Technische Fachhochschule Berlin
Fachbereich VI Informatik und Medien

Erweiterung von JUnit um eine Testfallverwaltung

vorgelegt von

Bernd Steindorff

Berlin 2007

verantwortl. Hochschullehrer:	Prof. Dr. Grude
Thema ausgegeben am:	02.05.2007
Abgabe der Arbeit:	01.08.2007
Matrikel-Nr.:	722663

Inhaltsverzeichnis

Einleitung.....4

1. Aufgabenstellung.....6

2. Lizenz der Diplomarbeit.....7

3. Aufbau der Arbeit.....8

4. Grundlagen und Techniken.....9
- 4.1 Java, Umgebung.....9
- 4.2 JUnit.....9
- 4.3 Vorgehensweise Extreme Programming.....14
- 4.4 Datenbank.....20
- 4.5 Sonstige benutzte Software.....21
- 4.6 Alternativen zur Testfallverwaltung.....22

5. Lizenz der Testfallverwaltung.....24

6. Motivation für die Testfallverwaltung.....28
- 6.1 Vorhandene Oberflächen für JUnit 4.....28
- 6.2 Fehlende Historisierung.....29
- 6.3 Sammlungen von Tests.....30
- 6.4 Probleme in nicht-testgetriebenen Projekten.....30
- 6.5 Nutzen der Testfallverwaltung.....31

7. Lösungsansatz.....32
- 7.1 Vorgehensweise.....32
- 7.2 Test-Klassen mit JUnit ausführen.....35
- 7.3 Daten-Modell.....36
- 7.4 Klassen-Ausführer oder Ausführer mit Übersetzer.....36
- 7.5 Klassenlader und Klassenpfad.....36
- 7.6 Persistenz der Test-Ergebnisse.....40
- 7.7 Oberfläche der Testfallverwaltung.....46
- 7.8 Ausnahmen.....47

7.9 Probleme bei der Plattformunabhängigkeit....48
7.10 Auslieferung der Testfallverwaltung....48

8. Lösungsausführung....50
8.1 Vorgehensweise Extreme Programming....50
8.2 Test-Klassen mit JUnit ausführen....54
8.3 Daten-Modell....54
8.4 Klassenlader und Klassenpfad....57
8.5 Persistenz der Test-Ergebnisse....60
8.6 Oberfläche der Testfallverwaltung....64
8.7 Ausnahmen....69
8.8 Probleme bei der Plattformunabhängigkeit....69
8.9 Auslieferung der Testfallverwaltung....69

9. Angaben zur Programmierung....71

10. Bewertung....73

11. Rückblick auf den Verlauf der Arbeit....74

Erklärung....75

Danksagungen....76

Anhang A Benutzte Software....77

Anhang B Literatur....78

Anhang C Dokumentation der Oberfläche....79

Anhang D Sonstige Anhänge....87

Einleitung

„Testgetriebene Entwicklung“ (engl. test-driven development) ist ein Schlagwort in der agilen Software-Entwicklung. Bekannt wurde diese Programmiertechnik unter anderem durch `Extreme Programming [be03]` und `[xp02]` als Vorgehensweise zur agilen Software-Entwicklung.

In der Entwicklung mit Java dominiert das quelloffene Test-Rahmenprogramm (engl. framework) JUnit, welches das Testen im Kleinen (Modultest, engl. unit test) abdeckt - einer von zwei Aspekten der testgetriebenen Entwicklung. Aktuell ist die Version 4, welche eine weitreichende Überarbeitung erfahren hat und auf Java 1.5 umgestellt wurde.

Die alte Version 3 beinhaltete eine eigene graphische Oberfläche für AWT und Swing. JUnit 4 bringt nun keine eigene graphische Oberfläche mehr mit, da die Entwickler von JUnit keinen Sinn darin sehen, in solch einem kleinen Rahmenprogramm auf Swing oder AWT zurückgreifen zu müssen, wenn doch die Integration in Entwicklungsumgebungen (z. B. Eclipse) weit voran geschritten sei.

Die Suche nach eigenständigen graphischen Oberflächen für JUnit 4 blieb im April 2007 erfolglos. Von den großen Java-Entwicklungsumgebungen Eclipse und Netbeans kann seit der Veröffentlichung von JUnit 4 im Februar 2006 nur Eclipse damit Tests ausführen. Betrachtet man Eclipse genauer, dann fällt die schlechtere Unterstützung von JUnit 4 gegenüber JUnit 3 auf. Es lässt sich nur noch eine Test-Klasse ausführen.

Der Entwickler könnte dadurch mit zwei Problemen konfrontiert werden:

- Er verliert die Motivation, jedes Mal alle Test-Klassen einzeln auszuführen.

- Er erstellt nur noch eine Test-Klasse, die vollkommen überladen und unstrukturiert ist.

Neben der fehlenden Möglichkeit, Test-Klassen als Sammlung auszuführen, bietet Eclipse nur eine einfache Historisierung an. JUnit 3 hat dies als einfacher Ausführer gar nicht angeboten. Die Historisierung der Ergebnisse wird dagegen vom Autor für wichtig angesehen. Ein Szenario hierzu:

- Ein Projekt wird mit mehreren Mitarbeitern bearbeitet. Der Quellcode wird über ein System zur Versionsverwaltung (z. B. Subversion) ausgetauscht.

- Es wird testgetrieben entwickelt, d. h. es wird zuerst ein Test erstellt, der dann durch den Programmcode erfüllt wird.

- Vor dem Übertragen des bearbeiteten Quellcodes in die Versionsverwaltung wird das lokale Verzeichnis aktualisiert. Bei einem Mitarbeiter laufen nun 15 von 100 Tests nicht mehr. Eine Problembeseitigung der 15 fehlerhaften Tests wird einige Zeit in Anspruch nehmen. Das Projekt muss überarbeitet werden.

- Die nächsten Überarbeitungsschritte verändern auch das Ausführungsergebnis der Tests. Denkbar ist, dass nach der 1. Überarbeitung nur noch 14 Tests, als nächstes noch 12 und dann wieder 16 scheitern.

Ist nun mehr als ein Test gescheitert, so ist für den Entwickler der Fortschritt bei der Problembeseitigung nur schwer erkennbar. Es fehlt ihm der Überblick über die letzten Test-Ausführungen. Es lässt sich ohne externes Speichern der Testergebnisse nichts über den Fortschritt sagen.

Ziel der Diplomarbeit ist es, JUnit 4 um eine graphische Oberfläche zu erweitern. Diese soll mehrere Test-Klassen ausführen und die Ergebnisse speichern können, sowie die Ergebnisse wieder abrufbar machen.

1. Aufgabenstellung

Es soll eine graphische Benutzeroberfläche in Java erstellt werden, mit der sich JUnit 4-Test-Klassen als Sammlung ausführen und speichern lassen.

Die graphische Oberfläche muss es ermöglichen, eine oder mehrere Test-Klassen auszuwählen. Alternativ muss nach Test-Klassen in einem Klassenpfad gesucht werden können. Schließlich sollen die gefundenen Tests in den Klassen mit JUnit ausgeführt werden.

Die Historisierung soll zusätzlich jedes Ergebnis einer Ausführung speichern und später wieder abrufbar machen. Es sind geeignete Ansichten zu erstellen, um

- Vergleiche zwischen den einzelnen Ausführungsergebnissen machen zu können
- und ein einzelnes Ergebnis einer aktuellen oder gespeicherten Ausführung abrufen zu können.

Die Daten der Test-Klassen und die Ausführungsergebnisse sollen persistent gespeichert werden, d. h. sie sollen über die Programmlaufzeit hinaus bestehen bleiben, um auf die Daten in einer folgenden Programmausführung zuzugreifen.

Da Java als Programmiersprache plattformunabhängig ist, sollte auch die Testfallverwaltung zu möglichst vielen Plattformen kompatibel sein.

2. Lizenz der Diplomarbeit

Diese Diplomarbeit wird unter der Creative Commons Lizenz veröffentlicht. Die Kurzform der Lizenzausgestaltung ist *by-nc-sa*, d.h.

Erlaubt ist:

- das Werk darf vervielfältigt, verbreitet und öffentlich zugänglich gemacht werden,
- das Werk darf bearbeitet werden.

Zu folgenden Bedingungen:

- by: Namensnennung des Autors ist erforderlich,
- nc: Das Werk darf nicht für kommerzielle Zwecke verwendet werden,
- sa: Bei einer Bearbeitung muss das neue Werk unter vergleichbaren Bedingungen veröffentlicht werden.

Weitere Details bei creativecommons.org:

deutscher Kurztext

english license summary

3. Aufbau der Arbeit

Die Diplomarbeit baut auf der Aufgabenstellung in Kapitel 1 auf.

In Kapitel 3 *Grundlagen und Techniken* wird die tatsächlich benutzte Software aufgeführt und erklärt. Dabei wird ein Vergleich existierender Software gezogen und die Vorgehensweise, wie die Testfallverwaltung entwickelt wird, erläutert.

Die Benutzung von freier Software im Rahmen der Testfallverwaltung wird lizenzrechtlich in Kapitel 4 *Lizenz der Testfallverwaltung* beleuchtet. Die Auswertung wirkt sich auf die Lizenzwahl für die Testfallverwaltung aus.

Die Aufgabenstellung wird in Kapitel 5 *Motivation für eine JUnit-Testfallverwaltung* umfangreicher erläutert. Hier wird verdeutlicht, warum die Testfallverwaltung benötigt wird.

In Kapitel 6 *Lösungsansatz* werden die eigenen Ansätze erarbeitet und beschrieben, die sich aus der Aufgabenstellung ergeben. Daraus resultiert, wie die Testfallverwaltung umzusetzen ist. Dazu werden auch Probleme diskutiert, die auftreten, wenn programmiert wird.

In Kapitel 7 *Lösungsausführung* wird schließlich beschrieben, wie die Testfallverwaltung umgesetzt wird, wobei auch Probleme aufgeführt werden, die sich zusätzlich im Laufe der Programmierung ergeben haben.

Weitere Details zur Programmierung, die nicht in Zusammenhang mit der Ausführung stehen, werden in Kapitel 8 *Angaben zur Programmierung* aufgeführt.

Eine abschließende Bewertung der erarbeiteten Ansätze und Lösungen wird in Kapitel 9 *Bewertung* beschrieben.

Kapitel 10 *Rückblick auf den Verlauf der Arbeit* enthält den persönlichen Rückblick auf die Diplomarbeit.

4. Grundlagen und Techniken

In diesem Kapitel werden Grundlagen, benutzte Techniken und verwendete Programme aufgeführt und erklärt.

Bei den Techniken ist darauf geachtet worden, dass quelloffene und frei erhältliche Software benutzt wird. Dieser Schritt begründet sich darin, dass das Rahmenprogramm JUnit, um das es in der Testfallverwaltung geht, auch frei erhältlich ist.

4.1 Java, Umgebung

Es wird die aktuelle Version 6 des Java Development Kit (JDK) in der Standard Edition (Java SE) von Sun benutzt. Die Grundlage für die graphische Oberfläche befindet sich mit den AWT- und Swing-Paketen bereits im JDK.

Als Entwicklungsumgebung wird Eclipse SDK 3.3 mit den Erweiterungen (engl. plugins) Subclipse, Mylyn, Buckminster und Metrics eingesetzt.

Das zugrunde liegende Betriebssystem ist Kubuntu 6.10 mit einem Linux-Kernel 2.6.17.

4.2 JUnit

Das Rahmenprogramm (engl. framework) JUnit wurde zum Schreiben von Modultests in Java entwickelt. Es soll den Programmierer dabei unterstützen, auf einfache Weise Tests zu schreiben. Benutzt wurde die Version 4.3.1.

Erwähnenswert sind die Hauptentwickler Erich Gamma und Kent Beck. Beck hat die Vorgehensweise für Software-Entwicklung Extreme Programming im gleichnamigen Buch [be03] beschrieben, währenddessen Gamma sich einen Namen mit dem Buch über Entwurfsmuster gemacht hat.

JUnit ist verfügbar bei Sourceforge:
`http://sourceforge.net/projects/junit`

Es ist quelloffen und steht unter der Common Public License (CPL).

Aufgabe des Rahmenprogramms

JUnit stellt das verwendete Rahmenprogramm dar, um Modultests durchzuführen. Im Bereich der Java-Programmierung ist es der Standard für Modultests, wobei es ähnliche Rahmenprogramme für andere Sprachen wie NUnit für C#, CppUnit für C++ und diverse weitere gibt.

Der Begriff Modultest (engl. unit test) setzt sich aus den Wörtern Modul und Test zusammen. Diese beiden Wörter drücken das Einsatzgebiet von JUnit aus. Abstrakt dargestellt versteht man unter einem Modul eine zusammenhängende Kette von Befehlen, die einen Verarbeitungsschritt im Programm vornehmen. Ein Modul meint in Java einen Programmteil, der einen Verarbeitungsschritt darstellt. Dieser Programmteil kann eine einzelne Methode, mehrere Methoden, eine Klasse oder sogar mehrere zusammenhängende Klassen umfassen.

Das zweite Wort Test sagt etwas über die Aufgabe aus, die dargestellt werden soll. Der Duden erläutert den Begriff als Probe oder Prüfung, was heißt, dass hier keine neue Funktionalität dargestellt wird. Vielmehr soll Vorhandenes dahingehend geprüft werden, ob eine versprochene Funktionalität tatsächlich gegeben ist.

Die Philosophie hinter den Modultests ist, die Programmteile einzeln auf ihre Funktionalität zu testen. Dies soll losgelöst von anderen Programmteilen geschehen. Normalerweise wird sich ein Modultest in Java daher mit einer oder wenigen Methoden in nur einer Klasse beschäftigen.

Ein größeres Software-Projekt braucht daher viele Modultests, um die einzelnen Programmteile zu testen.

Die Tests von größeren, zusammenhängenden Einheiten werden dagegen dem Bereich der Systemtests überlassen.

Einführung mit einem 1. Beispiel-Test

Wenn ein Test erstellt werden soll, ist es notwendig zu wissen, was zu testen ist. Daher wird nachfolgend ein kurzer Quellcode-Ausschnitt aufgezeigt, der getestet werden soll (wird testgetrieben entwickelt, so wird erst der Test und danach der Programmcode geschrieben, siehe Kapitel 3.3):

```
public class MathSimple {
    public int sum (int a, int b){
            return a + b;
    }
}
```

Soll der Code getestet werden, so richtet man eine neue Klasse, welche die Tests aufnehmen wird. Den Testcode vom Programmcode zu trennen ist sinnvoll, es ist aber keine Voraussetzung dafür, dass eine neue Klasse erstellt wird. Im Sinne der Trennung der Zuständigkeiten von Klassen (engl. separation of concern) sollte sich der Code von Tests und Implementierung nicht in einer gemeinsamen Klasse befinden.

In der Diplomarbeit wird im Folgenden zwischen Testcode und Programmcode unterschieden. Testcode ist der Quellcode der Test-Klassen, Programmcode die funktionelle Implementierung des Programms.

Als Standard-Konvention für die Test-Klasse hat sich das Präfix oder Suffix `Test` vor bzw. hinter dem Namen der zu testenden Klasse durchgesetzt.

Im Sinne von JUnit wird eine Klasse erst dann zur Test-Klasse, wenn sie mindestens eine öffentliche Methode mit der Annotation `@org.junit.Test` enthält, die keine Parameter übernimmt und keine Rückgabe macht. Außerdem muss die Klasse einen Standard-Konstruktor enthalten (wird er nicht explizit vom Programmierer eingefügt, so erledigt der Java-Übersetzer das automatisch). Jeder einzelne Modultest stellt einen Testfall dar, das Wort Test wird als gleichbedeutendes Synonym verwendet.

Ein Testfall für die obige Klasse könnte so aussehen:

```
public class MathSimpleTest {
    @org.junit.Test
    public void testSum(){
        MathSimple m = new MathSimple ();
        org.junit.Assert.assertEquals (4, m.sum(1,3));
    }
}
```

Die Test-Methode `testSum` erstellt erst ein neues `MathSimple`-Objekt. Die Prüfung erfolgt in der (statischen) `assertEquals`-Methode. Die beiden Argumente werden über die `equals`-Methode des 1. Arguments miteinander verglichen. Die Vergleichsmethode kann nur mit Objekten umgehen, im Fall von primitiven Datentypen (einzige Ausnahme ist `double`) erfolgt ein Autoboxing durch den Java-Übersetzer.

Nach diesem Muster verlaufen die Tests mit JUnit. Test-Methoden lassen sich leicht schreiben. Es werden diverse Hilfsmethoden bereitgestellt, u.a. lassen sich damit geeignete Testumgebungen aufbauen. Von den statischen Vergleichsmethoden mit dem Präfix `assert` der Klasse `org.junit.Assert` gibt es insgesamt 32 Stück in verschiedenen Varianten.

Das Charakteristische an JUnit (bis Version 3) ist, dass die Oberflächen je nach Testergebnis einen farbigen Balken anzeigen. Sind alle Tests erfolgreich verlaufen, so wird ein grüner Balken angezeigt. Ist mindestens ein Test fehlerhaft, so wird der Balken rot.

Ein Fehler stellt in JUnit eine nicht vorhergesehene Programmausführung oder ein fehlgeschlagener Vergleich dar. Wird dagegen ein erwarteter Fehler abgefangen, ist der Testfall erfolgreich. (JUnit 3 hat hier zwischen Abbruch und Fehler unterschieden).

Bei Modultests spricht man klassischerweise von White-Box-Tests. Der Name kommt daher, dass man über die Implementierung des zu testenden Codes Bescheid weiß.

