HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Framework zur Aufzeichnung automatisierter Integrationstests auf Systemtestebene

Rainer Weinhold

Konstanz, 01.07.2008

DIPLOMARBEIT

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

an der

Hochschule Konstanz

Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang SE

Thema: **Framework zur Aufzeichnung automatisierter Integrationstests auf Systemtestebene**

Diplomand: Rainer Weinhold,
Gottfried Keller Straße 14
78476 Konstanz

1. Prüfer: Prof. Dr. Christian Johner
2. Prüfer: Christian Baranowski
SEITENBAU GmbH
Robert-Gerwig-Str. 10-12
D-78467 Konstanz

Abgabedatum: 01.07.2008

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Rainer Weinhold, geboren am 29.08.1981,

(1) dass ich meine Diplomarbeit mit dem Titel:

„Framework zur Aufzeichnung automatisierter Integrationstests auf Systemtestebene“

bei SEITENBAU unter Anleitung von Herrn Christian Baranowski selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angeführten Hilfen benutzt habe;

(2) dass ich die Übernahme wörtlicher Zitate, Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 01.07.2008 Rainer Weinhold

Zusammenfassung (Abstract)

Thema:	Framework zur Aufzeichnung automatisierter Integrationstests auf Systemtestebene
Diplomand:	Rainer Weinhold
Firma:	SEITENBAU
Betreuer:	Christian Baranowski
Abgabedatum:	01.07.2008
Schlagworte:	Softwaretest, Recorded Test Muster, Java, AspectJ, JUnit, Test-Runner, Paramterized Test Muster, Record and Playback

Ziel dieser Diplomarbeit war es, ein Framework zur Aufzeichnung von Integrationstests zu entwickeln. Umgesetzt wurde hierzu das Recorded Test Muster. Die Aufzeichnung der Integrationstests erfolgt während der Bedienung des Systems an der Oberfläche, also auf Systemtestebene.

Die Aufzeichnung wurde mithilfe des AOP-Frameworks AspectJ implementiert. Die aufgezeichneten Daten werden in einer Datenbank abgelegt. Später können aus den abgelegten Daten in einem weiteren Schritt JUnit-Tests erstellt werden.

An Stelle des von JUnit mitgelieferten ParameterizedTestRunners wurde eine eigene Implementierung des Parameterized Test Musters für JUnit erstellt, um dieses auf das automatische Generieren von Tests zu optimieren. Zur Generation wurde eine Template Engine eingesetzt.

Das Aufzeichnen von Integrationstests auf der Systemtestebene konnte durch das implementierte Framework an existierenden Projekten untersucht werden. Die daraus gewonnenen Erkenntnisse wurden ausgewertet und konnten so zu einer Bewertung des Ansatzes herangezogen werden.

Inhaltsverzeichnis

1 Kapitel

Einleitung

1.1 Motivation

Beim Schreiben von Softwaretests sind zu überprüfenden Funktionen öfters komplex und können nicht ohne weiteres durch einfache, verständliche Tests geprüft werden. Dies betrifft insbesondere Projekte, die mit Vorgehensmodellen entwickelt sind bei denen Tests erst gegen Ende erstellt werden. Dadurch ergibt sich die praktische Testbarkeit erst zu einem späten Zeitpunkt im Projekt. Dies betrifft auch bereits fertig gestellte Projekte, so genannte Wartungsprojekte.

Bei diesen Projekten sind oft Funktionen zu finden, für welche ein manuelles Erstellen von Tests zu aufwändig wäre. Hier müsste also erst das Design der Software auf eine Testbarkeit hin angepasst werden. Dies würde aber eine Änderung am Quellcode mitbringen, welcher wegen der fehlenden Tests eben nicht gegen Beschädigung gesichert werden kann. Daher gibt es die unterschiedlichsten Bestrebungen das Erstellen von Tests zu automatisieren, um auch sonst nicht prüfbare Funktionen testen zu können.

Für Systemtests hat sich hier schon seit längerem das Muster „Recorded Test“ [Meszaros, 2007] etabliert. Bei diesem Muster wird das zu testende System (engl. System under Test, SUT) ausgeführt. Im Hintergrund werden dabei Benutzereingaben als Testdaten aufgezeichnet. Die dadurch aufgezeichneten Daten können nun als Test-Skript zur Absicherung eingesetzt werden. Dieses Muster ist besonders interessant für Projekte, die ohne automatisierte Softwaretests umgesetzt wurden, denn dadurch lassen sich relativ schnell eine Menge an Regressionstests erstellen.

1.2 Problemstellung

SEITENBAU ist ein mittelständisches Software-Unternehmen das individuelle Kundenlösungen im Web-, JEE- und SOA- Umfeld entwickelt. Um diese Projekte in einer hohen Qualität umsetzen zu können, spielen automatisierte Softwaretests auf allen Testebenen eine wichtige Rolle. SEITENBAU ist immer wieder auf der Suche nach neuen Methoden für effiziente, automatisierte Softwaretests.

Daher soll in dieser Diplomarbeit die folgende Methode untersucht werden: Erstellen von Integrationstests durch Aufzeichnung auf der Systemtestebene.

Die Methode ergab sich im Rahmen der Diplomarbeit von [Baranowski, 2007]. Dort wurde aufgezeigt dass es interessant sein kann, einen Test Recorder zu implementieren, welcher Tests auf der Systemtestebene aufzeichnet, aber dabei Integrations- oder Komponententests erstellt. Wie ein solcher

TestRecorder konkret umgesetzt werden kann, soll im Rahmen der Diplomarbeit evaluiert werden.

Welche Techniken eignen sich besonders für die Realisierung? Hierzu soll ein Prototyp erstellt werden der sich erweitern lässt (Recorder Framework). Anhand des Prototypes kann dann die Qualität der Testmethode gemessen werden, z.B. welcher Abdeckungsgrad durch die aufgezeichneten Tests erreicht werden kann. Dabei soll z.B. der Quellcode untersucht werden, welcher ohne Komponenten- und Integrationstests entwickelt wurde. Besonders interessant ist der Aspekt wie gut Tests für Programmteile mit niedriger Testbarkeit mittels eines Recorders erstellt werden können. Diese Tests sollten in einer solchen Form sein, dass sie durch einen Tester angepasst werden können, wenn sich Änderungen am SUT ergeben.

1.3 Fragestellung

Wie und in welcher Qualität können Integrationstests auf Systemtestebene aufgezeichnet werden?

1.4 Teilfragen

Damit das Vorgehen zur Umsetzung besser strukturiert werden kann, wurde die Fragestellung in die folgenden Teilfragen unterteilt:

Mit welchen Methoden und Werkzeugen kann das Recorded Test Muster umgesetzt werden?

Für eine Umsetzung des Recorded Test Musters sind unterschiedlichste Methoden und Werkzeuge denkbar. Dazu werden die Methoden und Werkzeuge auf die jeweiligen Vor- und Nachteile untersucht. Wichtig ist es hier Werkzeuge zu finden, welche keine Änderungen am zu testenden Projekt benötigen.

Wie können aus den aufgezeichneten Daten, Tests generiert werden?

Da Softwaresysteme ständigen Änderungen unterworfen sind, müssen die erstellten Tests anpassbar sein. Daher ist für die Tests neben einer guten Lesbarkeit auch eine Änderbarkeit durch den Entwickler, also eine Robustheit gegen unterschiedliche Refactoring Muster wünschenswert.

Wie gut ist die Qualität der generierten Tests?

Die generierten Tests werden im letzten Schritt auf ihre Test-Qualität hin analysiert. Hierzu wird ein größeres bestehendes Projekt der Firma SEITENBAU genutzt.

1.5 Zielsetzung

Ziel ist es, eine Bewertung der möglichen Methoden und Werkzeuge zur Umsetzung des Recorded Test Musters zu erarbeiten. Hierzu wird im Rahmen dieser Diplomarbeit ein Prototyp implementiert. Durch diesen wird es möglich die Praxistauglichkeit an bestehenden Projekten zu untersuchen. Der Prototyp wird dabei in Form eines Frameworks umgesetzt, wodurch dieser durch Erweiterungen effizient an die Architektur eigener Projekte angepasst werden kann.

1.6 Symbolik

Enthaltene UML-Diagramme [Oesterreich, 2006] sind nach den Vorschlägen von [Fowler, 2004b] umgesetzt. Lediglich in Klassendiagrammen wird, der Übersicht halber, auf Getter- und Setter-Methoden verzichtet. Daher gilt für alle öffentlichen Felder in Klassen, dass diese jeweils zugehörige Getter- und Setter-Methoden besitzen und nur über diese modifiziert werden können.

Bei der Bewertung wird an mehreren Stellen ein Daumen - hoch Symbol () bzw. Daumen - herunter Symbol () genutzt, um Abschnitte mit Vor-/Nachteilen hervorzuheben.

1.7 Gliederung

Neben dieser Einleitung enthält die Diplomarbeit die folgenden Kapitel:

Kapitel 2 befasst sich mit den Grundlagen zur Durchführung dieser Diplomarbeit. Es werden die theoretischen Ansätze der einzelnen Werkzeuge und Methoden für die Umsetzung im Prototyp vorgestellt und bewertet.

Kapitel 3 beschreibt die Umsetzung des Prototypen. Dieser wird hier auf einer konzeptionellen Basis beschrieben.

Kapitel 4 zeigt die gewonnenen Erkenntnisse aus dem Einsatz des Prototypen an Beispielprojekten und einem realen Projekt auf.

Kapitel 5 diskutiert verschiedene weiterführende Ideen, welche im Rahmen dieser Diplomarbeit aufgekommen sind.

Kapitel 6 beendet die Diplomarbeit mit einer abschließenden Bewertung.

Dem letzten Kapitel folgt nach Literatur-, Abbildungs- und Abkürzungsverzeichnis noch ein Anhang mit einigen beispielhaften Auszügen aus den Generierten-Tests.

2 Kapitel
Grundlagen

In diesem Kapitel wird auf die Methoden und Werkzeuge zur Umsetzung des Recorded Test Musters eingegangen. Nach einem allgemeinen Überblick werden diese einzeln vorgestellt und bewertet.

2.1 Recorded Test Muster

Im Rahmen dieser Diplomarbeit soll das Recorded Test Muster nach [Meszaros, 2007] umgesetzt werden (Abbildung 2.1).

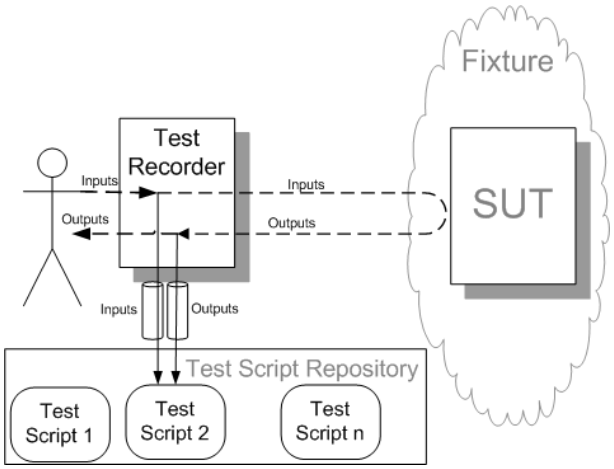

Abb. 2.1: Recorded-Test-Muster entnommen aus [Meszaros, 2007]

Dieses Muster beschreibt, wie für ein bestehendes SUT Testdaten aufgezeichnet werden können. Hierzu wird der Test-Recorder als transparenter Proxy zwischen dem Bediener und dem zu testenden System installiert. Alle Eingaben des Benutzers werden durch den Proxy an das SUT übergeben, wie auch alle Rückgabewerte wieder zurück durch den Proxy geleitet werden.

Dadurch kann der Proxy in einem Repository die zugehörigen Ein-/Ausgabepaare als Testdaten abspeichern. Für den Bediener ändert sich hierbei nichts am System, er bedient dies weiter wie gewohnt. Einzig, versteckt im Hintergrund, werden Testdaten aufgezeichnet. Durch dieses Vorgehen können relativ schnell auch größere Mengen an Testdaten aufgezeichnet werden.

Bei dem von [Meszaros, 2007] vorgeschlagenen Vorgehen liegen im Repository bereits Testskripte vor. In diesem Fall würde das Wiedergeben der Testskripte einem Data-Driven Test [Meszaros, 2007] entsprechen, da sowohl Daten als auch Logik aus dem Repository kommen. Diese Diplomarbeit wird

allerdings das Erstellen der Testskripte getrennt vom Testrecorder durchführen. Hierzu wird ein Testgenerator eingesetzt, der aus den aufgezeichneten "Roh"-Daten erst die Testskripte erstellt.

Daher ist der Prototyp in zwei Hauptkomponente aufgeteilt: Den Testrecorder zur Aufzeichnung der Testdaten in das Repository, und den Testgenerator zum Erzeugen der eigentlichen Tests aus den aufgezeichneten Daten.

2.1.1 Record and Playback

Unter der Bezeichnung "Record and Playback Test" [Meszaros, 2007] ist das Recorded Test Muster in verschiedenen Werkzeugen umgesetzt.

Diese Werkzeuge zeichnen (record) meist die Benutzer-Interaktion auf einer Oberfläche auf und generieren daraus Testskripte. Zum Ausführen der Testskripte ist der Benutzer dann nicht mehr erforderlich, da dessen Eingaben aus der Aufzeichnung wiedergegeben (playback) werden können.

Web-Oberflächen können beispielsweise mit dem freien Werkzeug Selenium [Huggins u. a., 2008] getestet werden. Dieses zeichnet Aktionen eines Benutzers auf einer Webseite auf und kann diese später in einem Browser wiedergeben.

Aber auch auf grafischen Benutzeroberflächen von Rich Client Anwendungen kann diese Methode angewandt werden. Es gibt Werkzeuge, welche beliebige Oberflächen wie Win32 oder DirectX aufzeichnen können. Ein kommerzieller Vertreter ist hier unter anderem HP Mercury.

2.1.2 Umsetzung des Recorded Test Musters für Integrationstests

Anders wie bei den verfügbaren *Record and Playback* Werkzeugen, soll in dieser Diplomarbeit eine Umsetzung für Integrationstests erstellt werden. Der hierfür zu implementierende Prototyp arbeitet daher direkt auf den Komponenten eines Systems. Dabei müssen die an Methodenaufrufen übergebenen und zurückgegebenen Werte aufgezeichnet werden. Bei nicht zustandslosen Objekten muss zusätzlich deren Zustand gesichert werden. Dies kann auch externe Zustände wie Datenbanken umfassen.

Für die Aufzeichnung wird hierzu der Testrecorder in das bestehende Projekt integriert, d.h. dieses wird um die Funktionalität zum Aufzeichnen erweitert. Dieses Injizieren des Testrecorders in ein Projekt wird als Instrumentation desselbigen bezeichnet.

2.2 Abgrenzung

Neben der Aufzeichnung von Tests, gibt es noch alternative Methoden zum automatisierten Erstellen von Tests. In diesem Unterabschnitt soll ein kurze Abgrenzung zu den alternativen Vorgehensweisen erfolgen.

2.2.1 Statische Codeanalyse

Ein mögliches Vorgehen für das automatische Erstellen von Tests ist eine Quellcode-Analyse. Dabei wird ein Baum über alle Verzweigungen einer Methode erstellt. Aus diesem können Äquivalenzklassen [Andreas Spiller, 2005] gebildet werden die zur Generierung von Tests genutzt werden können.

Mit *JUnit Generator* von [Agitar Software, 2008] gibt es ein Werkzeug, das durch eine Art Codeanalyse JUnit Tests erstellt. Agitar liefert eine Integration in die Entwicklungsumgebung mit zum direkten Erstellen der Tests.

2.2.2 TPTP

Die Eclipse Test & Performance Tools Platform [Cole u. a., 2008], kurz TPTP, liefert eine einheitliche Plattform für Test- und Profiling-Werkzeuge. In diesem Rahmen ist auch ein sogenannter API-Recorder [Safabakhsh, 2006] verfügbar welcher in der Lage ist, auf Methodenebene Testdaten aufzuzeichnen. Allerdings hat sich beim Testen gezeigt, dass dieser Teil noch äußerst instabil und unzuverlässig ist. Daher konnte TPTP nicht zum Testen des Ansatzes herangezogen werden.

2.3 Methoden und Werkzeuge zur Testaufzeichnung

In diesem Abschnitt soll eine Bewertung verschiedener Methoden und Werkzeuge zur Testaufzeichnung erfolgen.

2.3.1 Kriterien der Instrumentation

Im Folgenden werden verschiedene Möglichkeiten zum Aufzeichnen von Testdaten betrachtet. Dabei werden die folgenden Anforderungen an die Methoden gestellt:

- konfigurierbar
  Der Testrecorder sollte für unterschiedliche Projekte leicht anzupassen sein. Für eine neu aufzuzeichnende Methode sollte daher das Erstellen/Ändern einer Konfiguration ausreichen.

- nicht invasiv
  Zum Zeitpunkt der Aufzeichnung ist der Quellcode des Projekts noch nicht durch Tests abgesichert, daher sollte der Testrecorder auch keine Änderungen daran vornehmen.

- entfernbar
  Nach dem Aufzeichnen sollte der Recorder wieder entfernbar sein. Die meisten Projekte stehen vermutlich unter einer Quellcodeverwaltung, können dadurch wieder in einen alten Zustand zurückgesetzt werden. Allerdings sind dann auch alle gewollten Änderungen auf dem letzten Stand aus der Quellcodeverwaltung, und damit verloren. Daher muss der Testrecorder sich rückstandslos entfernen können.

2.3.1.1 Manuelles Einbauen

Die einfachste Methode zur Aufzeichnung stellt ein direktes Einbauen der Aufzeichnungsroutinen in den Quellcode dar. Listing 2.1 zeigt eine kleine Klasse mit einer Methode, für welche Testdaten aufgezeichnet werden sollen.

```
1 public class Dummy {
2     public int aMethod(String text) {
3         // Verarbeiten der Parameter
4         // und zurückgeben eines Wertes.
5         return ...;
6     }
7 }
```

Listing 2.1: Eine einfache Methode

Wird in diese Klasse manuell der Testrecorder eingebaut, erhält man eine Klasse wie in Listing 2.2.

```
1 public class Dummy {
2     public int aMethod(String text) {
3         TestRecorder.recordParameter( text );
4         // Der original code ...
5         int result=... // Original Ergebnis wird zugewiesen
6         TestRecorder.recordReturn(result);
7         return result;
8     }
9 }
```

Listing 2.2: Manuell eingefügter Testrecorder

Hier übergibt Zeile 3 den Parameter `text` an den Testrecorder, welcher diesen nun abspeichern kann. In den nächsten Zeilen wird der ursprüngliche Quellcode ausgeführt. Erst in Zeile 6 wird der Testrecorder erneut aufgerufen, indem ihm der Rückgabewert übergeben wird. Damit der Rückgabewert aufgezeichnet werden kann, wird dieser in Zeile 5 zuerst in einer temporären Variablen abgelegt. Der Testrecorder hat nun einen Satz an Ein-/Ausgabewerten in das Repository als Testdaten gespeichert. Diese Methode hat die folgenden Vor- und Nachteile:

Das Aufzeichnen der übergebenen Parameter erfolgt ohne Schwierigkeiten. Durch sprachliche Mittel können die Aufzeichnungsfunktionen unabhängig von den konkreten Datentypen der Parameter umgesetzt werden.

Schwieriger wird die Aufzeichnung von Rückgabewerten. In den meisten Progammiersprachen können mehrere return-Anweisungen vorhanden sein. Jede der Anweisungen ist daher zu finden und anzupassen. Wird innerhalb der Methode eine Ausnahme ausgelöst, führt dies ebenfalls zu einem Verlassen der Methode. Dies aufzuzeichnen ist im gegebenen Beispiel nicht möglich. Einzig durch das Einbetten des kompletten Inhaltes in einen großen try-catch-Block könnte der Testrecorder eine Ausnahme aufzeichnen.

Eine Alternative kann durch das Refactoring Muster “Methode extrahieren” [Fowler, 2005] erreicht werden. Hier wird der komplette Inhalt der Methode in eine zweite ausgelagert. Das Listing 2.3 zeigt den Quellcode nach dem Extrahieren der Methode.

```
1  public class Dummy {
2      public int aMethod(String text) {
3          TestRecorder.recordParameter( text );
4          try {
5              int tempResult = extractedMethod(text);
6              TestRecorder.recordReturn(tempResult);
7              return tempResult;
8          } catch( Throwable theThrowable ) {
9              TestRecorder.recordException( theThrowable );
10             throw theThrowable;
11         }
12     }
13     private int extractedMethod(text) {
14         // Orginal Code
15     }
16 }
```

Listing 2.3: Manuelle Testaufzeichnung durch Methodenextraktion

Der urspüngliche Progammcode ist nun in eine eigene Methode ab Zeile 13 ausgelagert. Für das Aufzeichnen wird in Zeile 3 der Übergabeparameter gesichert. Der Aufruf der ausgelagerten Funktion erfolgt in Zeile 5. Nun kann der Rückgabewert durch einen einzigen Aufruf an den Testrecorder übergeben werden. Durch den nun zentralen try-catch-Block (Zeile 4-11) können auch Ausnahmen aufgezeichnet (Zeile 9) werden.

2.3.1.2 Aufzeichnung durch manuellen Proxy

Betrachtet man den letzten Ansatz, das Extrahieren in eine eigene Methode, so stellt man fest, dass sich der Aufzeichnungscode kaum zwischen unterschiedlichen Methoden unterscheidet. Nach dem Extrahieren differenziert der Aufzeichungscode nur noch durch die Datentypen und den Methodenamen der extrahierten Methode.

Bei objektorientierten Programmiersprachen erfolgt ein Methodenaufruf immer auf einer Instanz der Klasse. Diesen Umstand kann man durch das Einsetzen eines Proxies [Gamma u. a., 1995] ausnutzen. Ein Proxy wird dabei anstelle der ursprünglichen Klasse instanziert und delegiert die Methodenaufrufe an eine innere Instanz der ursprünglichen Klasse. Dadurch kann ein Proxy zusätzliche Funktionalität vor und nach einem Methodenaufruf einfügen, und daher für das Aufzeichnen eingesetzt werden.

Es ist nun nötig alle Instanzierungen der betroffenen Klasse durch die des Proxies zu ersetzen. Also müssen alle new-Operatoren geändert werden:

```
1 Dummy instance = new Dummy( "parameter" );
```

Anstelle der urspünglichen Klasse muss nun der Proxy instanziert werden:

```
1 Dummy instance = new ProxyDummy( "parameter" );
```

Dank Polymorphie ist am restlichen Quellcode keine weitere Änderung mehr vorzunehmen. Der Proxy ist in Listing 2.4 erzeugt im Konstruktor eine Instanz der Original-Klasse (Zeile 4). Die eigentliche

Methode ab Zeile 6 ist gleich wie bei der vorherigen Methodenextraktion, einzig in Zeile 9 wird der Aufruf an die Original-Instanz delegiert.

```
1  public class DummyProxy {
2      Dummy myDelgate=null;
3      public ProxyDummy( String aParameter ) {
4          myDelgate=new Dummy( aParameter );
5      }
6      public int anMethod( String text ) {
7          TestRecorder.recordParameter( text );
8          try {
9              int tempResult= myDelgate.anMethod( text );
10             TestRecorder.recordReturn( tempResult );
11             return tempResult;
12         } catch( Throwable t) {
13             TestRecorder.recordExcpetion( t );
14             throw t;
15         }
16     }
17 }
```

Listing 2.4: Manueller Proxy

Werden Muster, wie etwa Dependency Injection (DI) [Fowler, 2004a], Fabrikmethoden [Gamma u. a., 1995] oder Abstrakte Fabrik [Gamma u. a., 1995] eingesetzt, kann die Instanzierung der Klassen zentral ausgetauscht werden. Dann ist dieses Vorgehen einfach umzusetzen.

Wird allerdings kein derartiges Muster eingesetzt, ist es nötig, jede Instanzierung ausfindig zu machen und durch die Instanzierung des Proxies zu ersetzen. Damit ist gegenüber dem manuellen Extrahieren kaum ein Vorteil vorhanden. Einzig beim Entfernen der Proxy Klasse kann der Compiler vergessene Programmstellen aufzeigen. Beim vorherigen Extrahieren der Methode waren Änderungen am Quellcode immer nur lokal, wodurch dort auch das Entfernen gut möglich ist. Bei einem Proxy sind die zu ändernden new-Operatoren verstreut über das ganze Projekt und daher schwieriger zu entfernen. Zusätzlich ist es durch die genutzte Vererbung nicht möglich private Methoden zu überschreiben. Dadurch sind private Methoden mit einem Proxy nicht aufzeichenbar.

2.3.1.3 Dynamische Proxies

Der vorgeschlagene Proxy muss allerdings immer noch spezifisch für jede Klasse entwickelt werden. Dies kann in vielen Fällen durch den Einsatz sogenannter dynamischer Proxies verallgemeinert werden. Hierbei wird von der Programmiersprache selbst oder durch entspechende Erweiterungsbibliotheken die Möglichkeit geschaffen dynamisch, also zur Laufzeit, einen Proxy zu erzeugen.

Ein dynamischer Proxy ist unabhängig von der Klasse. Dadurch kann der Recorder ebenfalls unabhängig von der konkreten Klasse umgesetzt werden.

In der Sprache Java sind dynamische Proxies nur für Schnittstellen erzeugbar. Somit wird es schwierig einen dynamischen Proxy für eine Klasse zu erzeugen, welche mehr Methoden bereitstellt als die

von ihr implementierten Schnittstellen. Da außerdem Schnittstellen nur öffentliche Methoden enthalten können, sind somit auch nur öffentliche Methoden aufzeichenbar. Bei dem manuell erstellten Proxy waren es möglich geschützte (protected) Methoden aufzuzeichnen. Diese beiden Einschränkungen müssen allerdings nicht für jede Programmiersprache gelten. Die Einschränkungen können im Falle von Java durch den Einsatz der cglib [Baliuka u. a., 2008] Bibiothek umgangen werden. Diese erlaubt es auch, dynamische Proxies für Klassen zu erzeugen.

2.3.1.4 Der Java ClassLoader

Ein ClassLoader ist für das Laden der Bytecoderepräsentation einer Java Klasse zuständig. Da es möglich ist den ClassLoader durch eine eigene Implementierung zu ersetzen, kann dadurch auch eine Manipulation am Bytecode vorgenommen werden.

Der ClassLoader kann an zentraler Stelle ausgetauscht werden. Dadurch kann der Testrecorder sehr einfach ein-/ausgeschaltet werden.

Ein entspechend implementierter ClassLoader delegiert das Laden von Klassen nicht mehr an seine Eltern ClassLoader, sondern fängt diese direkt ab. Dies ist kein Problem, entspricht aber nicht dem empfohlenen vorgehen für ClassLoader [Wunderlich, 2005]. Die eigentliche Manipulation am Bytecode ist nicht einfach, weswegen auf vorhandene Bibliotheken, wie beispielsweise Javaassist [Chiba, 2008], zurückgegriffen werden sollte.

2.3.1.5 Nutzung eines Java Agent

Die Sprache Java kennt seit dem JDK 1.5 sogenannte Java Agents. Ein Agent kann zur Laufzeit Manipulationen am Bytecode von Klassen durchzuführen. Hierdurch kann direkt die Zielmethode instrumentiert werden.

Durch die Bytecode Manipulation ist es möglich private Methoden aufzuzeichnen. Anders als der Ansatz mit dem Classloader, ist ein Java Agent für ein nachträgliches Manipulieren am Bytecode gedacht. Der Agent wird während dem Starten als zusätzlicher Parameter an die Laufzeitumgebung übergeben. Dadurch ist keine Änderung mehr am Quellcode nötig. Das Ein- und Ausschalten des Recorders kann somit durch das Angeben eines Startparameters erreicht werden.

Ist der Zugang zur Laufzeitumgebung eingeschränkt, kann das Übergeben von zusätzlichen Parametern schwieriger sein. Bei einem parallelen Einsatz mehrerer Agents gibt Java keine Garantie über die Reihenfolge.

