/scftbr/, Zugriff am 19.02.2015

Architekturtests, der zweite Teil bestand aus der Neuimplementierung eines Features mit den Architekturtests als „Wächter“ über die Architekturentscheidungen. Dabei stellte sich heraus, dass die nachträgliche Einführung solcher Tests in Legacy-Software auf den Erhalt der bisherigen, nicht immer hohen, Codequalität zu beschränken ist. Die Neuentwicklung hingegen wird durch den Einsatz der Tests „von Anfang an“ dahingehend unterstützt, dass entlang der in den Tests fixierten Architekturentscheidungen entwickelt und dies als Unterstützung empfunden wird.

So muss man differenzieren und sagen, dass das Framework kein „Allheilmittel“ für schleichenden Architekturverlust darstellen kann und will. Was es leistet, ist, Entwickler dabei zu unterstützen, Architektur- und Designentscheidungen einzuhalten durch die regelmäßige Ausführung der Tests, ebenso häufig wie Unit-Tests.

In meiner Berufslaufbahn bin ich seit über einem Jahrzehnt im IT-Umfeld unterwegs; mein erster Studienabschluss als Diplom-Informatiker (FH) liegt zum Zeitpunkt der Fertigstellung dieser Arbeit 13 Jahre zurück. Das Masterstudium der Informatik an der Freien Universität Berlin sollte dem in fachlicher Hinsicht sowohl mehr Breite als auch Tiefe verleihen und tat dies erfolgreich. Diese Bereicherung war es auch, die mir die klare Abgrenzung der Arbeit zuweilen schwer machte. Schließlich geht es wohl jedem, der von seinem Fachgebiet überzeugt ist, ähnlich, immer weitere Aspekte und Technologien, erforschen zu wollen, um daraus neue Erkenntnisse zu gewinnen.

Mein besonderes Interesse gilt der Testautomation einerseits und Softwarequalität andererseits. Beides zusammen brachte mich auf die Idee, Tests zu entwickeln, die Codequalität und Architekturentscheidungen so selbstverständlich und regelmäßig sicherstellen wie spezifizierte Funktionalität. Der römische Dichter Ovid (43 v. Chr. bis 17 n. Chr.) sagte schon: „Steter Tropfen höhlt den Stein.“

Abbildungsverzeichnis

Abbildung 1: Testquadranten (Quelle: [CG09])....20
Abbildung 2: Testquadranten mit Architekturtests (Quelle: [CG09] und eigene Darstellung)....21
Abbildung 3: Testpyramide (Quelle: [CG09])....22
Abbildung 4: Jenkins, Ansicht 'Testergebnisse' (Quelle: Eigene Darstellung)....27
Abbildung 5: Softwareinfrastruktur bei mybet (Quelle: Eigene Darstellung)....29
Abbildung 6: Vererbungsstruktur der Seitenelemente des mFES (Quelle: Eigene Darstellung)....30
Abbildung 7: Lage einzelner Seitenelemente im Bereich 'Mein Konto' auf mybet.com (Quelle: Eigene Darstellung)....31
Abbildung 8: Lage einzelner Seitenelemente im Bereich 'Sportwetten' auf mybet.com (Quelle: Eigene Darstellung)....31
Abbildung 9: Blockdiagramm der Module des BetShop (Quelle: Interne Dokumentation)....32
Abbildung 10: Testinfrastruktur bei mybet (Quelle: Eigene Darstellung)....34
Abbildung 11: SimpleEntityData mit einigen erbenden Klassen (Quelle: Eigene Darstellung)....36
Abbildung 12: Paketstruktur der mWS Sports Bets Engine (Quelle: Eigene Darstellung)....37
Abbildung 13: Vererbungsstruktur der Basisklassen (Quelle: Eigene Darstellung)....48
Abbildung 14: Vererbungsstruktur der Regel-Basisklassen (Quelle: Eigene Darstellung)....51
Abbildung 15: Vererbungsstruktur der Bedingungsklassen (Quelle: Eigene Darstellung)....51
Abbildung 16: Vererbungsstruktur der Regelgenerierer (Quelle: Eigene Darstellung)....53
Abbildung 17: Vererbungsstruktur der Conventions Tests (Quelle: Eigene Darstellung)....55
Abbildung 18: Vererbungsstruktur der Metrics Tests (Quelle: Eigene Darstellung)....57
Abbildung 19: Vererbungsstruktur der Pattern Tests (Quelle: Eigene Darstellung)....59
Abbildung 20: Vererbungsstruktur der Dependency Tests, Teil 1 (Quelle: Eigene Darstellung)....61
Abbildung 21: Vererbungsstruktur der Dependency Tests, Teil 2 (Quelle: Eigene Darstellung)....62
Abbildung 22: Klassendiagramm der Hilfsklassen (Quelle: Eigene Darstellung)....66
Abbildung 23: Auszug aus der pom.xml des Moduls test/architecture (Quelle: Eigene Darstellung)....68
Abbildung 24: Instabilitäts-Abstraktheits-Diagramm (Quelle: http://team.fh-kl.de/uploads/media/Codequalitaet_Codemetrik.pdf, Zugriff am 02.04.2015, Folie 57)....94