Weitergehende Techniken in JUnit

Tests lassen sich nicht nur einfach mit JUnit schreiben, es stellt auch diverse Techniken zur Verfügung, um das Testen noch zu unterstützen.

Die Techniken werden seit JUnit 4 ebenfalls in Annotationen ausgedrückt, wie bereits die vorgestellte Annotation `@org.junit.Test`. Es gibt fünf weitere Annotationen, welche sich alle im Paket `org.junit` befinden.

Die Annotationen `@Before` und `@After` werden für Methoden mit beliebigem Namen benutzt, die vor bzw. nach dem Ausführen jedes Testfalls aufgerufen werden (in JUnit 3 waren dies die Methoden `setUp` und `tearDown`). Auch sie müssen öffentlich sein, dürfen keine Parameter übernehmen und müssen ohne Rückgabewert sein. Die `@Before`-Methode baut eine Testumgebung auf. Sie wird pro Testfall einmal aufgerufen, um jedem Testfall der Klasse die gleiche Test-Umgebung zu bieten. Hier können z.B. Objekt-Variablen initialisiert werden. Die `@After`-Methode baut die Test-Umgebung wieder ab. Es sind mehrere `@Before`- und `@After`-Methoden möglich, wobei die Ausführungsreihenfolge nicht bestimmbar ist. Tritt ein Fehler in einem Testfall auf, so wird von JUnit auf jeden Fall die `@After`-Methode ausgeführt.

Neu in JUnit 4 sind die Annotationen `@BeforeClass` und `@AfterClass`. Sie bauen auch eine Test-Umgebung auf bzw. ab, dies wird jedoch nur einmal pro Test-Klasse ganz am Anfang bzw. ganz am Ende gemacht. Diese beiden Annotationen sind dann sinnvoll, wenn es aufwändiger ist, eine Testumgebung aufzubauen. Dies könnte zum Beispiel bei einem Zugriff auf das Netzwerk oder eine Datenbank der Fall sein. Da es nur einen einmaligen Aufbau der Umgebung gibt, sollten gegenseitige Nebenwirkungen der Tests zueinander bedacht werden. Auch hiervon sind mehrere Methoden in nicht bestimmbarer Ausführungsreihenfolge möglich.

Die fünfte Annotation ist `@Ignore`. Mit dieser Annotation lassen sich nur Methoden versehen, die bereits mit `@Test` markiert sind. So lässt sich der Test außer Betrieb nehmen, ohne dass er komplett auskommentiert werden muss. Der Programmierer wird im Ergebnis der Test-Ausführung daran erinnert, dass es noch offene Testfälle gibt, so dass es kein Vergessen von Tests mehr gibt (sowohl absichtlich, als auch unabsichtlich).

Für die Annotation `@Test` gibt es optionale Parameter:

- `timeout`: Es lässt sich eine Zeitspanne in Millisekunden festlegen, in welcher der Test erfolgreich laufen muss. Überschreitet er die Zeitspanne, dann wird der Test mit einem Fehler beendet.

- `expected`: Es lässt sich eine erwartete Ausnahme (engl. exception) angeben. Der Test wird mit einem Fehler beendet, wenn die erwartete Ausnahme nicht eintritt.

  Ein Beispiel-Testfall dazu:

```
@org.junit.Test (expected = MathSimpleException.class)
public void testSumFail(){
    MathSimple m = new MathSimple ();
    org.junit.Assert.assertEquals (4, m.sum(-1,3));
}
```

Bei letzter Methode lässt sich allerdings nur die Klasse der Ausnahme überprüfen. Der Test scheitert, wenn eine andere Ausnahme auftritt. Bei dieser Variante ist es außerdem schwer, die Ausnahme im Detail zu untersuchen. Möchte man zum Beispiel den Text der Ausnahme testen, so ist der Parameter `expected` nicht hilfreich, hier muss mittels eines `try/catch`-Befehls gearbeitet werden.

Da JUnit die Ausführungsreihenfolge der Test-Methoden nicht sicherstellt, sollten die Testfälle nicht voneinander abhängig sein. Jeder Testfall sollte innerhalb seiner Test-Umgebung daher nur einen Programmteil losgelöst vom Rest des Programmcodes prüfen und sich dabei nicht auf andere Testfälle verlassen. JUnit führt die Testfälle aus, indem jeweils ein eigenes Objekt der Test-Klasse pro Testfall erzeugt wird. Dadurch soll sichergestellt werden, dass jeder Testfall die gleiche Umgebung nutzt und unabhängig von den anderen ist.

Weitergehende Informationen findet man in der Dokumentation von JUnit und in `[we05]`.

Probleme aufgrund des Versionssprungs von 3 zu 4

Bei dem Versionssprung von JUnit 3 auf 4 wurde eine umfangreiche Überarbeitung (engl. refactoring) von JUnit vorgenommen. Das Hauptaugenmerk lag darin, mit JUnit noch einfacher Tests schreiben zu können. Der Aufbau des Rahmenprogramms wurde komplett überarbeitet, wobei in diesem Zusammenhang auf Java 1.5 umgestellt wurde.

Zu den vorhandenen Möglichkeiten, Tests zu schreiben, sind noch einige Hilfsmethoden hinzugekommen. Die zentralen Methoden zur Zusicherung (`org.junit.Assert.assert*`) sind erhalten geblieben und wurden noch erweitert.

Neben der Architektur gibt es auch grundsätzliche Unterschiede, wie die Tests ausgeführt und präsentiert werden können. JUnit 3 hatte eine eigene (graphische) Oberfläche mitgebracht. Dort ließen sich mehrere Test-Klassen auswählen, allerdings nur einzeln ausführen. Wollte man mehrere Test-Klasse ausführen, wird eine Klasse benötigt, die eine `junit.framework.TestSuite` (Test-Sammlung) zusammenstellt. Diese Klasse fasste alle auszuführenden Klassen in einer speziell präparierten `suite`-Methode zusammen. Diese konnte sowohl von der eigenen Oberfläche als auch in einer integrierten Lösung wie Eclipse ausgeführt werden. Der Programmierer konnte einzelne Klassen und Sammlungen von Klassen aus einer Oberfläche starten, weil beide eine gemeinsame Schnittstelle implementiert (`junit.framework.Test`) hatten.

JUnit 4 bringt keine Oberfläche mehr mit. Es ist lediglich möglich, die Tests ausführen zu lassen und das Ergebnis-Objekt von JUnit dann manuell nach den benötigten Informationen abzufragen. Dafür lassen sich in JUnit 4 mehrere Test-Klassen an den Test-Ausführer des Rahmenprogramm `org.junit.runner.JUnitCore` übergeben. Die Rückgabe ist ein Objekt der Klasse `org.junit.runner.Result.` Es enthält alle Ergebnisse der Ausführung.

Unter der Bedingung, dass die Test-Klassen (dem Java-Ausführer) bereits bekannt sind, fehlt JUnit 4 nur die Möglichkeit, das Ergebnis der Ausführungen aller übergebenen Test-Klassen anzuzeigen.

4.3 Vorgehensweise Extreme Programming

Extreme Programming (XP) ist die Vorgehensweise, wie das Projekt vorangetrieben und entwickelt wird. Es wird bereits im Grundlagen-Kapitel die Vorgehensweise beschrieben, und nicht erst in Kapitel 6 *Lösungsansatz* darauf eingegangen. XP kann nicht bei allen Lesern als bekannt vorausgesetzt werden. Außerdem wird die daraus stammende Programmiertechnik der testgetriebenen Entwicklung, welche JUnit nah steht, besonders hervorgehoben.

Im Gegensatz zu den klassischen Vorgehensweisen, wie z. B. dem deutschen V-Modell, setzt XP weniger darauf, Papierberge zu erstellen und tagelang zu planen. Es will schnell zu einem akzeptablen Ergebnis für die Entwickler und den Kunden / die Benutzer führen.

Es wurden XP-Techniken festgelegt, um eine qualtitativ hochwertige Software mit XP zu produzieren. Einer der ersten Entwickler, der diese Techniken ausformulierte, war Kent Beck in [be03]. Die Techniken basieren auf Werten und Prinzipien, die ebenso für XP formuliert wurden. Es gibt vier XP-Werte: Einfachheit, Kommunikation, Feedback und Mut. Diese Werte gehen über in zwölf XP-Prinzipien, die in der Diplomarbeit nicht genauer genannt werden.

Die XP-Techniken schließlich begründen sich auf den Werten und Prinzipien. Sie sollen den Entwicklern helfen, sich an die Werte und Prinzipien zu halten.

Die Techniken kurz im Überblick:

- Kunde vor Ort: Der Kunde als Auftraggeber kennt die fachlichen Anforderungen für das Projekt. Er sollte am besten vor Ort sein, um den Entwicklern schnell Informationen aus fachlicher Sicht zu geben.

  In [xp02] wird die Rolle des Kunden aufgeteilt in den Auftraggeber und den Benutzer. Das klassische XP macht hier keinen Unterschied. Bei der Diplomarbeit gibt es sehr wohl einen großen Unterschied. Der Auftraggeber ist der betreuende Professor. Der oder die Benutzer sind andere Programmierer, die die Testfallverwaltung für ihre JUnit 4-Tests benutzen wollen.
- Planungsspiel: Die Kunden und Entwickler legen in einem Planungsspiel am Beginn der nächsten Iteration fest, welchen Umfang die Iteration hat.
- Metapher: Die Kernideen des Systems sollen wenige klare Metaphern ausdrücken.
- Kurze Releasezyklen: In kurzen Abständen soll der Kunde voll funktionsfähige, neue oder geänderte Ausschnitte des Systems erhalten. Diese kann der Kunde dann gleich bewerten und somit die weitere Entwicklung beeinflussen. Ein Releasezyklus umfasst maximal drei Monate.
- Testen: Modultests sollen einzelne Systembaustein testen. Die fachlichen Anforderungen werden durch Systemtests ausgedrückt.
- Einfaches Design: Das Design des Systems soll möglichst einfach sein. Dadurch verringert sich die Entwicklungszeit und der Kunde versteht das System leichter. Außerdem soll damit nicht auf „Vorrat“ programmiert werden, da es Programmierer-Ressourcen an nicht

benötigte Programmteile bindet und zukünftige Anforderungen noch gar nicht im Detail bekannt sein können.

- Überarbeitung (engl. refactoring): Ein ungünstige Systemstruktur sollte behoben werden, bevor die Weiterentwicklung behindert wird. Abgesichert werden sie durch Modultests.
- Programmieren in Paaren: Zwei Entwickler programmieren an einem Rechner. Das erhöht die Qualität, und das Wissen kann unter den Entwicklern schnell verbreitet werden. Faustregel: 1 + 1 > 2.
- Gemeinsame Verantwortlichkeit: Alle Entwickler sind gemeinsam für das Projekt verantwortlich. Der Quellcode sollte allen Entwicklern zur Verfügung stehen.
- Fortlaufende Integration: Änderungen am Quellcode sollen schnell in ein System zur Versionsverwaltung integriert werden. Dadurch gibt es keine Parallelentwicklung und Konflikte können früh erkannt und beseitigt werden.
- Programmierstandards: Um das Programmieren in Paaren und die gemeinsame Verantwortlichlichkeit zu unterstützen, werden gemeinsame Programmierstandards gebraucht.
- 40-Stunden-Woche: Die Entwickler sollen regelmäßig nicht mehr als 40 Stunden arbeiten. Langfristig unterstützt dies die Kreativität der Entwickler und lässt sie motiviert bleiben.

Extreme Programming sieht normalerweise vor, dass alle aufgeführten Techniken eingesetzt werden. Sie sind nicht unabhängig zu betrachten, sondern stehen in Beziehung zueinander. Als Beispiel hängt die Technik Überarbeitung vom Testen ab. Man sichert eine große Überarbeitung ab, indem Modultests gemacht werden. Ohne Modultests kann nicht geprüft werden, ob die Funktionalität nach der Überarbeitung immer noch vorhanden ist.

Die Techniken sind jedoch auf die individuellen Projekte anzupassen. Das muss hier geschehen, denn die Diplomarbeit stellt eine Einzelarbeit dar, so dass viele Annahmen des XP nicht zutreffen, so fehlen z. B. ein Entwicklerteam und ein richtiger Kunde. Im Kapitel 6 *Lösungsansatz* wird genauer beschrieben, welche Techniken angewendet werden und welche angepasst, oder sogar weggelassen werden.

Da der Autor nicht alle Techniken umsetzen kann, wird besonders auf Testen großen Wert gelegt. Hier gibt es eine eigene Programmiertechnik, die testgetriebene Entwicklung. Von der testgetriebenen Entwicklung wird der Teil der Modultests verstärkt eingesetzt. Nachfolgend findet eine kurze Einführung darin statt.

Testgetriebene Entwicklung

Bei dieser Programmiertechnik werden die Tests vor dem zu testenden Programmcode geschrieben. Eine Weiterentwicklung des Software-Projektes soll nur dann geschehen, wenn dies durch einen Test initiiert wird. Die Tests sollen dabei automatisiert laufen, um sie jederzeit schnell ausführen zu können.

Die testgetriebene Entwicklung besteht aus zwei Vorgehensweisen. Zum einen gibt es die Modultests und zum anderen die Systemtests. Die Modultests testen dabei einzelne Programmteile, währenddessen die Systemtests große Ausschnitte des gesamten Software-Projektes testen sollen.

Die Testfallverwaltung wird in seinen Kern-Bestandteilen (die Bereiche, die nicht als Oberfläche dargestellt werden) testgetrieben entwickelt. Allerdings werden dabei nur Modultests eingesetzt. Dieses Verfahren wird gewählt, da sich die Testfallverwaltung aufgrund seines Einsatzgebietes selbst mit Modultests beschäftigt. Systemtests werden nicht erstellt, da sich diese als zu aufwändig für die Diplomarbeit darstellen.

Im Nachfolgenden wird daher eine kurze Einführung in die testgetriebene Entwicklung mit Modultests gegeben.

Modultests

Die Modultests testen einzelne Programmteile unabhängig von anderen Programmteilen. In Java wird ein Programmteil normalerweise eine oder wenige Methoden in einer Klasse umfassen. Um ein großes Software-Projekt mit Modultests abzudecken, müssen demnach alle Programmteile von Modultests getestet werden. Damit wird mit der Projektgröße auch die Anzahl der Tests in etwa linear ansteigen.

Die Modultests werden auch White-Box-Tests genannt, da der erstellende Programmierer in der Regel den Programmcode implementiert. Der Programmierer kennt in der testgetriebenen Entwicklung die Implementierung des Programmcode noch nicht, da er zuerst die Tests schreibt. Weil er im Allgemeinen aber gleich nach den Tests den Programmcode schreibt, wird er bereits eine Vorstellung des Code-Gerüstes in Gedanken haben. Damit hat er Einblick in den zu testenden Code. Spätestens wenn der Test- und Programmcode geschrieben sind, ist es ein White-Box-Test, da die Implementierung offen liegt.

Vorgehen bei Modultests

Wird testgetrieben entwickelt, so läuft das Schema für Modultests wie folgt ab:

1. Es wird ein einfacher Test geschrieben, welcher anfangs mangels (funktioneller) Implementierung scheitern muss. Der Test muss die Erwartung an den zu prüfenden Programmcode ausdrücken.

2. Der Test wird mit so wenig Code wie möglich erfüllt. Im einfachsten Fall kann dies eine statische Rückgabe sein. Der Test sollte nun das erste Mal erfolgreich laufen.

3. Nun geht es daran, weitere Tests für die getestete Methode zu schreiben und diese Tests immer wieder zu erfüllen. Dabei sollte es nicht mehr als einen fehlgeschlagenen Test geben. In Schritt 3 sollte auch umfangreich überarbeitet (engl. refactoring) werden.

In JUnit wird charakteristisch der grüne Balken für ein erfolgreiches Ergebnis der Tests angezeigt, bei mindestens einem Fehler der rote Balken. Übertragen auf die testgetriebene Entwicklung bedeutet dies, dass vom 1. zum 2. Schritt immer von rot nach grün entwickelt wird. Im 3. Schritt sollte das Ergebnis immer den grünen Balken anzeigen.

Der 3. Schritt ist der wichtigste Schritt von allen. Er wird so lange wiederholt, bis der Programmierer überzeugt ist, den Programmcode mit genügend Tests auf seine erwartete Funktionsweise hin geprüft zu haben. Dies bedeutet, nicht annähernd unendlich viele Kombinationen zu testen, sondern sinnvolle Bereiche mit Tests abzudecken.