2.3.1.6 Bytecode Manipulation durch AOP

Seit einiger Zeit gibt es das Paradigma der Aspektorientierten Programmierung (AOP) [Kiczales u. a., 1997]. Diese verfolgt den Ansatz, wiederkehrende Funktionalität in sogenannte Aspekte auszulagern. Die wiederkehrende Funktionalität wird in einem Aspekt implementiert und kann über sogenannte Pointcuts an beliebig definierbaren Stellen in das Programm eingebaut werden. Das Einbauen der Aspekte erfolgt meist über eine Manipulation am Bytecode.

Mit AspectJ [Colyer u. a., 2008] ist eine bekannte und mächtige Implementierung von AOP für Java verfügbar. Zur Umsetzung führt AspectJ einige neue Sprachkonstrukte ein. Der Aspekt wird ähnlich einer Java Klasse geschrieben und enthält als Implementierung normalen Javacode. Ein kleiner Aspekt ist in Listing 2.5 dargestellt.

```
1 public aspect NullAspect {
2     protected pointcut pc_MainCall() :
3           call( void com.example.MainApp.main(..) );
4
5     Object around()  : pc_MainCall()  {
6         Object theReturnValue=proceed();
7         return theReturnValue;
8     }
9 }
```

Listing 2.5: Einfacher Aspekt

Der Aspekt besteht aus den folgenden Elementen:

Zeile 2 - Ein Pointcut beschreibt die Stelle im Quellcode, für welche ein Aspekt aktiviert wird. Hier wird als Beispiel die main-Methode der Klasse MainApp beschrieben. Es werden dabei nur Methoden ohne Rückgabewert betrachtet. Durch Wildcards können Pointcuts auch unscharf angegeben werden, so wird hier der Pointcut mit "(..)" auf Methoden mit beliebiger Anzahl an Parametern aktiv.

Der eigentliche Aspekt ist in der Zeile 5-8 codiert. Zeile 5 beschreibt den Aspekt als around [Böhm, 2006] Aspekt, welcher an allen Stellen die durch den Pointcut pc_MainCall spezifiziert sind, aktiviert wird. Ein around Aspekt umschließt einen Methodenaufruf, kann diesen also auch komplett abfangen. In Zeile 6 wird mit dem Aufruf der proceed() Methode der eigentliche Originalcode aufgerufen.

Durch das Einfügen des Aspektes direkt durch den Compiler, werden keine Änderungen am Quellcode und an der Laufzeitumgebung nötig. Der Aspekt ist unabhängig von der eigentlichen Methodensignatur und kann daher funktionell einmal implementiert werden. Einzig der Pointcut, also die Stelle an welche der Aspekt eincompiliert wird, ist spezifisch für die jeweilige Methode.

Im Normalfall muss das Projekt zum Einbauen des Aspektes neu compiliert werden. AspectJ kann dies aber auch direkt auf Bytecodeebene. Dadurch kann ein Aspekt auch in ein bereits übersetztes Projekt eingebaut werden.

2.3.2 Ausgewählte Technologie

Neben den betrachteten Ansätzen sind auch weitere Ansätze denkbar. Beispielsweise gibt es für Java mit der Java Platform Debugger Architecture (JPDA) [Sun Microsystems Inc., 2004] die Möglichkeit während der Laufzeit Anwendungen zu überwachen. Der bekannteste Einsatz ist hier sicher der Debugger. Aber auch Performance Tools können diese Schnittstelle nutzen um Daten über das Laufzeitverhalten einer Anwendung zu erheben.

Für den implementierten Prototyp wurde der AOP Ansatz gewählt. Durch den Einsatz von AspectJ, kann für Java auf eine gut getestete Bytecodemanipulation zurückgegriffen werden. Zudem gibt es bereits weitreichende Tooluntersützung. Mit den AspectJ Development Tools [Chapman u. a., 2008],

kurz AJDT, gibt es eine Integration von AspectJ in die Eclipse Entwicklungsumgebung. AJDT liefert neben der Integration des AsepctJ Compilers (ajc) auch Unterstützung in Form einer Autovervollständigung und ähnlichem. Hierdurch sind Aspekte sehr komfortabel nutzbar. Durch den Einsatz von AspectJ ist zur Laufzeit das zusätzliche Einbinden der AspectJ Runtime Bibliothek nötig. Allerdings ist in allen Fällen sowieso mindestens die Testrecorder-Bibliothek als Abhängigkeit hinzuzufügen.

Der Aspekt kann entweder spezifisch an eine bestimmte Methode angehängt werden, oder durch den Einsatz von Wildcards auf alle Methoden angewandt werden. In letzterem Fall muss dann allerdings im Aspekt entschieden werden ob die aufgerufene Methode aufgezeichnet werden soll. Da dies Auswirkungen auf die Performance des Systems haben kann, wird in den folgenden Unterpunkten dies abgeschätzt.

Performanceabschätzung von AspectJ

Zur Abschätzung der Performance wurde ein kleines Testprogramm (Sequenziagram 2.2) geschrieben (Quellcode im Anhang A.1). Das Programm soll die Methode `subMethodCall` aufzeichnen. Damit durch das Programm die Auswirkung des Aspektes in anderen Methoden messbar wird, wird die Zielmethode nicht direkt aufgerufen, sondern über die rekursive Methode `callSubMethodCall` erst nach mehreren Verschachtelungen. Durch die Rekursion wird erreicht, dass der Laufzeitverlust eines unscharfen Aspektes sich auf die `callSubMethodCall` Methode auswirkt. Dadurch ist der Zeitverbrauch eines allgemeinen Aspekts abschätzbar.

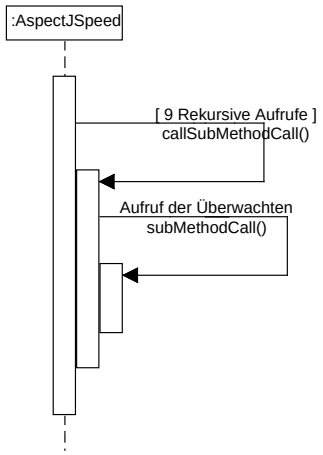

Abb. 2.2: Sequenzdiagramm derPerformanceabschätzung.

Für aussagekräftige Durchschnittswerte wurden jeweils 10000000 Aufrufe mit einer Rekursionstiefe von 10 gemittelt. Die Ergebnisse sind in der Tabelle 2.1 dargestellt.

Lauf	Beschreibung	Zeitaufwand *[ms]*
1	Programm ohne Aspekte	32
2	Spezifischer Aspekt für Zielmethode	52
3	Bedingung im Aspekt enthalten	87
4	Aspekt mit Bedingung auf allen Methoden aktiv	381

Tab. 2.1: AspectJ - Performance

Lauf 1 Das Programm wurde ohne den Einsatz von Aspekten gestartet. Dies ergibt eine Referenzzeit von $32ms$ welche das Originalprogramm benötigte.

Lauf 2 Nun wurde ein Aspekt, der genau auf die Zielmethode angepasst ist, definiert und eincompiliert. Da der Aspekt genau auf der Zielmethode aktiv ist, sind im Aspekt keine weiteren Überprüfungen mehr nötig, dieser ruft einfach mit `proceed()` die Originalmethode. Die mehr verbrauchten $52 - 32 = 20ms$ kommen alleine durch die Indirektion des Aspektes zustande.

Lauf 3 Als nächstes wurde dem Aspekt eine if-Bedingung hinzugefügt um sicherzustellen, dass der Aspekt sich in der gewünschten Methode befindet. Dies ist eigentlich überflüssig, da der Aspekt ja sowieso nur auf der Zielmethode aktiviert ist. Daher kostet die zusätzliche if-Bedingung auch nur weitere $87 - 52 = 35ms$.

Lauf 4 Im letzten Lauf wurde nun der Pointcut mit wildcards versehen und ist somit bei jedem Methodenaufruf aktiv. Insbesondere findet nun bei jedem Aufruf der rekursiven Methode eine Überprüfung statt. Diesmal benötigt das Programm mit $381 - 52 = 329ms$ wesentlich mehr Zeit.

Beim Einsatz eines allgemeinen Aspektes, könnte der Testrecorder auch direkt zur Laufzeit angepasst werden. Dadurch wäre es möglich zur Laufzeit die Aufzeichnungsmethode zu konfigurieren. Zum Ändern der aufzuzeichnenden Methode wäre so kein ändern am Aspekt nötig und somit kein Neustart erforderlich.

Da ein allgemeiner Aspekt alle Methoden des Systems verlangsamt, ist es sinnvoller den Aspekt spezifisch auf die Aufzuzeichnende Methode anzupassen um das Verhalten des Systems nicht zu verändern.

Gewählte Aspekt Art Gewählt wurde der spezialisierte Aspekt, da hier weniger Einfluss auf das zu Testende System genommen wird.

2.3.3 Speicherung der Testdaten

Im Recorded Test Muster werden die aufgezeichneten Daten in ein sogenanntes Repository abgelegt. Für die Umsetzung des Prototypen muss dieses Repository implementiert werden. Hierbei sind zwei Eigenschaften zu betrachten:

- Datenformat für den Zustand von Objekten.
- Containerformat, in welches die Objekte gespeichert werden.

Zur Beantwortung dieser Fragen werden jeweils verschiedene Methoden zur Lösung betrachtet und bewertet.

2.3.3.1 Datenformat für den Zustand von Objekten

Während der Aufzeichnung werden von verschiedenen Objekten (Parameter, Rückgabewerte etc.) Momentaufnahmen erzeugt. Dies entspricht in objektorientierten Sprachen dem Wert der Felder einer Objektinstanz. Dabei können diese Felder wiederum weitere Instanzen halten, wodurch eigentlich ein Objektbaum zu speichern ist. Zur Testausführung muss dieser in einen gleichen Objektbaum wieder hergestellt werden. Dabei sind folgende Vorausetzungen zu erfüllen:

- lesbar
  Damit ein Tester die aufgezeichneten Daten betrachten und bewerten kann ist es wünschenswert, wenn die Datenrepräsentation so gewählt ist, dass diese von einem Menschen lesbar ist.

- nicht invasiv
  Wie auch schon bei der Aufzeichnung darf der Einsatz keinen Einfluss auf die Funktionalität des Projektes haben. Es darf also keine Manipulation am Quellcode nötig sein.

- filterbar
  Da es nötig sein kann bestimmte Objekte von einer Sicherung auszuschließen, muss eine Filterung bereits beim Aufzeichnen möglich sein.

- instanzierbar
  Es dürfen keine Voraussetzungen für den Aufbau der Objekte gemacht werden, beispielsweise müssen auch nicht öffentliche Felder gesichert werden können. Beim Wiederherstellen der Objekte ist zu beachten, dass nicht jede Klasse einen parameterlosen, öffentlichen Konstruktor besitzen muss. Aber auch diese Klassen müssen wiederherstellbar sein.

Java Serialisierung

Java selbst bringt einen Mechanismus der Serialisierung mit. Hierbei wird ein Objektbaum in einen Byte-Stream gewandelt. Dieser kann gespeichert werden und zu einem späteren Zeitpunkt zurück in eine Instanz des Objekts gewandelt werden.

 Dieses Vorgehen ist Teil von Java und kann beliebige Objektbäume sichern. Es kommt auch mit zirkulären Referenzen zurecht und hat sich bewährt.

 Durch das binäre Format ist es für einen Tester nicht möglich sich ein gespeichertes Objekt anzusehen. Es können nur Objekte serialisiert werden, welche das Marker-Interface “Serializable” implementieren. Dadurch sind Änderungen am Quellcode nötig. Dies gilt auch für den Ausschluss von Feldern. Dies kann nur durch das Einfügen des `transient` Schlüsselwortes direkt im Quellcode erreicht werden.

JAXB

Mit der Java Architecture for XML Binding, kurz JAXB, ist es möglich Java Objekte in eine XML Repräsentation zu überführen. Aus diesem XML kann dann wieder eine Instanz erzeugt werden.

Das Vorgehen ist standardisiert [Fialli und Vajjhala, 2003; Kawaguchi, 2006]. Durch die Speicherung als XML wird eine gute Lesbarkeit für den Entwickler erreicht.

JAXB implementiert ein Binding, was gegenüber einer Serialisierung auf mehr Vorwissen aufbaut. Beispielsweise muss jede zu speichernde Klasse vorher bekannt sein. Dies erzeugt zusätzlichen Overhead.

XStream

Mit der Bibliothek XStream [Schaible u. a., 2008] gibt es für Java eine Alternative zur mitgebrachten Java Serialisierung. XStream ermöglicht es beliebige Java Objekte direkt nach XML zu serialisieren. Dabei ist die Programmierschnittstelle sehr einfach gehalten. Eine einfache Serialisierung nach XML sieht wie folgt aus:

```
1 String xml = new XStream().toXml(meinObjekt);
```

Diese Zeile serialisiert bereits den kompletten Objektbaum des übergebenen Objekts (`meinObjekt`) nach XML. Genauso einfach kann aus der XML Darstellung wieder eine Instanz erzeugt werden:

```
2 Object copy = new XStream().fromXml(xml);
```

Durch das Speichern als XML ist auch hier eine gute Lesbarkeit vorhanden. Neben der einfachen API bietet die Bibliothek aber auch wesentlich mächtigere Funktionen an. Es ist ohne weiteres möglich einzelne Felder auszuschließen. Dies kann dabei über entsprechende Methodenaufrufe an der XStream Instanz eingestellt werden. Dadurch ist XStream komplett transparent und es sind keinerlei Eingriffe in den Quellcode nötig.

Felder die mit dem Schlüsselwort `transient` markiert sind, ignoriert auch XStream. Dies kann daher in Ausnahmefällen zu Problemen führen. Da `transient` allerdings eher selten eingesetzt wird, ist im Rahmen dieser Diplomarbeit dieser Effekt nicht aufgetreten und sollte in den meisten Situationen durch das Anpassen der Tests umgehbar sein.

Gewählte Speichertechnik

Die Wahl fiel auf XStream, da hier die Lesbarkeit erhalten bleibt, aber gleichzeitig eine einfache und doch mächtige API zur Verfügung steht. XStream kann sowohl mit privaten Feldern umgehen als auch Objekte deserialisieren, welche keinen Standardkonstruktor bereitstellen. Auch sind Klassen mit privaten Konstruktoren kein Problem für XStream.

2.3.3.2 Persistenz

Die erzeugten Abzüge der Objektzustände müssen durch das Repository persistent gemacht werden. Hierzu werden im folgenden zwei mögliche Methoden bewertet.

Basierend auf Dateien

Jeder erzeugte Abzug wird in eine eigene Datei gespeichert. Gruppierungen können mithilfe von Ordnerstrukturen erreicht werden.

 Die Objekte liegen in jeweils eigenen Dateien vor, dadurch ist es sehr einfach, gezielt ein bestimmtes Objekt zu betrachten. Da als Datenformat XML eingesetzt wird, ist der Einsatz gängiger XML Editoren möglich. Daher kann zur Ansicht und Bearbeitung auf leistungsfähige Editoren zurückgegriffen werden. Da auch die Ordnerstruktur durch Dateisystembetrachter navigierbar ist, sind somit keine weiteren Werkzeuge zu entwickeln.

 Je nach Umfang der Aufzeichnung können sehr schnell viele Objekte anfallen. Dies führt zu einer großen Anzahl an Dateien. Dadurch wird auch eine Ablage in Ordnern nicht wesentlich übersichtlicher. Für die Dateinamen muss zur Auseinanderhaltung auf IDs zurückgegriffen werden. Dies ist allerdings für einen Menschen relativ schlecht zu verstehen. Je nach Art des Projektes kann das Ausführungsverzeichnis variieren. Dadurch kann die konkrete Lokalisierung der Dateien schwierig werden. Solange die Ausführung auf dem gleichen Rechner stattfindet, kann dies durch absolute Pfadangaben weitgehend umgangen werden. Allerdings können je nach System auch einfach Schreibrechte auf der Dateisystemebene fehlen.

Basierend auf einer Datenbank

Um eine Unabhängigkeit von der physikalischen Lokation zu erreichen, bietet sich der Einsatz einer Datenbank an. Hier werden die Objekte, bzw. deren XML Repräsentation in Tabellen abgelegt.

 Solange keine restriktiven Firewalls im Netzwerk eingesetzt werden, sollte dieser Ansatz auch über Rechnergrenzen hinweg funktionieren. Durch den Einsatz von OR-Mappern [Ambler, 2003] ist es sehr einfach möglich, das Repository zu implementieren. Dadurch wird auch eine Unabhängigkeit gegenüber der konkreten Datenbanksoftware erreicht.

 Das Betrachten der aufgezeichneten Daten ist nur indirekt durch einen Datenbankbrowser möglich. Dieser erkennt die XML Daten zwar als Text, unterstützt aber keine weiteren Features, wie es ein XML Editor würde.

Gewählte Persistenztechnik

Der Einsatz einer Datenbank scheint sinnvoll, da dies sehr einfach durch den Einsatz von OR-Mappern implementiert werden kann. Bei einem dateibasierten Ansatz hätten die Referenzen zwischen Teilen der Dateien über IDs verwaltet werden müssen, was quasi zu einer manuellen Implementierung einer Datenbank führt, und somit einen erheblichen Mehraufwand bedeutet hätte. Zusätzlich wird das Problem der Lokalisierung von Dateien umgangen.

2.4 Methoden und Werkzeuge für die Test Generierung

Nach der Aufzeichnung liegen die rohen Testdaten im Repository vor. Um nun das System damit testen zu können, müssen diese in eine ausführbare Form transformiert werden. Diese Transformation wird im Folgenden als Test-Generierung bezeichnet. Die dazu verfügbaren Methoden werden in diesem Abschnitt betrachtet.

2.4.1 Testmethoden

Die folgenden Arten von Tests können aus den aufgezeichneten Daten generiert werden:

Integrationstests Diese testen das Verhalten zwischen den Komponenten eines Systems. Diese Testart kann je nach Art der zu testenden Methoden sehr einfach implementiert werden. Beispielsweise reicht bei zustandslosen Webservices das Aufrufen der Serviceschnittstelle und das Überprüfen der zurückgegebenen Werte aus. Es sind bei Integrationstests normalerweise keine Mockobjekte nötig.

Es reicht prinzipiell, ein zu testendes Objekt während der Aufzeichnung mit dem kompletten Zustand zu sichern, so dass der Zustand zum Testzeitpunkt wiederhergestellt werden kann. Nicht alle so gespeicherten Testdaten müssen aber auch zum späteren Testzeitpunkt weiterhin gültig sein. Beispielsweise wird eine aufgezeichnete Datenbankverbindung zu einem späteren Zeitpunkt ungültig sein.

Komponententests Komponententests unterscheiden sich von Integrationstests dahingehend, dass eine einzelne Komponente isoliert getestet wird. Es werden also alle umgebenden Objekte mit Hilfe von Mockobjekten simuliert. Damit die Mockobjekte automatisch erzeugt werden können, müssen allerdings schon während der Aufzeichnung alle an die Umgebung gemachten Methoden- und Feldzugriffe mit in das Repository gespeichert werden.

2.4.2 Unit Framework

Die zu erstellenden Softwaretests müssen automatisierbar sein. Nur dadurch ist es möglich, diese erneut auf das Projekt anzuwenden und dieses so gegen Veränderungen abzusichern. Dadurch können diese Tests auch für Regression-Tests oder für continuous integration [Fowler und Foemmel, 2000] eingesetzt werden.

Aus einem Vorschlag von [Beck, 1994] wurden eine Reihe von Test Frameworks entwickelt. Diese Frameworks dienen zur Automatisation von Komponententets, engl. Unit-Tests. Daher wird diese Art von Framework meist als xUnit bezeichnet, wobei x der Platzhalter für die jeweilige Sprache ist. Ein bekannter Vertreter ist beispielsweise JUnit für Java.

Da sich JUnit in der Javawelt durchgesetzt hat, werden die meisten Entwickler und Tester bei damit erstellten Tests keine Verständnis Probleme haben.

Grundsätzlich wäre es auch möglich die Testdaten direkt während der Testausführung aus dem Repository herauszulesen. Man könnte also einen allgemeinen Testcode implementieren, welcher sich je

nach Testdaten anders verhält. Bei diesem Vorgehen ist allerdings vieles der Testlogik starr, kann also nicht nachträglich angepasst werden. Außerdem sollten die Tests mit dem restlichen Projekt unter der Quellcode-Verwaltung stehen, wodurch diese auch automatisiert auf einem Server ausgeführt werden können.

Durch die gute Unterstützung in Entwicklungsumgebungen und build-Systemen ist das Generieren von JUnit-Tests als beste Lösung anzusehen. Die erstellten JUnit-Tests können unter die Quellcode-Verwaltung gestellt werden und mit dem sich ändernden Projekt mitwachsen. Hierzu bietet es sich auch an, die wirklich benötigten Testdaten aus dem Repository herauszulösen und paralell zum Test als Dateien zu speichern.

2.4.2.1 Parameterized Test

Mit dem Muster Parameterized Test beschreibt [Meszaros, 2007] wie ein Test entwickelt werden kann, bei welchem jeder Testfall sich nur in den Testdaten, aber nicht in der Testlogik unterscheidet. Der Ablauf ist in Abbildung 2.3 dargestellt.

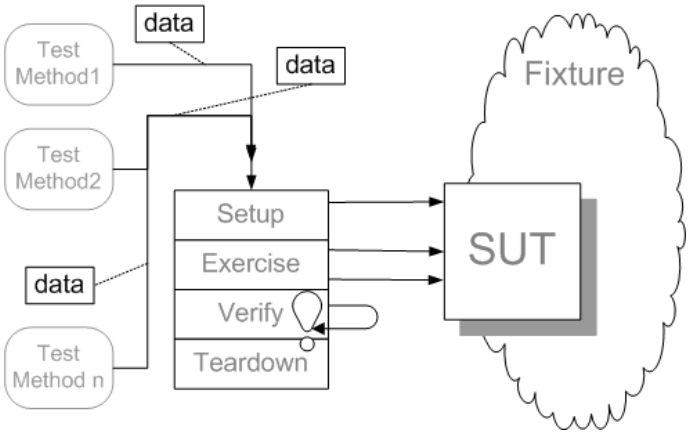

Abb. 2.3: Parameterized Test Muster entnommen aus [Meszaros, 2007]

Das JUnit Framework bringt eine eigene Implementierung des Parameterized Tests mit. Dabei werden die einzelnen Daten der Testfälle über Konstruktorparameter injiziert. Listing 2.6 zeigt wie der Parameterized Test über die `@RunWith` Annotation aktiviert wird und im Konstruktor (Zeile 5-8) die Parameter in lokale Felder gesichert werden.

```
1 @RunWith(Parameterized.class)
2 public class WandlerTest {
3   int fWert;
4   String fAusgabe;
5   public WandlerTest( int wert, String ausgabe) {
6     fWert=wert;
7     fAusgabe=ausgabe;
8   }
```

Listing 2.6: JUnit ParameterizedTestRunner

Die Daten kommen aus einer statischen Methode die über eine `@Parameters` Annotation markiert ist. Die in Listing 2.7 abgedruckte Version liefert ein zweidimensionales Feld zurück. Eine Spalte entspricht den Konstruktor-Parametern (int, String) und eine Zeile je einem Testdatensatz.

```
9   @Parameters
10  public static Collection getTestDaten(){
11    return Arrays.asList(new Object[][] {
12        { 4, "vier"}
13        { 2, "zwei"}
14      });
15  }
```

Listing 2.7: JUnit Parameters Methode

Für jeden angegebenen Testdatensatz wird eine neue Instanz der Testklasse erstellt, und die Testdaten als Parameter übergeben. Die Daten werden daher meist im Konstruktor in eigenen Feldern gesichert, um in den einzelnen Testmethoden zur Verfügung zu stehen. Im Test (Listing 2.8) werden die gespeicherten Felder überprüft.

```
16  @Test
17  public void wandle() {
18    assertEquals( fAusgabe , Wander.wandle( fwert ) );
19  }
```

Listing 2.8: Die Testmethode des Parameterized Tests.

Durch einen Parameterized Test ist es möglich datenabhängige Tests zu erstellen. Für den Einsatz der JUnit Implementierung spricht, daß keine weiteren Abhängigkeiten neben JUnit für die Testausführung erforderlich sind.

Die JUnit Implementierung hat allerdings den Nachteil, dass die Testdaten über eine statische Methode bereitgestellt werden.

Durch den Einsatz der `@Parameters` Annotation ist es zwar möglich, mehrere solcher Methoden anzugeben, deren Testdaten werden aber zu einer gemeinsamen Liste zusammengefasst. Somit ist pro Testklasse auch nur eine Datenquelle möglich.

Bei manuell geschriebenen Tests verfolgt man normalerweise das Ziel, für jede zu testende Methode eine zugehörige Testmethode zu erstellen. Das Ablegen der aufgezeichneten Daten direkt im JUnit Test ist zwar möglich, führt aber bei größeren XML Repräsentationen schnell zu einer schlechten Lesbarkeit. Die Daten sollten daher besser in eine Datei ausgelagert werden. Daher sollte eine eigene Implementierung des Parameterized Test Musters angestrebt werden, welche diese Nachteile umgeht.

Die Integration in JUnit muß aber gewährt bleiben, da sich die Tests so nahtlos in bestehende Testwerkzeuge integrieren lassen. JUnit kann hierzu durch einen sogenannten Test-Runner erweitert werden, durch welchen der Testablauf mit einer eigenen Klasse kontrolliert werden kann. Das Bereitstellen eines Test-Runners ermöglicht eine Integration mit bestehenden Umgebungen.

2.4.3 Generieren der Tests

Zur Generierung ist eine Transformation der Testdaten aus dem Repository in die für den JUnit-Test benötigten Artefakte nötig. Die Artefakte umfassen neben der eigentlichen JUnit-Testklasse auch die gespeicherten Testdaten. Zum generieren bietet sich der Einsatz einer sogenannten Template Engine an. Eine Template Engine liest eine Template Datei ein und ersetzt enthaltene Platzhalter zur

Laufzeit durch entsprechende Werte. Durch dieses Vorgehen können sehr gut Inhalte erzeugt werden die größtenteils aus statischen Teilen bestehen. In unserem Fall können beispielsweise die Rümpfe für Klassen und Methoden im Template abgelegt werden. Zur Laufzeit wird dann lediglich der Klassen- bzw. Methodename angepasst.

Apache Velocity

Die Velocity Engine [Glass-Husain u. a., 2008] kommt aus dem Apache Projekt. Parameter können durch einen Context übergeben werden, auf welchen man im Template zugreifen kann. Neben dem einfachen Einfügen von Werten gibt es auch verschiedene Sprachkonstrukte wie Schleifen und Bedingungen. Dadurch können beispielsweise auch Java-Collections oder Arrays durchlaufen werden.

Eine einfaches Hello World Template ist in Listing 2.9 dargestellt.

```
1 #set( $foo = "World" )
2 Hello $foo
```

Listing 2.9: Apache Velocity Beispiel

Die Velocity Engine liefert für dieses Template "Hello World" zurück. Die Variable `$foo` wird in diesem Fall direkt im Template zugewiesen, könnte aber auch über besagten Context bezogen werden und dadurch vom Programm geändert werden.

Die Templates werden zur Laufzeit übersetzt. Vorstellbar wäre hier auch die Möglichkeit, dass ein Nutzer die Templates einfach lokal für eigene Zwecke modifizieren kann.

Für Eclipse gibt es keine direkte Editorunterstützung, aus Sicht von Eclipse sind die Templates daher einfache Textdateien. Dadurch fehlen Features wie beispielsweise Autovervollständigung. Da die Transformation erst zur Laufzeit erfolgt, werden Syntaxfehler auch erst zur Laufzeit gefunden.

JET

Aus dem Eclipse Modeling Framework [Merks u. a., 2008] stammen die Java Emitter Templates (JET). Der Syntax der Templates ist an den von Java Server Pages angelehnt. Die Templatedatei wird bereits während der Entwicklung durch ein Eclipse Plugin in eine Klasse umgewandelt. Diese wird über den normalen Compiler übersetzt. Zur Laufzeit wird daher die Templatedatei nicht mehr benötigt, da diese nun in Form der Klasse vorliegt.