Literaturverzeichnis

[CG09] Crispin, L.; Gregory, J.: Agile Testing. A Practical Guide for Testers and Agile Teams. Addison-Wesley, 2009

[HF11] Humble, J.; Farley, D.: Continuous Delivery. Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, 2011

[DP07] Duvall, P. M. et al.: Continuous Integration. Improving Software Quality and Reducing Risk. Addison-Wesley, 2007

[EJ00] Eckstein, J.: Agile Softwareentwicklung im Großen. Ein Eintauchen in die Untiefen erfolgreicher Projekte. dpunkt-Verlag, 2000

[FM03] Fowler, M.: Patterns für Enterprise Application-Architekturen. mitp-Verlag, Bonn, 2003

[MR08] Martin, Robert C.: Clean Code. A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008

[US14] Uchiyama, S. et al.: Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning. In: Journal of Software Engineering and Applications, Band 7/2014, Seiten 983-998. Scientific Research Publishing, 2014

[GH96] Gamma, E. et al.: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 1996

[SJ09] Smith, J.: WPF-Anwendungen mit dem Model-View-ViewModel-Entwurfsmuster. MSDN Magazin, Ausgabe Februar 2009, Microsoft Corp., 2009 https://msdn.microsoft.com/de-de/magazine/dd419663.aspx, abgerufen am 11.12.2014

[HM14] Hunger, M.: Neo4j 2.0. Eine Graphdatenbank für alle. entwickler.press, 2014

[HM77] Halstead, M. H.: Elements of software science. Elsevier, New York, 1977

[PR08] Pichler, R.: Scrum. Agiles Projektmanagement erfolgreich einsetzen. dpunkt-Verlag, 2008

[MJ15] McManus, J. et al: CERT Oracle Coding Standard for Java. CERT Software Engineering Institute, Carnegie Mellon University, USA, 2015 https://www.securecoding.cert.org/confluence/display/java/CERT+Oracle+Coding+Standard+for+Java, abgerufen am 18.03.2015

[BA10] Bardas, A.: Static Code Analysis. Kansas State University, USA, 2010 ftp://ftp.repec.org/opt/ReDIF/RePEc/rau/jisomg/WI10/JISOM-WI10-A10.pdf, Zugriff am 23.03.2015

[SA05] Spillner, A. et al.: Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified Tester. 3. Auflage, dpunkt-Verlag, 2005

[MT76] McCabe, T.: A Complexity Measure. IEEE Transactions on Software Engineering, Band SE-2, Nr. 4, 1976 http://www.literateprogramming.com/mccabe.pdf, abgerufen am 03.12.2014

Glossar

Das Glossar dient dazu, die Sportwetten-spezifischen Begriffe zu erläutern, von denen mehrere im Text der Arbeit vorkommen. Dafür wurde auf die mybet-FAQ-Seite (https://help.mybet.com/, Zugriff am 12.02.2015) zurückgegriffen.

Begriff	**Erklärung**
Mainbookwette	Als *Mainbook* wird das Wettangebot bezeichnet, das →Events enthält, die noch nicht begonnen haben. Dementsprechend heißen Wetten auf solchen Events Mainbookwetten.
Livewette	Eine *Livewette* enthält mindestens einen →Tipp auf ein →Event, dass zum Zeitpunkt der Wettabgabe läuft.
Markt	Ein *Markt* besteht aus mindestens zwei →Outcomes, deren reziproke Wahrscheinlichkeiten der →Quoten zusammen 100% ergeben. Ein Markt ist also eine zusammengehörige Gruppe von möglichen Ergebnissen, die sich untereinander jedoch ausschließen, zu einem →Event. Beispiel: Der Markt „1 X 2“ enthält drei Outcomes für • Gewinn Heimmannschaft (1), • Unentschieden (X) und • Gewinn Gastmannschaft (2). 2.00 4.00 4.00 für „1 X 2“ bedeutet also, dass der Buchmacher eine 50%-ige Wahrscheinlichkeit sieht, dass die Heimmannschaft gewinnt, und jeweils 25% Wahrscheinlichkeit, dass das Spiel unentschieden ausgeht oder die Gastmannschaft gewinnt.
Event	Ein *Event* ist ein Ereignis (z. B. ein Fußballspiel), auf dem Wetten in mindestens einem Markt abgegeben werden können.
Outcome	Ein *Outcome* ist ein möglicher Ausgang eines Marktes bei einem Event. Ein Outcome muss, um tippbar zu sein, mit einer Quote > 1.00 belegt sein.
Tipp	Ein *Tipp* ist ein Outcome (mit Quote) im →Schein.