Wird das Schema für Modultests in der testgetriebenen Entwicklung auf das obere Beispiel in der JUnit-Einführung (Kapitel 3.2) angewendet, so schreibt man zuerst den Test :

```
public class MathSimpleTest {
    @org.junit.Test
    public void testSum(){
    MathSimple m = new MathSimple ();
    org.junit.Assert.assertEquals (4, m.sum(1,3));
    }
}
```

Damit der Test überhaupt gestartet werden kann, muss wenigstens die Klasse `MathSimple` mit der zu testenden Methode `sum` angelegt werden. Man könnte auf noch niedrigerem Niveau anfangen, in dem man keine Klasse `MathSimple` erstellt und so zuerst eine Fehlermeldung des Übersetzers produziert wird. Dies wird sogar in [we05] empfohlen, da

man sich so sicher sein kann, dass der Test läuft. Die zu testende Methode `sum` lautet:

```
public class MathSimple {
    public int sum(int a, int b) {
        return 0;
    }
}
```

Nun kann JUnit das erste Mal den Test ausführen. Der Balken bleibt allerdings rot. Die Fehlermeldung sagt, dass eine 4 erwartet wurde, tatsächlich hat die Methode eine 0 zurückgegeben. Die genaue Meldung lautet:

```
java.lang.AssertionError: expected:<4> but was:<0>
```

Die Tests in der testgetriebenen Entwicklung sollten immer in kleinen Schritten erfüllt und so einfach wie möglich programmiert werden. Um den Test das erste Mal erfolgreich zu durchlaufen, muss die Methode `sum` wie folgt aussehen:

```
public int sum (int a, int b){
    return 4;
}
```

Damit ist lediglich ein Testfall durch eine „harte“ Rückgabe abgedeckt. Um die Funktionalität der `sum`-Methode zu erhöhen, schreibt man weitere Tests:

```
@org.junit.Test
public void testSum2(){
    MathSimple m = new MathSimple();
    org.junit.Assert.assertEquals (20, m.sum(10,10));
}
```

Damit führt das Ergebnis der Test-Ausführung wieder zu einem Fehler. Das Ziel sollte schließlich sein, durch weitere sinnvolle Testfälle eine korrekte Berechnung zu erreichen, sie könnte so aussehen:

```
public int sum (int a, int b){
    return a + b;
}
```

Wie hoch sollte der Grad der Test-Abdeckung von Modultests sein?

Es gilt generell, dass so viele Tests angestrebt werden müssen, bis der Programmierer sich sicher ist, dass der Code die gewünschte Funktionalität bereitstellt. Das obere Beispiel sollte auf einfache Weise veranschaulichen, dass man sich durch viele einfache Tests schrittweise an die gewünschte Funktionalität heran arbeitet.

Beim Testen gibt es Software-Komponenten, die man als vertrauenswürdig voraussetzen sollte. Dazu zählen die Java-Standardbibliothek, benutzte Drittanbieter-Bibliotheken und Trivial-Klassen. Bei den ersten beiden Komponenten muss man ein gewisses Vertrauen in den Code setzen. Sie lassen sich zwar mit eigenen Tests prüfen, allerdings werden sie meist so verwendet, als wären sie fehlerfrei. Bei Trivial-Klassen wie Bean-Klassen ist auch davon auszugehen, dass sie einwandfrei funktionieren, da sie keinerlei Anwendungslogik bereitstellen. Daher sollten die Tests sich auf die selbst programmierte Funktionalität beschränken.

Modultests werden während der gesamten Entwicklung über angestrebt. Es wird immer zuerst ein Test geschrieben, bevor weiterentwickelt wird. Wird ein Fehler außerhalb der Testverfahren gefunden, so lässt sich dieser auch beheben, indem man den gefundenen Fehler zuerst mit einem Test sein fehlerhaftes Verhalten nachweist. Der Test sollte die Funktionsweise ausdrücken, die als Erwartung an den Programmcode gestellt wird. Erst nachdem der Fehler getestet wurde, wird er auch behoben.

Spätestens nachdem die Funktionalität auf einfache Art implementiert ist, muss auch der Code überarbeitet werden. Denn die Qualität des Programmcode zeichnet sich nicht nur durch die Funktionalität aus, sondern auch durch die Struktur. Ist die Struktur schlecht aufgebaut, würde die Weiterentwicklung behindert werden. Die Qualität der Software bleibt nur bestehen, wenn auf Funktion und Struktur gleichermaßen geachtet wird. Tests stellen bei der Überarbeitung sicher, dass noch alle erwarteten Funktionen vorhanden sind.

Auch lässt sich eine Überarbeitung (engl. refactoring) testgetrieben erledigen. Zieht man beispielsweise eine Schnittstelle ein, so wird diese zuerst im Test eingebaut und danach implementiert.

4.4 Datenbank

Als Persistenzlösung kommt das objektorientierte Datenbanksystem db4o von db4objects zum Einsatz. Es wird die aktuelle Version 6.1 verwendet. Gegenüber dem relationalen Datenbanksystem ist die Einbettung in die

Programmierung einfach, da nativ objektorientiert mit der Datenbank gearbeitet werden kann.

Es entfällt ein aufwändiges und fehleranfälliges Abbilden (engl. mapping) der Daten vom relationalen zum objektorientierten Datenmodell.

Das Datenbanksystem db4o ist verfügbar unter: `http://www.db4o.de`

Zum reinen Datenbanksystem-Paket gehört der *Object Manager* als Oberfläche zur Betrachtung der gespeicherten Objekte dazu. Er wird von db4objects als Java-Programm bereitgestellt. Es eignet sich gut, um Abfragen zu machen.

Db4o ist quelloffen und unter einer dualen Lizenz verfügbar. Die so genannte Community-Version wird im Rahmen dieses Projektes benutzt, was bedeutet, dass die GNU General Public License (GPLv2) zur Anwendung kommt.

4.5 Sonstige benutzte Software

Erstellung von Grafiken

Die Grafiken, welche als Symbole (engl. icon) in der Testfallverwaltung benutzt werden, sind alle mit dem Zeichen- und Bildbearbeitungs-programm Krita von KDE erstellt. Krita steht unter der GPLv2.

Werkzeug zur Vereinfachung des Build-Prozess

Als Werkzeug für den Build-Prozess wird Ant eingesetzt. Es ist als Erweiterung von Eclipse vorhanden und wurde benutzt, um einfach eine Jar-Datei aus dem Projekt zu erzeugen. Ant steht unter der Apache License 2.0

Austausch mit Subversion-Server

Um Dateien mit einem Subversion-Server (System zur Versionsverwaltung) auszutauschen, wird Subclipse als Erweiterung für Eclipse installiert. Subclipse steht unter der EPL 1.0.

Erstellung von Graphen

yEd Graph Editor wird zum Erstellen eines Diagramms der Objektgraphentiefe benutzt. yEd ist das einzige Programm, welches nicht quelloffen, aber frei verfügbar ist.

Protokollierung

Um nicht alle Ausgaben auf der Konsole ungesteuert auszugeben, wird log4j eingesetzt. Log4j steht unter der Apache License 2.0.

UML-Modellierung

Alle erstellten Klassendiagramme werden mit ArgoUML 0.24 erstellt. Es ist unter der BSD License verfügbar.

4.6 Alternativen zur Testfallverwaltung

Hier soll ein kurzer Überblick gegeben werden, welche Alternativen zur Entwicklung der Testfallverwaltung bestehen.

Es kommen drei Alternativen in die nähere Betrachtung:

- Quelloffene Programme
- Kommerzielle Programme
- Integrierte Lösungen in Entwicklungsumgebungen

Quelloffene Programme

Erste Anlaufstelle für die Suche nach quelloffenen Programmen war Sourcefourge.net. Eine umfangreiche Suche im April 2007 unter allen verwalteten Projekten mit dem Begriff „JUnit“ führte zu über 230 Ergebnissen. Darunter waren viele Projekte, die über den Status der Anmeldung nicht hinaus gekommen sind. Des Weiteren fiel ein Großteil weg, der sich nicht um das Testen in Java-Umgebungen kümmerte, sondern für andere Programmiersprachen nach der Vorlage von JUnit entwickelt wurde.

Übrig blieben nur wenige brauchbare und aktuelle Projekte, die jedoch entweder Erweiterungen für JUnit um fehlende Funktionen darstellten, oder sich nur um Spezialgebiete (z. B. DbUnit) kümmerten. Ein Projekt, welches unabhängig von einer Entwicklungsumgebung eine graphische Oberfläche bereit stellte, befand sich nicht darunter.

Kommerzielle Programme

Die Suche nach kommerziellen Oberflächen zur Ausführung von JUnit-Tests blieb im April 2007 erfolglos.

Integrierte Lösungen

Die integrierten Lösungen werden am Beispiel von Eclipse und Netbeans erläutert. Diese beiden stellen die am meisten benutzten Entwicklungsumgebungen für Java dar. Beide bieten eine Integration von JUnit an. Ihr großer Vorteil ist, dass die Entwicklungsumgebungen über die Klassen Bescheid wissen. Es bedarf keiner großen Konfiguration des Klassenpfades. Der Klassen- und Paketname sind bereits bekannt. So ist

es für den Klassenlader einfach, die zu testenden Klassen an das JUnit-Rahmenprogramm weiterzureichen.

Der Nachteil besteht darin, dass man gezwungen ist, die Entwicklungsumgebung zu starten, nur um die Tests auszuführen.

Integrierte Lösungen: Eclipse 3.3

In Eclipse wird sowohl JUnit 3.8.2 als auch Version 4.3.1 unterstützt. Die Ausführung ist hier intuitiv. Hat man eine Klasse, die Tests enthält, so biete Eclipse eine Ausführung mit JUnit an. Es wird eine separate Ansicht (engl. view) eingefügt. Sie bietet alle Informationen zu den ausgeführten und nicht-ausgeführten Test-Methoden. Ist ein Test fehlgeschlagen, so wird die entsprechende Fehlermeldung von JUnit angezeigt.

Es wird sogar eine Historisierung der Tests innerhalb der gleichen Eclipse-Sitzung vorgenommen. Allerdings ist es nur möglich, eine Klasse auf einmal zu testen. Es lässt sich demnach kein komplettes Paket mit mehreren Test-Klassen schnüren. Dies ist in Eclipse nur Test-Klassen für JUnit 3 vorbehalten, für die sich eine `TestSuite` erstellen lässt.

Integrierte Lösungen: Netbeans 5.5

Netbeans unterstützt in der aktuellen Version 5.5 nur JUnit 3.8.1. Auf den Internetseiten von Netbeans wird die Benutzung von JUnit 3.8.1 ausführlich beschrieben. Zu JUnit 4 gibt es nur wenige Informationen:

- Man soll die JUnit 4-Tests über einen Adapter (aus JUnit 4) mit der alten Oberfläche von JUnit 3 ausführen.
- Tests in JUnit 4 lassen sich unter Einbindung des neuen Rahmenprogramm als Klassen-Pfad-Bibliothek mit dem eigenen Ausführer `org.junit.runner.JUnitCore` ausführen. Dies geschieht allerdings ohne jede graphische Integration in die Benutzeroberfläche von Netbeans.
- Die Entwickler von Netbeans wollen eine Integration von JUnit 4 nicht selbst übernehmen. Sie wollen, dass die Entwickler von JUnit dies erledigen.

Netbeans ist demnach als Entwicklungsumgebung für JUnit 4-Tests nicht brauchbar einzusetzen.

5. Lizenz der Testfallverwaltung

Bei jedem Software-Projekt stellt sich die Frage, unter welche Lizenz man sein Werk stellt. Es gibt die gängigen Lizenzmodelle proprietär und quelloffen, bei wenigen Projekten auch eine duale Lizenz.

Mit proprietär ist hier gemeint, dass ein Projekt nicht entgeltfrei vertrieben wird und der Quellcode nicht offen gelegt ist. Quelloffen dagegen meint hier, dass nicht nur der Quellcode offen liegt, sondern das Projekt auch unter einer „freien“ Lizenz wie der GPL verfügbar ist. In der Praxis sind diese beiden gegensätzlichen Lizenztypen nicht immer so genau abgegrenzt und definiert. Die harte Trennung soll die Gegensätze verdeutlichen und gilt auch nur im Rahmen dieser Arbeit.

Erwähnt werden soll aber auch, dass es Mischvarianten, z. B. Pretty Good Privacy (besser als PGP-Verschlüsselung bekannt) gibt, deren Quellcode zwar offen gelegt ist, die Weitergabe und Veränderung aber verboten ist, was freier Software widerspricht.

Leider ist der Bereich Lizenzen der freien Software unübersichtlich. Dies fängt schon bei der Begriffsdefinition *freie Software* an, die je nach Interpreter und Lizenz sehr unterschiedlich ausfällt. Hier fällt auch oft der Begriff *Open Source*, der genauso kritisch in seiner Definition zu betrachten ist. Die entsprechenden Lizenzen sind unterschiedlich ausgestaltet. Nicht zuletzt führt das zu Inkompatibilitäten zwischen den verschiedenen Lizenzen.

Eine gute Definition von freier Software ist bei der Free Software Foundation zu finden:

`http://fsfeurope.org/documents/whyfs.de.html`

Im Rahmen dieser Arbeit wird quelloffene Software benutzt. Einzige Ausnahmen sind hier db4o (welches unter einer dualen Lizenz steht) und yEd (welches nur frei erhältlich ist).

Zu beachten sind nur die Lizenzen, bei der die Software nicht nur benutzt wird (wie z. B. Eclipse), sondern eine `import`-Anweisung (Verlinkung) benutzt wird:

- GNU General Public License 2 (GPLv2) – db4o
- Common Public License 1.0 (CPL) – JUnit
- Apache License 2.0 – log4j

Die ersten beiden Lizenzen erlauben es nicht, sich zu der Software zu verbinden oder sie zu erweitern und die erstellte Sofware unter eine andere Lizenz zu stellen (strenges Copyleft). Das bewirkt, dass davon abgeleitete Software wieder unter der Ursprungslizenz veröffentlicht werden muss. Die einzig liberalere benutzte Lizenz (im Vergleich zur GPLv2 und CPL) ist die Apache License, die allerdings inkompatibel zur GPLv2 ist.

Zur Kompatibilität / Inkompatibilität der GPL mit anderen freien Lizenzen hat die Free Software Foundation eine Übersicht im Internet:

`http://www.fsf.org/licensing/licenses/index_html`

Nach der vorliegenden Übersicht darf das Projekt damit nicht veröffentlicht werden, da die GPL und CPL nicht gleichzeitig bei einer Veröffentlichung zum Einsatz kommen dürfen. Wäre dies möglich, würde die Testfallverwaltung unter die GPL gestellt werden.

Es sind zwei Alternativen diskutierbar, welche sich bei einer Internet-Recherche ergeben haben:

- Nichtauslieferung der betroffenen Rahmenprogramme:

  Es werden die betroffenen Rahmenprogramme JUnit, db4o und log4j nicht im Paket der Testfallverwaltung mit ausgeliefert. Damit ist zwar eine Verbindung im Programm (in übersetzten Klassen-Dateien) gegen die entsprechenden Rahmenprogramme geschaffen, aber die Rahmenprogramme liegen nicht vor, so dass eine aktive Verbindung nicht zustande kommen kann. So gesehen ist hier die rechtliche Frage offen, ob die nicht funktionierende Verbindung bereits ein Lizenzbruch ist. Die gleiche Problematik zeigt sich an prominenter Stelle im Linux-Kernel, bei dem es regelmäßig Diskussionen um Grafik-Hardware gibt, die sich mit proprietären Software-Treibern mit dem Kernel verbinden möchten.

- Es findet keine Veränderung im Sinne der Lizenzbestimmungen statt:

  In [if07] wird zum Copyleft geschrieben, dass Veränderungen und Weiterentwicklungen der Software wieder unter der gleichen Lizenz veröffentlicht werden müssen (in Bezug auf die GPLv2). Überträgt man diese Auslegung des Copyleft auch auf die CPL, so müsste es möglich sein, die Software unter irgendeiner Lizenz zu veröffentlichen. Es wird zwar eine Verbindung zu den betroffenen Rahmenprogrammen JUnit und db4o geschaffen, aber es werden keine Veränderungen an diesen vorgenommen. Dies könnte man gleichsetzen mit einer Benutzung des Betriebssystems Ubuntu, welches auf dem GPL-lizensierten Linux-Kernel aufsetzt. Wird nun eine neue Software für dieses Betriebssystem entwickelt, so muss es auch nicht automatisch unter der GPL stehen. Es findet lediglich eine Benutzung der Bibliotheken statt (bzw. der über dem Linux-Kernel liegenden Schichten).

Liest man dazu die umfangreiche FAQ-Liste zur GPL der Free Software Foundation (`http://www.fsf.org/licensing/licenses/gpl-faq.html`), so werden beide Punkte für rechtlich nicht möglich erklärt. Zum ersten Punkt findet sich in der FAQ-Liste ein Beispiel, in dem es um das Verbinden zu GPL-Modulen geht. Lässt sich das Programm in allgemeiner Weise mit dem Modul verbinden, z. B. über einen `fork`-Befehl oder es wird nur die `main`-Methode aufgerufen, so ist man nicht an die GPL gebunden. Da jedoch Klassen im Programm importiert werden, erscheint dies nicht als allgemein genug ausreichend.

Zur Lizenzproblematik wurde der Autor von *Testgetriebene Entwicklung mit JUnit & Fit* [we05] befragt, seine Antwort lautete: *„Tut mir leid, kenne mich mit dem Lizenzrecht nicht gut genug aus, um Dir fachmännischen Rat geben zu können.“* Es erscheint fraglich, dass ein Autor 200 Seiten über den Umgang mit einem Test-Rahmenprogramm verfasst und sich nicht mit der Lizenz befasst, unter der es steht.

Auch einer der Entwickler von JUnit, Kent Beck wurde befragt, seine Antwort lautete: *„Thank you for the question about licensing. Unfortunately, I am not an expert on open source licensing issues so I don't have any advice for you.“* Kent Beck ist bei Agitar Programmierer, die eine auf JUnit aufsetzende proprietäre Software verkaufen. Dort kann JUnit als Binär-Datei eingesetzt werden (dies ist zusammen mit GPL-Software nicht möglich). Es wird vermutet, dass daher die Wahl auf die CPL fiel, da JUnit so zwar frei ist, aber auch in ein kommerzielles Programm integriert werden kann.

In [if07] wird als eine Option beschrieben, dass der oder die Urheber (zumindest nach deutschen Recht) die Software unter anderer Lizenz freigeben können. Herr Beck wurde zur Freigabe befragt, jedoch blieb er eine Antwort schuldig.