Eine einfaches Hello World Template ist in Listing 2.10 dargestellt. Dieses Template produziert ein "Hello World" beim Ausführen.

```
1 <% String foo="World"; %>
2 Hello <%= foo%>
```

Listing 2.10: JET Beispiel

Durch die Wandlung in eine Javaklasse kann im Template selbst der komplette Javasprachumfang genutzt werden. Zudem sind syntaktische Fehler bereits zum Entwicklungszeitpunkt erkennbar.

Daneben gibt es für Eclipse auch einen Editor der Autovervollständigung unterstützt. Da zur Laufzeit eine normale Javaklasse ausgeführt wird, kann auf den leistungsfähigen Debugger von Eclipse zurückgegriffen werden.

Die Transformation des Templates in eine Javaklasse erfolgt durch das Eclipse Plugin, weswegen der Einsatz von Eclipse zur Entwicklung unbedingt nötig ist.

Gewählte Template Engine

Die Wahl fiel auf JET, da dies sehr leistungsfähig ist und durch die Integration in Eclipse eine gute Hilfestellung während der Entwicklung liefert. Eventuell ist es für eine Integration in Maven möglich auf einen Eclipse Headless build [Barchfeld u. a., 2008] zurückzugreifen.

2.4.4 Test Evaluation

Der four-phase-Test nach [Meszaros, 2007] enthält als letzte Phase die Evaluation. In dieser Phase wird das SUT auf das erfolgreiche Ausführen des Tests überprüft. Bei einer zustandslosen Methode reicht hier das Vergleichen des Rückgabewertes gegen den aufgezeichneten Wert aus.

Das Vergleichen von Objekten erfolgt dabei über die equals Methode, welche daher sinnvoll überschrieben sein muss. Dies ist allerdings nicht immer gegeben, und müsste daher implementiert werden, was zu einer Änderung am Quellcode führt.

Vergleichen auf XML Basis In [Baranowski, 2007] wurde daher vorgeschlagen, den Vergleich auf der Basis eines XML Baumes durchzuführen. Da Objekte nach einer Serialisierung mit XStream als XML vorliegen, kann dessen Baum zum Vergleich herangezogen werden. Hierzu kann beispielsweise XMLUnit [Bacon und Martin, 2008] eingesetzt werden, welches ein Vergleichsfunktion für XML Bäumen mitbringt.

Filterung Bei Objektbaum-Elementen, die sich beispielsweise durch ein aktuelles Datum unterscheiden, ist eine Filterung vor dem eigentlichen Vergleich sinnvoll. Eine Filterung kann dabei auf Basis der XML Repräsentation allgemeingültig implementiert werden. Hierzu bietet sich XSLT [Elliotte Rusty Harold, 2005] an, welches eine Transformation von XML nach XML oder in beliebig andere Darstellungen ermöglicht.

Dadurch kann vor dem eigentlichen Vergleich zweier XML Bäume die gleiche Transformation auf beide Bäume angewandt werden, wodurch die problematischen Felder aus beiden entfallen und somit nicht in den Vergleich einbezogen werden.

Vergleichen externer Daten Neben der trivialen Situation einer zustandslosen Methode müssen bei Tests häufig auch noch weitere Umgebungszustände mit überprüft werden. Beim Testen der DAO Schicht muss beispielsweise auch die Datenbank mit in die Tests einbezogen werden. Da dies außerhalb der normalen Java Umgebung stattfindet, muss der Prototyp einen Mechanismus vorsehen um alternative Vergleiche einbinden zu können.

2.5 Auswertung der erzeugten Testqualität

Zur Auswertung der Testqualität werden Test Metriken [Andreas Spiller, 2005] herangezogen.

2.5.1 Metriken

Der Einsatz von Metriken ist aus zwei Gründen interessant. Durch Metriken können Methoden gefunden werden, für die im Normalfall Tests nur sehr schwierig erstellt werden können. Auf den dann aufgezeichneten Testes kann durch weitere Metriken bewertet werden wie gut diese zur Absicherung von Änderungen geeignet sind.

Bei normalen Projekten wird in den meisten Fällen das Auswählen der Methode durch äußere Umstände definiert. Soll zum Beispiel die Architektur einer Komponente geändert werden, dann ist es natürlich sinnvoll Tests für diese Komponente aufzuzeichnen. Im Rahmen dieser Diplomarbeit wird eine Bewertung des Recorded Test Musters erfolgen, weshalb gezielt nach schwer zu testenden Methoden gesucht wird. Methoden mit einem hohen McCabe Maß [McCabe, 1976] oder großer Anzahl an Parametern sind hier relevant.

Die Qualität der aufgezeichneten Tests wird über eine Testabdeckung qualifiziert. Daneben gibt es auch eine Menge an Testfällen die als grundlegend anzusehen sind und daher mindestens testbar sein müssen.

2.5.2 Grundlegende Testsituationen

Neben den Testfällen auf einem Projekt muss der Testrecorder und Testgenerator auch die folgenden Standard Situationen behandeln können. Genaue Details finden sich im Abschnitt 4.1.

- Datentypen und Arrays
  Java kennt verschiedene Datentypen welche auch als Felder vorkommen können.

- Sonderwerte
  Objekten kann der Wert null zugeweisen werden.
  Methoden können keine Parameter oder Rückgabewerte besitzen.

- Statische Methoden/Variablen
  Felder und Methoden können statisch sein.

- Wiederherstellen von Objektzuständen
  Bei der Testausführung muss das Testobjekt in den gleichen Zustand gebracht werden der bei der Aufzeichnung bestand.

- Nicht öffentliche Methoden
  Methoden können neben geschützt (protected) oder privat (private) sein.

- Fabriken, Singeltons
  Eingesetzte Muster nach [Gamma u. a., 1995].

- Ausnahmen
  Auch Fehlerfälle müssen testbar sein.

- Equals
  Vergleiche in Java erfolgen über eine spezielle equals Methode.

- Threads
  Ist die Aufzeichung möglich wenn mehr als ein Threads läuft.

- Zustandsbehaftete Objekte
  Einige Objekte haben zusätzlich einen Zustand auserhalb der Java Virtual Machine (JVM).

- dynamischer Proxy

3 Kapitel
Durchführung

Nach den im vorherigen Kapitel vorgestellten Grundlagen, wird in diesem Kapitel auf deren Umsetzung im Prototypen eingegangen. Erkenntnisse die aus dem Einsatz dieses Prototypen gewonnen wurden werden dann im nächsten Kapitel vorgestellt.

3.1 Anwendungsfälle

Der Prototyp umfasst die in Abbildung 3.1 dargestellten Anwendungsfälle. Diese werden im Normalfall sequentiell abgearbeitet, sind aber an sich voneinander unabhängig.

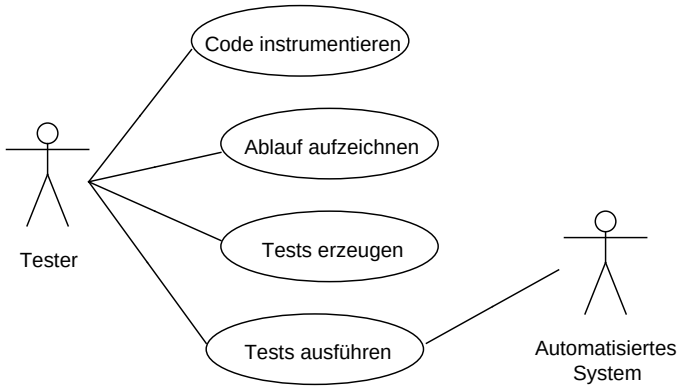

Abb. 3.1: Anwendungsfälle des Prototypen

Code instrumentieren Damit Testdaten für ein Projekt aufgezeichnet werden können muss dieses instrumentiert werden. Hierfür werden für konfigurierte Methoden die nötigen Dateien zur Instrumentation erzeugt.

Ablauf aufzeichnen Das Projekt kann nun instrumentiert und ausgeführt werden. Während der Bedienung zeichnet das Projekt nun Testdaten auf und legt diese im Repository ab.

Tests erzeugen Aus den im Repository vorhandenen Testdaten können nun die Tests zur Absicherung des Projektes generiert werden.

Tests ausführen Die erstellten Tests können zur Absicherung eingesetzt werden. Die Ausführung ist unabhängig von Eingaben und ist daher auch automatisiert als Regressionstest einsetzbar.

3.2 Projekt Struktur

Für die Programmiersprache Java hat sich der Einsatz des Projektmanagements und Build-Werkzeuges Maven [van Zyl u. a., 2008] angeboten. Maven übernimmt unter anderem das Verwalten der Abhängigkeiten. Der Prototyp liegt in zwei getrennten Maven-Projekten vor, welche in Abbildung 3.2 mit den wichtigsten enthaltenen Packages dargestellt sind.

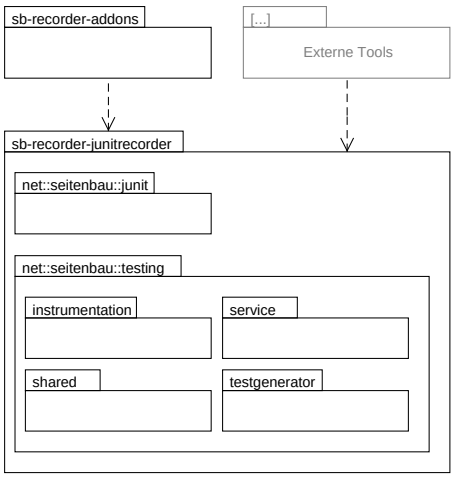

Abb. 3.2: Projektstruktur

Das *sb-recorder-junitrecorder* Projekt enthält den Kern des Frameworks. Im *sb-recorder-addons* Projekt finden sich Erweiterungen für das Framework. Das *addons* Projekt kann daher als erste Anlaufstelle beim Implementieren von eigenen Erweiterungen dienen. Dort ist beispielsweise das Testen von Datenbanken abgelegt. Der Platzhalter "externe Tools" steht für eigene Entwicklungen, wie Frontends, oder Erweiterungen zum Framework.

Externe Tools, der erwähnte Platzhalter für Erweiterungen und Werkzeuge.

sb-recorder-junitrecorder enthält die Kern-Implementierung des Frameworks und ist in folgende packages aufgeteilt:

- instrumentation
  Alle Klassen die für das Erzeugen der Instrumentation nötig sind.

- testgenerator
  Klassen die für das Erzeugen der JUnit-Tests benötigt werden.

- service
  Enthält zustandslose Zugangspunkte für eine Steuerung durch externe Tools.

- shared
  Klassen, zum Beispiel das Repositorymodell, welche an mehreren Stellen benötigt werden, aber nicht eindeutig einem Teil zuzuordnen sind.

- junit
  Die Implementierung des Parameterized Test für JUnit ist unabhängig vom eigentlichen Framework, weswegen dies in einem eigenen Package liegt.

sb-recorder-addons Dieses Projekt enthält verschiedene Erweiterungen, wie die schon erwähnte Aufzeichnung von Datenbanken.

3.3 Code Instrumentation

In einem ersten Schritt wird das bestehende Projekt für eine Aufzeichnung der Testdaten vorbereitet. Hierzu wird dieses mithilfe der in Kapitel 2.3.2 ausgewählten AOP-Technik instrumentiert.

Das Framework ist unabhängig von der eingesetzten Instrumentations-Technologie gehalten. Das Erzeugen der für die jeweilige Technik nötigen Artefakte (Dateien, Datenbanken etc.) erfolgt durch einen spezifischen Generator. Dieser ist austauschbar und wird durch ein Konfigurations-Modell gesteuert.

3.3.1 Anwendungsfälle

Das Erstellen der Instrumentation kann durch die in Abbildung 3.3 dargestellten Anwendungsfälle erreicht werden.

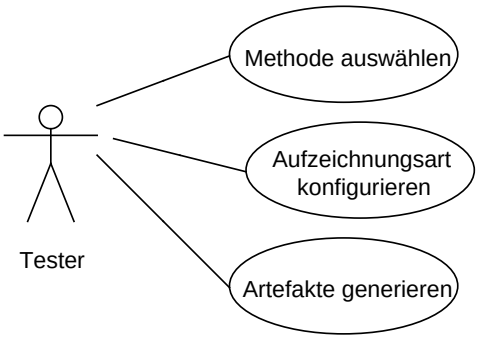

Abb. 3.3: Anwendungsfälle für die Instrumentation

Methode auswählen Vor der Generation der Artefakte ist es nötig die aufzuzeichnende Methode festzulegen. Dies ist allerdings meist bereits durch den Grund für die Aufzeichnung vorgegeben. So könnten Änderungen an einer Komponente anstehen, welche durch ein vorheriges Aufzeichnen der

Methoden abgesichert werden soll. Die Entscheidung welche konkreten Methoden aufgezeichnet werden sollen, ist daher abhängig vom Projekt und im Rahmen dieser Diplomarbeit nicht näher spezifizierbar. Am Ende muss eine Menge an Methodensignaturen feststehen für welche Testdaten aufgezeichnet werden sollen.

Aufzeichnungsart konfigurieren Nach der Auswahl der zu aufzuzeichnenden Methode wird bestimmt welche Daten festgehalten und in das Repository gespeichert werden.

Artefakte generieren Mithilfe der erstellten Konfiguration können nun die nötigen Artefakte für die Instrumentation erzeugt werden. Die Erzeugung erfolgt durch einen sogenannten Generator. Für den AOP basierten Ansatz ist ein Generator mit AspectJ implementiert.

3.3.2 Der Generator

Um das Framework unabhängig von einer konkreten Implementation zu halten, erfolgt die Generierung der jeweils nötigen Artefakte (meist werden dies Dateien sein) durch einen Generator. Das Klassendiagramm für Generatoren ist in Abbildung 3.4 dargestellt.

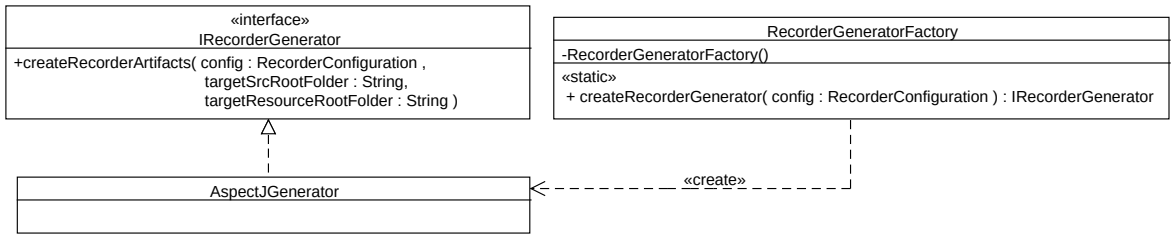

Abb. 3.4: Die IRecorderGenerator Schnittstelle

Als Basis dient das Interface `IRecorderGenerator`, welches von einem Generator implementieren werden muss. Ein solcher Generator erzeugt dann die zur Instrumentation nötigen Artefakte in der `createRecordArtifacts` Methode.

Generell werden in Java zwei Arten von Dateien unterschieden. Quellcode, wird vom Compiler übersetzt und daher in compilierter Form ausgeliefert. Dagegen liegen Ressourcen auch zur Laufzeit in gleicher Form vor. Dies ist der Grund warum `createRecordArtifacts` zwei Verzeichnisse als Parameter erwartet: Das Wurzelverzeichnis für Quellcodedateien (`targetSrcRootFolder`) und das Wurzelverzeichnis für Ressourcen (`targetRessourceRootFolder`).

Als erster Parameter wird die Konfiguration übergeben, welche die Zielmethode und Daten festlegt. Der konkrete Aufbau der Konfiguration wird im nächsten Abschnitt beschrieben.

Der bereitgestellte `AspectJGenerator` nutzt aus der Konfiguration die Informationen über die aufzuzeichnende Methode, um daraus den Aspekt spezifisch für diese Methode zu generieren. Alle weiteren Konfigurationsoptionen werden vom Generator nicht ausgewertet und als Konfigurationsdatei abgespeichert. Diese wird als Ressource abgelegt und kann so zur Laufzeit wieder eingelesen werden.

Damit externe Tools kein Wissen über jeden Generator benötigt wird, kann der Generator auch durch die `RecorderGeneratorFactory` instanziert werden. Die `RecorderGeneratorFactory` instanziert und initialisiert den in der Konfiguration angegebenen Generator.

3.3.3 Konfiguration des Recorder Generators

Das Konfigurationsmodell (Abbildung 3.5) dient zum Festlegen der zur Laufzeit aufzuzeichnenden Daten. Dies umfasst neben den Methoden auch eine Konfiguration welche Parameter und Rückgabewerte aufzuzeichnen sind.

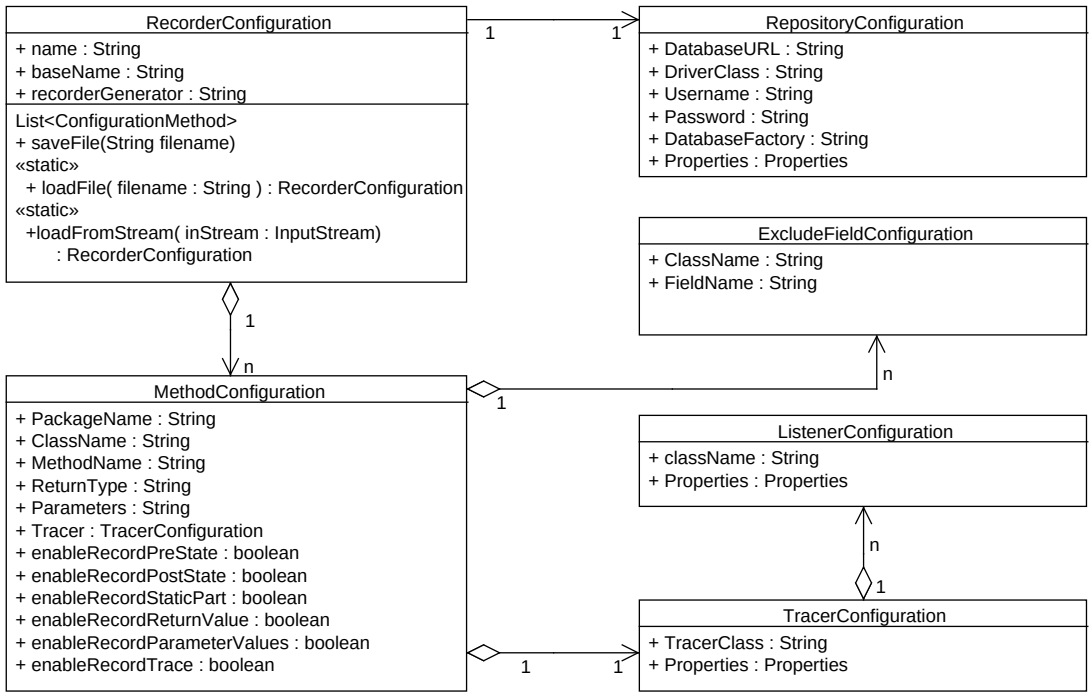

Abb. 3.5: Konfigurations Modell des RecorderGenerator's

RecorderConfiguration Wurzelelement, mit statische Methoden zum Laden und Speichern. Die erwähnte `RecorderGeneratorFactory` nutzt zum Erzeugen des Generators den FQN[1] aus dem `recorderGenerator` Feld. Das Feld `baseName` kann dazu genutzt werden ein Präfix vor die generierten Artefakte zu setzen. Dies ist nützlich um Kollisionen von Namen zu vermeiden.

RepositoryConfiguration Beschreibt wie das Repository angesprochen wird. Die Implementierung des Repositories kann durch die Angabe eines FQNs im Feld `DatabaseFactory` ausgetauscht werden. Wurde hier keine Klasse angegeben oder implementiert diese nicht die Schnitttelle `IDaoFactory`, wird die mitgelieferte Datenbank-Implementierung genutzt.

[1] **F**ully **Q**ualified **N**ame. Der komplette Namen (package und Klassenname) einer Klasse/Schnittstelle. Beispielsweise: com.example.recording.MainClass

MethodConfiguration Spezifiziert die Signatur der aufzuzeichnenden Methode. Die Methodensignatur ist in ihre Bestandteile zerlegt, wobei für die Parameter nur die Datentypen anzugeben sind, die Parameternamen sind nicht Teil der Signatur.

Über die Setter zu den enableRecord* Feldern kann hier nun auch festgelegt werden was für Grundlegende Daten in das Laufzeitmodell aufgezeichnet werden sollen.

- PreState : Aufzeichnung des Objektzustandes vor dem Aufruf
- PostState :Aufzeichnung des Objektzustandes nach dem Aufruf
- StaticPart : Aufzeichnen der Statischen Felder
- ReturnValue : Aufzeichnen des Rückgabewertes
- ParameterValues : Aufzeichnen der übergebenen Parameter
- Trace : Aufzeichnen von Unter- Methoden/Operatoren aufrufen

ExcludeFieldConfiguration Es kann sinnvoll sein, einzelne Felder bereits während der Aufzeichnung auszuschliessen. Dies kann durch das Nutzen der `ExcludeFieldConfiguration` erreicht werden.

TracerConfiguration / ListenerConfiguration Diese zwei Klassen dienen zur Konfiguration des eingesetzten Tracer bzw. daran angehängter Listener. Dieses Konzept wird in den nächsten Abschnitten näher erläutert. In der Konfiguration selbst wird lediglich für den konkreten Tracer bzw. Listener der Klassenname als FQN angegeben. Hierzu gibt es in der `TracerConfiguration` das Feld `TracerClass` und in der `ListenerConfiguration` das Feld `className`.

3.3.4 Erzeugter Aspekt

Der für den Prototypen erstellte AOP Generator speichert die Konfiguration als Ressource und erzeugt einen Aspekt pro konfigurierter Methode. Damit der erzeugte Aspekt nicht immer wieder die gleichen Codezeilen enthält und das Don't-Repeat-Yourself Prinzip (DRY) [Andrew Hunt, 2003] verletzt, wird der generierte Aspekt durch eine Vererbung aus klein gehalten. Das Klassendiagramm in Abbildung 3.6 zeigt diese Aspekthierarchie. Der generierte Aspekt wird dabei als `${generated}_Aspect` bezeichnet.

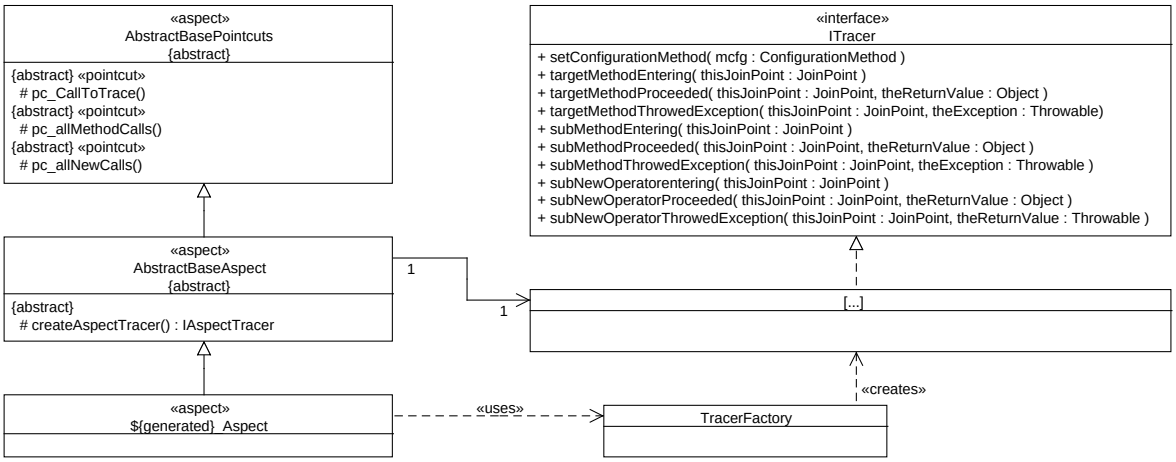

Abb. 3.6: Hierarchie der Aspekte

Der `AbstractBasePointcuts`-Aspekt enthält mehrere abstrakte Pointcuts. Mit dem pc_`CallToTrace` Pointcut wird die eigentliche Aufzeichnungsroutine aktiviert. Die anderen beiden Pointcuts dienen zum Aufzeichnen aller innerhalb dieser Methode stattfindenden Unteraufrufe. Daher werden diese beiden Pointcuts nur für das Aufzeichnen von Mock Objekten bei Komponententests benötigt.

Der eigentliche Aspekt für das Aufzeichnen ist in `AbstractBaseAspect` abgelegt. In Listing 3.1 ist ein Ausschnitt daraus abgedruckt.

```
1  /** Around the Method to record */
2  Object around() : pc_CallToTrace() {
3    getAspectTracer().targetMethodEntering(thisJoinPoint);
4
5    Object theReturnValue = null;
6    try {
7      theReturnValue = proceed();
8    } catch (Throwable t) {
9      getAspectTracer().targetMethodThrowedException(thisJoinPoint, t);
10     ReflectionHelper.throwException(t);
11   }
12
13   getAspectTracer()
14     .targetMethodProceeded(thisJoinPoint, theReturnValue);
15   return theReturnValue;
16 }
```

Listing 3.1: Auszug aus AbstractBaseAspect

Das eigentliche Aufzeichnen der Daten ist nicht im Aspekt abgelegt, sondern wird an einen Tracer delegiert. Ein Tracer implementiert die Schnittstelle ITracer welche verschiedene Methoden zur Benachrichtigung bereitstellt. Die Benachrichtigungen enthalten das Betreten (*`Entering`) und Verlassen (*`Proceeded`) einer Methode/Operation sowie eine möglicherweise geworfene Exceptions (*`ThrowedException`). Diese drei Arten können auf der aufzuzeichnenden Methode (`targetMethod`*) oder auf einem innerhalb dieser stattfindenden Untermethodenaufruf (`subMethodCall`*) bzw. Aufruf eines new-Operators (`subNewOperator`*) erfolgen.

Der Ausschnitt zeigt wie die Übergabeparameter in Zeile 3 erfasst werden. Geworfene Ausnahmen werden durch einen try-catch-Block abgefangen und in Zeile 9 an den Tracer übergeben. Damit die Ausnahmen dadurch nicht verschluckt werden, wirft Zeile 10 die gefangene Ausnahme erneut. Das Werfen der Ausnahme erfolgt nicht über eine `throw` Anweisung, sondern mithilfe von Reflections. Dies ist nötig, da eine `throw` Anweisung vom Compiler als checked Exception zum Übersetzungszeitpunkt geprüft wird. Infolgedessen müsste diese im Projekt gefangen werden, wodurch eine Änderung am Quellcode nötig wird. Dies kann durch den Einsatz von Reflections umgangen werden. Für alle aufzuzeichnenden Methoden kann somit ein Aspekt genutzt werden.

Wurde keine Ausnahme geworfen, wird in Zeile 13,14 der Rückgabewert gesichert und mit Zeile 15 die Methode verlassen. Wird der Aspekt auf einer `void` Methode angewandt, ignoriert AspectJ die return Anweisung entsprechend.

Der Tracer wird im Aspekt mithilfe einer Fabrikmethode [Gamma u. a., 1995] `createAspectTracer` instanziert. Der generierte Aspekt muss daher die abstrakten Pointcuts und die `createAspectTracer` Methode überschreiben. Zum Instanzieren des Tracers wird auf die `TracerFactory` zurückgegriffen welche aus dem Konfigurationsmodell, das ja zur Laufzeit aus der Ressource geladen wurde, alle nötigen Informationen herausliest.

3.3.4.1 Tracer

Die konkreten Implementierungen eines Tracers können von einer der in Abbildung 3.7 dargestellten Klassen abgeleitet werden.