Schein

Ein (Wett-)*Schein* enthält mindestens einen Tipp und den Wetteinsatz.
Beispiel: Ein Wettschein als Kombiwette mit Einsatz 10€.

Kombiwette

Die *Kombinationswette* verbindet mehrere →Einzelwetten miteinander. Um hiermit einen Gewinn zu erzielen, müssen alle auf dem Kombinationsschein enthaltenen Einzelwetten richtig getippt werden. Die Quoten der Einzelwetten werden miteinander multipliziert.

Einzelwette

Die *Einzelwette* enthält einen Tipp, der richtig getippt werden muss, damit die Wette gewinnt.

Systemwette

Eine *Systemwette* setzt sich aus mehreren Kombiwetten zusammen. Mit einem einzigen Wettabschluss werden mehrere Tipps bzw. Tipp-Kombinationen gespielt. Es reicht zum Gewinn aus, wenn nur eine oder mehrere Teilwetten richtig sind. Beispiel:
Ein System „3 aus 5“ (kombinatorisch $\binom{5}{3}$ = 10 Wetten) enthält fünf Tipps, aus denen zehn 3er-Kombis gebildet werden. Mindestens drei Tipps müssen richtig sein, damit die Wette gewinnt.

Mehrwegwette

Bei einer *Mehrweg-Einzelwette* werden für ein Event mehrere Tipps abgegeben, wobei jeder Tipp als ein Weg und damit als eine Einzelwette abgegeben wird. Der Einsatz für die Mehrwegwette wird durch die Anzahl der Wege geteilt und pro Einzelwette gesetzt.
Als *Mehrweg-Kombiwetten* werden Wetten bezeichnet, in denen in mindestens einem Event mehrere Wege getippt wurden. Auch hier wird pro Weg eine Kombiwette abgegeben.

Quote

Mathematisch gesehen ist die *Quote* die reziproke Wahrscheinlichkeit, mit der ein Outcome, das mit der Quote belegt ist, eintritt, also im Falle eines Tipps auf das Outcome der Tipp gewinnt. Beispiel: Bei einer Quote von 2.00 geht der Buchmacher davon aus, dass das Outcome mit 50% Wahrscheinlichkeit eintritt. 1.00 entsprächen 100% (wird nie verwendet), 1.50 66,67% oder 4.00 25% Wahrscheinlichkeit. Die Quote bestimmt die Höhe des Gewinns, der ausgezahlt wird, wenn die Wette gewinnt. Dabei wird der Einsatz mit der Quote multipliziert.

Anhang

Der folgende Auszug ist dem *Projekt-Abschlussbericht – Softwareprojekt zu Anwendungen effizienter Algorithmen: „Approximation von Punktfolgen durch Kreisbogenketten“*, vorgelegt von Sven Adfeldt, Maximilian Claus, Samuel Eickelberg und Georg Nührenberg im Sommersemester 2013 bei Prof. Dr. Günter Rote, entnommen.

„1.1 Ziel

Der erforderliche Speicherplatzbedarf von Punktfolgen in einer Zeichnung ist stets linear gegeben mit der Anzahl der Punkte. Oft lassen sich aber solche Punktfolgen mit einer signifikant kleineren Anzahl von Kreisbögen approximieren, so dass der erforderliche Speicherplatzbedarf bei möglichst ähnlichem Informationsgehalt sinkt.

Die hier entwickelte Software hat das Ziel, solche Approximationen aus Punktfolgen zu berechnen und grafisch darzustellen. Daneben bietet sie eine Schnittstelle zum grafischen Editor Ipe, so dass eine Weiterverarbeitung der Approximation durch diesen möglich ist.

1.2 Aufgabenbeschreibung

Die Aufgabe ergibt sich aus der Zielstellung: Es sollte eine Software entwickelt werden, die mittels Kreisbogenketten Punktfolgen approximieren kann. Als Inspiration sollte das Paper “*Approximation of an open polygonal curve with a minimum number of circular arcs and biarcs*” dienen, aus dem die Datenstruktur “Biarc”, also “Doppelkreisbogen” abgeleitet wurde und uns die Idee der Richtungsbestimmung gab.