Deutsche Gerichte haben sich bisher nur wenig mit den Details der freien Lizenzen beschäftigt. Die GPL wurde bei mehreren groben Verstößen mehrfach bestätigt (zuletzt in http://www.heise.de/newsticker/meldung/78530). Inwieweit deutsche Gerichte eine Veröffentlichung der Testfallverwaltung unter der GPL, also frei und unentgeltlich, aber inkompatibel zur CPL, bewerten würden, konnte nicht geklärt werden.

Im Rahmen der im Juni 2007 erfolgten Überarbeitung der GPL-Lizenzbestimmungen zu Version 3 ist nun die Apache License 2.0 kompatibel, dafür ist die GPLv3 nicht kompatibel zur GPLv2. Die Lizenzproblematik in Zusammenhang mit der CPL besteht nach wie vor.

Um rechtliche Probleme zu vermeiden, sollte eine Veröffentlichung der Testfallverwaltung mit den benutzten Rahmenprogrammen unterbleiben.

6. Motivation für die Testfallverwaltung

Im Szenario der Einleitung ist von einer testgetriebenen Entwicklung ausgegangen worden, bei der immer vom roten zum grünen Testbalken hin gearbeitet wird. Es darf maximal ein Test nicht funktionieren, so dass Schritt für Schritt entwickelt werden kann.

Dies funktioniert, so lange man allein entwickelt, noch verhältnismäßig gut. Sicherlich passiert es, dass eine Änderung am Code mehr als einen Test scheitern lässt, dies sollte sich aber schnell beheben lassen.

Arbeiten mehrere Programmierer an einem Projekt, das sich über ein System zur Versionsverwaltung (z. B. Subversion) austauscht, so kann es zu größeren Problemen kommen. Besonders dann, wenn die gleichen Code-Teile bearbeitet werden. Dies fällt wahrscheinlich erst auf, wenn man vor dem Hochladen des geänderten Code noch einmal das lokale Verzeichnis mit der Versionsverwaltung synchronisiert. Hier kann es schnell dazu führen, dass eine größere Anzahl von Tests nicht läuft.

Nachfolgend werden vier Probleme verdeutlicht, die aufzeigen, dass es einige Gründe für die Testfallverwaltung gibt.

6.1 Vorhandene Oberflächen für JUnit 4

Eine Oberfläche für JUnit 4 bietet bislang nur Eclipse, als eine der großen integrierten Entwicklungsumgebungen. Netbeans kann bislang mit JUnit 4 nicht umgehen.

Es gibt diverse Entwickler, die ohne große Entwicklungsumgebung programmieren. Diese sind auf einen unabhängigen Test-Ausführer angewiesen. Sie könnten zwar auch direkt den JUnit-Ausführer benutzen, müssen dabei aber Abstriche bei der Bedienung und der Ergebnisaufbereitung und -speicherung in Kauf nehmen.

Andere quelloffene Projekte für JUnit-Oberflächen wurden im April 2007 bei Sourceforge nicht gefunden.

Wird unabhängig von Eclipse entwickelt, ist ein neues Programm wie die Testfallverwaltung sinnvoll.

6.2 Fehlende Historisierung

Laufen mehrere Tests über eine längere Phase nicht, dann können Probleme auftauchen, diese nachzuvollziehen. Hierzu ist eine Historisierung der Test-Ergebnisse sinnvoll. So kann man auch später noch feststellen, an welchen Stellen die Schwierigkeiten aufgetreten sind.

Ein gegenüber der Einleitung leicht erweitertes Szenario soll dies verdeutlichen:

- nachdem das lokale Verzeichnis mit der Versionsverwaltung synchronisiert wurde, zeigen 15 von 100 Tests einen roten Balken,
- die erste Überarbeitung beseitigt einen Fehler, 14 Tests misslingen noch,
- die zweite Überarbeitung beseitigt noch zwei Fehler, 12 Test misslingen noch,
- die nächste Überarbeitung führt zu einem Rückschritt, es funktionieren plötzlich 16 Tests nicht mehr,
- bei der letzten Überarbeitung des Tages gehen 16 Tests immer noch nicht.

Wichtig ist es demnach, den Überblick zu behalten. Das beinhaltet nicht nur die Anzahl der misslungenen Tests zu wissen, sondern auch detailliert die Fehler in Erfahrung bringen zu können. Nur so lässt sich ein sinnvoller Vergleich anstellen, um auf Probleme besser aufmerksam zu werden.

Diese Vorgehensweise lässt sich nur bewerkstelligen, wenn eine Historie der vergangenen Tests gespeichert wurde. Ist dies der Fall, so kann man jedes Ergebnis eines Test-Laufs einzeln nach den Fehlschlägen durchsehen. Zusätzlich kann man dann die Fehler miteinander vergleichen und auch feststellen, warum die letzten Tests noch funktionierten oder fehlschlagen.

Beim Vergleich der Fehler fällt möglicherweise auf, dass nach der letzten Code-Implementierung ein anderer Fehler entstanden ist. Hat man nun zusätzlich die alten Test-Ergebnisse vorzuliegen, so kann dies eine Behebung des Fehlers vereinfachen.

Die Anzahl der misslungenen Tests sagt auch nichts darüber aus, ob es sich dabei jeweils um die gleichen Tests handelt. Hier ist vorstellbar, dass es sich selbst bei gleich gebliebener Anzahl um verschiedene (misslungene) Tests bei mehreren Läufen handelt.

Als Vergleich werden wieder die integrierten Lösungen betrachtet. Eclipse bietet die Ergebnisse von zehn vergangenen Test-Läufen in der gleichen Sitzung an, das Maximum kann dabei erhöht werden. Persistent gespeichert werden die Ergebnisse nicht.

Daraus folgt, dass eine Historisierung der Test-Läufe sinnvoll ist und eine Erleichterung bei der Fehlersuche darstellt.

6.3 Sammlungen von Tests

Da sich die Tests normalerweise auch über mehrere Klassen verteilen, ist es zudem immens wichtig, alle Tests auf einmal starten zu können.

Die Tests bei JUnit erstrecken sich normalerweise über mehrere Klassen. Zum einen wird der Programmierer die Test-Klassen sinnvoll nach Zugehörigkeit zu Programm-Klassen strukturieren wollen. Zum anderen ist die Zugehörigkeit auch abhängig vom Aufbau der Test-Umgebung, die je nach Test-Klasse unterschiedlich sein kann.

Gerade den integrierten Lösungen fehlt seit JUnit 4 jedoch die Möglichkeit, mehrere Test-Klassen zusammen auszuführen. Eclipse bietet es nur an, eine Test-Klasse auszuführen.

Sind für die im obigen Szenario (siehe Kapitel 5.2) 100 Tests zehn Test-Klassen erstellt, so müsste jede einzeln ausgeführt und manuell ausgewertet werden. Das verlangt vom Programmierer viel Mehrarbeit, die redundant ist. Außerdem verlangt es große Disziplin, nach jeder Implementierung alle Tests laufen zu lassen. Auf lange Sicht würden darunter die Tests und damit auch die Programmqualität leiden. Vernachlässigt man die Test-Ausführungen, kann man keinen Fortschritt bei der Problembeseitigung feststellen oder übersieht Probleme in anderen Tests. Dies dürfte dazu führen, dass das Testen komplett unterbleibt.

Ein Ausführer, der eine Sammlung an Test-Klassen aufnimmt, ist demnach wichtig, um die Tests automatisiert laufen lassen zu können und den Überblick über alle Test-Klassen zu behalten.

6.4 Probleme in nicht-testgetriebenen Projekten

In normalen Java-Projekten, die nicht testgetrieben entwickelt werden, könnte gerade eine Historisierung noch wichtiger sein. Schließlich werden die Tests erst nach dem Programmcode erstellt. Ist in diesen Projekten nicht von vornherein auf gute Testbarkeit und saubere Strukturen geachtet worden, so wird es größere Probleme geben, die Tests zu erstellen.

Da die Entwickler auch nicht der Philosophie des „Schritt-für-Schritt Programmierens“ folgen, wird nicht darauf geachtet, dass nach Möglichkeit maximal ein Test fehlschlägt. Läuft nun mehr als ein Test nicht, ist eine Historisierung nach Kapitel 5.2 sinnvoll.

Besonders wenn der Programmcode stark überarbeitet (engl. refactoring) wird, kann es in normalen Projekten zu einer Reihe von misslungenen Tests kommen. Dies ist die Folge der Herangehensweise, bei der zuerst im normalen Code überarbeitet wird und danach die Tests angepasst werden. Auf diesem Weg ist die Historisierung auch sinnvoll, so dass sich ebenso in normalen Projekten deren Verfügbarkeit lohnen würde. Zusätzlich sollten sich auch hier die Tests in einer Ansammlung ausführen lassen.

6.5 Nutzen der Testfallverwaltung

Mit der Testfallverwaltung macht man sich unabhängig von den Entwicklungsumgebungen, die JUnit 4 noch nicht gut oder gar nicht unterstützen. Andere Oberflächen wurden nicht gefunden, daher hat sie hier ein erstes Alleinstellungsmerkmal.

Die Testfallverwaltung speichert die Ergebnisse über die Programm-laufzeit hinaus. Damit macht sie es einfacher Test-Ergebnisse zu vergleichen und hilft damit die Test-Qualität zu erhöhen. Da Eclipse als einziges nur eine temporäre Speicherung durchführt, ist das zweite Alleinstellungsmerkmal, die Test-Ergebnisse persistent zu machen. Das dritte Alleinstellungsmerkmal ist, Test-Ergebnisse vergleichen zu können.

7. Lösungsansatz

Hier werden die Ansätze erläutert, die sich aus der Aufgabenstellung ergeben. Es werden die eingesetzten Techniken und Grundlagen auf die Aufgabenstellung reflektiert und diskutiert.

Es wird zuerst die Vorgehensweise erläutert (Kapitel 6.1). Danach wird damit begonnen, Tests mit JUnit 4 ausführbar zu machen (Kapitel 6.2). Die Daten werden in das Daten-Modell (Kapitel 6.3) überführt, um sie speichern zu können. Weitere Test-Klassen kommen über den Klassenlader in der Testfallverwaltung hinzu (Kapitel 6.4 und 6.5). Die Daten müssen für die Historisierung persistent gemacht werden (Kapitel 6.6). Eine Darstellung ist dann über die graphische Oberfläche realisiert (Kapitel 6.7). Zur Oberfläche gehört auch eine entsprechende Fehlerbehandlung (Kapitel 6.8). Java als Programmiersprache bringt eine hohe Plattformunabhängigkeit mit, die Testfallverwaltung steht dem nicht nach (Kapitel 6.9). Die Auslieferung an den Endnutzer wird so einfach wie möglich gemacht (Kapitel 6.10).

7.1 Vorgehensweise

Bei den Vorgehensweisen gibt es neben Extreme Programming (XP) noch diverse klassische Vorgehensweisen (z. B. Wasserfallmodell, V-Modell). Sie sollen den Entwicklern einen Ablauf geben, wie die Software erstellt wird. Dafür werden oftmals verschiedene Phasen der Entwicklung fest definiert, die schrittweise abzuarbeiten sind.

Bei den klassischen Modellen werden normalerweise zuerst umfangreich Informationen gesammelt, bevor die erste Zeile Code geschrieben wird. Beim Wasserfallmodell beispielsweise nennt sich die erste Phase Analyse. Darin wird ein (Lasten-) und Pflichtenheft erstellt, welches detailliert die Anforderungen an die zu erstellende Software beschreibt. Aufgrund des hohen Detaillierungsgrads muss der Kunde genaue Vorstellungen haben, was er benötigt. Diese hat er oftmals aber gar nicht bzw. es sind meist nur ungenaue Vorstellungen vom Ziel vorhanden.

Der gesamte Prozess der klassischen Entwicklung ist mit viel Bürokratie verbunden, die viele Dinge von vornherein zu regeln versucht. So ist das Modell für spätere Änderungen anfällig. Bis man zu ersten sichtbaren Ergebnissen kommt, vergeht oftmals eine wesentlich größere Zeitspanne im Vergleich zu einer agilen Entwicklung.

Extreme Programming dagegen zählt zu den agilen Methoden, eine Software zu entwickeln. Die Bürokratie bei der Entwicklung soll verringert werden. Dafür soll schneller ein Ergebnis für den Kunden

ersichtlich und auch benutzbar sein. So soll der Kunde bereits während des laufenden Projektes von den Fortschritten profitieren. Außerdem kann der Kunde schnell seine bisherige Vorstellung von der Software in der Praxis überprüfen und gegebenenfalls rechtzeitig auf die Entwicklung Einfluss nehmen. Zu den Techniken in XP gehört es auch, besonderen Wert auf die Qualität der Software zu legen, durch z. B. Testen oder Programmieren in Paaren. Während bei letztem Punkt in klassischen Projekten sofort an die Kosten gedacht wird, begegnet XP dem Kostenargument mit einer wesentlichen Qualitätssteigerung der Software. Für mehr Informationen wird auf [be03] und [xp02] verwiesen.

Da dem Autor der Vorteil einer qualitätsbewussten Entwicklung mit XP wichtiger ist, als die langwierige (klassische) Entwicklung über Phasen mit viel Bürokratie, wird nach XP vorgegangen.

Extreme Programming

Bei XP stehen die einzusetzenden Techniken (siehe Kapitel 3.3) besonders im Vordergrund. Im Rahmen der Diplomarbeit als Einzelarbeit konnten nicht alle Techniken angewendet werden. XP sieht vor, dass auch Techniken an die individuellen Gegebenheiten angepasst werden, das ist, wenn möglich, auch erfolgt. Einige Techniken müssen weggelassen werden, da XP annimmt, dass es ein Entwicklerteam gibt und ein richtiger Kunde existiert. Die kurze Beschreibung der Techniken in Kapitel 3.3 gibt schon einen Aufschluss darüber, inwieweit eine Umsetzung im Rahmen der Arbeit erfolgen kann.

Zusätzlich zu den aufgeführten Techniken werden noch Benutzer-Geschichten (engl. user stories) und Aufgaben (engl. tasks) verwendet. Benutzer-Geschichten sind Beschreibungen des Kunden. Sie enthalten die Anforderungen des Kunden aus seiner Sicht an die Software. Dabei wird nicht erwartet, dass der Kunde genaue Angaben macht. Vielmehr sollen kleine Geschichten dabei entstehen, die eine Eigenschaft des Systems beschreiben. Die Empfehlung in [xp02] lautet, dass auch mindestens ein Entwickler an der Beschreibung beteiligt ist. Für die Testfallverwaltung müssen aus der Aufgabenstellung in Zusammenarbeit mit dem Kunden die Geschichten geschrieben werden.

Diese werden ergänzt um Aufgaben, die keine Eigenschaft der Software für den Kunden beschreiben. Die Aufgaben sind aber notwendig, um die Software intern in einer sauberen Struktur zu halten. Große Überarbeitungen werden während der Programmierung festgehalten, um sie später abzuarbeiten. Ergeben sich große Überarbeitungen in der Testfallverwaltung, wird dies in einer Aufgabe festgehalten.

Für XP als Vorgehensweise hat sich der Autor erst im Laufe der Diplomarbeit bewusst entschieden. Daraus resultiert, dass viele Techniken bereits von Beginn an implizit benutzt werden (z. B. das Testen). Andere Techniken wurden nur noch indirekt etabliert (z. B. das Planungsspiel).

Nachfolgend wird auf die XP-Techniken eingegangen:

- Kunde vor Ort: Als Kunde und Auftraggeber fungiert der betreuende Professor Dr. Grude, nachfolgend Kunde genannt. Ihn vor Ort (in der heimischen Wohnung) zu haben, ist nicht umsetzbar und macht auch keinen Sinn. Dafür finden regelmäßige Treffen im 2-wöchigen Rythmus in der TFH Berlin statt. Dazwischen steht der Kunde per E-Mail und telefonisch zur Verfügung.

  Benutzer konnten nicht hinzugezogen werden, da derzeit keine Veröffentlichung der Software möglich ist.

- Planungsspiel: Die Benutzer-Geschichten werden vom Kunden zu Beginn der Arbeit mit Prioritäten versehen, dies geschieht implizit, obwohl die Vorgehensweise noch nicht nach XP fest steht.

  Im Rahmen der Kundenbesprechungen werden die nächsten Iterationsschritte in Zusammenhang mit den Benutzer-Geschichten besprochen. Auch die Iterationen werden implizit besprochen, ohne sie als XP-Technik anzusehen.

- Metapher: Die Metaphern ergeben sich aus der Aufgabenstellung und den Benutzer-Geschichten.

- Kurze Releasezyklen: Es kommt nicht dazu, dass in Releasezyklen gearbeitet werden kann, da der Zeitraum der Diplomarbeit von drei Monaten dies nicht zulässt. Bei Abgabe der Arbeit wird das erste Release herausgegeben. Auf dem Weg dahin kann es aber mehrere Iterationen geben, die dem Kunden vorgestellt werden sollten.

- Testen: Modultests werden für besonders wichtig erachtet, da einige wegzulassende XP-Techniken kompensiert werden sollen. Außerdem betrifft die Testfallverwaltung wegen des Bezugs zu JUnit die Modultests selbst. Systemtests sind zu komplex, um sie im Rahmen der Diplomarbeit umzusetzen, daher fallen sie weg. Dazu kommt noch, dass Systemtests vom Kunden zu erstellen sind, der nur eingeschränkt zur Verfügung steht.