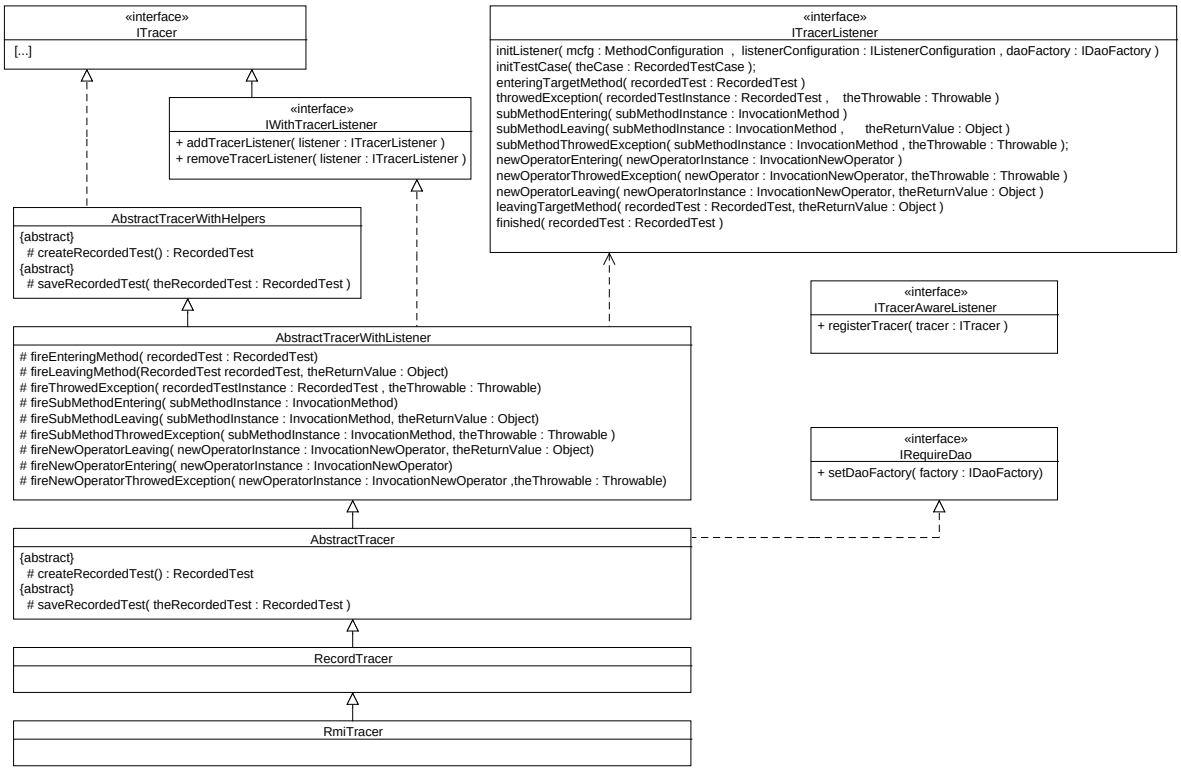

Abb. 3.7: Tracer des Recorders

Zur Laufzeit übernimmt je Aspekt ein Tracer das Sichern der Testdaten. Damit diese Delegation auch durch eigene Funktionalität erweitert werden kann wurde die Möglichkeit geschaffen, dass sich Listener an einen Tracer anmelden können.

AbstractTracerWithHelper Stellt eine Standardimplementierung für die `ITracer` Schnittstelle bereit. Diese Klasse baut aus den Aufrufen des Aspektes ein Laufzeitmodell auf. Dieses Modell enthält die in das Repository zu speichernden Daten. Detailiert wird auf das Laufzeitmodell in Abschnitt 3.4.5 eingegangen.

AbstractTracerWithListener Stellt eine Implementierung für Listener, welche die Schnitttelle `ITracerListener` implementieren, bereit. Neben der Verwaltung der angemeldeten ITracerListener, werden für abgeleitete Klassen verschiedene `fire*` Methoden zur Benachrichtigung der angemeldeten Listener bereitgestellt.

ITracerListener Ist das Interface, welches alle angemeldeten Listener implementieren müssen. Angemeldete Listener bekommen durch diese Schnittstelle die gleichen Ereignisse mitgeteilt wie der Tracer. Einziger Unterschied ist die Unabhängigkeit von AspectJ. Anstelle des AspectJ spezifischen `thisJoinPoint` Parameter wird den Listenern das Laufzeitmodell übergeben. Für TracerListener die einen direkten Zugang zum Tracer benötigen, gibt es eine zusätzliche Schnittstelle `ITracerAwareListener`, durch welche eine Injizierung des aktiven Tracers möglich ist.

RecordTracer Für den Prototypen stellt der `RecordTracer` die Standardimplementierung bereit. Der `RecordTracer` baut in Abhängigkeit von der Konfiguration das Laufzeitmodell auf und schreibt dieses in das Repository.

RmiTracer Dieser Tracer stellt einige experimentelle Features bereit, beispielsweise die Möglichkeit eine RMI-Verbindung zur Fernwartung aufzubauen.

IRequireDao Dient als Marker Interface für das injizieren der Repository Verbindung. Das Repository wird durch die `IDaoFactory` Schnittstelle bereitgestellt.

3.3.4.2 TracerFactory

Da hinter der ITracer Schnittstelle mehrere unterschiedliche Klassen mit verschiedensten Abhängigkeiten liegen, ist das Instanzieren aufwendig. Es gibt deshalb mit der `TracerFactory` (Abbildung 3.8) die Möglichkeit das Erzeugen des Tracers zu vereinfachen.

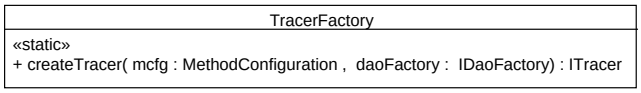

Abb. 3.8: TracerFactory

Basierend auf der `MethodConfiguration` instanziert die `TracerFactory` den konkreten Tracer und die Tracer-Listener, wobei letztere gleich am Tracer angemeldet werden. Ebenfalls wird die übergebene `IDaoFactory` in Tracer injiziert welche das Marker Interface `IRequireDao` implementieren.

3.3.5 Instrumentation Service

Für externe Tools gibt es über die in Abbildung 3.9 dargestellte Klasse einen zustandslosen Zugangspunkt zum Recordergenerator.

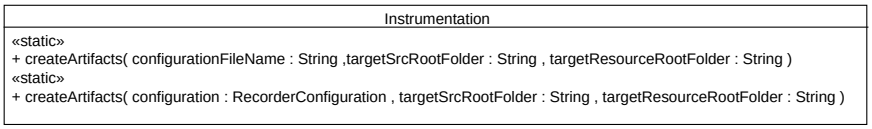

Abb. 3.9: Service zur Instrumentation

Die Klasse stellt zwei `createArtifacts` Methoden bereit, welche das Erzeugen und Ausführen eines Generators abstrahieren. Einmal kann das Konfigurations Modell als Instanz übergeben werden, während in der zweiten Ausprägung der Pfad zu einem gespeicherten Modell erwartet wird.

Beide Methoden erzeugen dann mithilfe der Konfiguration den konkreten Generator und starten diesen mit den übergebenen Verzeichnissen für Quellcode und Ressourcen. Ein externes Tool braucht für die Generation daher kein Spezialwissen über den Generator.

3.3.6 Beispiel Recorder-Generator

Im Folgenden soll ein Auszug aufgezeigt werden, wie das beschriebene Konfigurations Modell zum Erzeugen der Artefakte genutzt wird.

Als erstes wird das Repository für die aufzuzeichnenden Daten konfiguriert:

```
1  RepositoryConfig repository = new RepositoryConfig();
2
3  repository.setDriverClass ("com.mysql.jdbc.Driver");
4  repository.setDatabaseURL ("jdbc:mysql://localhost/sbtestrecorder");
5  repository.setUsername    ("root");
6  repository.setPassword    ("");
```

Das Repository ist in diesem Fall ein lokaler MySQL Server. Solange die Repository Implementierung nicht ausgetauscht wird, können hier alle von Hibernate unterstützten Datenbanken genutzt werden.

Nach dem Repository wird die aufzuzeichnende Methode konfiguriert:

```
7       MethodConfiguration methodConfiguration = new MethodConfiguration(
8               "org.example",
9               "SomeClass",
10              "int",
11              "getMax",
12              "int,int"
13          );
```

Der automatisch genutzte `RecordTracer` gibt zur Laufzeit keine Statusinformationen aus. Dies kann aber über das Anmelden des `ConsoleListener` an den Tracer nachgeholt werden:

```
14      methodConfiguration.getTracer().addListener( ConsoleListener.class );
```

Als nächstes wird das konfigurierte Repository sowie die Methoden-Konfiguration in das `RecorderConfiguration` Modell eingehängt. Die Recorder-Konfiguration unterstützt ein Repository, aber mehrere Methoden-Konfiguration:

```
15      RecorderConfiguration configuration = new RecorderConfiguration();
16      configuration.setRepository(repository);
17      configuration.addMethodConfiguration(methodConfiguration);
```

Durch den Instrumentation-Service können nun die zur Instrumentation benötigten Artefakte erzeugt werden:

```
18      Instrumentation.createArtifacts(
19              configuration,
20              "src/main/java",
21              "src/main/resources"
22          );
```

Dies sind konkret der Aspket “src/main/java/org/example/SomeClassAspect.aj” und die oben zusammengebaute Konfiguration als Ressource “src/main/resources/org/example/SomeClass.record.xml”.

3.4 Instrumentieren und Ausführen

Nachdem die zur Instrumentation benötigten Artefakte generiert sind, können diese nun zur Instrumentation genutzt werden. Das Vorgehen zum Instrumentieren und das Verhalten während der Aufzeichnung soll in diesem Abschnitt betrachtet werden.

3.4.1 Anwendungsfälle

Das Vorgehen ist dabei teilweise stark von der Art des Projektes abhängig. Bei einer Swing Anwendung reicht ein Compilieren und Ausführen aus. Anwendungen die in einem Container laufen, müssen gepackt und deployed werden. Abbildung 3.10 zeigt die vier Anwendungsfälle welche die meisten Situationen abdecken.

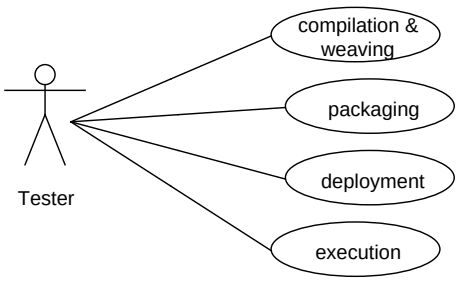

Abb. 3.10: Anwendungsfälle zur Ausfühung des instrumentierten Projektes

compilation & weaving AspectJ kennt mehrere Wege um einen Aspekt zur Laufzeit auszuführen. Im Rahmen dieser Diplomarbeit wurden die Aspekte durch den AspectJ Compiler direkt in das Projekt compiliert.

packaging Beim packaging werden alle compilierten Dateien und Ressourcen in ein Archiv verpackt. Für den Recorder ist hier zusätzlich zu den bestehenden Ressourcen auch ein Einbetten der Konfigurationsdatei als Ressource nötig.

deployment Das gepackte Projekt wird beispielsweise auf einen Webserver deployed. Bei nicht lokalen Installationen ist sicherzustellen, dass der Zugriff auf das Repository möglich ist.

execution Die Anwendung wird ausgeführt während der Aspekt im Hintergrund Testdaten in das Repository schreibt.

3.4.2 Compilation & weaving

Das Compilieren der Aspekte wurde mit Eclipse und Maven getestet.

Eclipse

In Eclipse reicht die Installation der AspectJ Development Tools (AJDT) für eine AspectJ Unterstützung aus. Nach der Installation und dem Aktivieren der AspectJ-Unterstützung werden die Aspekte automatisch im Hintergrund compiliert. Bei Webanwendung ändert sich dadurch nichts am Deployment, da die Klassen bereits automatisch instrumentiert wurden.

Maven

Ein Compilieren kann durch das AspectJ-Maven-Plugin erfolgen. Dieses muss in der `pom.xml` im Plugin Abschnitt aktiviert werden (Siehe Listing 3.2). Dadurch wird an Stelle des normalen Java Compilers

der AspectJ Compiler genutzt. Da Teile des Prototypen auf Java Generics zurückgreifen ist es zudem nötig, die Compiler Kompatibilität auf Version 1.5 zu erhöhen.

```
<plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>aspectj-maven-plugin</artifactId>
        <configuration>
                <source>1.5</source>
                <complianceLevel>1.5</complianceLevel>
        </configuration>
        <executions> <execution>
                <goals>
                        <goal>compile</goal>
                </goals>
        </execution> </executions>
</plugin>
```

Listing 3.2: Pom.xml Sektion für aspectj-maven-plugin

Alternativ können die Aspekte auch von einer eigenen JAR-Datei aus eincompiliert werden. Hier kann beispielsweise eine beliebige Maven Dependency angegeben werden. Listing 3.3 zeigt die hierfür nötigen Programmzeilen.

```
  <configuration>
    <weaveDependencies>
        <weaveDependency>
            <groupId>com.example.recorder</groupId>
            <artifactId>recorder-inst</artifactId>
        </weaveDependency>
     </weaveDependencies>
     ...
```

Listing 3.3: Aspekte in den Bytecode integrieren

Die Möglichkeit den Aspekt in den Bytecode zu compilieren ermöglicht es die Aspekte getrennt von dem zu instrumentierenden Projekt zu halten. So kann ein zweites Maven Projekt dazu genutzt werden die Aspekte und Ressourcen zu halten. Dieses Projekt kann in das lokale Repository installiert werden und so als Abhängigkeit in das zu instrumentierende Projekt eingebunden werden. Die Instrumentation des Projekts erfolgt so fast komplett transparent. Als einziges ist hier dann die Maven Konfiguration zu ändern, welche aber durch den Profilen sehr gut kontrollierbar ist.

3.4.3 Execution

Wird nun das durch die Aspekte instrumentierte Projekt ausgeführt, zeichnet dieses Testdaten auf. Durch das Eincompilieren der Aspekte ändert sich am Startvorgang nichts. Das instrumentierte Projekt baut während der Ausführung ein Laufzeitmodell (Abbildung 3.11) des Programmablaufes auf.

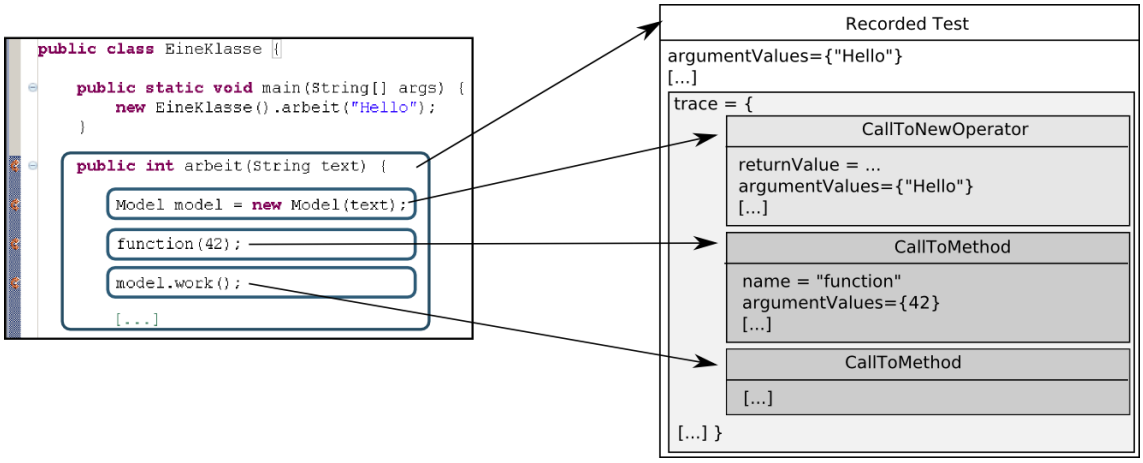

Abb. 3.11: Aufbau des Laufzeitmodelles

Im Beispiel wird eine Methode `arbeit` aufgezeichnet. Je Aufruf der Methode wird ein RecordedTest Objekt angelegt, welches die Übergabeparameter und Rückgabewerte enthält. Werden während der Ausführung der Zielmethode weitere Untermethoden/Operatoren aufgerufen, werden diese in der Liste `trace` gespeichert. Das Aufzeichnen des Traces ist über die Konfiguration steuerbar, bietet sich für Komponententests an, um daraus später entsprechende Mock-Objekte zu erstellen.

3.4.4 ObjectSnapshot

Während der Aufzeichnung ist es nötig den Zustand von Objekten, beispielsweise Parametern, zu sichern. Zur Serialisierung selbst wird das in Kapitel 2.3.3 gewählte XStream eingesetzt. Da XStream allerdings nur die nicht statischen Teile speichert, wurde für die Sicherung des Zustandes die Klasse ObjectSnapshot definiert, welche auch den statischen Zustand speichern kann.

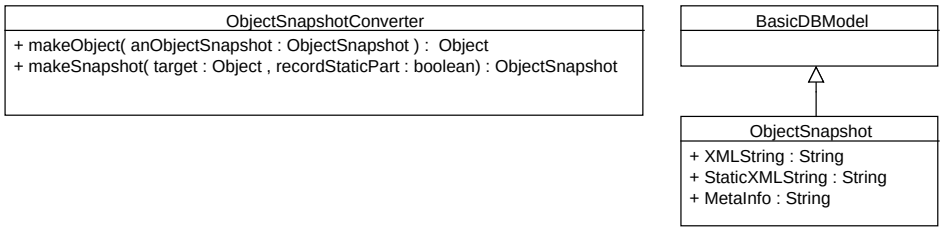

Abb. 3.12: Klassen zur ObjectSnapshot Verwaltung

Abbildung 3.12 zeigt die eingesetzten Klassen. Die `BasicDBModel` Klasse dient lediglich als Basisklasse, da die `ObjectSnapshot` Klasse in die Datenbank abgelegt werden muss. Die `ObjectSnapshotConverter` Klasse stellt Funktionen zur Transformation eines Objektes, nach bzw. wieder zurück, aus einer `ObjectSnapshot` Instanz bereit.

Die Klasse `ObjectSnapshot` enthält nur die rohen XML Daten. Dadurch ist es einfach möglich, diese in die Datenbank abzubilden. Da der `ObjectSnapshotConverter` als Instanz gehalten werden kann, ist es möglich diesen an einer zentralen Stelle zu initialisieren und beispielsweise Filterungen zu setzen.

3.4.5 Das Laufzeitmodell

Die Abbildung der aufgezeichneten Daten in das Repository, also in die Datenbank, wurde mithilfe von Hibernate [King u. a., 2008] als OR-Mapper umgesetzt. Daher wird das Modell der Repository hier nur in Form eines Java Modelles (Abbildung 3.13) beschrieben und nicht als Datenbanktabellen.

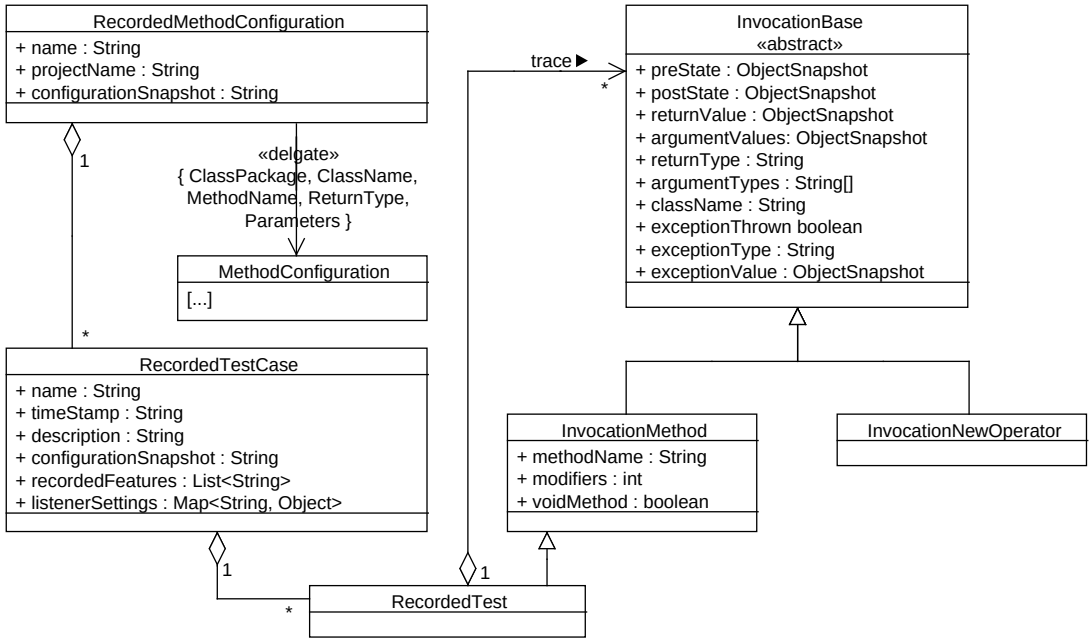

Abb. 3.13: Datenmodell der Aufzeichnung

RecordMethodConfiguration ist das Wurzelelement, welches gegenüber der `MethodConfiguration` um zusätzliche Informationen, wie dem Projektnamen, erweitert ist. Da die restlichen Daten schon für die Methoden-Konfiguration während der Generation angefallen sind, werden diese über eine Delegation an eine `MethodConfiguration` Instanz gehalten.

RecordedTestCase dient zum Gruppieren von Testläufen. Während sich die `RecordMethodConfiguration` bei einem Neustart der Anwendung nicht ändert, wird ein neuer RecordedTestCase bei jedem Neustart der Anwendung angelegt. Vorstellbar wäre es auch, während der Ausführung eine neue Instanz anzulegen. Dadurch wird es möglich, noch während der Aufzeichnung Testfälle zu gruppieren, beispielsweise nach getesteter Fehlerklasse.

RecordedTest entspricht einem Aufruf der aufzuzeichnenden Methode. Es werden die ganzen Statusdaten aus `InvocationMethod` geerbt. Einzig eine trace-Liste ist hier zusätzlich vorhanden, in welcher weitere Untermethoden / Operatorenaufrufe abgelegt werden können.

InvocationBase ist die Basisklasse für einen Aufruf einer Methode oder eines Operators. Neben dem Status des Objektes vor und nach dem Aufruf, werden auch Meta Informationen, wie der FQN der Klasse gespeichert.

InvocationMethod erweitert die `InvocationBase` Klasse um Felder, die zusätzlich beim Aufruf von Methoden anfallen. Dies sind der Methodenname, Modifizierer (`static`) und Rückgabetyp.

InvocationNewOperator dient zur Markierung, dass ein new-Operator aufgerufen wurde. Dies dient der Testgenerierung zum Generieren von unterschiedlichen Tests.

3.4.6 Interaktion zwischen Tracer und Listener

Der normale `RecordTracer` baut ein Laufzeitmodell auf, welches an die angemeldeten Listener pro Methodenaufruf weitergegeben wird. Abbildung 3.14 zeigt die Interaktionen zwischen Aspekt, Tracer und Listener beim Aufruf einer Methode.

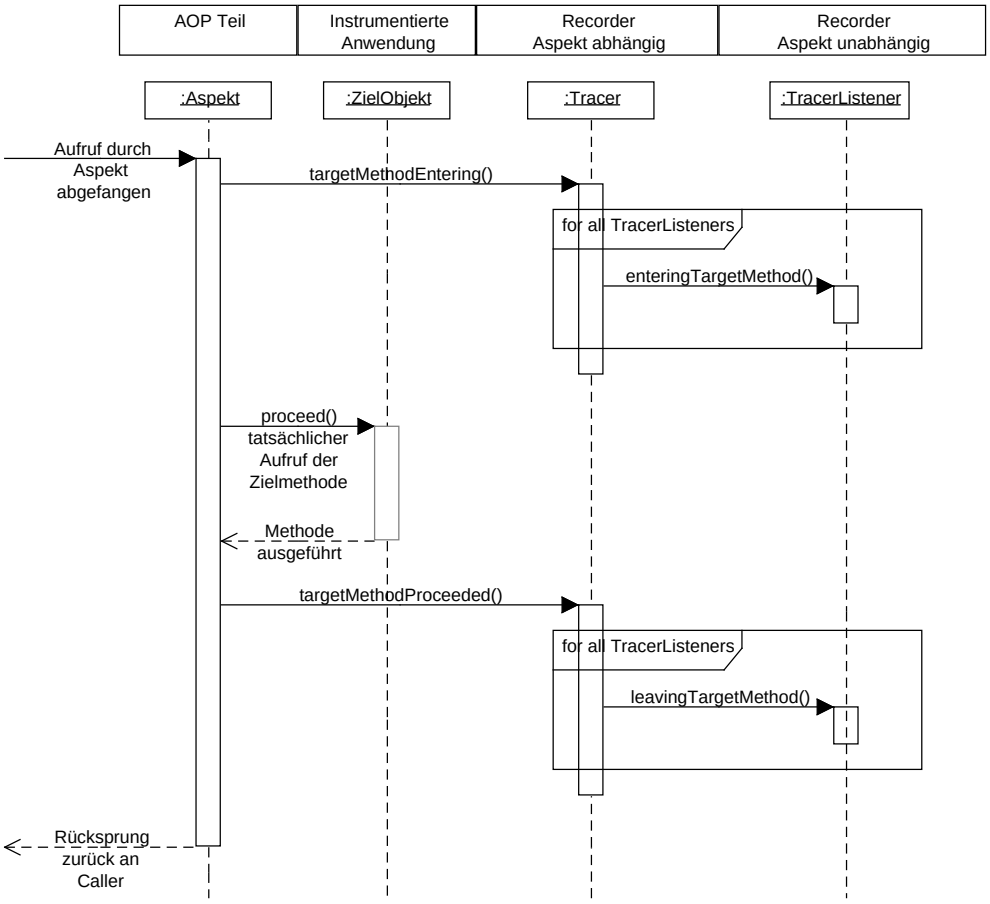

Abb. 3.14: Ablaufdiagramm Tracer und Listener von instrumentiertem Code

Dadurch, dass der Aspekt direkt in den Bytecode eincompiliert ist, existiert auf der Bytecodeebene keine wirkliche Trennung zwischen Aspekt und eigentlicher Anwendung. Für das Sequenzdiagramm und während der Programmierung kann der Aspekt aber als eigenständige Einheit betrachtet werden.

Beim Aufruf der aufzuzeichnenden Methode wird der Methodenaufruf durch den Aspekt abgefangen und weiter an den Tracer delegiert. Der Tracer baut nun das Laufzeitmodell auf und benachrichtigt

die angemeldeten Listener. Diese können nun zusätzliche Daten erheben, beispielsweise eine Sicherung der Datenbank. Nachdem alle Listener über den bevorstehenden Methoden-Aufruf informiert wurden, wird dieser ausgeführt.

Im Sequenzdiagramm wurden mögliche Untermethodenaufrufe ignoriert und angenommen, dass die Methode sofort beendet wird. Wäre eine Ausnahme geworfen, würde der Aspekt den Tracer stattdessen über die Ausnahme informieren. Beim gezeigten Fall baut der Tracer den Rückgabewert in das Laufzeitmodell ein und informiert die angemeldeten Listener.

3.4.7 Repository Service

Die nun aufgezeichneten Daten liegen im Repository vor. Damit externe Tools auf das Repository neutral von der eigentlichen Implementierung zugreifen können, gibt es die in Abbildung 3.15 dargestellte zustandslose Klasse `RepositoryService`.

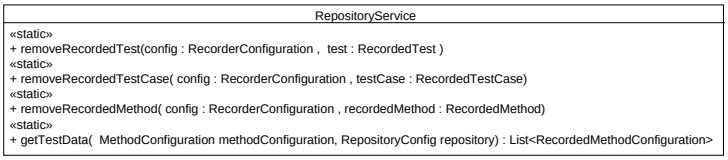

Abb. 3.15: Repository Service

Die Methoden der Klasse erwarten lediglich die Übergaben der Konfiguration. Aus dieser wird dann dynamisch die Konfiguration für das Repository gelesen. In welcher Form das Repository tatsächlich implementiert wurde, muss der Anwender der Schnittstelle nicht wissen. Eine korrekte RecorderConfiguration bzw. die darin enthaltenen RepositoryConfiguration reichen aus.