Im Projekt sollte es lediglich um die Anwendung einfacher Heuristiken gehen. Eine komplette Implementierung der Algorithmen im oben genannten Paper hätte den gegebenen Projektrahmen bei weitem gesprengt. Mehrere dieser Heuristiken wurden getestet, mit sehr unterschiedlichen Ergebnissen, die im weiteren Verlauf diskutiert werden.

2 Umsetzung

Für die Umsetzung haben wir uns für die Programmiersprache Java sowie ihr angegliederten Bibliotheken (im Wesentlichen Java SE) entschieden. Alle Projektteilnehmer hatten Erfahrung mit Java, so dass uns ein schneller Start in das Projekt möglich war. Im Folgenden werden die weiteren eingesetzten Technologien, die Architekturentscheidungen sowie die verwendeten Algorithmen erläutert.

2.1 Technologien

Zusätzlich zu Java und seinen Standardbibliotheken verwenden wir JAXB (Java API for XML

Binding), MulTEx (Multi-Tier Exception Handling Framework), Google Guice (für Dependency Injection) und JUnit (für automatisierte Tests).

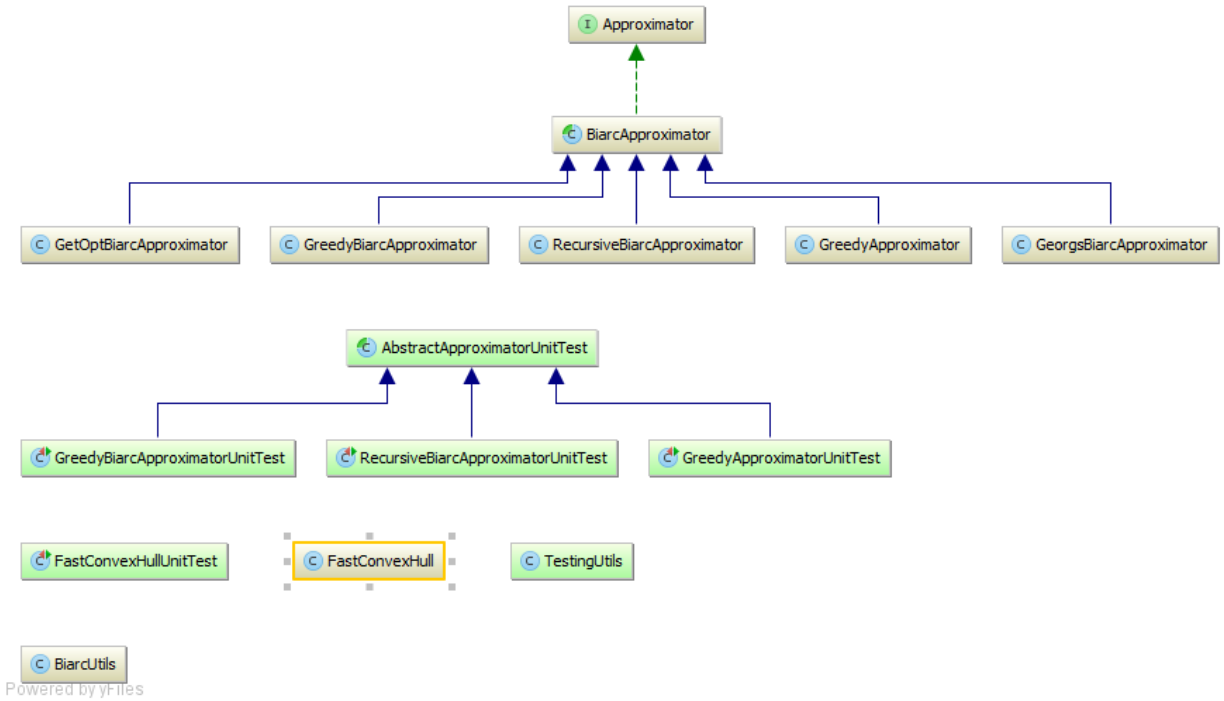
[…]

2.2 Architektur

Als Architektur haben wir einen mehrschichtigen Aufbau gewählt, untergliedert nach

- core - Algorithmen, Zustandsklassen, JAXB-generierte Klassen
- geom - Datentypen (geometrische)
- ui - Benutzeroberfläche
- io - Datenein- / -ausgabe (Dateioperationen), Datenmodelle

[…]