  Unter der Auflistung der Techniken wird der Ansatz für die testgetriebene Entwicklung genauer erläutert.

- Einfaches Design: Es gibt einen festen Satz an Anforderungen, die in den Benutzer-Geschichten beschrieben werden und umgesetzt werden sollen. In Absprache mit dem Kunden werden die Programmteile umgesetzt. Dabei ist nicht mehr als nötig zu programmieren.

- Überarbeitung (engl. refactoring): Die Software durch ständige Überarbeitungen strukturell sauber zu halten, ist wichtig, da das Projekt später quelloffen veröffentlicht werden soll.
- Fortlaufende Integration: Es soll ein reines Werkzeug zur Versionsverwaltung an der TFH Berlin genutzt werden. Allerdings gilt auch hier, dass diese Technik zurzeit wenig sinnvoll ist, da die Versionsverwaltung nur vom Autor benutzt wird. Es kann so zu keinen Konflikten mit anderen Integrationen kommen, da nur der Autor integriert.
- Programmierstandards: Obwohl die Arbeit eine Einzelarbeit ist, werden Programmierstandards für wichtig erachtet, da das Projekt später veröffentlicht werden soll. Es werden die Java Code Conventions als Vorlage genommen. Daraus werden eigene Standards abgeleitet und erarbeitet.

Auf Programmieren in Paaren, Gemeinsame Verantwortlichkeit wird nicht näher eingegangen, da Diplomarbeit als Einzelarbeit geleistet wird. Auf die 40-Stunden-Woche wird auch nicht näher eingegangen.

In der Lösungsausführung wird beschrieben, wie die umsetzbaren Techniken angewendet werden.

Testgetriebene Entwicklung

Der Ansatz sollte der testgetriebenen Entwicklung entsprechen. Das heißt, zuerst ist ein Test zu schreiben und dann den zugehörigen Code implementieren. Umzusetzen ist es in allen Programmteilen, in denen es sinnvoll anwendbar ist.

Die Modultests sollten die Methoden abdecken, die über den Bereich der Trivial-Methoden hinausgehen. Als Trivial-Methoden werden die funktionslosen `get`- und `set`-Methoden ausgeschlossen. Außerdem sollen nur sinnvolle Testfälle entwickelt werden.

7.2 Test-Klassen mit JUnit ausführen

Die Hauptfunktion der Testfallverwaltung soll es sein, Test-Klassen mit JUnit 4 auszuführen. Dazu muss an den JUnit-Ausführer eine Reihung von Klassen (`java.lang.Class<?>`) übergeben werden. Als Ergebnis erhält man ein Objekt der Klasse `org.junit.runner.Result`. Dieses ist im Daten-Modell aufzunehmen, um die Historisierung zu praktizieren.

7.3 Daten-Modell

Die Testfallverwaltung wird folgende Daten speichern müssen:

- Die aktuellen Test-Klassen.
- Alle Ergebnisse der Test-Ausführungen (`org.junit.runner.Result`).
- Ein Objekt zur Zusammenfassung, welches zu den Test-Klassen und deren Ausführungsergebnissen noch einen Namen und einen Klassenpfad speichert. Dieses Objekt wird im Folgenden als Projekt bezeichnet.

Durch die Verwendung einer objektorientierten Datenbank ist man beim Modellieren nicht daran gebunden, datenbank-gerecht im relationalen Sinn zu entwerfen. Es lässt sich demnach das volle Spektrum der objektorientierten Programmierung einsetzen.

7.4 Klassen-Ausführer oder Ausführer mit Übersetzer

Führt man nur Test-Klassen aus, die bereits übersetzt sind oder bietet man zusätzlich an, die Quell-Dateien zu übersetzen?

Ein reiner Ausführer hat den Vorteil, dass er die Klasse nur laden und mit JUnit ausführen muss. Dazu braucht der Ausführer den vollqualifizierten Klassennamen, um sie zu laden. Das setzt eine erfolgreiche Übersetzung voraus.

Ist die Klasse noch nicht übersetzt, müsste dieser Schritt vor dem Laden geschehen. Das ist um einiges aufwändiger, da die Klassen aus einem laufenden Programm mit den richtigen Argumenten übersetzt werden müssten. Tritt dabei ein Fehler auf, so muss dieser dem Benutzer erst angezeigt und dann von diesem behoben werden.

Da dieser Vorgang, die Klassen auch zu übersetzen, zu aufwändig und fehleranfällig ist, wird in dieser Arbeit davon ausgegangen, dass die Klassen bereits in Bytecode übersetzt sind.

7.5 Klassenlader und Klassenpfad

Die im Bytecode vorliegenden Dateien der Test-Klassen müssen vom Klassenlader geladen werden, um sie in der Java virtuellen Maschine (JVM) benutzen zu können. Neben dem vollqualifizierten Klassennamen (z. B. `de.berndsteindorff.junittca.TcaMain`) muss sich die Klassen-Datei im Klassenpfad befinden, damit der Klassenlader sie finden und laden kann.

Die Testfallverwaltung muss die Test-Klassen laden, um sie dann mit JUnit ausführen zu können. Dazu werden hier zwei verschiedene Wege beschrieben. Der erste Weg ist, den vom Benutzer zu definierenden Klassenpfad abzusuchen. Der zweite Weg ist, eine beliebige Klasse durch einen Dialog vom Benutzer auswählen und laden zu lassen (auch aus dem Klassenpfad).

Als Grundvoraussetzung gilt, dass die importierten Klassen der Test-Klassen sich im Klassenpfad befinden müssen.

Der Klassenlader muss noch die weitere Funktion aufweisen, die ausführbaren Test-Klassen vor jeder Test-Ausführung erneut einzulesen. So wird festgestellt, wenn die Klasse verändert wird. Ist sie verändert und sie kann neu geladen werden, muss die Testfallverwaltung nicht bei jeder Veränderung am Quellcode geschlossen werden. Damit lässt sie sich einfacher benutzen, denn die Veränderungen werden dynamisch zur Laufzeit einbezogen.

Damit muss der Klassenlader drei Funktionen erfüllen:

- Absuchen des Klassenpfades nach Test-Klassen.
- Laden beliebig ausgewählter Dateien als Test-Klasse.
- Neu laden der verwalteten Test-Klassen.

Diese werden in den folgenden Abschnitten beschrieben.

Absuchen des Klassenpfades

Am Einfachsten ist es, die Test-Klassen zu laden, indem davon ausgegangen wird, dass sie sich im Klassenpfad befinden. Allerdings müssen die Klassen der Testfallverwaltung mit Namen bekannt gemacht werden, um sie laden zu können. Der Klassenname liegt jedoch bei Übersetzung der Testfallverwaltung nicht vor. Da der Klassenname nicht bekannt ist, müsste man alle Verzeichnisse des Klassenpfades durchgehen und nach Klassen suchen, in denen sich Tests befinden. Problematisch wird das jedoch bei vielen Klassen und umfangreichen Verzeichnisstrukturen. So kann dieser Vorgang die Ladegeschwindigkeit maßgeblich verringern.

Unter dem Klassenpfad versteht man im Zusammenhang mit der Java-Programmierung eine Variable, welche Verzeichnisse, Jar-Dateien und Klassen-Dateien enthält. Die Variable wird beim Start der virtuellen Maschine (Java Virtual Machine) einmalig ausgelesen, hier wird nach Klassendateien über den vollqualifizierten Klassennamen gesucht. Sie gilt bis zum Stopp der virtuellen Maschine. Es gibt zwei Möglichkeiten den Klassenpfad zu setzen:

- Eine System- oder Benutzerumgebungsvariable mit der Bezeichnung `CLASSPATH` wird gesetzt, z. B.

```
Für Linux:
$ export CLASSPATH='/home/bernd/workspace/junittca/bin'

Für Windows:
$ SET CLASSPATH="C:\Eigene Dateien\workspace\junuttca/bin"
```

  Weitere Pfade können unter Windows mit einem Semikolon, unter Linux mit einem Doppelpunkt angehängt werden.

- Beim Ausführen des Projektes mit der Java virtuellen Maschine wird der Klassenpfad über die Option `-cp` gesetzt, z. B.

```
$ java -cp /home/bernd/workspace/junittca/bin
      de.testpackage.TestMain
```

Es sollten einige Kriterien zur Dateiauswahl getroffen werden, um nicht alle Klassen laden zu müssen. Ein Namenspräfix nach dem Muster `Test*.class` bzw. Suffix `*Test.class` mag zwar üblich sein, ist jedoch von JUnit nicht vorgeschrieben. In JUnit 4 lassen sich Test-Klassen nur daran erkennen, dass sie Methoden mit der Annotation `@Test` enthalten. Das lässt sich jedoch erst auslesen, wenn die Klasse erfolgreich vom Klassenlader geladen wurde.

Wird die Testfallverwaltung als jar-Datei ausgeliefert, so gibt es ein Problem mit dem Standard-Klassenpfad. Ein Start des Programms über den Befehl `java -jar` bewirkt, dass kein weiterer Klassenpfad angenommen wird. Beide o. g. Verfahren, den Klassenpfad zu setzen, scheitern dann. Es muss ein neuer Klassenlader geschrieben werden, der nach dem Standard-Klassenlader aufgerufen wird, um weitere Klassen aus einem separaten Pfad nachzuladen. Der Benutzer muss daher einen eigenen Klassenpfad definieren.

Als Klassenpfad für den Standard-Klassenlader werden auch einzelne `.class`-Dateien, sowie `.jar`-Dateien angenommen. Bei der Testfall-verwaltung erscheint das nicht sinnvoll, einzelne `.class`-Dateien als Pfad zu akzeptieren. Der Benutzer müsste alle weiteren Test-Klassen sowie alle importierten Klassen auch einzeln angegeben.

Einzelne `.jar`-Dateien als Pfad anzunehmen, wird für die Testfall-verwaltung auch nicht als sinnvoll angesehen. Tests werden laufend zur erstellten Software programmiert und überarbeitet. Programmierer werden nicht nach jeder Übersetzung ihrer Klassen eine Jar-Datei daraus erstellen.

Es werden daher zurzeit nur Verzeichnisse als Pfade angenommen. Wird ein anderer Pfad als ein existierendes Verzeichnis eingegeben, so sollte eine Ausnahme erzeugt und dem Benutzer eine Fehlermeldung angezeigt werden.

Hinzufügen einer beliebigen Test-Klasse

Als Alternative sollte es auch möglich sein, Test-Klassen dynamisch hinzuzufügen. Darunter wird verstanden, eine beliebige Klasse der Testfallverwaltung hinzuzufügen. Die Auswahl könnte über ein Dialogfenster geschehen, wobei die Klassen-Dateien einzeln oder in Gruppen hinzugefügt werden können. Das dynamische Hinzufügen ist schneller, da Dateien gezielt ausgewählt werden.

Der Nachteil ist jedoch, dass sich die Klassen möglicherweise nicht im Klassenpfad befinden. So wäre es denkbar, dass die Dateien zwar ausgewählt worden sind, allerdings vom Klassenlader nicht gefunden werden.

Ein eigener Klassenlader ist auch hier notwendig, der mit einem benutzerdefinierten Klassenpfad läuft.

Automatisches Neuladen der Test-Klassen vor einer Test-Ausführung

Die Test-Klassen sollten vor jeder Ausführung dynamisch zur Laufzeit erneut eingelesen werden, so dass die Testfallverwaltung nicht nach jeder Ausführung geschlossen werden muss.

Ansonsten würde eine Veränderung des Test- oder Programmcodes bei laufender Testfallverwaltung sich nicht auf weitere Ausführungen auswirken. Der Standard-Klassenlader und auch der eigene Klassenlader „kennen“ bei laufender Testfallverwaltung die Klasse deshalb, weil sie sie bereits eingelesen haben. Werden dann neue Objekte der Test-Klassen erzeugt, so werden sie von der virtuellen Maschine nach dem Muster der alten Klasse erzeugt. Die Testfallverwaltung müsste jedes Mal geschlossen werden, um den Klassenlader neu zu starten, damit er den veränderten Code erkennen kann.

Es muss daher ein eigener Klassenlader erstellt werden, der die Klassen-Dateien erneut einliest. So wirken sich Veränderungen an den Klassen auch bei laufender Testfallverwaltung auf die verwalteten Test-Klassen aus. Der Klassenlader benutzt dann den Klassenpfad, der ihm zum neu laden der Klassen übergeben wird.

Als Nachteil wirkt sich hier aus, dass ein eigener Klassenlader wesentlich mehr Laufzeit benötigt, da zum einen bei jeder Ausführung erstmal die

Klassen neu geladen werden und zusätzlich davor noch ein neuer Klassenlader erzeugt werden muss.

Die Vorteile, dass die Testfallverwaltung die Klassen überhaupt laden kann und zur Laufzeit alle Änderungen an den verwalteten Klassen mitbekommt, wiegt jedoch den Nachteil deutlich auf.

7.6 Persistenz der Test-Ergebnisse

Die Persistenz, also das Speichern der benutzten Daten über die Programmlaufzeit hinaus, ist heutzutage in vielen Projekten ein wichtiger Faktor.

Im Rahmen der Testfallverwaltung werden auch Daten anfallen, die über die Programmlaufzeit hinaus gespeichert werden sollen.

Alle Daten, die im Ansatz zum Daten-Modell beschrieben sind, müssen auch gespeichert werden. Aus quantitativen Gesichtspunkten wird dies keine große Datenmenge sein:

- Die Ausführung der Tests wird manuell gesteuert, die Daten fallen daher nicht in regelmäßigen Zeitabständen an.
- Die Anzahl der Modell-Klassen ist gering, die Anzahl der zu speichernden Variablen ebenfalls.
- Der Teil der Daten, der schwer schätzbar und variabel ist, stellt die Anzahl der Test-Klassen dar. Selbst in größeren Projekten dürfte die Anzahl der Test-Klassen im maximal dreistelligen Bereich bleiben.

  Je Ausführung kommt ein großer Daten-Block als Ergebnis hinzu.
- Die Testfallverwaltung ist erstmal als Einzel-Benutzer-System konzipiert, so dass keine gleichzeitigen Datenänderungen von parallelen Benutzern geschehen können. Dies schränkt die Menge der produzierten Daten zusätzlich ein.

Durch den letzten Punkt wird auch klar, dass eine Netzwerkfähigkeit keine Voraussetzung für die Persistenz-Lösung ist.

Die verhältnismäßig kleine Datenmenge und der ausreichende Einzel-Benutzer-Modus machen es möglich, verschiedene Lösungen in Betracht zu ziehen:

- Relationales Datenbanksystem
- Objektorientiertes Datenbanksystem
- Speicherung als XML-Datei

Ausgeschlossen wurde von vornherein die Serialisierung durch die Standard Java Klassen. Bei der Standard-Serialisierung werden die Objekt-Daten durch die Klasse `java.io.ObjectOutputStream` in eine Datei als Byte-Strom serialisiert. Werden die Daten wieder gelesen, geschieht der entgegengesetzte Prozess mit Hilfe der Klasse `java.io.ObjectInputStream`. Diese Art, Daten zu speichern, erscheint als veraltet, da es im Vergleich wesentlich bessere Lösungen gibt. Der Vorteil der Standard-Serialisierung liegt in der automatischen Serialisierung / Deserialisierung der Objektdaten. Der große Nachteil ist jedoch, dass es sich nur um eine einfache Datei handelt. Die Datei wird von den Stream-Klassen nicht wie innerhalb eines richtigen Datenbankbanksystems gehandhabt, wozu man z. B. den Umgang mit Transaktionen zählt.

Die aufgezählten Lösungen werden im Folgenden betrachtet.

Relationales Datenbanksystem

Das Konzept eines relationalen Datenbanksystems ist die am meisten verbreitete Technik zur Persistierung. Es fußt darauf, dass die verwaltete Datenmenge - die relationale Datenbank - in einem relationalen Datenbankmodell verwaltet wird. Die Verwaltungssoftware nennt sich Datenbankmanagementsystem.

Die aktuellen relationalen Datenbankmanagementsysteme (RDBMS) stellen ausgereifte Lösungen dar. Große Hersteller sind hier IBM (mit DB2), Oracle, Microsoft, MySql und PostgreSql. Die RDBMS sind seit den 70er Jahren im Einsatz und haben heute den größten Marktanteil unter den Datenbanksystemen.

Entscheidend bei den RDBMS ist das Datenmodell, das relational ist. Die Daten werden in Tabellen (den Relationen) abgespeichert. Die Tabellen wiederum enthalten Datensätze, die je Zeile ein Tupel darstellen. In Konkurrenz zum Datenmodell der Programmiersprache wird dazu ein eigenes (relationales) Datenbankmodell (Entity Relationship-Modell) erstellt.

Neben dem zweiten Datenmodell liegt ein weiteres Problem in den unterschiedlichen Datentypen. Ein relationales Datenbanksystem will die Anfragen an die Datenbank von verschiedenen Programmen ermöglichen, so muss es losgelöst von der jeweiligen Programmiersprache sein. Damit geht einher, dass die Datentypen oftmals nicht denen der Programmiersprache entsprechen, sondern eigene Datentypen mitgeliefert werden. Beispielsweise gibt es bei MySql 5.1 elf numerische Datentypen, während Java nur sieben numerische (primitive) Datentypen hat. Daraus können Probleme bei der Zuordnung der Datentypen entstehen.