3.5 Test Generierung

Nachdem nun Testdaten aufgezeichnet sind, können Tests generiert werden.

3.5.1 Anwendungsfälle

Das Generieren der Tests lässt sich in die in Abbildung 3.16 dargestellten Anwendungsfälle aufteilen.

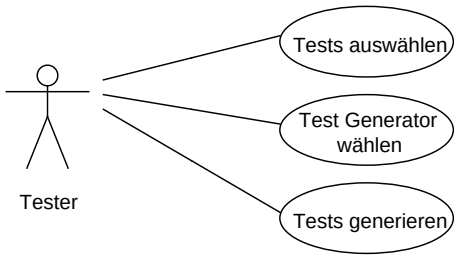

Abb. 3.16: Anwendungsfälle der Testerstellung

Test-Daten auswählen Während der Aufzeichnung werden mehr Testdaten aufgezeichnet als eigentlich benötigt werden. Daher müssen in einem ersten Schritt die gewünschten Daten ausgewählt werden.

Generator wählen Abhängig von den zur Verfügung stehenden Testdaten können unterschiedliche Arten von Tests generiert werden. Die möglichen Arten werden durch jeweilige Generatoren bereitgestellt.

Tests generieren In diesem letzten Schritt wird der konfigurierte Test erstellt.

3.5.2 Implementierter Parameterized Test Runner

Wie schon festgestellt wurde, ist es sinnvoll eine eigene Implementierung des Parameteriezd Test Musters bereitzustellen. Dadurch können die Nachteile der JUnit Implementation ausgeglichen werden:

- Anzahl Datenquellen
  Jede JUnit Testklasse kann nur eine Liste mit Testdaten enthalten.

- Dateibasierte Datenquelle
  Bei JUnit werden die Testdaten direkt in der Testklasse abgelegt. Die XML Repräsentationen können bei größeren Objekten schnell anwachsen und so die Lesbarkeit stark verschlechtern.

Die Klassen der eigenen Implementierung des Parameterized Test Musters sind in Abbildung 3.17 dargestellt.

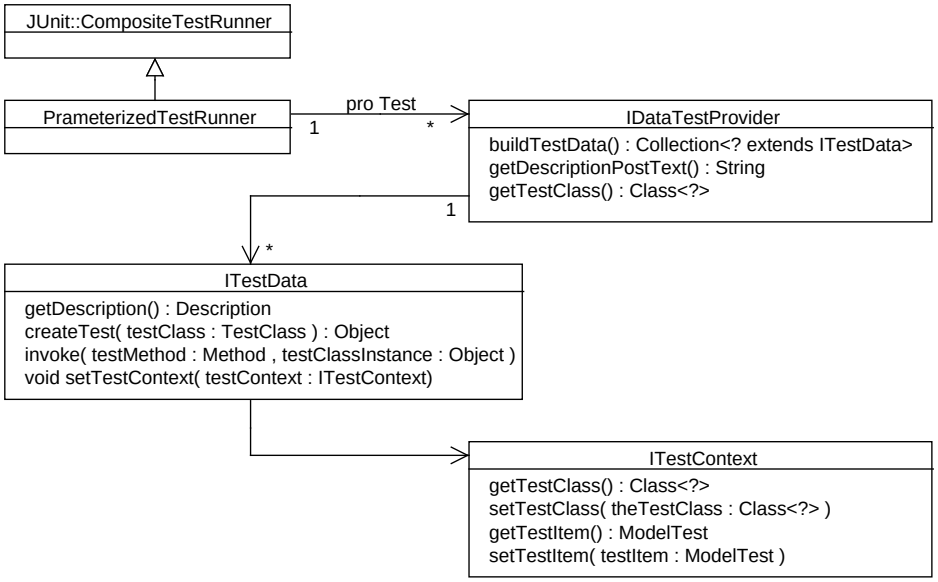

Abb. 3.17: Parameterized TestRunner Implementation

Der `ParameterizedTestRunner` wird von einem JUnit `CompositeTestRunner` abgeleitet, wodurch dieser sich nach außen wie ein normaler Test Runner verhält und sich so mit JUnit-Werkzeugen integriert.

JUnit übergibt die Kontrolle über das Ausführen eines Tests an einen Testrunner, wenn dieser durch die Annotation `@RunWith` angegeben wird. Für den `ParameterizedTestRunner` würde die Annotation an der Testklasse wie folgt aussehen:

```
1 @RunWith(ParameterizedTestRunner.class)
2 public class ATestClass { ...
```

Ähnlich dem JUnit-Vorgehen wird eine Methode durch eine Annotation als Test definiert. Die Annotation nennt sich `ParameterizedTestMethod` und kennt als weiteren Parameter einen Datenprovider, welcher dafür zuständig ist, die eigentlichen Testdaten bereitzustellen. Hierzu muss der Datenprovider das Interface `IDataTestProvider` implementieren. Im folgenden Auszug wird eine XML-basierte Implementierung gewählt. Der gewählte XmlDataProvider lädt mithilfe von XStream die Test Objekte aus einer XML Datei.

```
3 @ParametizedTestMethod(XmlDataProvider.class)
4 @XmlDataOption(file = "justATest_positiveNumbers.xml")
5 public void justATest( boolean returnValue, int number) {
6     // ...
7 }
```

Ein Datenprovider stellt dem `ParameterizedTestRunner` die Testdaten als eine Menge von `ITestData` Elementen zur Verfügung. Jede dieser Daten wird zum gegebenen Zeitpunkt mit dem Aufrufen der Testmethode beauftragt. Dadurch können unterschiedliche Provider die Daten in anderer Form an den Test übergeben.

Mithilfe der `@ParameterizedTestField` Annotation kann eine Variable in der Testklasse annotiert werden, in welche ein `ITestContext` injiziert wird. Dieser stellt einen Wrapper um eine ITestData-Instanz dar. Über dieses Feld kann aus dem Test direkt auf die Testdaten zugegriffen werden.

```
8 @DataDrivenTestField
9 protected ITestContext myContext;
```

3.5.3 TestGenerator

Die Generierung der Tests folgt einem ähnlichen Aufbau wie das Generieren der Instrumentation. Auch hier wird ein Generator für das eigentliche Erzeugen der Tests eingesetzt. Abbildung 3.18 zeigt das Klassendiagramm der `ITestGenerator` Schnittstelle.

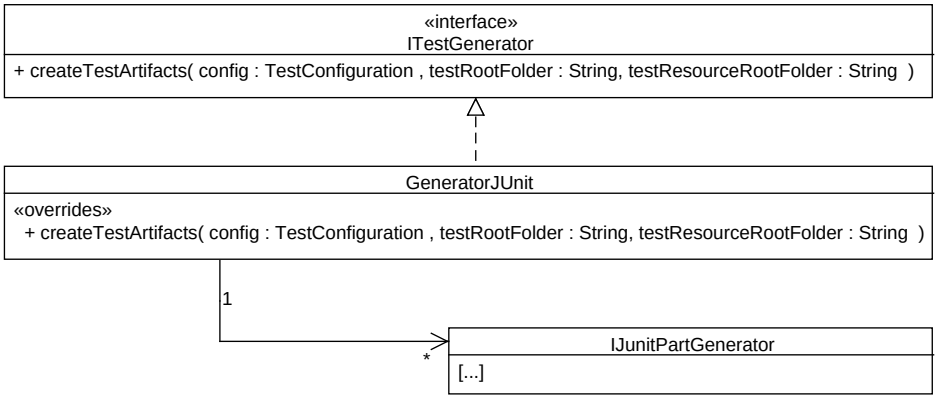

Abb. 3.18: Test Generator Schnittstelle

Der Hauptunterschied gegenüber der Instrumentation ist hier, dass der JUnit Generator nur das Grundgerüst des Tests erzeugt. Der eigentliche Inhalt kann flexibel durch sogenannte Part-Generatoren erweitert werden.

Der Generator wird auch hier mit einem eigenen Konfigurations-Modell gesteuert, welches der Methode als ersten Parameter übergeben wird. Die zwei weiteren Parameter geben wieder ein Sourcecode- und Ressourcenverzeichnis an.

GeneratorJUnit

Der JUnit Generator erzeugt die folgenden Daten:

- JUnit-Test
  Die eigentliche JUnit-Test Klasse.

- Daten
  Damit der Test und alle Daten ohne Probleme in ein Source-Verwaltungssystem eingecheckt werden können, werden die Testdaten aus der Datenbank herausgelöst und in Dateien gespeichert.

3.5.4 Part-Generatoren

Mithilfe der in Abbildung 3.19 dargestellten Part-Generatoren können die vom GeneratorJUnit erzeugten JUnit-Tests um Inhalt erweitert werden. Hierfür ruft der GeneratorJUnit während der Generatrion des Tests angemeldete Part-Generatoren über die IJunitPartGenerator Schnittstelle auf.

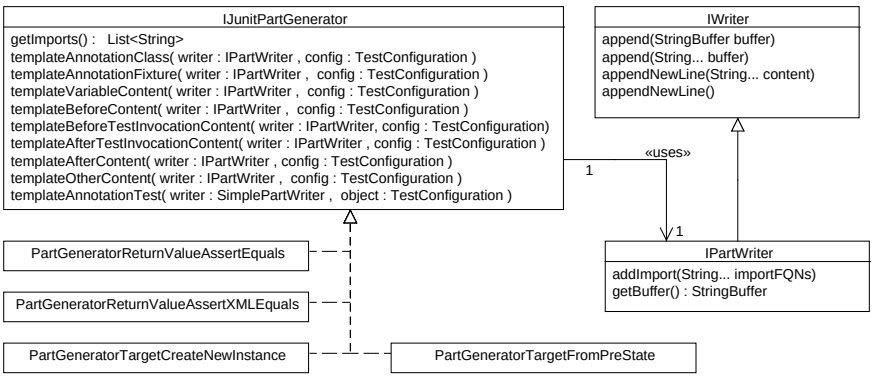

Abb. 3.19: Part-Generatoren Hierarchie

IWriter/IPartWriter Stellt einen Wrapper zur Abstraktion des Ausgabezieles dar. In den Methoden des `IJunitPartGenerator` wird diese Schnittstelle dazu genutzt um eine Unabhängigkeit vom Ausgabemedium (Hier eine Testklasse) zu erhalten.

IJunitPartGenerator Während der `GeneratorJUnit` den Test erzeugt, kann an definierten Stellen zusätzlicher Quellcode in den Test eingebracht werden. Beispielsweise kann durch die `templateBeforeContent` Methode Testcode in die `@Before` Methode injiziert werden. Beispielsweise könnte so eine Umgebungsvariable bei jedem Testlauf initialisiert werden. Details zu den verfügbaren Injektions-Punkten finden sich im JavaDoc, bzw. lassen sich aus den Methodennamen ablesen.

PartGeneratorReturnValueAssertEquals Fügt einen Vergleich des Returnwertes in den Test auf Basis der JUnit `AssertEquals` Methode ein.

PartGeneratorReturnValueAssertXMLEquals Vergleicht ebenfalls den Returnwert, allerdings nicht über `AssertEquals` sondern über einen XML Vergleich. Hierzu werden auch die in der Test-Konfiguration (Siehe nächster Abschnitt) festlegbaren Filter berücksichtigt und beispielsweise eine XSLT-Filterung vorzunehmen.

PartGeneratorTargetCreateNewInstance Ein normal generierter JUnit Test erzeugt keine Instanz der Klasse der zu testenden Methode. Dies wird über diesen Part-Generator eingefügt, welcher durch einen new-Operator die Instanz erzeugt.

PartGeneratorTargetFromPreState Durch diesen Part-Generator kann der Zustand aus dem während der Aufzeichnung gespeicherten PreState wiederhergestellt werden.

3.5.5 Modell TestConfiguration

Die Test-Konfiguration (Abbildung 3.20) steuert das Erstellen der Tests und hält die Testdaten als Liste von `RecordedMethodConfiguration` Instanzen vor. Diese können so direkt aus der Datenbank entnommen werden. Zusätzlich kann mit dem Feld `baseName` ein Prefix vor die generierte Dateien gesetzt werden, damit keine Kollisionen in den Namen entstehen. Als weiteres Feld können noch zusätzlich Filter für XML Vergleiche konfiguriert werden.

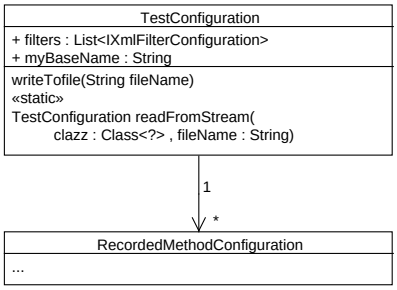

Abb. 3.20: Test Konfigurations Modell

3.5.6 Test Generator Service

Auch für den Test Generator gibt es einen zustandslosen Service (Abbildung 3.21).

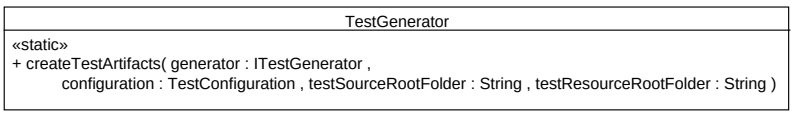

Abb. 3.21: Test Generator Service

Die Klasse TestGenerator stellt mit der `createTestArtifacts` Methode einen neutralen Zugangspunkt bereit.

3.5.7 Beispiel zur Test Generation

Im Folgenden wird ein Beispiel für das Konfigurieren und Generieren von Tests aufgezeigt.

Als erstes muss, wie bei der Instrumentation, ein Repository und eine Methoden Konfiguration erzeugt werden:

```
1  RepositoryConfig repository = ...; // Repository festlegen
2  MethodConfiguration methodConfig = new MethodConfiguration(
3          "int","org.example","SomeClass","getMax","int, int" );
```

Durch diese Beiden kann über den RepositoryService eine Liste aller für diese Methode aufgezeichneten Daten abgefragt werden.

```
4 | List daten = RepositoryService.getTestData( methodConfig, repository );
```

Die erhaltenen Testdaten werden dann der Test Konfiguration zugewiesen:

```
6 |     TestConfiguration configuration = new TestConfiguration();
7 |     configuration.setTestData(daten);
```

Zur Generation der JUnit-Tests wird nun der `GeneratorJUnit` benötigt:

```
8 |     GeneratorJUnit generator = new GeneratorJUnit();
```

Der GeneratorJUnit erzeugt nur das JUnit-Parameterized-Test Grundgerüst. Überprüfungen müssen mithilfe von Part-Generatoren in den Test hinein generiert werden:

```
9  | generator.addPartGenerator(new PartGeneratorTargetCreateNewInstance());
10 | generator.addPartGenerator(new PartGeneratorReturnValueAssertEquals());
```

Nachdem nun Generator und Test Konfiguration aufgesetzt wurden, können nun die Artefakte über den TestGenerator Service erzeugt werden.

```
11 |     TestGenerator.createTestArtifacts(generator, configuration,
12 |             "src/test/java", "src/test/resources");
```

Der Generator erzeugt nun eine JUnit-Testklasse “src/test/java/org/example/SomeClassTest.java” und eine zugehörige XML-Testdaten Datei “src/test/resources/org/example/SomeClassTest.xml” für den Parameterized Test. Abhängig von den konfigurierten Part-Generatoren können noch weitere Ressourcen erzeugt werden. Abbildung 3.22 zeigt eine Ressourcenstruktur mit zusätzlichen Dateien verschiedener Part-Generatoren.

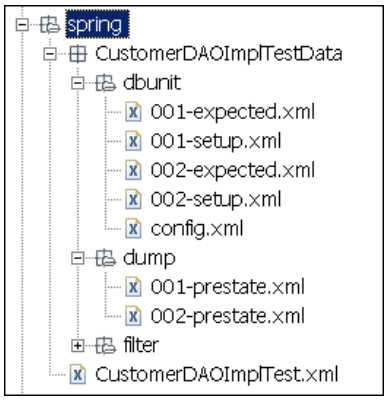

Abb. 3.22: Ressourcen Struktur eines generierten Testes

3.6 Ausführen der Tests

Die generierten Tests können normal mit JUnit ausgeführt werden. Da alle Testdaten als Ressourcen abgespeichert sind, können die Tests nun unter eine Quellcodeverwaltung gestellt werden oder auch automatisiert auf einem dedizierten Testrechner ausgeführt werden.

4 Kapitel
Ergebnisse

Dieses Kapitel beschreibt Erkenntnisse, welche durch den Praxiseinsatz des Recorders gewonnen wurden.

4.1 Grundfunktionalität

Im Abschnitt 2.5.2 wurden verschiedene Programmkonstrukte beschrieben, welche vom Recorder aufgezeichnet, und für welche Tests generierbar sein müssen. Die folgenden Unterabschnitte fassen diese Anforderungen zusammen und beschreiben deren Umsetzung, sowie mögliche Probleme. Als Grundlage für die Tests dienen verschiedene Testprogramme, welche die jeweilige Anforderung gezielt getestet haben.

4.1.1 Datentypen

Java kennt zwei verschiedene Arten von Datentypen: primitive und komplexe Datentypen. Beide können in Parametern oder Rückgabewerten auftreten. Daher müssen beide Arten aufzeichenbar sein.

```
1 int    zahl = 42;   // Primitiver Datentyp
2 String text = "Hello World";   // Komplexer Datentyp
```

Der primitive Datentyp aus Zeile 1 wird von XStream in folgendes XML umgesetzt:

```
3 <int>42</int>
```

Die komplexe Datentyp aus Zeile 2 in folgendes:

```
4 <string>Hello World</string>
```

Die obigen zwei Elemente könnten auch als Variablen Teil einer Klasse sein. Dann erzeugt XStream das in Listing 4.1 abgedruckte XML.

```
1 <com.example.AsMembers>
2   <zahl>42</zahl>
3   <text>Hello World</text>
4 </com.example.AsMembers>
```

Listing 4.1: Komplexes serialisiertes Objekt

Nun enthalten die XML-Tags zusätzliche Metadaten. Der Name des äußerste Tags (Zeile 1) entspricht dem FQN der Klasse, während die Tags in Zeile 2 und 3 den jeweiligen Variablennamen wiedergeben. Durch diese Metadaten kann XStream beim deserialisieren die Klasse mit den enthaltenen Variablen wiederherstellen.

AspectJ wandelt alle Parameter in Objekte um, wodurch im Aspekt keine primitive Datentypen auftauchen. Das Autoboxing wird daher schon von AspectJ durchgeführt wodurch an XStream nur Objekte übergeben werden.

Primitive Datentypen liegen nach der Deserialisierung als komplexe Datentypen vor. Für den Aufruf der Methode aus dem Test müssen mögliche primitive Datentypen daher erst zurückgewandelt werden. Dies geht im Prototypen bereits automatisch, da die deserialisierten Objekte im ParameterizedTestRunner vorliegen. Dieser ruft die eigentliche Testmethode über Reflections auf, welche das Autoboxing übernimmt, da der Testgenerator die Testmethode mit der entsprechenden Signatur erzeugt hatte.

Damit liegen beim Ausführen der Testmethode die Parameter als primitive Datentypen vor. Mit diesen kann nun die eigentlich zu testende Methode aufgerufen werden. Im Test selbst sind primitive Datentypen möglicherweise speziell zu behandeln, da zum Beispiel kein `null` Wert zugewiesen werden kann. Daher muss der Testgenerator bei primitiven Datentypen die Initialisierung gesondert durchführen, bei komplexen Datentypen kann immer `null` zugewiesen werden.

4.1.2 Sonderwerte

Neben den verschiedenen Datentypen können Methoden auch keine Parameter oder Rückgabewerte aufweisen. Bei derartigen Funktionen wird von AspectJ eine leere Parameterliste bereitgestellt. Daher benötigen diese Methoden keine gesonderte Behandlung im Aspekte.

Des Weiteren können Objekte den Wert `null` aufweisen. Dies wird von Xstream nach „`<null/>`“ serialisiert und bei der Deserialisierung wieder korrekt als null zurückgegeben.

4.1.3 Statische Variablen

Variablen einer Klasse können auch statisch sein, dann ist nur eine Instanz pro Klasse vorhanden. XStream speichert nur die normalen Variablen, statische werden ignoriert.

Die eigentlichen XML Daten werden im Recorder durch die Klasse `ObjectSnapshot` gekapselt. Diese Klasse hält hierzu die XML Daten als Zeichenfolge. Zusätzlich ist hier auch eine Zeichenfolge für statische Variable vorgesehen. Durch die Klasse `ObjectSnapshotConverter` können auch die statischen Anteile eines Objektes gesichert werden. Hierzu werden alle statischen Variablen in einer Map abgelegt. Diese wird durch XStream nach XML transformiert und in `ObjectSnapshot` gespeichert.

Die statischen Variablen der Klasse

```
1 class StaticClass {
2   static int statischerWert = 42;
3 }
```

werden durch den `ObjectSnapshotConverter` gespeichert als

```
<map>
  <entry>
    <string>statischerWert</string>
    <int>12</int>
  </entry>
</map>
```

Die gespeicherte Map enthält als Schlüssel/Werte-Paar den Namen des statischen Feldes, sowie dessen Wert als XML Repräsentation.

4.1.4 Wiederherstellen von Objektzuständen

Für die Testausführung ist es nötig, das Testobjekt in einen definierten Zustand zu versetzen. Hierzu kann während der Testaufzeichnung der Zustand beim Aufruf gesichert werden, um diesen im Test zum Wiederherstellen zu nutzen.

Das AspectJ Schlüsselwort `thisJoinPoint` bietet Aspekten den Zugang zur Instanz des aktuellen Objektes. Genau wie Parameter ist dies eine Objekt-Instanz und kann daher genau wie diese mit XStream gespeichert werden. Dadurch kann bei der Testausführung der Objektzustand durch ein deserialisieren wiederhergestellt werden.

4.1.5 Arrays

Da Felder in Java durch die Klasse `Array` abgebildet werden, unterschiedet sich das Aufzeichnen nicht von dem anderer Objekte. XStream serialisiert beispielsweise das folgende String-Feld

```
String[] { "Suppen", "Teller" };
```

in die folgende XML Darstellung:

```
<string-array>
  <string>Suppen</string>
  <string>Teller</string>
</string-array>
```

Im XML erscheint hinter dem Datentyp des Feldes der Zusatz “-array”, mit dem XStream Felder markiert. Dieses XML kann nun gespeichert, und später von XStream wieder zurück in ein String-Feld gewandelt werden.

4.1.6 Nicht öffentliche Methoden

Methoden können in ihrer Sichtbarkeit gegenüber anderen Klassen eingeschränkt sein. Am problematischsten erscheint hier der Einsatz von privaten Methoden, da diese schon bei einer Vererbung nicht geändert werden können.

Da der Aspekt direkt in den Bytecode injiziert wurde, gibt es für diesen keinen Unterschied in der Sichtbarkeit der Methode. Der Aspekt und damit auch der Testrecorder muss für die Aufzeichnung privater Methoden nicht angepasst werden.

Im Test werden die zu testenden Methoden direkt über eine Instanz der Klasse aufgerufen:

```
1 testInstance.eineMethode("Hallo");
```

Dieses Vorgehen funktioniert allerdings nur auf öffentlichen (public) oder geschützen (protected) Methoden. Soll eine private Methode aufgerufen werden, dann ist dies nur über Reflections möglich:

```
1 Method method = ReflectionHelper.getMethod( PrivateClass.class,
2                         "eineMethode", new Class[] { String.class });
3 method.setAccessible(true);
4 method.invoke(testInstance, "Hallo");
```

In Zeile 3 wird durch `setAccessible` das Aufrufen der Methode in Zeile 4 erlaubt.

War die Methode bereits während der Aufzeichnung privat, dann erzeugt der Testgenerator bereits den korrekten Methodenaufruf durch Reflections. Wurde die Methode aber erst nachträglich in ihrer Sichtbarkeit geändert, fehlt der Aufruf durch Reflections, weswegen der Test manuell angepasst werden muss.

4.1.7 Singeltons

Das Singelton-Entwurfsmuster [Gamma u. a., 1995] beschreibt wie sichergestellt werden kann, dass zur Laufzeit nur eine einzige Instanz einer Klasse existiert. Dazu wird der Konstruktor einer solchen Klasse als privat deklariert. So kann keine andere Klasse eine Instanz anlegen. Um die Klasse nutzen zu können wird in dieser eine statische Instanz gehalten, auf welche durch eine öffentliche, statische Methode zugegriffen werden kann.

Es gibt dabei zwei Situationen zu betrachten: der Singelton wird als Parameter übergeben oder eine Methode aus der Singelton Klasse soll aufgezeichnet werden.

Im ersten Fall stellt sich der Vorgang sehr einfach dar, da hier durch den Einsatz von XStream keine Komplikationen auftreten. Das Aufzeichnen eines übergebenen Singeltons ist unproblematisch, da bereits eine Instanz übergeben wird. Für die Testausführung muss diese wiederhergestellt werden, was trotz privatem Konstruktor von XStream unterstützt wird.

Auch das Aufzeichnen einer Methode des Singeltons ist unproblematisch, da das Verstecken des Konstruktors für die Aufzeichnung keine Auswirkung hat, und das Wiederherstellen durch XStream keine Probleme bereitet.

Da im Normalfall die Funktionen des Singeltons unabhängig von der statischen Instanz sind, ist es nicht nötig die statischen Variablen mit aufzuzeichnen. Die statische Instanz des Singeltons ist nur für die öffentliche Zugangsmethode interessant, alle anderen Methoden im Singelton sind unabhängig davon implementiert. Ausnahmen sind weitere öffentliche, statische Methoden. Beim Aufzeichnen dieser ist das gleiche wie beim Aufzeichnen statischer Methoden (4.1.12) zu beachten.

4.1.8 Fabriken

Sowohl abstrakte Fabriken, wie auch Fabrikmethoden dienen zum Vereinfachen der Objektinstanzierung. Bei beiden Vorgehen wird die Instanzierung einer konkreten Klasse in Methoden ausgelagert.

Aus Sicht des Recorders und der generierten Tests stellt dies kein Problem dar, da es sich um einen normalen Methodenaufruf handelt.

4.1.9 Ausnahmen

Sprachen wie Java kennen das Konzept von Ausnahmen, zum frühzeitigen Verlassen einer Methode um einen Fehlerzustand zu signalisieren.

Der folgende kurze Auszug aus dem Aspekt zeigt den Aufruf der Zielmethode (Zeile 2), der komplett durch einen try-catch-Block umschlossen ist. Daher kann der Aspekt alle Ausnahmen abfangen und aufzeichnen.

```
1  try {
2    theReturnValue = proceed();
3  } catch (Throwable theThrowable) {
4    ...
5  }
```

Bei der Testausführung sind Ausnahmen speziell zu behandeln. Dies wird momentan durch einen direkten `try catch` Block in der Testemethode erreicht:

```
1  try {
2    actualReturnValue = testInstance.calculate(param1, param2);
3  } catch (Throwable theThrowable) {
4    actualException = theThrowable;
5    // Überprüfung ob gespeicherte Exception gleich der geworfenen ist.
6  }
```

Durch Annotationen wurde in JUnit4 die Lesbarkeit solcher Tests verbessert. Der Folgende Test erwartet eine `NullPointerException`. Nur wenn diese auch im Test geworfen wurde, war der Test erfolgreich.

```
1 @Test(expected=NullPointerException.class)
2 public void nullPtrTest() {
3     Object obj = null;
4     obj.toString();
5 }
```

Das Nutzen der expected Eigenschaft von JUnit4, ist für den Generator schwieriger umzusetzen, da hier beim Generieren je geworfener Ausnahmenklasse ein eigener Parameterized-Test erstellt werden müsste. Dies ist der Grund warum es momentan als ein try-catch-Block implementiert ist.

Eine mögliche Lösung wie der `ParameterizedTestRunner` ähnlich JUnit4 mit Ausnahmen umgeht, wird im Abschnitt 5.2.4 vorgestellt.