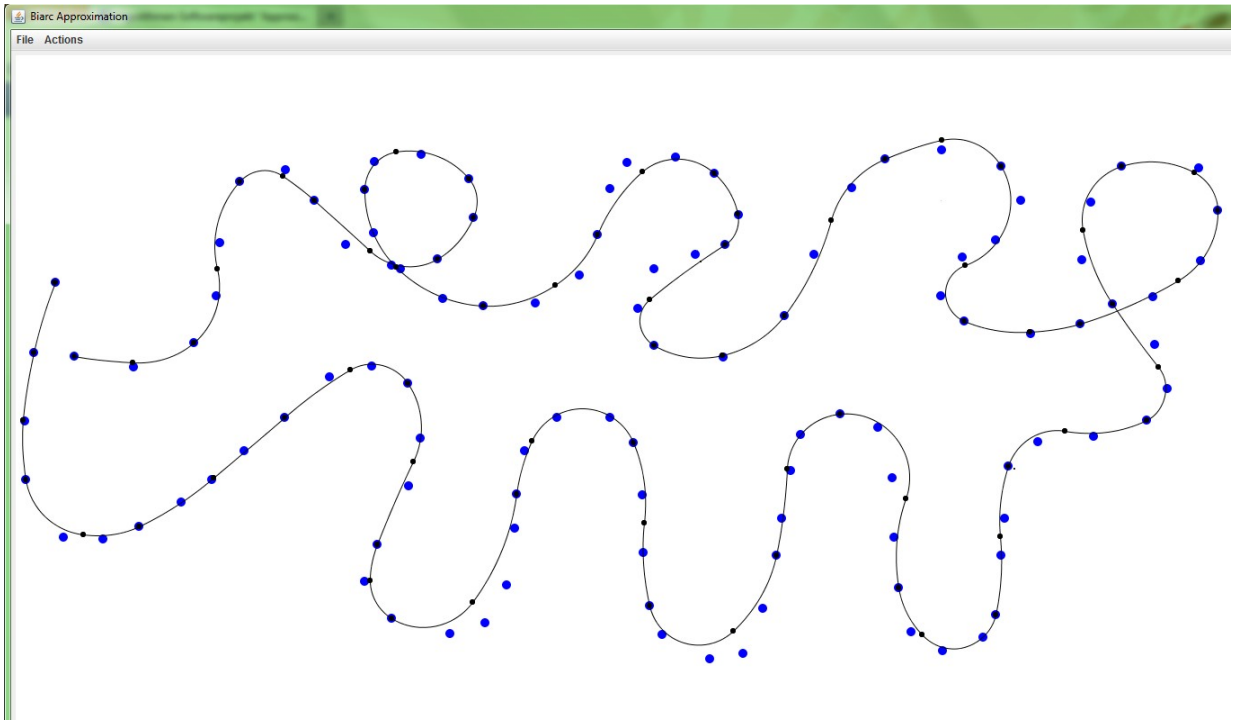


[…]

2.3 Algorithmen

Zur Berechnung der Menge aus Biarcs zur Approximation der gegebenen Punktmenge wurde eine Greedy-Strategie verwendet, die von vorne beginnend jeweils für eine gewisse Menge an Punkten einen optimalen einzelnen Biarc berechnet, der mit der Endsteigung des letzten Biarcs beginnt.

[…]



[…]

3.1 Was wurde erreicht?

Die hier implementierte Approximation von allgemeinen Punktfolgen liefert gute Ergebnisse. Selbst Schleifen und Kreise werden problemlos verarbeitet. Dies war eine der Hauptherausforderungen. Es wurde ein gut benutzbares GUI geschrieben, das die Punktfolge sowie ihre Approximation optisch leicht unterscheidbar darstellt. Darüber hinaus wurde eine Dateischnittstelle implementiert, die es ermöglicht, aus Ipe Dateien einzulesen, die Approximation zu erzeugen und diese ebenso in Ipe weiter zu verarbeiten.

Während der Implementierung wurden Tests geschrieben, die sicherstellen, dass das Programm in seinen Bestandteilen und zusammenhängend richtig arbeitet („korrekt“ wäre hier zu viel gesagt, da wir keine formalen Korrektheitsbeweise durchgeführt, sondern automatisierte Tests geschrieben haben).“

Hinweise zur CD-ROM

Auf der beiliegenden CD-ROM befinden sich diese Arbeit in elektronischer Form (*Adobe Portable Document Format*) und die gesamten Quellcodes der beiden Module *test/architecture* und *share/architecture-rules* sowie des neu geschriebenen Kreisbogenketten-Projekts.

Ordnerstruktur:

`/share/architecture-rules`	Regelgenerierer für die Architekturtests
`/test/architecture`	Architekturtest-Framework
`/biarcs`	Kreisbogenketten-Projekt
`/`	Stammordner, enthält diese Arbeit als PDF