Es ist eine eigene Sprache namens Structured Query Language (SQL) entwickelt worden. Damit werden Definitionen, Abfragen und Manipulationen auf Datenbanken erledigt. SQL lässt sich auf mehreren Datenbanksystemen benutzen, da die meisten es als Standard-Sprache unterstützen. Außerdem zieht die JDBC-Datenbankschnittstelle (JDBC = Java Database Connetivity) eine Abstraktion ein, welche die eigene Implementierung von der der Datenbank löst. Eine Beispielabfrage könnte so aussehen:

```
ResultSet rs = connection.createStatement
        .executeQuery(„SELECT * FROM Project;“);
```

Die Ergebnismenge `rs` vom (Typ `ResultSet`) enthält eine Liste mit den gefundenen Datensätzen. Iteriert man über die Menge, so lässt sich jedes Feld eines Datensatzes über seinen Datenbank-Bezeichner (als `String`) einzeln erfragen. Den erfragten Wert kann man dann einer Variablen zuweisen.

Die Datenmodelle aufeinander abzubilden, ist fehleranfällig, wenn man es manuell vornimmt. Um diesen Prozess zu vereinfachen und es dabei weniger fehleranfällig zu machen, gibt es inzwischen leistungsstarke Software-Komponenten, die als Bindeglied fungieren. Im Java-Bereich hat sich z. B. Hibernate (http://www.hibernate.org) etabliert. Dies erfordert zusätzlichen Aufwand bei der Implementierung.

Vorteile von RDBMS sind, dass sie:

- Aufgrund der langen Verfügbarkeit und Entwicklung ausgereift sind.
- Eine hohe Geschwindigkeit aufweisen.
- Eine größere Unabhängigigkeit von einem speziellen Anbieter durch Benutzung einer Datenbankschnittstelle wie JDBC sicherstellen.
- Stark skalierbar sind. Dies ist für die Testfallverwaltung nicht wichtig, da die Datenmenge gering ist.
- Viele nützliche Funktionen rund um Sicherheit und Transaktionalität anbieten.

Nachteile sind, dass sie:

- Ein Abbilden (engl. mapping) der Daten zwischen dem Datenbankmodell und dem der Programmiersprache notwendig machen.
- Meist nicht die gleichen Datentypen der benutzten Programmiersprache haben (beispielsweise unterscheiden sich die numerischen Datentypen zwischen MySql 5.1 und Java).
- Durch die Benutzung von SQL fehleranfälliger in der Programmierung sind, besonders bei Überarbeitungen
- Einen eindeutigen Schlüssel für jeden Datensatz einer Relation aufweisen müssen (beispielsweise ein extra Feld, dass eine Nummer aufnimmt).

Objektorientiertes Datenbanksystem

Ein objektorientiertes Datenbanksystem besteht aus der objektorientierten Datenbank und einem dazugehörigen Datenbankmanagementsystem. Zusammen wird dies als objektorientiertes Datenbankmanagementsystem (ODBMS) bezeichnet.

Die Lösungen sind erst seit der größeren Verbreitung von objektorientierten Programmiersprachen entwickelt worden. Ihr Datenmodell soll sich nativ an das der Programmiersprache (hier Java) anbinden. Demnach sollen die Objekte mit ihren Variablen in der Form gespeichert werden, wie sie die Java Virtuelle Maschine verwaltet. Das macht es für den Programmierer einfach, da er sich nicht darum kümmern muss, dass mehrere Datenmodelle aufeinander abgebildet werden müssen. Das garantiert auch, dass es keine abweichenden Datentypen, wie etwa bei relationalen Datenbanksystemen, gibt.

Die näher betrachtete objektorientierte Lösung ist db4o von db4objects. Db4o ist eine kleine Software-Lösung, die speziell für eingebettete Anwendungen, Echtzeit- Steuerungssysteme und Software-Pakete auf mobilen und Desktop Plattformen entwickelt wurde. Die Entwicklung geschieht in schnellen Intervallen, so dass in kurzen Abständen neue Versionen mit größerem Funktionsumfang oder Geschwindigkeitssteigerung erscheinen.

Aufgrund der gering zu erwartenden Datenmenge ist db4o hier perfekt geeignet.

Db4o ist sowohl für Java als auch .NET verfügbar, dass heißt, es ist für die beiden Sprachen nativ entwickelt. Bei db4o besteht ein Unterschied darin, ob die Daten nur lokal gespeichert werden oder im Netzwerk bereit stehen sollen. Da die Testfallverwaltung erst einmal nur für einen einzelnen Benutzer erstellt wird, reicht die lokale Variante. Die

Datenbank wird so ähnlich verwendet, als öffne man eine Datei und wird innerhalb der gleichen virtuellen Maschine benutzt. Das erspart, die Datenbank netzwerkfähig zu machen.

Soll später die Netzwerkfähigkeit hinzukommen, so sind am bestehenden Code kaum Änderungen zu machen. Es muss nur ein Server aufgesetzt werden. Im Sinne von db4o ist damit gemeint, dass ein Server gestartet werden muss. Dies kann innerhalb der gleichen virtuellen Maschine, aber auch in einer separaten Sitzung auf dem gleichen oder einem fremden Rechner geschehen.

Abfragen und Manipulationen lassen sich direkt in der Programmiersprache in objektorientierter Weise ausdrücken (in db4o wird dies als Native Queries bezeichnet). Eine Beispielabfrage könnte so aussehen:

```
ObjectSet<Project> list = db.query( new Predicate<Project> () {
    public boolean match ( Project candidate ) {
        return true;
    }
});
```

Das Ergebnis der Abfrage wird eine Liste mit allen Projekten sein. Über die `match`-Methode (der inneren Klasse `Predicate` als Argument der `query`-Methode) lässt sich die Ergebnismenge beeinflussen. Die Rückgabe von `true` bewirkt hier, dass alle gespeicherten Projekte zurückgegeben werden. Iteriert man über die Ergebnismenge, so erhält man direkt das jeweilige Objekt zurück, also die einzelnen Projekte. Beim Auslesen der Datenbank sind bis zu einer konfigurierten Ebene im Objektgraphen auch die Variablen des Projektes, deren Variablen usw. gefüllt.

Das Datenmodell für eine Klasse richtet db4o automatisch ein, wenn ein Objekt (einer Klasse) das erste Mal gespeichert wird. Änderungen am Datenmodell der Klassen werden von db4o bis zu einer gewissen Komplexität ohne Probleme übernommen. Den Programmiercode inkl. der Variablen kann man auch weitergehend überarbeiten (engl. refactoring), ohne dass man dem Code für die Persistenz besondere Aufmerksamkeit schenken muss.

Vorteile von ODBMS sind, dass sie:

- Einfach zu implementieren sind.
- Den Programmierer das objektorientierte Datenmodell und deren Datentypen beibehalten lassen.

Vorteile speziell von db4o sind:

- Dass die Datenbank innerhalb der gleichen virtuellen Maschine läuft. Sie kann aber auch schnell netzwerkfähig umgestellt werden.
- Dass der eindeutige Schlüssel in der Objektidentität liegt.
- Dass Abfragen auf der Datenmenge mittels Native Queries direkt in Java möglich sind.
- Dass db4o schnell weiter entwickelt wird.

Nachteile sind, dass sie:

- Eine direkte Bindung an einen speziellen Anbieter aufbauen. Dies betrifft sowohl die Wahl der Programmiersprache als auch die starke Bindung der Implementierung. Es fehlt eine Abstraktionsschicht wie etwa JDBC.
- Schwer vergleichbar bei der Geschwindigkeit im Verhältnis zu relationalen Datenbanksystemen sind. Es gibt zwar unter `http://www.db4o.com/about/productinformation/benchmarks/` einen Vergleich mit anderen frei erhältlichen relationalen Datenbanksystemen, jedoch fehlen hier die großen Anbieter wie Oracle, IBM und Microsoft.

Speicherung in XML-Datei

XML-Dateien speichern ihre Daten im simplem Textformat, allerdings in hierarchisch strukturierter Form.

Es gibt verschiedene Verarbeitungsstrategien wie DOM, SAX und StaX. Für Java gibt es diverse Rahmenprogramme, die eine dieser Verarbeitungsstrategien umsetzt.

Diese Rahmenprogramme als XML-Verarbeitern sind darum bemüht, die Geschwindigkeit und den Speicherverbrauch zu optimieren. Sie bringen normalerweise keine Datenbankfunktionalität wie etwa Transaktionen oder Netzwerkfähigkeit mit.

Die Objekte müssen wie beim relationalen Datenmodell vom Objektmodell auf die Struktur der XML-Datei abgebildet werden. Ähnlich dem Entity Relationship-Modell gibt es eine Beschreibung in Form einer DTD oder eines Schemas.

Da die Nachteile bei der Speicherung in einer XML-Datei überwiegen, scheidet diese Persistenz-Lösung aus.

Entscheidung für eine Persistenz-Lösung

Zur abschließenden Entscheidung über die Persistenz-Lösung stehen zur Auswahl, ein relationales (RDBMS) oder ein objektorientiertes Datenbanksystem (ODBMS) zu benutzen.

Die ODBMS bieten ihre Stärken darin, sich an die Programmiersprache anzulehnen. RDBMS sind dagegen unabhängiger von der Implementierung. Außerdem sind RDBMS ausgereifter aufgrund ihres zeitlichen Entwicklungsvorsprungs.

Die Testfallverwaltung stellt keine großen Ansprüche an eine Datenbank. Die großen Vorteile der objektorientierten Ausrichtung von db4o führen dazu, dass sie verwendet wird. Die Unabhängigkeit die sich durch RDBMS ergibt, ist bei diesem kleinen Projekt nicht erforderlich und erfordert viel zusätzlichen Aufwand, der anderweitig sinnvoller investiert werden kann.

Da db4o im Einzel-Benutzer-Modus eingesetzt werden soll, muss es wenigstens möglich sein, die entsprechende Datenbank-Datei an andere Benutzer weiterzugeben. Der Benutzer soll steuern können, wo und wie die Datenbank-Datei gespeichert wird. So können verschiedene Datenbank-Dateien benutzt werden. Außerdem kann man die Datei so weiterleiten oder sie auf anderen Rechnern verwenden.

Die Datenbank-Datei soll der Behälter für Projekte sein. Im Nachfolgenden wird eine Datenbank-Datei als Projekt-Datei bezeichnet.

7.7 Oberfläche der Testfallverwaltung

Die Oberfläche soll dem Benutzer eine komfortable Möglichkeit bieten, die Testfallverwaltung zu benutzen. Er soll damit folgende Aktionen ausführen können:

- Eine Projekt-Datei anlegen und öffnen.
- Ein Projekt anlegen, öffnen, speichern und löschen.
- Zum Projekt einen Namen und einen Klassenpfad speichern.
- Einen oder mehrere Test-Klasse/n über einen Auswahldialog dem Projekt hinzufügen.
- Test-Klassen im Klassenpfad suchen und dem Projekt hinzufügen.
- Entfernen von einzelnen oder allen Test-Klassen aus dem Projekt.
- Ausführen der verwalteten Test-Klassen.
- Überblick über die Historie und die verwalteten Klassen bieten.

- Ansicht für ein Test-Ergebnis bereitstellen.
- Ansicht für ein gespeichertes Test-Ergebnis bereitstellen.
- Ansicht um mehrere Test-Ergebnisse zu vergleichen.

Umgesetzt werden soll die Oberfläche mit den Grafik-Bibliotheken der Standard Java-Bibliothek. Dort stehen AWT und Swing zur Auswahl. Es wird die modernere Bibliothek Swing benutzt, da sie dem Benutzer mehr Komfort bietet und sich besser in die verschiedenen Betriebssysteme integriert. Natürlich werden dabei auch einige Klassen aus AWT benötigt und verwendet.

Die Oberfläche soll die geforderten Aktionen einfach darstellen, allerdings dabei ein gewisses Maß an Komfort bieten. Der Benutzer soll die Oberfläche möglichst intuitiv benutzen können. Wenn das nicht möglich ist, sollte er durch entsprechende Hinweise an sein Ziel geführt werden.

Bei der Programmierung von Oberflächen ist es üblich, dass die Ansichten von der Steuerung und der Logik getrennt sind. Die Testfall-verwaltung hat keine umfangreiche Anzahl von Ansichten. Daher sollen Entwurfsmuster nur dann eingesetzt werden, wenn sie passend sind. Nach XP soll erstmal so einfach wie möglich entwickelt werden. Im Hinblick auf spätere Erweiterungen soll trotzdem eine gewisser Grad an Entkoppelung passieren.

7.8 Ausnahmen

Während der Programmausführung können Fehler auftreten. Geschieht dies, muss entweder das Programm ordnungsgemäß beendet werden, oder der Benutzer muss die Möglichkeit haben, auf den Fehler zu reagieren.

In Java ist es üblich, dass sich der Programmierer nur um die geprüften Fehler, die Ausnahmen (engl. exception), kümmert. Die vom Java-Ausführer geworfenen Ausnahmen werden dabei abgefangen und sollten behandelt werden, ansonsten terminiert das Programm sofort. Wird eine Ausnahme abgefangen, so ist es dem Programmierer möglich, einen sinnvollen Umgang mit der Ausnahme zu finden.

In der Testfallverwaltung werden Ausnahmen hauptsächlich in Programmteilen auftreten, wo Klassen geladen oder Datenbank-operationen ausgeführt werden. Dies sind alles Ausnahmen, die auch der Benutzer erfahren sollte, um entsprechend darauf zu reagieren.

Daher sollten die abgefangenen Ausnahmen dem Benutzer so aussagekräftig wie möglich dargestellt werden. Er sollte dabei am besten ein Fenster angezeigt bekommen. Dies sollte eine Meldung enthalten, aus der auch kleine Entscheidungen vom Benutzer abgefragt werden können.

7.9 Probleme bei der Plattformunabhängigkeit

Da die benutzte Programmiersprache darauf bedacht ist, plattformunabhängig zu sein, muss die Testfallverwaltung auch auf verschiedenen Betriebssystemen funktionieren und getestet werden.

Der Autor verfügt über drei Rechner (privates Notebook und PC, sowie Labor-PC mit Terminal-Server-Anbindung in der TFH Berlin). Darauf werden zwei verschiedene Versionen von Kubuntu (Linux-Kernel mit Fenster-Manager), Windows XP und Windows Server 2003 eingesetzt. Da ein System mit Linux-Kernel einer Unix-Umgebung ähnlich ist, dürfte es trotz fehlender Tests unter einem richtigen Unix oder auch Mac OS zu keinen Problemen kommen. Andere Betriebssysteme konnten nicht getestet werden.

Um das Programm auf allen genannten Betriebssystemen lauffähig zu machen, muss entsprechend plattformunabhängig programmiert werden. Besonders betrifft das folgenden Programmcode:

- Dateioperationen: Besonders Verzeichnisstrukturen und -trenner unterscheiden sich, unter Windows ist „\“ und unter Linux/Unix „/“ der Verzeichnistrenner. Daneben gibt es noch weitere, hier relevante Abweichungen, wie die Bezeichnung von Geräten, Partitionen und das Heim-Verzeichnis des Benutzers.
- Klassenpfad: Der Pfadtrenner unterscheidet sich auf den verschiedenen Betriebssystemen.
- Oberfläche: Je nach verwendetem Erscheinungsbild (engl. look & feel) gibt es Unterschiede, wenn die Oberflächen gezeichnet werden.

7.10 Auslieferung der Testfallverwaltung

Nachdem die Lizenzproblematik geklärt ist, soll die Testfallverwaltung verbreitet werden. Damit sie möglichst leicht verbreitet werden kann, sollte es dem Benutzer möglichst einfach gemacht werden, das System auszuführen und zu benutzen.

Bei Java-Projekten bietet es sich an, die Software nicht einfach als Ansammlung von einzelnen Klassen in Verzeichnissen auszuliefern, sondern ein Jar-Archiv daraus zu erstellen. Ein Jar-Archiv fasst im einfachen Fall nur die Klassen in einer Datei mit der Endung `.jar`

zusammen. Zusätzlich lässt sich u. a. das Archiv noch komprimieren und mit einer speziellen Datei (Manifest) versehen, die die Hauptklasse (Klasse mit `main`-Methode, die den Startpunkt für die Programmausführung darstellt) angibt.

Ist in dem benutzten Betriebssystem der Befehl `java -jar` mit dem Java-Ausführer (z. B. `java.exe` unter Windows) verknüpft, so lässt ein Doppelklick die Hauptklasse ausführen.

Ein Jar-Archiv kann mit dem Kommandozeilenprogramm `jar` aus dem Java Development Kit (JDK) erstellt werden. Komfortabler und einfacher automatisierbar ist es, Ant als Build-Werkzeug zu benutzen. Man schreibt eine Build-Datei im XML-Format um Ant mit Anweisungen zu konfigurieren. Die Build-Datei lässt sich leicht in Entwicklungsumgebungen wie Eclipse bearbeiten und ausführen.

Die Testfallverwaltung soll als Jar-Archiv ausgeliefert werden, welches ohne weitere Installation lauffähig ist. Ant soll das Werkzeug darstellen, um das Jar-Archiv zu erstellen.

8. Lösungsausführung

Die Lösungsausführung beschreibt die Umsetzung des erarbeiteten Ansatzes.