4.1.10 Equals

Jedes Objekt in Java ist von der Klasse `Object` abgeleitet. Damit besitzt jedes Objekt automatisch eine `equals` Methode. Diese dient dazu zwei Objekte zu vergleichen. In der Implementierung, welche in `Object` hinterlegt ist, wird hierbei der Hashcode der beteiligten Objekte genommen. Der Hashcode ist im Normalfall die Objektreferenz. Daher gibt es keinen Unterschied zwischen der `equals` Methode und dem Gleichheits Operator ("`==`"). Erst wenn die `equals` Methode überschrieben wurde, wird der Inhalt der Objekte betrachtet, nicht deren Referenz.

Soll diese Methode aufgezeichnet werden, ist dies eine normale öffentliche Methode und daher möglich.

Die Problematik liegt hier allerdings in einem Nebeneffekt der beim Speichern der Objekte zur Laufzeit eintritt. Betrachtet man das folgende Beispiel, liefert der Vergleich eines Objektes mit sich selbst ein `true`.

```
1 Object obj = new Object();
2 System.out.println( obj.equals(obj) ); // true
```

Dies ist das erwartete Ereignis, da in diesem Fall die Objektreferenzen überprüft werden und diese identisch sind.

Wird nun aber während der Testaufzeichnung das Objekt als XML abgespeichert:

```
3 XStream xs = new XStream();
4 String xml = xs.toXML(obj); // xml = "<object/>"
```

Wird die Referenz nicht im XML abgelegt, daher erzeugt XStream bei jedem Aufruf ein neues Objekt:

```
5 Object instance = xs.fromXML( xml );
6 Object param1   = xs.fromXML( xml );
```

Der Vergleich mit der equals Methode liefert nun ein `false` zurück, da es sich bei `instance` und `param1` nicht mehr um ein und dasselbe Objekt handelt.

```
7 System.out.println( instance.equals(param1) ); // false!
```

Die Gleichheit zweier Objekte ist daher in vielen Fällen bei der Testausführung nicht mehr gegeben. Um dieses Problem zu adressieren wurde die ObjectSnapshot-Klasse um Meta-Informationen erweitert. In dieses Feld wird während der Aufzeichnung die aktuelle Referenz des Objektes abgelegt. Dadurch ist es möglich identische Objekte zu finden.

Während der Testausführung kann durch die gespeicherte Referenz eine Identifikation gleicher Objekte erfolgen. Wird ein Objekt deserialisiert, dessen Referenz schon geladen ist, dann ist ein Weg zu finden wie dieses Objekt deserialisiert werden kann, ohne eine neue Referenz zu erhalten. Hierzu ist entweder XStream anzupassen oder eine Hilfsklasse zu entwickeln. Daher ist das Behandeln von derartigen Duplikaten momentan noch nicht implementiert.

4.1.11 Threads

In der aktuellen Fassung ist der implementierte Tracer wegen dem Einsatz von statischen Daten nicht für das Aufzeichnen bei mehreren Threads geeignet.

4.1.12 Statische Methoden

Das Aufzeichnen von statischen Methoden unterscheidet sich größtenteils nicht von dem anderer Methoden, einzig kennen diese keinen `this` Zeiger, besitzen also keinen Objektzustand.

Daher wird im Aspekt bei statischen Methoden der Objektzustand nicht gesichert, aber vorhandene statische Variablen.

Bei den erzeugten Tests wird zum Aufrufen der Zielmethode die Klasse anstelle einer Instanz genutzt.

4.1.13 Zustandsbehaftete Objekte

Unter zustandsbehafteten Objekten sind nicht normale Java Objekte gemeint, sondern Objekte deren Zustand extern definiert ist. Beispielsweise eine Datenbankverbindung.

Getestet wurde hier das Verhalten von verschiedenen PrintStream Instanzen. Das Aufzeichnen einer PrintStream Instanz wird von XStream normal durchgeführt. Einzig ist das resultierende XML ist groß, da ein Puffer vorgehalten wird den XStream mitspeichert.

Für den Test wurde `System.out` und ein `FileOutputStream` genutzt. Das Ausführen der Testfälle funktioniert, allerdings sind die Streams nicht mit ihren ursprünglichen Zielen verbunden. Der `System.out` Stream gibt nichts auf die Konsole aus, und der `FileOutputStream` schreibt nichts in die Datei.

Generell stellen Objekte die einen externen Zustand kapseln ein Problem dar. Neben den Streams umfasst das eine ganze Reihe an Objekten: Datenbankverbindungen, RMI, HTTPSessions etc.

Im Falle der Streams wird diese fehlende Verbindung nicht festgestellt, sondern der Test läuft erfolgreich durch obwohl er eigentlich fehlschlagen sollte.

4.1.14 Dynamische Proxies

Wird eine Zielmethode durch einen dynamischen Proxy mit Funktionalität angereichert, gibt es einen Nebeneffekt. Dies lässt sich aus dem Sequenzdiagramm 4.1 ablesen.

Da die Instrumentation direkt in den Bytecode compiliert wurde, liegt der dynamische Proxy davor. Dadurch wird der Proxy immer vor dem Aspekt ausgeführt, dieser kann also nicht die Funktionalität des Proxies mit aufzeichnen. Ist der Proxy bei der Testausführung aktiv, dann ist der Test das aufrufende Programm, weswegen bei der Testausführung der Proxy komplett durchlaufen wird. Gegenüber den aufgezeichneten Daten, ohne Proxy, verhält sich das SUT anders, der Test schlägt fehl.

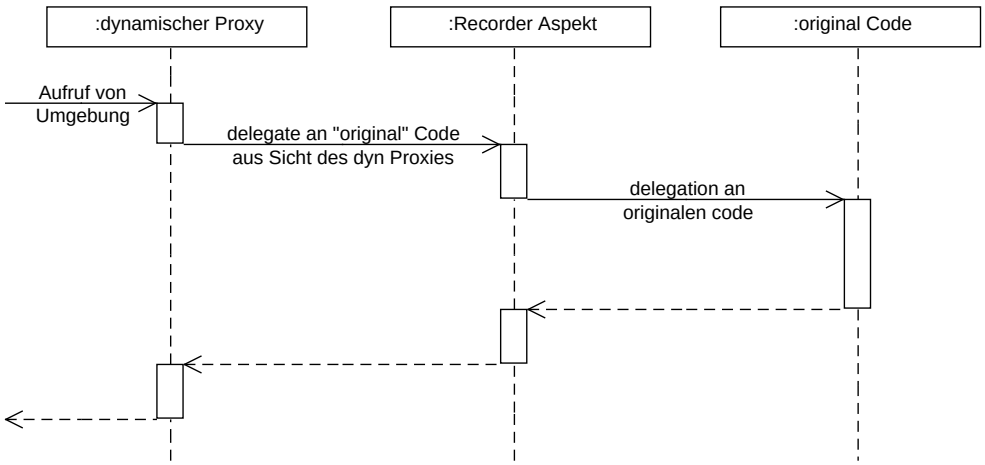

Abb. 4.1: Dynamischer Proxy vor eigentlichem Aspekt

Das Aufzeichnen von Methoden mit einem dynamischen Proxy ist daher nur durch Tricks möglich, in Abschnitt 4.2 wird eine Lösung beim Einsatz von Datenbanktransaktionen beschrieben.

4.2 Aufzeichnen von Datenbanken

Zum Testen von DAO-Schichten ist es nötig den Zustand des Systems und den der Datenbank zu testen. Hierbei gilt es verschiedene Eigenheiten zu beachten.

In den folgenden Unterabschnitten wird als erstes allgemein beschrieben wie eine Aufzeichnung aussehen würde. Die Details der konkreten Umsetzung aus diesen Erkenntnissen kommen dann im letzten Unterabschnitt. Vorweggenommen sei, dass die Bibliothek DBUnit zum Speichern und Wiederherstellen eingesetzt wird. DBUnit bietet bereits fertige Funktionen zum Sichern, Wiederherstellen und Vergleichen von Tabellen an. Eine Datenbankverbindung reicht hierzu aus.

4.2.1 Einfaches Aufzeichnen der Datenbank

Wird die Verbindung zur Datenbank in der aufzeichnenden Methode auf- und wieder abgebaut, dann ist das Aufzeichen problemlos implementierbar: parallel wird eine zweite Datenbankverbindung aufgebaut und über diese, vor dem Aufruf der Methode, die Tabellen gesichert. Diese Sicherung kann vor dem Testen zurück in die Datenbank übertragen werden, um diese in einen definierten Ausgangszustand zu bringen.

Durch den Einsatz von DBUnit können die Tabellen ebenfalls als XML gespeichert werden. Listing 4.2 zeigt eine leere “customer” Tabelle.

```
1 <dataset>
2   <table name="customer">
3     <column>id</column>
4     <column>name</column>
```

```
5   </table>
6 </dataset>
```

Listing 4.2: Customer Tabelle vor Änderung

Nachdem die Methode verlassen wurde, kann erneut eine Verbindung aufgebaut werden um die Tabellen wieder zu speichern. Beim Testen dient dieser zweite Abzug einem Vergleich, ob Änderungen korrekt durchgeführt wurden. Listing 4.3 zeigt die "customer" Tabelle nach dem Anlegen eines neuen Kunden "Spring Customer".

```
1 <dataset>
2   <table name="customer">
3     <column>id</column>
4     <column>name</column>
5     <row>
6       <value>1</value>
7       <value><![CDATA[Spring Customer]]></value>
8     </row>
9   </table>
10 </dataset>
```

Listing 4.3: Customer Tabelle nach Änderung

Die Funktionalität zum Speichern der Tabellen und Wiederherstellen wird bereits von DBUnit angeboten. Auch bei Vergleichen bietet DBUnit weitreichende Unterstützung an.

4.2.2 Mit Datenbank-Transaktionen

Wird die Datenbankverbindung bereits außerhalb der Zielmethode aufgebaut, wird das Aufzeichnen schwieriger. Hier werden Datenbanktransaktionen eingesetzt, wodurch Änderungen an der Datenbank nur innerhalb der gleichen Verbindung sichtbar werden. Erst durch ein Commit werden die Änderungen auch für weitere Verbindungen zur Datenbank sichtbar.

Der im vorherigen Abschnitt genutzte Ansatz einer zweiten Verbindung sieht daher keine Änderung auf der Datenbank.

Die Lösung besteht darin, anstatt einer zweiten Verbindung, die selbe Verbindung wie die Zielmethode zu nutzen. Dadurch kann der Testrecorder nun die tatsächlichen Änderungen sichern. Es ist daher eine Möglichkeit vorzusehen die aktuelle Datenbankverbindung an den Testrecorder zu übergeben.

Erfolgt das Verwalten der Datenbankverbindung manuell, kann hier kein allgemeingültiger Ansatz gefunden werden. Meist werden aber Hilfsbibliotheken eingesetzt, für die allgemeine Ansätze gefunden werden können.

4.2.3 Aufzeichnen von Transaktionen unter Spring

Wird das Springframework [Johnson u. a., 2008], kurz Spring, zur Transaktionsverwaltung eingesetzt, dann ist für dieses das Aufzeichnen allgemein lösbar. Spring nutzt für die Transaktionsverwaltung eine Kombination aus Annotationen und AOP Techniken.

Da Spring für das Instanzieren aller Objekte zuständig ist, wird das Transaktionsmanagement von Spring durch dynamische Proxies eingebaut. Dadurch liegen diese Proxies, wie in Abschnitt 4.1.14 beschrieben, vor dem eigentlichen instrumentierten Code des Recorders. Es werden Spring-Transaktionen, aus Sicht des Recorders, immer vor der aufzuzeichnenden Methode gestartet und erst danach beendet.

Im Fall von Spring ist dies allerdings einfach lösbar. Da auch die eigentliche Datenbankverbindung von Spring verwaltet wird, kann diese durch eine eigene Spring-Bean ausgelesen werden. Dadurch kann beim Einsatz von Spring die gleiche Datenbankverbindung genutzt werden wie für die Methode selbst, und es wird somit das Aufzeichnen einer DAO Schicht möglich.

4.2.4 Proxy für Datenbanken

Alternativ zum vorgeschlagenen Abgreifen der Connection ist es auch möglich einen Proxyansatz zu wählen. Dieses Vorgehen ist dabei am besten auf Spring Projekten möglich, da hier der Proxy zentral eingebaut werden kann.

Anstelle der eigentlichen Zielmethode wird der Proxy aufgezeichnet, dieser delegiert den Aufruf an die Zielmethode, welche mit einem dynamischen Proxy die Transaktion verwaltet. Nach dem Durchlaufen der Zielmethode schließt der dynamische Proxy die Transaktion und es können die Änderungen aus der Datenbank aufgezeichnet werden.

Dieser Ansatz funktioniert allerdings nur solange nicht eine weitere, äußere, Transaktion aktiv ist. Es ist daher nur sinnvoll auf einzelnen transaktionsbehafteten Methoden anzuwenden. Zusätzlich wird im generierten Test nicht die Zielmethode sondern der Proxy getestet. Da die Signaturen der Methoden aber identisch sind, reicht ein einfaches Ändern des Klassennamens aus.

4.2.5 Spring mit JPA/Hibernate

Wird die Java Persistence API (JPA) eingesetzt, tritt neben dem geschilderten Transaktions-Effekt noch ein Caching-Effekt auf.

JPA definiert ein Caching, bei welchem Änderungen an Objekten erst vor der nächsten Datenbankoperation synchronisiert werden. Beim Einsatz von JPA mit Spring wird durch den Spring-AOP-Proxy beim Verlassen der Methode eine Synchronisation vorgenommen. Da der Recorder-Aspekt noch vor dem Spring-Aspekt ausgeführt wird, sind die Änderungen noch nicht in der Datenbank gepeichert. Während der Testausführung wird der Spring-Aspekt mit ausgeführt, weswegen der Test alle Änderungen der Datenbank auslesen kann. Somit schlägt das Vergleichen zwischen dem Datenbank-Zustand beim Testen fehl.

Der Effekt muss daher schon während der Aufzeichnung unterbunden werden. Dies kann durch die `flush()` Methode des Persistenz Managers erreicht werden. Diese Methode erzwingt das Synchronisieren mit der Datenbank. Die Daten werden allerdings nur in der aktuellen Transaktion gespeichert, ein Rollback ist weiterhin möglich. Dieser Workaround hatte daher auch bisher keine Nebeneffekte gezeigt.

Genau wie die Verbindung zur Datenbank, kann auch der Persistenz Manager durch eine Spring-Bean einfach ausgelesen werden.

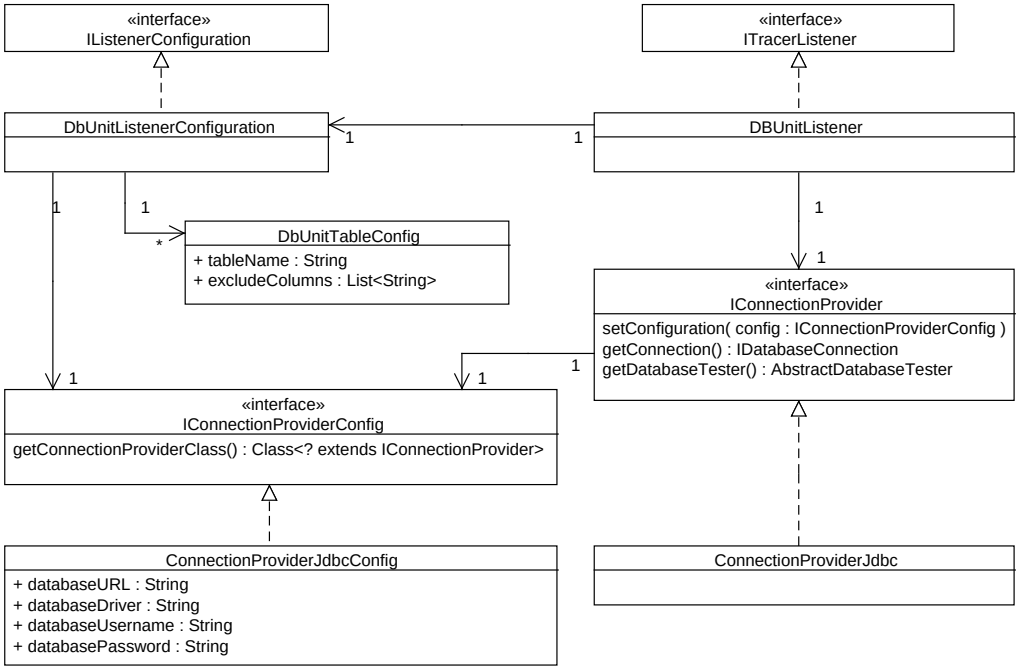

Abb. 4.2: DBUnit Implementation

4.2.6 DbUnitListener

Nach den vorgestellten Situationen welche bei Datenbanken auftreten können, soll hier der implementierte DbUnitListener beschrieben werden, der das Aufzeichnen von Datenbanken unterstützt.

Der DbUnitListener ist als Erweiterung für das Recorder-Framework implementiert. Es wurde dabei die Möglichkeit zum Anbinden eines Listeners an den Tracer genutzt. Hierzu werden die in Abbildung 4.2 dargestellten Klassen benötigt.

Die Hierarchie besteht aus den Klassen zur Konfiguration (links) und den Klassen, welche die Aufzeichnung durchführen (rechts).

Die Konfiguration kann durch das Implementieren der `IListenerConfiguration` Schnittstelle direkt in das Konfigurations-Modell des Recorders eingehängt werden. Damit nicht immer die komplette Datenbank gesichert werden muss, enthält die `DBUnitListernerConfig` eine Liste an aufzuzeichnenden Tabellen, wodurch sich das Speichern der Datenbankabzüge auf einzelne Tabellen beschränken lässt. Zusätzlich ist es möglich, je Tabelle, einzelne Spalten zu ignorieren.

Wie in den vorhergehenden Abschnitten gezeigt wurde, ist es nötig die eigentliche Datenbankverbindung an das jeweilige Zielsystem anzupassen. Um dies konfigurierbar und erweiterbar zu halten, wurden die eigentlichen Verbindunginformationen in einen sogenannten Connection Provider ausgelagert.

Dadurch können verschiedene Connection Provider für den DBUnitListener bereitgestellt werden und so an unterschiedlichste Projekte angepasst werden.

Das Listing 4.4 zeigt wie eine Konfiguration für den DbUnitListener aussehen könnte.

```
1 | ...
```

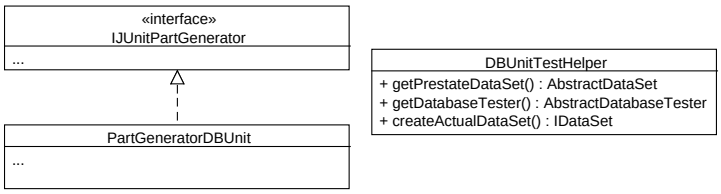

Abb. 4.3: Der DBUnit Part Generator

```
2  DbUnitListenerConfiguration duListenerConfig =
3      new DbUnitListenerConfiguration();
4
5  ConnectionProviderJdbcConfig jdbc = new ConnectionProviderJdbcConfig();
6  jdbc.setDatabaseDriver("com.mysql.jdbc.Driver");
7  jdbc.setDatabaseURL("jdbc:mysql://localhost/example1");
8  jdbc.setDatabaseUsername("root");
9  jdbc.setDatabasePassword("");
10
11 duListenerConfig.setConnectionProvider( jdbc );
12
13 DbUnitTableConfig tconf=new DbUnitTableConfig();
14 tconf.setTableName("customer");
15 tconf.addExcludeColumn("id");
16 duListenerConfig.add( tconf );
17
18 methodConfiguration.getTracer().addListener( duListenerConfig );
19 ...
```

Listing 4.4: DbUnitListener Konfiguration

Da die Konfiguration durch den AspectJ-Generator in eine XML-Konfiguration gespeichert wird, kann diese so zum Zeitpunkt der Aufzeichnung wieder geladen werden. Dieses wird dann durch die Tracer-Factory automatisch an den DbUnitListener übergeben.

Konfiguriert wurde in Listing 4.4 ein JDBC-Daten-Provider der eine Zweite Verbindung zu der Datenbank aufbaut und über diese die Tabelle “customer” sichert. Der DBUnitListener wird wie alle Listener in Zeile 18 an den Tracer angehängt.

4.2.7 PartGeneratorDBUnit

Die Klasse `PartGeneratorDBUnit` dient zum Generieren der Tests und ist das Gegenstück zum `DBUnitListener`. Abbildung 4.3 zeigt den Part Generator und eine Hilfsklasse für die Testausführung.

Der `PartGeneratorDBUnit` überschreibt die Methoden der `IJunitPartGenerator` Schnittstelle und baut während der Test-Generation den nötigen Programmcode für das Vergleichen der Datenbank ein. Das Hinzufügen des Part-Generators erfolgt gleich wie das anderer Part-Generatoren auf dem Generator:

```
1  GeneratorJUnit generator = ...;
```

```
2   generator.addPartGenerator( new PartGeneratorDBUnit() );
3   ...
```

Im erzeugten Test wird die Klasse DBUnitTestHelper eingesetzt um dem DatabaseTester von DBUnit die korrekten Testdaten zuzuweisen. Dies ist nötig, da durch den Parameteriezd Test die Testmethode mehrmals gerufen wird, und so die echten Testdaten dynamisch nachgeladen werden müssen.

4.3 Reales Projekt

Neben den Aufzeichnungen auf dem Testprojekt wurden zusätzlich Aufzeichnungen an einem größeren Projekt von SEITENBAU durchgeführt. Hierbei sind die in den folgenden Unterabschnittenen beschriebenen Effekte eingetreten.

4.3.1 Parameter Refactoring

In manchen Methodensignaturen stehen mehr Parameter als in der Methode selbst verarbeitet werden. Dies kann passieren wenn die Parameter durch ein externes Interface eingeführt werden. In folgendem Beispielcode wird eine `runIt` Methode einer Schnittstelle überschrieben, aber nur der erste Parameter wirklich ausgewertet.

```
1   public class BigClass implements IRunIt {
2     public int runIt(String args, DatabaseDump[] data) {
3       // Nutzt nur den Parameter args
4     }
5   }
```

In diesem Fall kann das Refactoring Muster “Methode Extrahieren” dazu genutzt werden eine Methode zu erzeugen, deren Signatur ohne den Balast auskommt:

```
1 public class BigClass implements IRunIt {
2     public int runIt(String args, DatabaseDump[] data) {
3       return runItExtracted(args);
4     }
5     protected int runItExtracted(String args) {
6       // Nutzt nur den Parameter args
7     }
8   }
```

Die extrahierte `runItExtr`acted Methode enthält nun keinen Parameter DataBaseDump mehr und kann nun aufgezeichnet werden. Der für diese Aufzeichnung erstellte Test überprüft allerdings die extrahierte Methode, nicht die Originalmethode.

Da für dieses Vorgehen ein Ändern am Projekt nötig ist, sollte es mit Vorsicht eingesetzt werden. Die bessere Lösung wäre es dem Testrecorder die Fähigkeit zu geben, einzelne Parameter nicht aufzuzeichnen. Dadurch ist keine Methode mehr zu extrahieren und nur eine Änderung an der Konfiguration nötig.

4.3.2 Objektgröße

Werden an Methoden sehr große Parameter übergeben kann dies während der Aufzeichnung zu OutOfMemory-Ausnahmen führen. Sind die übergebenen Parameter sehr groß, verbraucht XStream beim Transformieren zuviel Speicher.

Methoden mit speicherintensiven Objekten können nicht direkt aufgezeichnet werden. Die einzige Lösung stellt das im vorherigen Abschnitt gezeigte "Parameter Refactoring" dar. Dies setzt aber voraus, dass nicht alle Parameter genutzt werden.

Ein alternativer Vorschlag, um nur die tatsächlich genutzten Daten zu erfassen, wird im Abschnit 5.2.1 behandelt.

4.3.3 Umgebung aussetzen

Insgesamt ist das Wiederherstellen der Systemumgebung aufwändiger als erwartet. Vor allem bei globalen und statischen Zuständen ist dies durch den Testgenerator schwer abdeckbar. So müssen zum Beispiel Persistenz-Frameworks auch während der Testausführung initialisiert sein.

Die Initialisierung ist dabei stark vom jeweiligen Projekt abhängig und kann daher am besten durch einen projektspezifischen Part-Generator gelöst werden. Der Part-Generator baut dann in jeden Test projektspezifische Initialisierungsroutinen ein.

4.3.4 Tests mit Spring

Bei Spring Projekten ist es nötig während der Testausführung die Springumgebung wiederherzustellen. Im Normalfall wird dies durch den SpringTestRunner erledigt. Da dieser aber auch ein JUnit-TestRunner ist, kann der ParameterizedTestRunner nicht gleichzeitig eingesetzt werden.

Als Lösung wurde ein Listener-Konzept für den ParameterizedTestRunner implementiert. Wie in folgendem Testausschnitt zu sehen ist, wird durch eine Annotation ein `SpringTestContextListener` für die Testausführung mit angemeldet.

```
1 @RunWith(ParameterizedTestRunner.class)
2 @ParameterizedTestListener( { SpringTestContextListener.class } )
3 public class CustomerDAOImplTest {
4   ...
```

Der ParameterizedTestRunner informiert den SpringTestContextListener während der Ausfühung über das Instanzieren des Testes. Dadurch kann der Listener über einen Spring-`TestContextManager` den neu erzeugten Test initialisieren.

4.3.5 Persistenz Framework

Das Speichern und Wiederherstellen von Parametern stellt für den Recorder kein Problem dar. Wird allerdings ein Persistenz Framework eingesetzt, dann sind einige Parameter zusätzlich mit der Daten-

bank verbunden.

Werden diese Objekte aufgezeichnet, dann geht die Verbindung zur Datenbank verloren. Bei der Testausführung werden diese als normale Objekte wiederhergestellt. Beispielsweise, wird bei einem Hibernate Framework ein update durchgeführt:

```
1 | hibernateSession.update( anObject ); // Ausnahme : detached Entity
```

führt dies zu einer Ausnahme. Hibernate war nicht für das Laden des Objektes zuständig, da dies von XStream wiederhergestellt wurde, somit kann Hibernate keine Zuordnung zwischen Objekt und Datenbank zum Testzeitpunkt mehr treffen.

Wird anstelle eines Updates ein Save ausgeführt

```
1 | hibernateSession.saveOrUpdate( anOtherObject ); // keine Ausnahme aber
    |     evtl. falsch
```

speichert Hibernate das Objekt neu in die Datenbank. Dieser Fehler wird erst durch das Testen der Datenbank entdeckt, da im Originalprogamm nur eine Änderung stattgefunden hat, während im Test ein neuer Datensatz erstellt wurde.

Die Verbindung zwischen Datenbank und dem Objekt kann über eine `merge()` Operation wiederhergestellt werden. Dazu müsste vor dem Aufrufen der Persistenz Schicht ein `merge` durchgeführt werden. Dieses Vorgehen ist allerdings schwer automatisierbar, da dafür schon während der Aufzeichnung, je Objekt, die Verbindung zur Datenbank gespeichert werden muss. Dieses Phänomen tritt nicht nur direkt auf den Parametern auf, sondern auch in jedem darin enthaltenen Unterobjekt, und so muss der komplette Objektbaum durchlaufen werden.

4.3.6 SOA

Aufgrund des großen Aufwandes für das Erstellen und Aufzeichnen von Tests auf Objekten mit einem Zustand ist das Aufzeichnen auf zustandslosen Objekten sehr interessant. Beispielsweise konnte auf einer zustandslosen Methode mit einem Mac Cabe Maß von 19 eine 100%ige Anweisungs-Überdeckung erreicht werden.

Beim Aufzeichnen zustandslosen Objekte ist eine gute Testabdeckung und vor allem eine gute Initialisierbarkeit erreichbar. Daher ist der Ansatz auch sehr interessant für das Aufzeichnen von Serviceorientierte Architekturen (SOA).

4.3.7 Filter

Die Implementierte Filtertechnik ist zwar technisch in der Lage eine Filterung zu erreichen, das Modellieren der Ausschlusskriterien ist allerdings recht aufwendig und schwierig.

Da gerade bei realen Objektbäumen ist die kleine Darstellung im JUnit-Failure Trace sehr schwierig zu interpretieren um daraus eine Filter Konfiguration abzuleiten. Außerdem ist die Notwendigkeit zum Ausschluss auf mehr als einem Testdatensatz anzuwenden was vom Framework nicht unterstützt wird. Für eine Verbesserung dieses Zustandes finden sich im Abschnitt 5.1.5 Vorschläge zu finden.

4.4 Refactoring

Wie im Kapitel Grundlagen erwähnt, soll neben der reinen Testfunktionalität auch die Robustheit der Tests gegenüber Veränderungen betrachtet werden. Daher werden in diesem Abschnitt verschiedene Refactoring Muster betrachtet und deren Auswirkung auf die Tests bewertet.

Die Betrachtung beschränkt sich hierbei auf bereits generierte Tests. Werden Änderungen am Projekt gemacht, hat dies keine Auswirkung auf gespeicherte Testdaten im Repository. Werden die Tests also erst später aus dem Repository generiert, sind diese nicht mit dem geänderten Projekt lauffähig.

4.4.1 Feld umbenennen

Das Umbenennen eines Feldes ist kein Muster nach [Fowler, 2005], tritt aber in der Praxis auf und hat eine Auswirkung, weswegen dies hier behandelt wird.

Als Beispiel dient eine Klasse mit dem Feld `myState`, die als XML Abzug wie folgt aussieht:

```
1 <com.example.refactoring.RefactoringObject>
2   <myState>1024</myState>
3 </com.example.refactoring.RefactoringObject>
```

Wird nun der Name des Feldes `myState` geändert, dann kann XStream dieses Feld nicht mehr deserialisieren, denn XStream sucht den alten Feldnamen “myState”.

Es kann nun entweder der Name des Feldes im XML angepasst werden, oder kann eine XStream Annotation an das Feld angehängt werden:

```
1 @XStreamAlias("myState")
2 private int myInternalState = 1024;
```

Durch die Angabe der Alias Annotation wird nun von XStream das XML wieder korrekt deserialisiert. Das Ändern der XML Datei ist der Annotation vorzuziehen, da die Annotation eine Änderung am Projekt einführt. Allerdings fehlt hier momentan ein unterstützendes Werkzeug mit dem diese Änderung effizient vorgenommen werden könnte.

4.4.2 Feld kapseln

Die Sichtbarkeit eines Felds wird auf privat geändert. Der Zugriff kann nur noch durch Getter & Setter erfolgen. Während das Umbenennen eines Feldes Auswirkungen hatte, hat das Ändern der Sichtbarkeit keine, da XStream auch mit privaten Feldern zurechtkommt.

4.4.3 Methode verbergen

Die Sichtbarkeit einer Methode wird geändert.

Problematisch wird das Ändern der Sichtbarkeit erst wenn diese auf `private` geändert wird. Methoden die `public` oder `protected` sind, können vom Test normal aufgerufen werden. Private Methoden können nicht von anderen Klassen aufgerufen werden, also auch nicht vom Test. Daher erstellt der Generator bei privaten Methoden den Aufruf durch Reflections. Wird aber während der Testgeneration der Aufruf ohne Reflections erzeugt, ist der Test bei einer Änderung auf `private` nicht mehr lauffähig.

4.4.4 Methode umbenennen

Das Umbenennen einer Methode ist nur bei nicht öffentlichen Methoden speziell zu behandeln. Normalerweise werden beim Umbenennen durch die Entwicklungsumgebung bereits alle Quellcode-Stellen geändert.

Daher wird beim Umbenennen der Methode auch der JUnit Test automatisch angepasst. Private Methoden werden allerdings durch Reflections aufgerufen, weswegen hier der eigentliche Methodenname in einer Zeichenfolge abgelegt ist. Diese wird daher von einer Entwicklungsumgebung nicht erkannt und daher ignoriert.

Für private Methoden ist daher ein manuelles Anpassen der Tests nötig, für alle anderen Methoden wird dies von Seiten der Entwicklungsumgebung korrekt behandelt.

4.4.5 Parameter entfernen

Wird ein Parameter aus einer Methode entfernt, führt dies zu keinem direkten Nebeneffekt, da die Entwicklungsumgebung das Entfernen auch im Test vornimmt.

Allerdings ist der entfernte Parameter immer noch in der XML Datei vorhanden und wird daher auch für die Testausführung deserialisiert. Da er aber nicht mehr genutzt wird, ist dies nun überflüssig. Das Entfernen aus der XML Datei ist allerdings je nach Größe der Datei aufwändig. Zusätzlich muss beim Entfernen auch die Signatur der Testmethode angepasst werden.

4.4.6 Parameter hinzufügen

Wie auch beim Entfernen eines Parameters sorgt auch hier die Entwicklungsumgebung für einen weiterhin compilierbaren Test.

Allerdings wird es beim Hinzufügen nötig, den neuen Parameter auch mit Werten zu füllen. Daher ist die XML Datei zu ändern und die Signatur der Testmethode anzupassen.

4.5 Voraussetzungen für die Testaufzeichnung

Der Prototyp stellt an das System, in welchem die Aufzeichnung durchgeführt wird, die folgenden Anforderungen:

Funktionalität Sollte "getestet" sein. Da das Ziel der Testaufzeichnung das Erstellen von Softwaretests ist, erscheint dies als Wiederspruch. Es sind hier aber nicht Tests im Sinne von Softwaretests gemeint, sondern im Sinne von Funktionalität. Bei Wartungsprojekten ist dies beispielsweise durch die Benutzung gegeben. Ein später aus der Aufzeichnung generierter Test kann nichts anderes testen als das Verhalten der Anwendung während der Aufzeichnung, daher muss mindestens der aufzuzeichnende Teil der Anwendung fehlerfrei sein.

Oberfläche Während der Aufzeichnung muss die Anwendung bedient werden können. In den meisten Anwendungen wird dies durch eine Benutzeroberfläche erfolgen. Dabei wird ein Teil oder die ganze Anwendung wie bei einem normalen Systemtest genutzt, während der Test Recorder im Hintergrund Testdaten aufzeichnet. Unter der Oberfläche wird hier nicht nur eine enthaltene Oberfläche verstanden, sondern jegliche Art der Interaktion ist vorstellbar. Webservices können beispielsweise mit Tools wie [eviware software ab] angesprochen werden.

4.6 Welche Projekte eignen sich zur Testaufzeichnung?

Allgemein eignet sich die Methode für jegliche Projekte bei denen der zu testende Code bereits existiert.

Der Einsatz wird jedoch schwierig bei Projekten, welche mit Test Driven Development [Beck, 2003] entwickelt werden. Hier wird als erstes der Test geschrieben, und dann die eigentliche Methode. Daher ist ein Aufzeichnen hier nicht wirklich sinnvoll, weil maximal die sowieso schon geschriebenen Testdaten aufgezeichnet werden könnten.

Ansonsten hat der Ansatz keine größeren Einschränkungen. Einzig bei Wartungsprojekten hat sich das Initialisieren der Testumgebung in den generierten Test als aufwändiger herausgestellt als gedacht.

Interessant wird der Ansatz deshalb auf neu entwickelten Projekten, da auf eine gute Integration des Testrecorders geachtet werden kann. Gerade auch das Aufzeichnen von Service Schnittstellen ist sehr effizient realisierbar, da meist eine zustandslose Aufzeichnung ausreicht. Dadurch ist der Ansatz auch gut geeignet um beispielsweise SOA Anwendungen aufzuzeichnen.

4.7 Bewertung der Test-Methode

Initialisierungs-Problematik Gerade bei Projekten die schon existieren, gibt es oft Nebeneffekte, welche auf eine fehlerhafte Initialisierung zurückgehen. Viele Methoden greifen hier auf statische Objekte zu, welche vor der Testausführung initialisiert werden müssen.

Äquivalenzklassen werden nicht direkt getestet, wobei natürlich vom Benutzer an der Oberfläche meist nicht einfach wahllos Daten eingeben werden, sondern hier schon eine definierte Menge an gültigen und ungültigen Werten eingeben wird. Wieviele dieser Werte wirklich an der aufzuzeichnenden Methode ankommen, hängt stark von der Art des Fehlers und den darüberliegenden Schichten ab.

Wird in ein Textfeld die Eingabe bereits in der Oberfläche auf Zahlen geprüft, kann in einer aufzuzeichnenden Methode kein falscher String als Testdaten hereinkommen, und somit nicht als Fehlerfall aufgezeichnet werden.

4.8 Zu Java

Neben den Ergebnissen zum Prototypen sind dabei auch Grenzen der eingesetzten Programmiersprache Java aufgefallen.

getMethod Der Zugriff auf Field- oder Method Instanzen ist nur über dem Einsatz von Gettern möglich. Dabei wird hier der Name als Zeichenfolge übergeben. Diese Namen werden daher bei Refactoring Maßnahmen von Werkzeugen übersehen.

Type erasurement Beim Einsatz von Generics wäre es in vielen Fällen interessant, Zugang zum Typ der Klasse zu bekommen. Die Laufzeitumgebung von Java kennt allerdings keine Generics, weswegen ein `List<String>` zur Laufzeit als einfache `List` auftaucht. Dadurch kann zur Laufzeit keine Überprüfung von Übergabeparametern mit instanceof erfolgen. Dies führt beispielsweise bei der DAO Implementierung dazu, dass es mehrere Klassen gibt, welche lediglich eine einzige Template Method enthalten.

5 Kapitel
Diskussion

In diesem Kapitel wird auf weiterführende Ansätze eingegangen, welche während der Untersuchungen aufgekommen sind. Das Kapitel ist in zwei Unterabschnitte aufgeteilt: Verbesserungen durch ein Werkzeug und Verbesserungen im erstellten Framework.

5.1 Verbesserung durch Werkzeug

Als Nachteil für den täglichen Einsatz hat sich der Aufwand beim Konfigurieren des Testrecorders bzw. Testgenerators durch ein Java-Modell herausgestellt. Prinzipiell könnte die Konfiguration auch über eine XML Datei erfolgen, da die Konfigurations-Modelle bereits Methoden zum Laden und Speichern bereitstellen. Dies erhöht aber nicht die Bedienerfreundlichkeit, da anstatt Javacode XML editiert werden muss.

Interessant wäre ein Ansatz bei dem ohne viel Aufwand die Konfiguration erstellt werden kann. Hierzu bietet sich eine Integration in Entwicklungsumgebung wie beispielsweise Eclipse an. Dadurch könnte durch einfaches “Zusammenklicken” die Konfiguration erstellt werden.

Die folgenden Abschnitte beleuchten welche Möglichkeiten eine Integration eines solchen Werkzeuges in eine Entwicklungsumgebung bieten könnte.

5.1.1 Erweiterungen

Als Framework kann der Prototyp bereits mit eigenen Erweiterungen um Funktionalität ergänzt werden. Für ein Werkzeug ist daher der gesamte Umfang an Funktionalität nicht genau bekannt. Ein Benutzer könnte beispielsweise eine eigene Erweiterung einsetzten. Somit ist es für ein Werkzeug schwierig, die tatsächlich verfügbare Funktionalität in einer Oberfläche zu repräsentieren.

Damit ein Werkzeug in der Lage ist seine Oberfläche dynamisch an die verfügbaren Funktionen anzupassen müssen Erweiterungen bereits deskriptive Informationen mitbringen. Hierfür bietet sich eine Plugin-Struktur. Dies hat den Vorteil dass Werkzeuge nachträglich einfach erweitert werden können.

Eclipse-Plugin

Da das Werkzeug schon als Erweiterung entwickelt wird, beispielsweise als Eclipse-Plugin, kommt als Lösung auch ein Einbinden in die gleichen Strukturen in Betracht. Ein Eclipse-Plugin kann wieder

durch weitere Eclipse-Plugins erweitert werden. Jede für das Framework entwickelte Erweiterung muss dann allerdings auch ein zusätzliches Eclipse-Plugin mitliefern. Dadurch ist eine starke Abhängigkeit gegenüber Eclipse eingeführt.

Dieses Vorgehen kommt auch schwer mit Änderungen zurecht, da bei jeder Änderung an den Erweiterungen auch die lokale Eclipse Installation aktualisiert werden muss. Für projektspezifische Erweiterungen stellt dies einen hohen Aufwand dar.

jar-Plugin

Da Erweiterungen am Framework als direkte Abhängigkeiten in das Zielprojekt eingetragen werden, bietet es sich an dies auszunutzen. Ein als Eclipse Plugin ausgelegtes Werkzeug könnte die Abhängigkeiten des aktuellen Projektes einlesen und aus den jar-Archiven zusätzliche Informationen beziehen. Dadurch wird die Plugin Struktur auch unabhängig vom eigentlichen Werkzeug, eine Netbeans-Integration könnte die gleiche Schnittstelle benutzen.

Eine derartige Plugin-Struktur ist experimentell schon implementiert und wird über die in Abbildung 5.1 dargestellten Klassen bereitgestellt.

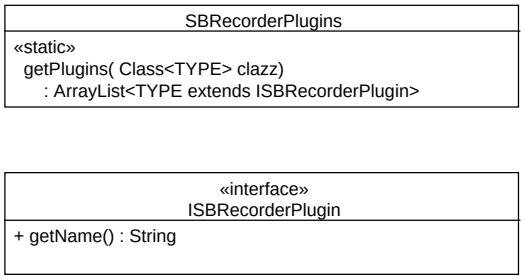

Abb. 5.1: Hilfsklassen für eine Plugin Architektur

Die `SBRecorderPlugins` Klasse stellt durch die Methode `getPlugins` eine Möglichkeit bereit, eine Liste aller verfügbaren Klassen zu erhalten, die eine angegebene Schnittstelle implementieren. Durch Definieren entsprechender Schnittstellen könnten so Werkzeuge angebunden werden.

Die Liste wird nicht durch das Durchlaufen aller vorhandenen Klassen erzeugt, sondern durch eine Dateistruktur. Hierzu muss im META-INF Verzeichnis ein Unterverzeichnis "service" vorhanden sein, in welchem für jede bereitgestellte Schnittstelle eine Textdatei mit dem konkreten Klassennamen liegt.

Möchte beispielsweise das *Addons* Projekt eine Plugin Klasse bereitstellen, muss in dessen META-INF-Verzeichnis ein Ordner mit dem Namen "service" angelegt werden. In diesem Order wird nun eine Datei mit dem Dateinamen "net.seitenbau.testing.shared.plugin.ISBRecorderPlugin" angelegt. Der Dateiname entspricht dabei dem FQN der Schnittstelle. In dieser Datei wird nun der FQN aller Klassen eingetragen, welche diese Schnittstelle implementieren. Beispielsweise:

```
1 net.seitenbau.testing.addons.plugin.DbUnitXStreamSweetner
```

SBRecorderPlugins liefert von nun an ein Class-Objekt für den `DbUnitXStreamSweetner` zurück. Diese kann instanziert werden und von einem Werkzeug zur Introspektion genutzt werden. Durch dieses Vorgehen können Erweiterungen verschiedene Schnittstellen implementieren durch welche das Werkzeug Auskunft über deren Fähigkeiten erhält.

Da das META-INF Verzeichnis auch gleich in ein gepacktes jar-Archiv eingebettet wird, funktioniert dieses Vorgehen auch dann noch. Dadurch können jar-Archive über ein firmeninternes Maven Repository verteilt werden. So ist es sehr einfach möglich projektspezifische Erweiterungen innerhalb des Projektteams zu verteilen. Bei Änderungen an der Erweiterung ist keine Änderung an der lokalen Eclipse Installation nötig. Es reicht ein Basis-Eclipse Plugin zu installieren. Alle weitere Funktionalität ist projektspezifisch und kann automatisch auf dem aktuellen Stand gehalten werden.

Für die im Folgenden vorgestellten Oberflächen wird ein derartiges System angenommen.

5.1.2 Recorder Generator

Das Generieren des Recorders besteht aus dem Auswählen einer Methoden und dem Konfigurieren der aufzuzeichnenden Daten.

Das Auswählen der aufzuzeichnenden Methode könnte elegant durch ein Eclipse-Plugin gelöst werden. Abbildung 5.2 zeigt wie aus der “Package Explorer” Ansicht über ein Context-Menü eine Methode ausgewählt werden kann.

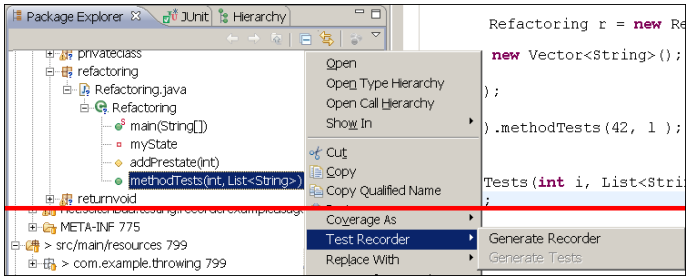

Abb. 5.2: Grafische Auswahl einer Methode

Das Eclipse Plugin kann direkt auf die von Eclipse bereitgestellten Informationen zugreifen und automatisch die Methodensignatur erkennen. Dies stellt bei komplexeren Methoden eine Vereinfachung dar, da so die FQNs der Methodensignatur nicht manuell zusammenzusetzen sind.

Ein beispielhafter Dialog könnt dann, ähnlich Abbildung 5.3, für die Konfiguration der aufzuzeichnenden Daten erscheinen.

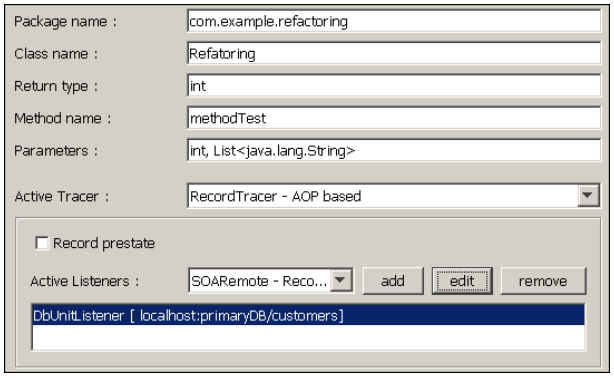

Abb. 5.3: Recorder grafisch konfigurieren

Einzig die Methodenkonfiguration ist durch das Framework fest vorgegeben. Die verfügbaren Tracer und Listener variieren jeweils, weswegen sich der Einsatz der vorgeschlagenen Plugin-Architektur anbietet.

Hier sind Strategien zu überlegen, wie sich eine jeweilige Erweiterung gut integrieren lässt. Für das Framework reicht ein Ableiten der Konfiguration von der `IListenerConfiguration` Schnittstelle. Die Plugin Schnittstelle muss eine Möglichkeit vorsehen um die nötigen Eingabefelder dem Werkzeug mitzuteilen. Dies kann von einer einfachen Liste an Eingabefeldern bis zu ganzen Dialogen gehen.

Die im Dialog gemachte Konfiguration werden im Hintergrund als `RecordConfiguration` aufgebaut und können direkt zur Generation der Instrumentations-Artefakte genutzt werden.

5.1.3 Test Auswahl

Die aufgezeichneten Testdaten liegen zunächst alle im Repository. Der Prototyp stellt durch den Repository-Service eine Möglichkeit bereit auf diese Testdaten zuzugreifen. Dies erfolgt allerdings programmatisch und ist daher benutzerunfreundlich, da es nur sehr schwierig ist gezielt einzelne Testdaten auszuwählen. Ähnlich Abbildung 5.4 bietet sich die Möglichkeit einer grafischen Aufbereitung des Repositories als Baumstruktur an.

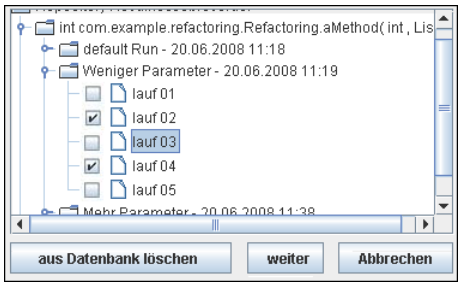

Abb. 5.4: Grafische Darstellung des Repositories

Ausgewählte Testfälle könnten einfach per Klick selektiert und dann gelöscht werden oder als Basis für eine Testgeneration dienen.

Neben einer rein grafischen Unterstützung ist auch eine Hilfestellung bei der Auswahl von Testfällen vorstellbar, beispielsweise durch das Finden von Duplikaten. Gerade bei vielen aufgezeichneten Testfällen treten Testdaten auf, welche identisch sind und daher nur einmal als Test generiert werden sollten.

Neben der Erkennung einfacher Duplikate sind auch kompliziertere Ansätze vorstellbar wie das Finden von ähnlichen Testfällen mithilfe einer Cluster-Analyse. Bei geeigneter Distanzfunktion können hier Testfälle mit ähnlichen Daten automatisiert gruppiert werden.

5.1.4 Test-Generator

Wie schon beim Generieren des Testrecorders sind auch bei den Tests unterschiedliche Konfigurationsoptionen zu setzen. Auch hier bietet sich die Eingabe in einem Dialog an.

Von Interesse wäre auch eine Abfragemöglichkeit, mit der ein Part-Generator bereits vor der Generierung testen kann, ob die ausgewählten Testdaten zur Generierung ausreichen oder ob mangels nicht aufgezeichneter Testdaten abgebrochen wird. So kann beispielsweise der DBUnit-Generator nicht ausgeführt werden, wenn während der Testaufzeichnung kein DBUnit-Listener aktiv war.

Da meist eine wiederkehrende Kombinationen von Generatoren genutzt werden wird, könnte es interessant sein diese als Profil vorzuhalten. Dadurch würde sich der Konfigurationsaufwand je Testgeneration noch wesentlich vereinfachen. Anstatt jedesmal mehrere Generatoren auszuwählen, reicht es einfach das Profil “Integrationstests” zu aktivieren.

5.1.5 Test Ausführung

Das Ausführen der Tests wird durch den Einsatz von JUnit zwar grafisch aufbereitet, trotzdem sind auch hier noch unterschiedlichste Erweiterungen denkbar.

Fehlermeldungen von XML-Vergleichen werden als Ausnahmen an JUnit gemeldet und wie alle Asserts dargestellt. Die Darstellung enthält zwar einen Vergleich zwischen dem aktuellen und dem erwarteten XML Element, wird aber bei größeren Bäumen schnell unübersichtlich.

Für fehlgeschlagene Vergleiche bietet es sich an das XML als Baum darzustellen (Abbildung 5.5). Dies erhöht die Lesbarkeit und es könnte direkt aus der Oberfläche heraus ein Element gewählt und in eine Filterkonfiguration zum Ausschluss übernommen werden.Optimal auch mit einer Funktion den Filter gleich auf mehrere Testfälle anzuwenden, da mit einer hohen Wahrscheinlichkeit ein fehlgeschlagenes Feld auch in anderen Testfällen zu einem Fehler führt.

Abb. 5.5: Filterung von Feldern

5.1.6 Integration in Maven

Neben einer Integration in eine Entwicklungsumgebung wie Eclipse erscheint auch eine Integration in Maven interessant.

Beispielsweise wäre das Generieren des Testrecorders basierend auf einer XML-Konfiguration vorstellbar. Dadurch könnte die Instrumentation on-the-fly erfolgen und für ein Projekt wäre nur eine XML-Konfiguration vorzuhalten, denn Aspekte und Ressourcen lassen sich komplett aus der XML-Konfiguration generieren.

Ähnliches gilt für den Testgenerator. Hier ist allerdings zusätzlich zur Konfiguration auch spezifizieren der Testdaten nötig. Da diese aus dem Repository kommen, benötigt das Maven-Plugin Zugang zu diesem. Daher muss ein effizienter Weg gefunden werden die zu generierenden Testfälle zu spezifizieren.

Mangels Dateien sind die on-the-fly erzeugten Tests auch nicht im Quellcodeverwaltungssystem, weswegen Modifikationen am Test nicht möglich sind.

Interessant wäre es auch die Integration von Maven mit AJDT zu erhöhen denn hier gibt es einen Aufholbedarf. Das verfügbare Maven Plugin synchronisiert nur die wenigsten Features von Maven nach AJDT.

5.2 Verbesserung der Methode

Neben einer Unterstützung durch ein Werkzeug gibt es hier Vorschläge zur Verbesserung der Methode durch den Prototypen.

5.2.1 Copy-on-write

Wie gesehen kann es beim Sichern größerer Objektbäumen zu Speicherproblemen kommen. Meist liegt dies an einem tiefen Objektbaum. Die Summe der einzelnen Elemente verbraucht zuviel Platz für eine komplette Serialisierung. Da oft nicht alle Elemente eines Parameters wirklich ausgewertet werden, bietet es sich an nur die tatsächlich genutzten Elemente zu speichern.

Da ein manuelles Ausschließen aufwändig ist, könnte als automatisierbare Lösung ein Ansatz ähnlich einem Copy-on-write (COW) genutzt werden.

Bei einem COW wird das eigentliche Kopieren eines Feldes erst bei dem ersten Schreibzugriff durchgeführt. Dadurch ist der Kopiervorgang performant und es können effizient größere Datenstrukturen kopiert werden.

Das Vorgehen basiert auf der Annahme, dass von einem übergebenen Parameter nicht alle enthaltenen Objekte auch tatsächlich genutzt werden. Anstatt den ganzen Parameter zu speichern, würde es ausreichen nur die tatsächlich in der Zielmethode genutzten Daten zu sichern.

Erreicht werden könnte ein derartiges Vorgehen durch einen Proxy. Anstelle des echten Parameters

wird ein dynamischer Proxy an die Zielmethode übergeben. Zugriffe auf den Parameter werden nun durch den Proxy geleitet und können detektiert werden. Bei einem Zugriff kann nun eine Kopie des Inhaltes abgespeichert werden. Nach Durchlaufen der Zielmethode sind jetzt nur die tatsächlich genutzen Elemente gespeichert.

Anstelle des sofortigen Sicherns wären es auch denkbar erste Trockenläufe durchzuführen um die am häufigsten benötigten Elementen zu isolieren. Im aktiven Aufnahmemodus wird dann für jeden Methodenaufruf die so erkannten Elemente gespeichert.

Dieses Vorgehen hat den Vorteil, dass für einen Methodenaufruf die üblichen Daten gesichert werden und nicht nur die spezifischen Daten für den einen Aufruf. Denn in beiden Fällen ist der gespeicherte Parameter nicht vollständig. Weswegen während der Testausführung kein vollständiges Objekt wiederhergestellt werden kann. Bei späteren Änderungen der Implementierung kann dies zu Problemen führen wenn die gleiche Funktionalität über ein anderes Element erreicht wird, welches aber nicht gespeichert wurde.

Mit XStream ist es zwar möglich Felder auszuschließen, also eine blacklist zu definieren, für das vorgeschlagene Vorgehen ist aber eine whitelist praktikabler. Daher müsste entweder die XStream Bibliothek angepasst werden oder eine andere Technik gewählt werden.

5.2.2 Remote Zugriff

Die Implementierung mit AOP erzeugt momentan für jede aufzuzeichnende Methode einen spezifischen Aspekt. Der Nachteil bei diesem Vorgehen ist, dass das Hinzufügen oder Entfernen von weiteren aufzuzeichnenden Methoden ein erneutes Compilieren und somit einen Neustart der Anwendung mit erfordert.

Würde anstelle eines spezifischen Aspekts ein allgemeiner genutzt, der auf allen Methoden aktiv ist und direkt beim Methodenaufruf entscheidet, ob eine Aufzeichnung stattfinden soll, könnte einfach zur Laufzeit die Methode geändert werden. Dies könnte über eine RMI-Verbindung von außen gesteuert werden.

Eine interessante Zwischenlösung könnte das Einsetzen eines Aspekts sein, welcher auf allen Methoden in einem package aktiv ist. Dadurch ist es möglich ohne größere Verluste der Perfomance zur Laufzeit die aufzuzeichnende Methode zu ändern.

Mit der Klasse `RmiTracer` ist solch ein experimentieller Tracer implementiert. Die vorhandene Implementierung beherrscht allerdings nur ein Remote logging, zum Ausgeben von Statusinformationen.