Die Vorgehensweise Extreme Programming wird zuerst beschrieben (Kapitel 7.1). Dann wird gezeigt, wie die Tests mit JUnit 4 ausgeführt werden (Kapitel 7.2). Das Datenmodell ergibt sich aus den Test-Ergebnissen und den verwalteten Klassen (Kapitel 7.3). Um die Klassen zu laden, wird ein eigener Klassenlader erstellt, der mehrere Funktionen für die Testfallverwaltung erfüllt (Kapitel 7.4). Mit db4o werden die Daten dann gespeichert. Dabei sind einige Besonderheiten im Umgang mit db4o beschrieben (Kapitel 7.5). Die graphische Oberfläche wird mit Swing realisiert und ausgestaltet, darin wird ein Entwurfsmuster eingesetzt, um die Oberfläche vom Kern der Anwendung zu entkoppeln (Kapitel 7.6). Erwartete Fehler (Ausnahmen) werden für den Benutzer aufbereitet und angezeigt (Kapitel 7.7). Die Plattformunabhängigkeit wird durch Java erleichtert, muss jedoch bei Pfaden und Verzeichnissen angepasst werden (Kapitel 7.8). Der Endnutzer bekommt die Testfallverwaltung als gepackte Zip-Datei geliefert (Kapitel 7.9).

8.1 Vorgehensweise Extreme Programming

Es werden zuerst die Benutzer-Geschichten und Aufgaben beschrieben, danach wird auf die angewendeten Techniken eingegangen.

Benutzer-Geschichten

Die Benutzer-Geschichten (engl. user stories) verdeutlichen aus Sicht des Kunden, welche Leistung das zu erstellende Programm zu erbringen hat.

Geschichte1: Der Benutzer startet eine graphische Benutzeroberfläche. Dort kann er Test-Klassen per Dialogauswahl öffnen oder aus einem bestimmten Pfad laden lassen. Die Tests werden mit JUnit ausgeführt. Das Ergebnis wird angezeigt.

Geschichte2: Der Benutzer lässt sich das Ergebnis eines zuvor ausgeführten Test-Laufes anzeigen.

Geschichte3: Der Benutzer kann zwei zuvor ausgeführte Test-Läufe auswählen und ihre Ergebnisse miteinander vergleichen lassen. Die Unterschiede werden dargestellt.

Geschichte4: Der Benutzer schließt die graphische Oberfläche, startet sie später wieder und kann auf das vorher bearbeitete Projekt zugreifen.

Aufgaben

Zu den Benutzer-Geschichten gibt es vier Aufgaben (engl. tasks), die unabhängig vom Kunden zu erledigen sind. Sie haben sich im Laufe der Entwicklung ergeben.

Aufgabe1: Die Klassen müssen über einen eigenen Klassenlader verfügen, da kein Klassenpfad vorher gesetzt werden kann, wenn die Testfallverwaltung aus einer Jar-Datei gestartet wird.

Aufgabe2: Die Klassen müssen über einen eigenen Klassenlader neu geladen werden, bevor sie mit JUnit ausgeführt werden. Ansonsten sind mögliche Veränderungen an den verwalteten Klassen nicht zur Laufzeit der Testfallverwaltung verfügbar.

Aufgabe3: Umstellung der Oberfläche auf Model-View-Controller-Muster.

Aufgabe4: Umstellung des Daten-Modells und des Klassenladers auf ein einfacheres Design.

Kunde vor Ort

Fachlich steht der Kunde zur Seite. Es wird jedoch in der Diplomarbeit darauf abgezielt, dass der Autor eigenes Wissen über das Anwendungsgebiet erwirbt, um die Testfallverwaltung zu erstellen.

Planungsspiel

Zu Beginn werden die Geschichten geplant. Da zu diesem Zeitpunkt XP noch nicht explizit benutzt wird, sind die Geschichten und das Planungsspiel nicht unter dem XP-Aspekt betrachtet, sondern formlos besprochen worden.

Für das erste Release hat die Geschichte1 die oberste Priorität bekommen. Danach folgend die Geschichte2, Geschichte4 und Geschichte3. Während der Arbeitstreffen werden die nächsten Iterationsschritte besprochen. Die Schritte orientieren sich an den Geschichten.

Metaphern

Es gibt vier Metaphern, die sich aus der Aufgabenstellung und den Benutzer-Geschichten ergeben:

- Test-Ausführer
- Historisierung von Test-Ergebnissen
- Ergebnis-Ansicht
- Ergebnis-Vergleich

Testen / Testgetriebene Entwicklung

Die Testfallverwaltung wird in seinen Kern-Programmteilen testgetrieben entwickelt. Alle Test-Klassen sind mit dem Suffix `Test` versehen.

Zu Beginn werden Lerntests geschrieben, um die Funktionsweise von JUnit zu erarbeiten. Die Lerntests gehen später in größeren Tests auf.

Das danach erstellte Daten-Modell wird nur an komplexeren Stellen getestet. Dazu zählen einige Konstruktoren sowie einige `get`- und `set`-Methoden. Außerdem werden die `equals`-Methoden getestet. Dabei wird ein Problem mit der `equals`-Methode der Klasse `java.lang.reflect.Method` festgestellt. Es lässt sich zwar mit Tests einkreisen, aber nicht beheben, da die Klasse aus der Java Standardbibliothek ist.

Tests für die Datenbank sind anfangs aufteilt in Tests, die auf die Festplatte schreiben, und Tests, die nur im Hauptspeicher arbeiten. Der Unterschied wird gemacht, da für jeden Modultest normalerweise eine eigene Testumgebung und damit eine separate Datenbank-Datei angelegt wird. Dieser Prozess ist jedoch zeitintensiv und belastend für die Festplatte. Die Aufteilung wird später wieder rückgängig gemacht, da der Ein- und Ausgabe-Adapter von db4o für den Hauptspeicher (`com.db4o.io.MemoryIoAdapter`) nicht erwartungsgemäß arbeitet (die Ausführung hält ohne einen Fehler vor dem Ende der Anweisungen an).

Die umfangreichsten Tests gibt es für den eigenen Klassenlader. Alle drei Funktionalitäten des Klassenladers werden ausgiebig getestet.

Soweit möglich, werden auftretende Ausnahmen simuliert und es wird getestet, ob sie korrekt geworfen werden. Der Parameter `expected` der Annotation `@Test` eignet sich nur bedingt, um die Ausnahmen zu prüfen. Daher werden die Ausnahmen mittels `try/catch`-Befehl abgefangen, um den genauen Text der Ausnahme zu testen.

Im Juli findet eine große Überarbeitung (Aufgabe4) des Daten-Modells statt. Davor funktionieren 33 Tests. Die Überarbeitung betrifft neben dem Daten-Modell stark den Klassenlader. Davon sind ca. drei Viertel der Tests betroffen. So wird damit begonnen, die Tests komplett neu zu schreiben. Zuerst wird das geänderte Daten-Modell getestet. Danach werden auch neue Tests für den Klassenlader erstellt, der ebenfalls komplett neu geschrieben wird. Die restlichen Tests werden nach und nach wieder aktiviert. Nach der Überarbeitung sind 41 Tests aktiv.

Einfaches Design und Überarbeitung

Diesen beiden Techniken zufolge werden Programm-Bereiche nach und nach entwickelt, dazwischen wird überarbeitet. Kleinere Überarbeitungen sind laufend geschehen, es werden nur vier große Überarbeitungen erläutert, die zu den Aufgaben führen.

Programmiert wird in der Reihenfolge, die bereits im Punkt zuvor beschrieben ist. Durch Tests gestützt, wird zuerst der Bereich um JUnit entwickelt. Es wird ein separates Software-Projekt als Test-Basis für die eigenen Programmteile erstellt. Das Daten-Modell und die Persistenz werden danach entwickelt.

Beim Klassenlader wird festgestellt, dass die erste Lösung zu einfach ist. Dort wird davon ausgegangen, dass der Standard-Klassenlader benutzt werden kann. Im Auslieferungszustand kann jedoch kein Klassenpfad übergeben werden. Das führt zur Aufgabe1 und der ersten Überarbeitung.

Als letzter großer Programmteil folgt die Oberfläche. Hier wird festgestellt, dass Klassen neu eingelesen werden müssen, um Veränderungen daran zu erfassen. Dies führt zur zweiten Überarbeitung in Aufgabe2. Im Verlaufe der weiteren Oberflächen-Programmierung wird die Struktur, wie Anzeige und Steuerung mit dem Daten-Modell zusammenarbeiten, unübersichtlich. Es ergibt sich die dritte Überarbeitung in Aufgabe3, nach der das Model-View-Controller-Muster eingeführt wird.

Nach Fertigstellung der Oberfläche findet eine Prüfung der Funktionalität statt. Dabei treten Probleme mit der Datendarstellung auf. Die Verursacher sind im Daten-Modell angesiedelt. Die Technik eines einfachen Designs wird dort nicht genügend beachtet, was zur Aufgabe4 mit der letzten großen Überarbeitung führt. Das Daten-Modell ist danach einfacher und die Oberfläche ist voll funktionsfähig.

Fortlaufende Integration

Am Tagesende werden die Arbeiten an den Dokumenten und dem Quellcode mit dem Subversion-Server des SWE-Labors der TFH-Berlin ausgetauscht. Dabei werden alle Dateien des Projektes übertragen und damit gesichert. So wird ohne Probleme an mehreren Arbeitsplätzen am Projekt gearbeitet.

Programmierstandards

Die Java Code Conventions werden als Vorlage genommen. Daraus werden eigene Standards abgeleitet und erarbeitet. Ein Großteil davon ist bereits am Beginn umgesetzt worden, der Rest folgte, nachdem die Programmierstandards ausgearbeitet sind.

In Anhang D findet man eine Übersicht der Programmierstandards.

8.2 Test-Klassen mit JUnit ausführen

Die in der Testfallverwaltung gespeicherten Test-Klassen werden als Reihung an den JUnit-Ausführer `JUnitCore.runTests(Class<?>... classes)` übergeben. Zurückgegeben wird ein Objekt der Klasse `Result`. Die Rückgabe wird im Daten-Modell festgehalten.

8.3 Daten-Modell

Das Daten-Modell lässt sich am Besten durch ein vereinfachtes Klassendiagramm beschreiben. Im nachfolgenden Diagramm sind nur die Daten-Klassen mit ihren Attributen, Beziehungen und Vererbungen zu sehen.

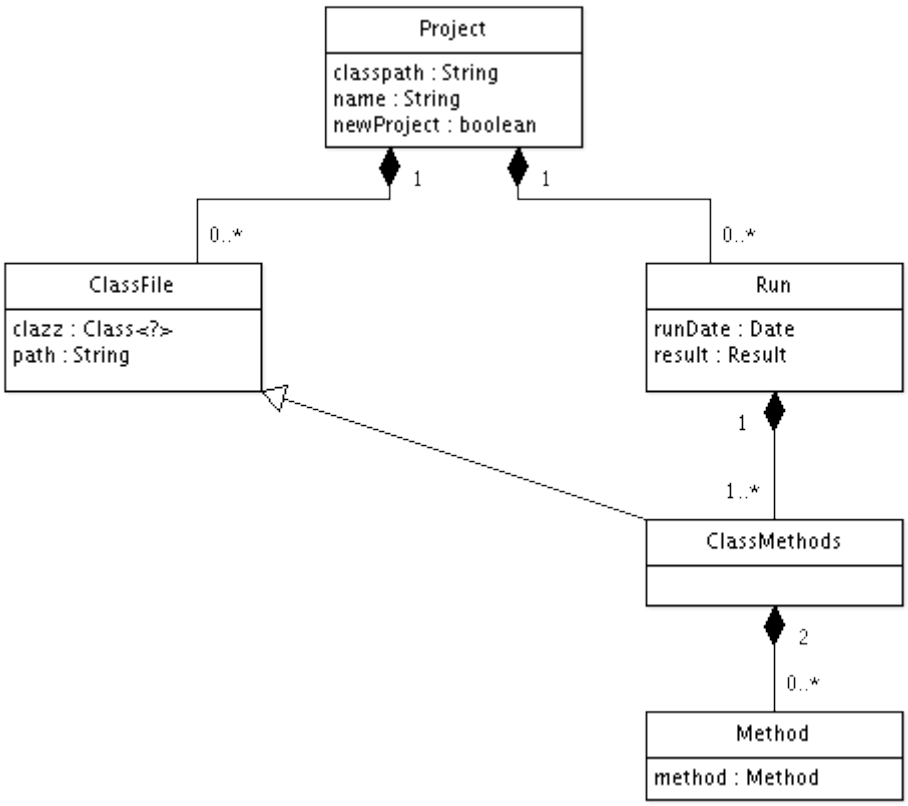


Abbildung 1: Vereinfachtes Klassendiagramm des Daten-Modell

Aufgrund der Benutzung einer objektorientierten Datenbank ist das Modell vollkommen nach objektorientierten Ansätzen erstellt.

Die elementare Datenklasse stellt das `Project` dar. Man kann sie sich auch als Wurzel vorstellen. Neben einem Namen enthält sie einen speziellen Klassenpfad und einen booleschen Wert, der angibt, ob das Projekt neu ist. Dazu werden alle hinzugefügten Test-Klassen (`ClassFile`) und deren vergangene Ausführungen (`Run`) jeweils in einer Liste gespeichert. Die Objekte der Klassen Run und ClassFile sind als Komposition aufgenommen. Durch diese Beziehung sind alle gespeicherten Objekte stark mit dem Projekt verbunden, wird es gelöscht, werden auch die gespeicherten Objekte gelöscht. Wird der Klassenpfad neu gesetzt, so wird überprüft, ob er ein existierendes Verzeichnis ist. Wird der Name geändert, so wird überprüft, ob es bereits ein anderes Projekt mit dem eingegebenen Namen gibt. In beiden Fällen wird eine Ausnahme geworfen, wenn die Prüfung nicht wie erwartet verläuft.

Eine Test-Klasse wird in der Klasse `ClassFile` zusammen mit dem Pfad (`String`) dargestellt. Es konnte nicht direkt die Standard-Klasse `java.lang.reflect.`Class<?> genutzt werden, da ihr einige Eigenschaften fehlen:

- Um die Anzeige in der Oberfläche geordnet darstellen zu können, müssen die Klassen sortierbar sein. Die Standard-Klasse `Class<?>` ist aber nicht sortierbar. Daher wird die Schnittstelle `Comparable<ClassFile>` in `ClassFile` übernommen, um so die Methode `compareTo(ClassFile other)` zu überschreiben. So lassen sich Listen mit diesem Typ sortierbar machen.
- Der Pfad wird in `Class<?>` nicht mitgespeichert, er wird jedoch benötigt, um die Klassen neu zu laden.

Die Klasse `Run` stellt eine einzelne Ausführung eines Testlaufs dar. Um eine Chronologie aufbauen zu können, enthält sie einen Datum- und Zeitstempel (`Date`). Das Ergebnis der JUnit-Ausführung (`Result`) wird festgehalten. Alle ausgeführten Klassen mit ihren Test-Methoden und ignorierten Test-Methoden werden in der Klasse `ClassMethods` in zwei Listen gespeichert. Diese Klasse ist eine Erweiterung von `ClassFile`. Wie auch in der Oberklasse sind die Variablen `final`. Die Objekte von `Run` werden in absteigender chronologischer Reihenfolge (der neueste Test-Lauf steht somit zuerst) in der Liste gespeichert.

Die Methoden werden in eine eigene Klasse `Method` ausgelagert, da sich die Standard-Klasse (`java.lang.reflect.Method`) nicht sortieren lässt.

Die Listen von `Project` (`ClassFile` und `Run`) und `ClassMethods` (2x `Method`) werden unveränderbar zurückgegeben. Diese Schutzmaßnahme

wird getroffen, weil die Listen nicht mehr veränderbar sein sollen. Die Klassen werden nur durch ihren Konstruktor gesetzt und sollen dann nur noch lesbar sein. Bei Listen ist dieser Punkt problematisch, da bei Rückgabe der Referenz auf die Liste darauf ansonsten Operationen wie `add(...)` ausführbar sind. Die Listen müssen demnach in einem schreibgeschützten Zustand versetzt werden. Die Klasse `Collections` aus dem `java.util`-Paket stellt die helfende Methode zum Schreibschutz (`unmodificable(...)`) bereit. Auf der damit zurückgegebene Liste lassen sich die Methoden `add`, `remove` und `set` nicht mehr ausführen, ohne dass es eine Ausnahme gibt. Beachtet werden muss, dass durch den Schutz nur die Listen unveränderbar werden, nicht jedoch die gespeicherten Objekte. Die Listen werden auch durch die Klasse `Collections` sortiert (`sort`-Methode).

In `ClassFile` und `ClassMethods` gibt es je eine `equals`-Methode. Sie erleichtern es, Tests zu schreiben. Die Vergleiche sind allerdings einfach aufgebaut, es werden nur Zeichenketten (`String`) verglichen. Bei `ClassMethods` werden zusätzlich die Methoden (`java.lang.reflect.Method`) verglichen. Alle anderen benutzen Klassen (wie `Class<?>` und `Result`) haben keine eigene `equals`-Methode und sind so nicht hilfreich für die eigene `equals`-Methode.

Alle Zugriffe auf die Klassen und Datenfelder sind auf das Notwendigste beschränkt. So gibt es nur eine `set`-Methode in allen Datenklassen. Wenn möglich, sind die Felder noch final gesetzt.