Damit der Tracer einen Callback registrieren kann, muss dieser erst in die RMI-Registry exportiert werden. Dies hat allerdings einen kleinen Nebeneffekt: beim Beenden der Anwendung wird weiterhin eine Referenz auf das exportierte Objekt gehalten, weswegen die Java-Laufzeitumgebung nicht beendet wird. Dies kann zu Problemen beim Beenden der Anwendung führen.

Bisher war die Konfiguration der Aufzeichnung in einer XML-Datei abgelegt. Durch eine Fernsteuerung gibt es Abweichungen zwischen der XML Datei und den tatsächlich aufgezeichneten Daten. Eine Lösung wäre das zusätzliche Speichern der Konfiguration im Repository.

5.2.3 Komponententests

Neben den Integrationstests ist es vorstellbar, Komponententests zu erzeugen. Bei Komponententests wird eine Komponente isoliert von ihrer Umgebung getestet. Hierzu wird die Umgebung mithilfe von Mock-Objekten simuliert. Zur einfachen Generierung von Mock-Objekten gibt es verschiedene Frameworks.

Die für die Mock-Objekte benötigten Daten könnten bereits während der Aufzeichnung vom Tracer erfasst werden. Das Laufzeitmodell ist hierfür bereits vorbereitet und für den `RecorderTracer` kann das Aufzeichnen in der `MethodConfiguration` aktiviert werden.

5.2.3.1 Mocking durch Proxy

Zum Erstellen von Mock-Objekten gibt es den Ansatz einen dynamischen Proxy einzusetzen. Die Bibliothek EasyMock macht es möglich für beliebige Schnittstellen ein Mock-Objekt zu erzeugen. Die Bibliothek EasyMock [Freese, 2008] setzt dies um indem ein dynamischer Proxy erzeugt wird, welcher erste alle Methodenaufrufe aufzeichnet um diese dann während der Testausführung wiederzugeben. Das folgende Listing 5.1 zeigt einen einfachen EasyMock Test.

```
1  HelloWorldText mock = createStrictMock( HelloWorldText.class);
2
3  expect( mock.getText("us") ) .andReturn("Hello World");
4
5  replay(mock);
6
7  assertEquals( "Hello World", new ClassUnderTest(mock).getMessage());
8
9  verify(mock);
```

Listing 5.1: Auszug eines EasyMock Testes

In Zeile 1 wird der Proxy für eine `HelloWorldText` Schnittstelle erzeugt. Dieses Proxy Objekt ist nun im “record” Modus, d.h. alle darauf stattfindenden Methodenaufrufe werden aufgezeichnet.

In der Zeile 3 wird so ein Aufruf der `getText` Methode aufgezeichnet, welche später einen Parameter “us” erwartet und den Rückgabewert “Hello World” zurückliefert. Durch Methoden wie `expect` lässt sich das gewünschte Verhalten elegant im Quellcode beschreiben. In Zeile 5 wird der Proxy dann in den “replay”-Modus geschalten. Von nun an werden die Aufrufe nicht mehr aufgezeichnet sondern wiedergegeben.

Im Beispiel wird auf der Klasse `ClassUnderTest` die `getMessage` Methode aufgerufen, welche wiederum die `getText` Methode der übergebenen `HelloWorldTest` Instanz aufruft. Der Aufruf findet dadurch also auf dem angelegten Proxy statt, welche nun den Übergabeparameter auf „us“ testet und „Hello World“ zurückliefert. Ob der Aufruf richtig stattgefunden hat, wird durch das AssertEquals in Zeile 7 festgestellt. Bei komplexeren Abläufen kann durch ein verify (Zeile 9) noch sichergestellt werden, dass alle zuvor aufgezeichneten Funktionen auch wirklich aufgerufen wurden.

Wie schon erwähnt ist EasyMock auf Interfaces beschränkt; es gibt aber mit dem *EasyMockClassExtensions* auch eine Version die mit Klassen zusammenarbeitet. Die größere Einschränkung ist aber

sicherlich die Voraussetzungen an den zu testenden Code, denn dieser muss für ein Injizieren des Proxies vorbereitet sein.

5.2.3.2 AOP Mocking

Das Austauschen von Abhängigkeiten, ohne eine Vorbereitung des Programms, kann durch den Einsatz von AOP Techniken erreicht werden. Diesen Weg geht die JMockit [Liesenfeld, 2008] Bibliothek. Diese ersetzt die Bytecode-Repräsentation einer Klasse zur Laufzeit. Das Austauschen erfolgt hier mit einem Java-Agent, welcher daher über einen Startparameter "-javaagent:jmockit.jar" angegeben werden muss.

Für den nachfolgenden JMockit Test werden wie die in Listing 5.2 abgedruckte Klasse genutzt.

```
1  public class HelloWorldText {
2          public String getText(String country) {
3                  // if country equals "us" ... etc.
4                  return "Hello World";
5          }
6  }
7
8  public class ClassUnderTest {
9          public String getMessage() {
10                 return new HelloWorldText().getText("us");
11         }
12 }
```

Listing 5.2: Klasse für das JMockit Beispiel

Der eigentliche Test (Listing 5.3) hat wenig Unterschied zu einem normalen Test. Einzig in Zeile 5 wird JMockit mitgeteilt, wie die Implementierung der Klassen `HelloWorldText` durch die der `HelloWorldTextMock` auszutauschen ist. In Zeile 7 wird die zu testende Klasse gerufen, diese instanziert in ihrem Code weiterhin die `HelloWorldText` Klasse, bekommt aber durch JMockit anstelle dieser die `HelloWorldTextMock` Klasse untergeschoben.

```
1  public class JMockitTest {
2
3          @Test
4          public void aTest2() {
5                  Mockit.redefineMethods(HelloWorldText.class,
                       HelloWorldTextMock.class);
6
7                  assertEquals("Hello World", new ClassUnderTest().
                       getMessage());
8          }
9
10         static public class HelloWorldTextMock {
11                 public String getText(String country) {
12                         assertEquals("us", country);
13                         return "Hello World";
14                 }
```

```
15 |             }
```

Listing 5.3: JMockit Test

Die `HelloWorldTextMock` Klasse kann den Methodenaufruf gegen die erwarteten Werte prüfen (Zeile 12) und "Hello World" zurückliefern (Zeile 13).

Durch den Einsatz von JMockit können auch Programme getestet werden die keine Injektion von Mock-Objekten erlauben. Dadurch wird es möglich aus den aufgezeichneten Testdaten automatisierte Komponententest zu erstellen.

5.2.3.3 Aufzeichnen des AOP Mocks

Ein alternativer Ansatz ist die Integration in die bei der Aufzeichnung eingesetzten Tracer.

Während der Aufzeichnung wird durch den Tracer jeder Aufruf an umgebende Komponenten gespeichert werden. Mit einem geänderten Tracer wäre es möglich das Modell der Aufzeichnung wiederzugeben. Hierzu wurde bereits eine Grundversion unter der Bezeichnung FlowControl implementiert. Dieses ist in Abbildung 5.6 dargestellt.

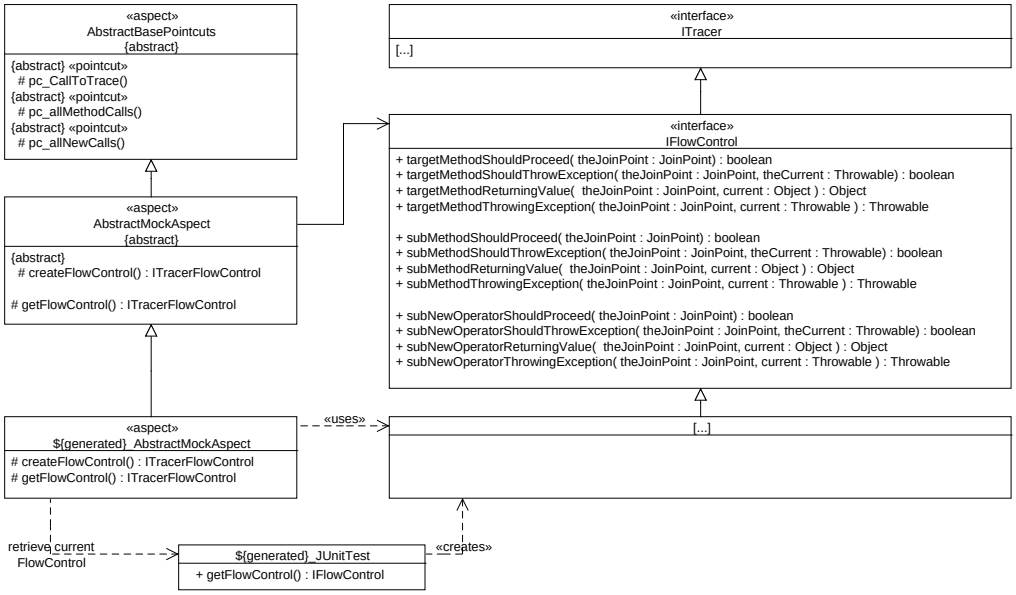

Abb. 5.6: AOP Mock

Der `AbstractMockAspect` ruft anstelle eines Tracers eine `IFlowControl` Instanz auf. Diese Schnittstelle basiert größtenteils auf der ITracer Schnittstelle, erweitert diese aber um Funktionen zum Definieren von Rückgabewerten. Dadurch ist es möglich, beim Aufruf einer Untermethode diese abzufangen und ein zuvor aufgezeichnetes Objekt als Mock-Objekt zurückzuliefern.

Der Ansatz ist mit dem `AbstractMockAspect` bereits implementiert. Einzig für die `IFlowControl` Schnittstelle gibt es nur eine `DefaultFlowControl` Implementierung, welche kein Mocking durchführt, sondern einfach die Originalmethoden aufruft. Hier wäre es also nötig einen FlowControl zu implementieren der die aufgezeichneten Objekte zurückliefert.

Dabei ist zu beachten, dass während der Aufzeichnung je Untermethodenaufruf die Objekte serialisiert wird. Wurden dabei zwei Methodenaufrufe aufgezeichnet, welche die gleiche Instanz eines Objektes übergeben bekommen, werden diese getrennt gespeichert und später von XStream in zwei unterschiedliche Instanzen wiederhergestellt. Dadurch werden in diesem Falle direkte Vergleiche der Objekte fehlschlagen. Aber auch beim Einsatz der equals-Methode kann dies zu ungewollten Effekten führen, nämlich dann, wenn die equals-Methode nicht überschrieben wurde, und so die Standard-Implementierung mit der hashcode Methode genutzt wird.

Zum Aufheben dieser Problematik ist es nötig die Referenznummern von Objekten bereits während der Testaufzeichnung zu speichern. Diese Information kann dann bei der Testaufführung dazu genutzt werden, um identische Objete zu erkennen und korrekt zu behandeln.

Das Speichern der Objekt-Referenz ist durch die `ObjectSnapshotMetaInfo` Klasse bereits vorgenommen. Auf der Testseite fehlt hier aber eine entsprechende Behandlung.

5.2.4 Ausnahmen

In Listing 5.4 ist ein generierter Test abgedruckt welcher überprüft, ob eine Ausnahme geworfen wurde. Gegenüber der in Listing 5.5 abgedruckten JUnit4 -Test ist dieser schwieriger zu lesen.

```
1  @DataDrivenTest(SBRecorderDataProvider.class)
2  @XmlDataOption(file = "InvokeThrowingClassTest.xml")
3  public void throwingTest( int param1 ) throws Exception {
4    Throwable actualException = null;
5
6    try {
7      testInstance.calculate(param1 );
8    } catch (Throwable theThrowable) {
9      actualException = theThrowable;
10     assertEquals(true, testContext().isException());
11
12     ObjectSnapshot expectedException = ...;
13     testContext.applyXmlFiltersAndAssertEquals(expectedException,
           actualException);
14   }
15 }
```

Listing 5.4: Parameterized Test für Ausnahmen

```
1  @Test(expected=NullPointerException.class)
2  public void nullPtrTest() {
3      Object obj = null;
4      obj.toString();
5  }
```

Listing 5.5: JUnit 4 Test mit erwarter Ausnahme

Ein direktes Umsetzen des JUnit-Tests für den Parameterized-Test-Runner ist schwierig, da der JUnit-Test immer die gleiche Ausnahme erwartet. Da der JUnit-Test nur einen Aufruf der Zielmethode durchführt ist dies dort kein Problem. Eine Annotation am Parameterized-Test, wie in Listing 5.6, stellt allerdings keine gute Lösung dar.

```
1 @DataDrivenTest(SBRecorderDataProvider.class,
2     expected=NullPointerException.class )
3 @XmlDataOption(file = "InvokeThrowingClassTest.xml")
4 public void throwingTest( int param1 ) throws Exception {
5   ...
```

Listing 5.6: Parameterized Test mit expected

Denn in dieser Version würde von allen Testdaten eine `NullPointerException` erwartet (Zeile 2). Da aber bei einem Parameterized Test sehr viele unterschiedliche Testdaten durchlaufen werden, wird es Testdaten geben, die überhaupt keine Ausnahme werfen oder die eine andere Klasse werfen.

Eine elegante Lösung ergibt sich, wenn die `expected` Deklaration in die XML Datei verlagert wird.

Der Testcode unterscheidet dann nicht mehr zwischen Test mit und ohne Ausnahmen. Einzig das Auftreten einer Ausnahme wird durch den Parameterized-Test-Runner geprüft. Die expected Information ist dann in der Testdaten-Datei wie in Listing 5.7 zu finden.

```
1 <test id="42" name="TestMitAusnahme">
2   <parameters><int>23</int></parameters>
3   <expected isException="true">
4     <expectedException>
5       <java.lang.NullPointerException>
6         <detailMessage>Was 23</detailMessage>
7       </java.lang.NullPointerException>
8     </expectedException>
9   </expected>
10 </test>
```

Listing 5.7: Parameterized Test Daten mit expected Eintrag

Dadurch ist bei Generieren der Tests kein Unterschiedung im JUnit-Test zu treffen.

6 Kapitel
Schlussfolgerungen und Ausblick

Es konnte gezeigt werden, dass das Recorded Test Muster für eine Aufzeichnung von Integrationstests auf Systemtestebene umgesetzt werden kann. Neben einer grundlegenden Bewertung hat sich die qualitative Bewertung auf bestehenden großen Projekten als schwierig herrausgestellt. Gerade das Erfassen und Wiederherstellen des Systemzustandes ist in vielen Fällen aufwändig und projektspezifisch. Einzig zustandslosen Methoden können gut aufgezeichnet werden, was den Ansatz für SOA Architekturen interessant macht.

Insgesamt ist der Ansatz nicht ohne Anpassungen in beliebigen Software Architekturen einsetzbar. Ein rein automatisiertes Anwenden ist nicht möglich, da bereits während der Aufzeichnung eine grobe Vorstellung über die gewünschten Tests vorliegen muss. Auch das Generieren der Tests erfordert ein fundiertes Wissen über Werkzeug und Projekt. Wie schon [Meszaros, 2007] anmerkte, ist daher in der Praxis eine Bedienung durch einen Entwickler oder dedizierten Tester erforderlich.

In Zukunft sollte vor allem Wert auf die Benutzbarkeit durch eine Integration in Entwicklungsumgebungen gelegt werden. Durch eine grafische Benutzeroberfläche und die zugrundeliegende Flexibilität des Frameworkes, könnten effizient projektspezifische Anpassungen vorgenommen werden. So kann der Testrecorder einen Platz im Werkzeugkasten eines Testers finden.

Literaturverzeichnis

[Agitar Software 2008] Agitar Software, Inc: *AgitarOn*. 2008. – URL http://www.agitar.com/

[Ambler 2003] Ambler, Scott W.: *Agile Database Techniques*. Wiley & Sons, 2003. – URL http://www.agiledata.org/essays/mappingObjects.html

[Andreas Spiller 2005] Andreas Spiller, Tino L.: *Basiswissen Softwaretest*. dpunkt.verlag, 2005

[Andrew Hunt 2003] Andrew Hunt, David T.: *Der Pragmatische Programmierer*. Hanser, 2003

[Bacon und Martin 2008] Bacon, Tim ; Martin, Jeff: *<xml-unit>*. 2008. – URL http://xmlunit.sourceforge.net/

[Baliuka u. a. 2008] Baliuka, Juozas ; Nokleberg, Chris ; Ramoska, Matas ; Biggs, Wes: *cglib*. 2008. – URL http://cglib.sourceforge.net/

[Baranowski 2007] Baranowski, Christian: *Einführung von automatisierten Tests in das SEITENBAU intramo Projekt*, HTWG - Konstanz, Diplomarbeit, 2007

[Barchfeld u. a. 2008] Barchfeld, Markus u. a.: *Build and Test Automation for plug-ins and features*. 2008. – URL http://www.eclipse.org/articles/Article-PDE-Automation/automation.html

[Beck 1994] Beck, Kent: Simple Smalltalk Testing: With Patterns. In: *The Smalltalk Report* vol. 4, no. 2 (1994), S. s. 16–18. – URL http://www.xprogramming.com/testfram.htm

[Beck 2003] Beck, Kent: *Test-Driven Development By Example*. Addison-Wesley, 2003

[Böhm 2006] Böhm, Oliver: *Aspektorientierte Progammierung mit AspectJ5*. dpunkt.verlag, 2006

[Chapman u. a. 2008] Chapman, Matt u. a.: *AJDT*. 2008. – URL http://www.eclipse.org/ajdt/

[Chiba 2008] Chiba, Shigeru: *Javassist*. 2008. – URL http://www.csg.is.titech.ac.jp/~chiba/javassist/

[Cole u. a. 2008] Cole, Oliver u. a.: *Test and Performance Tools Platform Project*. 2008. – URL http://www.eclipse.org/tptp/

[Colyer u. a. 2008] Colyer, Adrian u. a.: *AspectJ*. 2008. – URL http://www.eclipse.org/aspectj/

[Elliotte Rusty Harold 2005] Elliotte Rusty Harold, W. Scott M.: *XML in a Nutshell*. O'Reilly, 2005

[eviware software ab] eviware software ab: *soapUI; the Web Services Testing tool*. 2008. – URL http://www.soapui.org/

[Fialli und Vajjhala 2003] Fialli, Joe ; Vajjhala, Sekhar: *JSR31: Java Architecture for XML Binding*. 2003. – URL http://www.jcp.org/en/jsr/detail?id=31

[Fowler 2004a] Fowler, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. 2004. – URL http://martinfowler.com/articles/injection.html

[Fowler 2004b] Fowler, Martin: *UML Distilled : A Brief Guide To The Standard Object Modelling Language*. Addison-Wesley, 2004

[Fowler 2005] Fowler, Martin: *Refactoring*. Addison-Wesley, 2005

[Fowler und Foemmel 2000] Fowler, Martin ; Foemmel, Matthew: Continuous Integration. (2000). – URL http://www.martinfowler.com/articles/originalContinuousIntegration.html

[Freese 2008] Freese, Tammo: *EasyMock*. 2008. – URL http://www.easymock.org/

[Gamma u. a. 1995] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster*. Addison-Wesley, 1995

[Glass-Husain u. a. 2008] Glass-Husain, Will u. a.: *Apache Velocity*. 2008. – URL http://velocity.apache.org/

[Huggins u. a. 2008] Huggins, Jason u. a.: *Selenium*. 2008. – URL http://selenium.openqa.org/

[Johnson u. a. 2008] Johnson, Rod u. a.: *Spring Framework*. 2008. – URL http://www.springframework.org/

[Kawaguchi 2006] Kawaguchi, Kohsuke: *JSR222: Java Architecture for XML Binding*. 2006. – URL http://www.jcp.org/en/jsr/detail?id=222

[Kiczales u. a. 1997] Kiczales, Gregor u. a.: A s p e c t - O r i e n t e d P r o g r a m m i n g. (1997). – URL http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf

[King u. a. 2008] King, Gavin u. a.: *Hibernate*. 2008. – URL http://www.hibernate.org/

[Liesenfeld 2008] Liesenfeld, Rogério: *The JMockit Testing Toolkit*. 2008. – URL https://jmockit.dev.java.net

[McCabe 1976] McCabe, Thomas J.: *A Complexity Measure*. 1976. – URL http://www.literateprogramming.com/mccabe.pdf

[Merks u. a. 2008] Merks, Ed u. a.: *Eclipse Modeling Framework*. 2008. – URL http://www.eclipse.org/modeling/emf/

[Meszaros 2007] Meszaros, Gerard: *xUnit Test Patterns*. Addison-Wesley, 2007

[Oesterreich 2006] Oesterreich, Bernd: *Analyse und Design mit UML 2.1*. Oldenbourg, 2006

[Safabakhsh 2006] Safabakhsh, Borna: *TPTP API Recorder Tutorial*. 2006. – URL http://www.eclipse.org/tptp/test/documents/tutorials/API_Recorder/API_Recorder_Tutorial.htm

[Schaible u. a. 2008] Schaible, Jörg u. a.: *XStream*. 2008. – URL http://xstream.codehaus.org/

[Sun Microsystems Inc. 2004] Sun Microsystems Inc.: *Java Platform Debugger Architecture*. web. 2004. – URL http://java.sun.com/javase/technologies/core/toolsapis/jpda/

[Wunderlich 2005] Wunderlich, Lars: *Aspektorientierte Programmierung in der Praxis*. entwickler.press, 2005

[van Zyl u. a. 2008] Zyl, Jason van u. a.: *Maven*. 2008. – URL http://maven.apache.org/

Abbildungsverzeichnis

Abkürzungen

ajc AspectJ compiler

AJDT AspectJ Development Tools

AOP Aspect Oriented Programming

DRY Don't Repeat Yourself

EMF Eclipse Modeling Framework

GUI Graphical User Interface : Grafische Benutzerschnittstelle

HTTP Hyper Text Transfer Protocol

JDBC Java Database Connectivity

JDK Java Development Kit

JEE Java Platform, Enterprise Edition

JET Java Emitter Templates

JSP Java Server Pages

JVM Java Virtual Machine

SOA service oriented architecture : Serviceorientierte Architektur

SUT System under Test
Bezeichnung für das System das getestet werden soll.

XML Extensible Markup Language

XSL Extensible Stylesheet Language

XSLT XSL Transformation

A Anhang

A.1 Quellcodes

AspectJ Performance Test

Für das Testen der AspectJ Performance wurden die folgende Klasse und der folgende Aspekt eingesetzt.

Java - Klasse

```
1  package com.example.perf;
2
3  import java.util.List;
4  import java.util.Vector;
5
6  public class AspectJSpeed {
7
8          private static final int RUNS = 100;
9          private static final int LOOPS = 100000;
10         private static final int DEPTH = 9; // 0 == keine weiteren
               Unteraufrufe
11
12         public static void main(String[] args) {
13                 AspectJSpeed obj = new AspectJSpeed();
14
15                 List<Long> elap=new Vector<Long>();
16
17                 obj.run(); // mögliche Caches füllen
18
19                 for(int i=0;i<RUNS;++i) {
20                         long elapsedTime=obj.run();
21                         elap.add( elapsedTime );
22                         System.out.println("Lauf " + i + " elapsed :" +
                               elapsedTime);
23                 }
24                 Long sum=0L;
25                 for (Long elapsed : elap) {
```

```
26                          sum+=elapsed;
27                  }
28                  System.out.println("Statistik für " + RUNS + " Läufe mit
                        jeweils " + LOOPS + " Unteraufrufe :");
29                  System.out.println("Summe         : " + sum +"\tms");
30                  System.out.println("Durchschnitt  : " + sum/elap.size() +"
                        \tms");
31                  System.out.println("RUNS          : " + RUNS);
32                  System.out.println("LOOPS         : " + LOOPS);
33                  System.out.println("DEPTH         : " + DEPTH);
34          }
35
36          private long run() {
37                  long startTime = System.currentTimeMillis();
38
39                  long total = 0;
40                  for (int i = 0; i < LOOPS; i++)
41                  {
42                          String value=callSubMethodCall(0,"einWert");
43                          if( !value.equals("AspektFound") ) {
44                                  System.err.println("Aspect was not Active
                                        !!!" +value);
45                          }
46                          total += i;
47                  }
48
49                  long stopTime = System.currentTimeMillis();
50                  return stopTime - startTime;
51          }
52
53          private String callSubMethodCall(int depth,String param) {
54                  if(depth==DEPTH) {
55                          return subMethodCall(param);
56                  } else {
57                          return callSubMethodCall(depth+1, param);
58                  }
59          }
60
61          private String subMethodCall(String param) {
62                  return param + "1";
63          }
64
65  }
```

Listing A.1: Java Klasse für Performance Tests

Aspekt

```
1  package com.example.perf;
2
3  public aspect AspectJSpeedAspect {
4          pointcut pc_Catch() :
```

```
5                           execution(* *AspectJSpeed.*(..))
6   //                      execution(String com.example.perf.AspectJSpeed.
        subMethodCall(String))
7                           && !withincode(* *AspectJSpeedAspect(..))
8                           && !withincode(* *main(..))
9                           ;
10
11          pointcut pc_Null() : call(null null(null) ); // Deaktiviert
                Aspekt
12
13          Object around() : pc_Null() {
14                  Object obj=proceed(); // Orginal aufrufen
15                  if( thisJoinPoint.getThis()!=null && thisJoinPoint.
                        getThis().getClass() == AspectJSpeed.class) {
16                      if( thisJoinPoint.getSignature().toLongString()
17                                  .equals(
18                              "private java.lang.String com.example.
                                    perf.AspectJSpeed.subMethodCall(java.
                                    lang.String)"
19                                  )
20                              ) {
21                              return "AspektFound";
22                      }
23                  }
24                  return obj;
25          }
26
27          Object around() : pc_Catch() {
28                  Object obj=proceed(); // Orginal aufrufen
29                  return obj;
30          }
31  }
```

Listing A.2: Aspekt für Performance Test

JUnit Parameterized Test

Folgendes Listing zeigt einen JUnit-Parameterized-Test.

```
1   [...]
2
3   @RunWith(Parameterized.class)
4   public class XStreamControlCharTest {
5
6     private XStream myXStream;
7     private String myOrginalString;
8     private String myExpectedXml;
9
10    @Before
11    public void before() {
12      myXStream = new XStream(new HideControlCharXppDriver());
```

```
13    }
14
15    public XStreamControlCharTest(String orginalString, String expectedXML)
          {
16      myOrginalString = orginalString;
17      myExpectedXml = expectedXML;
18    }
19
20    @Parameters
21    public static Collection getMocks()
22    {
23      return
24        Arrays.asList(new Object[][] {
25         /* 0 */{"HelloWorld", "<string>HelloWorld</string>"}
26         /* 1 */, {"Hello World", "<string>Hello World</string>"}
27         /* 2 */, {"  Hello World", "<string>  Hello World</string>"}
28         [...]
29         /* 21 */, {"\\{0}", "<string>\\\\{0}</string>"}}
30        );
31    }
32
33    @Test
34    public void runTest()
35    {
36      String xml = myXStream.toXML(myOrginalString);
37      assertEquals(myExpectedXml, xml);
38
39      String neu = (String) myXStream.fromXML(xml);
40      assertEquals(myOrginalString, neu);
41    }
42  }
```

Listing A.3: JUnit Parameterized Tst

A.2 Tabellen

Folgende Tabelle Zeigt die gemessenen Daten für die Performance Abschätzung.

Lauf:	1	2	3	4
Summe [ms]	3266	5282	8767	38110
Durchschnitt [ms]	32	52	87	381
RUNS	100	100	100	100
LOOPS	100000	100000	100000	100000
DEPTH	9	9	9	9

A.3 Das Dokument

Historie

Datum	Beschreibung
01.08.2008	\setcounter{tocdepth}{2} auf 2 geändert.
01.07.2008	finale Version

Verwendete Werkzeuge

Dieses Dokument wurde Mithilfe LyX[1] geschrieben und durch LaTeX[2] gerendert. Eigene Grafiken wurden mit der Vektorgrafiksoftware Inkscape[3] gezeichnet. UML Diagramme mit UmlLet[4]. Die genutzen Daumen Synmbole stammen von [5].

[1] http://www.lyx.org
[2] http://www.latex-project.org
[3] http://www.inkscape.org
[4] http://www.umlet.com
[5] http://www.designofsignage.com