Durch die Oberfläche wird festgestellt, dass `ClassFile` und `ClassMethods` Probleme in der Darstellung machen. In der folgenden Überarbeitung ist das Daten-Modell einfacher geworden. Hierzu ein neues Klassendiagramm:

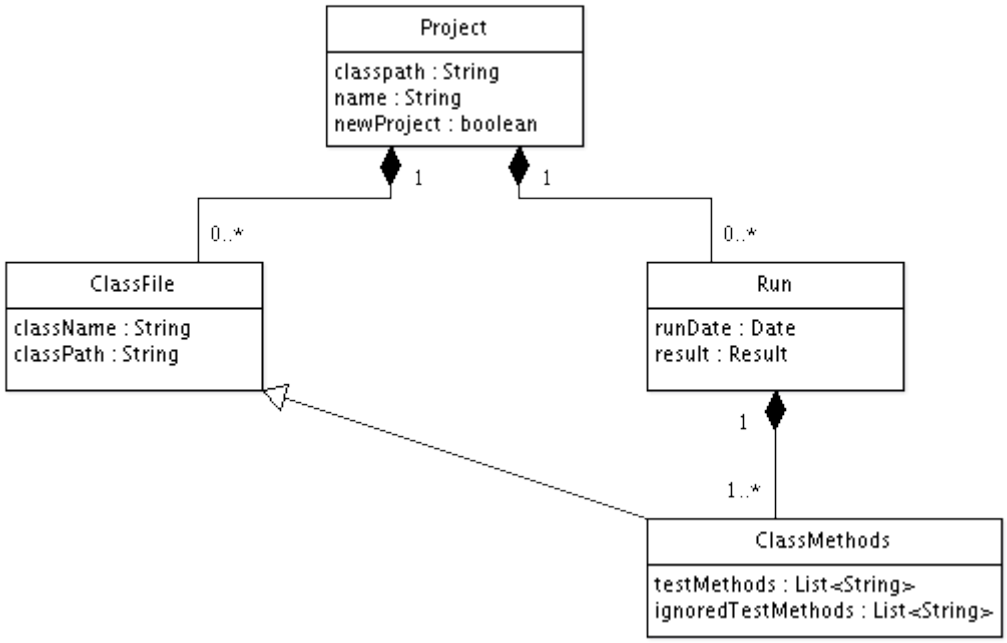


Abbildung 2: Neues Klassendiagramm des Daten-Modell

Im neuen Daten-Modell sind Class<?> und Method entfallen, an die Stelle sind Zeichenketten getreten. Der Klassenlader (siehe Kapitel 7.4) hat damit keine Probleme, denn er benötigt sowieso nur Zeichenketten als Pfad und Klassennamen.

8.4 Klassenlader und Klassenpfad

Vom Ansatz her gibt es drei Funktionen, in denen ein eigener Klassenlader benötigt wird:

- Durchsuchen eines definierten Klassenpfades nach Test-Klassen.
- Laden beliebig ausgewählter Dateien als Test-Klasse.
- Neu laden der verwalteten Test-Klassen.

Um Test-Klassen dem Projekt hinzuzufügen, wird für die ersten beiden Prozesse auch eine Schaltfläche in der Oberfläche bereitgestellt.

An sich könnte es einfach sein, Klassen zu laden. Hierfür gibt es die Methode `java.lang.Class.forName(String className)`. Das setzt allerdings zwei Bedingungen voraus:

- Man muss wissen, welche Klassen man laden will. Es wird der vollqualifizierten Klassennamen benötigt, wie etwa

  ```
  de.berndsteindorff.junittca.Test
  ```

- Die Klasse muss im Standard-Klassenpfad liegen.

Da besonders die zweite Bedingung bei Auslieferung als jar-Datei nicht erfüllt werden kann, benötigt man einen eigenen Klassenlader.

Bei beiden Varianten des Hinzufügens ist der vollqualifizierte Klassenname nicht bekannt. Man hat allerdings die Information, dass eine gefundene Datei in einem Pfad liegt (im Beispiel ist `/Pfad/zu/Projekt/bin` der angenommene Klassenpfad), z. B.

```
/Pfad/zu/Projekt/bin/de/berndsteindorff/junittca/Test.class
```

Nun muss zur Laufzeit der vollqualifizierte Klassenname entnommen werden. Der Programmierer kann es sich einfach machen, indem man einfach jeden möglichen Pfad probiert in einen Paketnamen umzuwandeln. Zu beachten ist hierbei noch, den betriebssystem-abhängigen Verzeichnis-Trenner (unter Linux/Unix „/“, unter Windows „\“) durch einen Punkt zu ersetzen. So könnten die ersten Versuche auf einfache Art anfangen, eine Klasse zu laden:

Dateiname: versuchter Klassenname:

```
Test.class           →      Test
junittca/Test.class  →      junittca.Test   usw.
```

Dies ist zeitaufwändig, gerade wenn die Paketnamen umfangreicher werden, außerdem werden eine Menge an Fehlern produziert, da alle nicht korrekt gefundenen Klassen eine Ausnahme werfen.

Der bessere und auch umgesetzte Weg ist, den vom Benutzer zu setzenden Klassenpfad zu nehmen und ihn vom Pfad zur Datei vorne abzuschneiden:

```
~~/Pfad/zu/Projekt/bin/~~de/berndsteindorff/junittca/Test.class
```

Als nächstes werden die Verzeichnis-Trenner in Punkte gewandelt:

```
de.berndsteindorff.junittca.Test.class
```

Zuletzt muss noch die Dateiendung entfernt werden:

```
de.berndsteindorff.junittca.Test
```

So kann die Klasse schließlich erfolgreich vom Klassenlader geladen werden. Alle abhängigen Klassen (über Import-Anweisungen) müssen ebenfalls im Klassenpfad vorhanden sein.

Bei der zweiten Funktion gilt das Prinzip um Klassen zu laden ebenfalls. Zusätzlich dazu kann es Probleme geben, die Klassen zu finden. Wenn sich eine Klasse nicht im Klassenpfad befindet, kann der Klassenlader diese nicht finden und löst eine entsprechende Ausnahme aus.

Beim Neuladen der bereits verwalteten Klassen tritt das Problem mit dem Klassenpfad in den Hintergrund, da die Klassen bereits identifiziert und geladen sind. Hierfür muss auch eine Funktion erstellt werden, da der Standard-Klassenlader nicht neu einliest.

Absuchen des Klassenpfades

Den Klassenpfad nach Test-Klassen abzusuchen, ist zeitaufwändig, da sich Test-Klassen nur daran erkennen lassen, dass sie Methoden enthalten, die mit der Annotation `@Test` versehen sind. Um den Ladevorgang zu beschleunigen, werden nur Klassen-Dateien mit den beiden Mustern

```
Test.*\\.class$
.*Test\\.class$
```

geladen und nach Test-Methoden durchsucht. Alle erfolgreich gefundenen Test-Klassen mit Test-Methoden werden dem Projekt hinzugefügt, sofern sie im Projekt noch nicht vorhanden sind.

Der eigene Klassenlader `FileClassLoader` ermittelt zu den im Pfad gefundenen Klassen den Klassennamen (durch die oben beschriebene Ersetzung). Danach werden die Klassen geladen und nach Methoden mit der Annotation `@Test` durchsucht. Erst wenn das zutrifft, wird die Klasse der Testfallverwaltung hinzugefügt. Zu der Klasse wird noch der Pfad gespeichert, um auch bei verändertem Pfad des Projektes die bereits hinzugefügten Klassen erfolgreich neu laden zu können.

Zu beachten ist, dass der Klassenpfad sinnvoll sein sollte. Es bringt Nachteile bei der Geschwindigkeit, wenn die Klassen im Wurzel-Verzeichnis bzw. deren Paketverzeichnis unter der Wurzel untergebracht sind. Die Suchmethode kann schließlich nicht wissen, in welchem Unterverzeichnis sich tatsächlich Klassen befinden. Die Suche geht rekursiv alle Unterverzeichnisse durch und sucht nach Dateien, die einem der beiden Muster entsprechen. Gibt der Benutzer einen Pfad an, in dem sich zwar Klassen befinden, die Paketstruktur sich aber erst in Unterverzeichnissen des angegebenen Pfades befindet, so wird eine Ausnahme geworfen.

Eine dynamische Veränderung des Suchmusters zur Laufzeit ist derzeit nicht geplant, könnte aber genau wie eine Klassenpfad-Variable benutzt werden.

Dynamisches Hinzufügen einer beliebigen Test-Klasse

Das dynamische Laden per Dialogfenster ist die Variante, die für den Benutzer transparenter ist. Er hat eine oder mehrere Dateien ausgewählt. Das erspart die Suche und ist daher schneller als einen kompletten Pfad abzusuchen.

Die Grundproblematik ist bereits oben beschrieben. Eine Datei muss nicht einer geladenen Klasse entsprechen und diese wiederum muss auch keine Tests enthalten. Auch hier übernimmt die Klasse `FileClassLoader` die Aufgabe, die ausgewählten Dateien als Klassen zu laden und auf Tests zu prüfen. Der Pfad, in dem die Klasse gefunden wird, wird ebenso gespeichert. Befindet sich die Klasse nicht im Klassenpfad, so wird eine Ausnahme geworfen.

Automatisches Neuladen der Test-Klassen vor einer Test-Ausführung

Die dritte Funktion von `FileClassLoader` ist es, die verwalteten Klassen der Testfallverwaltung neu zu laden, wobei die Klassen vom gespeicherten Pfad neu geladen werden. Sind sie dort nicht mehr zu

finden, gibt es eine Ausnahme. Der Benutzer muss sich dann darum bemühen, die Test-Klasse aus der Testfallverwaltung zu entfernen und sie gegebenenfalls neu zu laden.

8.5 Persistenz der Test-Ergebnisse

Mit dem objektorientierten Datenbanksystem db4o (der Firma db4objects) kann man die Projekt-Daten leicht abspeichern.

Das objektorientierte Datenmodell kann in der vorliegenden Form benutzt werden. Die Objekte werden komplett mit ihren Beziehungen zueinander in der Datenbank gespeichert.

Die Klasse `ProjectFileHandler` ist für die Persistenz verantwortlich. Sie kümmert sich um den Umgang mit dem einzelnen Projekt. Sie unterstützt den Standard-Satz an Datenbankoperationen nach dem CRUD-Prinzip. Damit ist gemeint, die Klasse erstellt Projekte (**c**reate), liest Projekte (**r**ead), aktualisiert Projekte (**u**pdate) und löscht (**d**elete) Projekte in der Datenbank.

Modus von db4o

Die Datenbank wird direkt in der gleichen Java virtuellen Maschine gestartet und nur zur Einzelplatzbenutzung eingerichtet. Die Datenbank stellt im Sinne von db4o eine einzelne Datei dar. Der Benutzer der Testfallverwaltung muss dazu eine vorhandene Projekt-Datei auswählen oder eine neue Projekt-Datei anlegen. Dem Benutzer wird dazu ein Dialogfenster angezeigt, in dem er Dateien auswählen oder anlegen kann. Die übliche Endung von db4o *.yap* wird ihm vorgegeben, ist aber keine Pflicht. Die gewählte Datei kann dann beliebig viele Projekte enthalten. Es ist denkbar, dass Projekte in verschiedenen Dateien abgelegt werden, um sie einfach zwischen verschiedenen Arbeitsplätzen austauschen zu können.

Bei größeren Projekten könnte es natürlich sein, dass es nicht ausreichend ist, die Datenbank-Datei auszutauschen. Hier wird ein Datenbank-Server benötigt. Auf den Einbau einer solchen Lösung wird hier verzichtet, da die Projekt-Daten den Einsatz eines „großen“ Datenbank-Servers kaum rechtfertigen. db4o eignet sich hier ideal als kleine, objektorientierte Lösung.

Problem der Objektidentität

Db4o hat auch Nachteile, die sich aus der Objektorientiertheit ergeben. Die Objektidentität stellt dabei ein Problem dar (im Rahmen von relationalen Datenbanken wird dieses Problem komplett anders gelöst).

Die Objektidentität ist in Java normalerweise etwas, was man nicht sieht. Sie wird aber von der virtuellen Maschine im Hintergrund für jedes Objekt mitgeführt. Die virtuelle Maschine vergibt jedem Objekt eine eindeutige Identifizierung, wenn sie das Objekt erstellt. Die Identifizierung entspricht oftmals der Position im Speicher, auch Objektreferenz genannt. Die Identität von Objekten lässt sich mit dem Operator „==“ prüfen (im Gegensatz zu `equals`, das die Objekte nur auf gleiche Zustände prüft).

Db4o nutzt einfach die Objektidentität der Java-Welt. Alle in einer db4o-Sitzung benutzten Identitäten sind bekannt. Es lässt sich innerhalb einer Sitzung ein Objekt erstellen, in der Datenbank speichern, das Objekt mehrmals abändern und wieder speichern. Für db4o stellt dies keine Probleme dar, selbst dann nicht, wenn man die Objekte mehrmals speichert.

Sobald jedoch die Sitzung zu db4o geschlossen und eine neue gestartet wird, kann db4o mit den noch im Speicher befindlichen Objekten der virtuellen Maschine nichts mehr anfangen. Ein kleines Szenario hierzu:

Schritt	**Anzahl der Objekte**		**Bemerkung**
	virt. Maschine	**db4o**	
neues Projekt *project1* wird erstellt	1	0	
neue db4o-Sitzung, *project1* darin wird gespeichert	1	1	
project1 wird geändert und erneut gespeichert	1	1	
db4o-Sitzung wird geschlossen	1	1	
project1 wird wieder geändert und in einer neuen db4o-Sitzung (mit obiger Datei) gespeichert	1	2	nicht gewünscht, dass db4o das *project1* als neu speichert

Eine Speicherung würde zu einem neuen Objekt in der Datenbank führen, was oftmals nicht gewünscht ist. Die Empfehlung in `[vi07]` ist, die Sitzung nicht zu schließen, damit die Objektidentitäten zwischen db4o und der virtuellen Maschine immer auf dem gleichen Stand sind.

Anfangs sorgt das Singleton-Muster dafür, dass die Sitzung von db4o immer offen gehalten wird. Das Singleton-Muster wird in der Klasse `ProjectFileHandler` dafür eingesetzt, dass die Datenbank immer geöffnet bleibt. Außerdem wird die Projekt-Datei so höchstens einmal geöffnet. Ansonsten könnte es zu einer Ausnahme kommen, da db4o alleinigen Dateizugriff benötigt.

Nachdem jedoch die Oberfläche auf das Muster Model-View-Controller (MVC) umgestellt wird, ist das Singleton-Muster in `ProjectFileHandler` nicht mehr notwendig. Ein Objekt von `ProjectFileHandler` wird nach dem MVC-Muster nur noch ein Mal beim Start der Testfallverwaltung erzeugt. Dafür wird die Struktur des `ProjectFileHandler` stark überarbeitet, so dass dieser maximal eine Datei öffnet und diese die ganze Sitzung über offen hält oder auf Anforderung des Benutzers eine andere Datei öffnet.

Aktivierungstiefe / Aktualisierungstiefe

Db4o wurde ursprünglich für mobile Geräte entwickelt. Die Entwickler haben festgelegt, dass beim Umgang mit einem Objekt nicht die volle Tiefe des Objektgraphen behandelt wird.

Unter einem Objektgraphen wird verstanden, dass eine Klasse Variablen besitzen kann, die ebenfalls Klassen darstellen. Ist eine der Variablen einer Klasse selber eine Klasse, dann erhöht sich die Tiefe des Objektgraphen um eins, ausgehend von der Wurzel-Klasse. Ein Beispiel zum Objektgraphen anhand der Klasse `Project`:

Tiefe 1		Tiefe 2		Tiefe 3	
`Project`	→	`List<Run>`	→	`Run`	...

In db4o können in zwei verschiedenen Bereichen die Graphentiefen beeinflusst werden. Dies betrifft die Aktivierung und die Aktualisierung. Beim erstmaligen Speichern eines Objektes ist die Tiefe unwichtig, da der komplette Graph gespeichert wird.

Bei der Aktivierung werden Objekte aus der Datenbank in den Hauptspeicher geladen. Es wird bis zu einer vorher festgelegten Tiefe geladen. Dies soll verhindern, dass besonders tiefe Objektgraphen, wie etwa bei einer verketteten Liste, den Hauptspeicher zu stark belegen.

Bei der Aktualisierung ist ähnlich der Aktivierung eine Graphentiefe festgelegt, bis zu der db4o die Objekte in der Datenbank aktualisiert.

Beide Tiefen-Werte sind global (auf alle Klassen der gleiche Wert) und lokal (auf eine spezielle Klasse) festlegbar. Der globale Standardwert für

die Aktivierung liegt bei Tiefe 5 und der der Aktualisierung bei Tiefe 1. Db4o rechnet ab Tiefe 0, welche da behandelte Objekt selbst betrifft. Die referenzierten Unterelemente und primitiven Variablen befinden sich in Tiefe 1.

Aufgrund des Daten-Modells ergibt sich eine größere Objekttiefe, als sie standardmäßig gesetzt ist. So muss ein neuer Wert ermittelt und gesetzt werden. Da die Testfallverwaltung eine gering zu erwartende Datenmenge hat, ist es nicht wichtig, sparsam in Bezug auf den Hauptspeicher zu sein. Daher sollte die komplette Graphentiefe geladen werden.

Die Graphentiefe wird mit manueller Durchsicht des Quellcodes (der benutzten Klassen) und mit Hilfe des Eclipse-Debuggers ermittelt.

Bis auf zwei Ausnahmefälle wird die Graphentiefe 12 errechnet. Die erste Ausnahme bezieht sich auf die Klassen `Class` und `Method` (beide aus dem Paket `java.lang`), hier besteht das Problem der Nachvollziehbarkeit, z. B. hat `Class` 13 verschiedene referenzierte Datentypen (Klassen und primitive Datentypen) als Unterelemente. Diese Unterelemente verzweigen wiederum stark. Deshalb wurden diese beiden Klassen nur bedingt in die Berechnung mit einbezogen.

Die zweite Ausnahme liegt in vier verwendeten Klassen, die auf sich selbst zeigen (`Class`, `Method`, `Description` und `Throwable`). Die unübersichtlichen Klassen `Class` und `Method` sowie die vier rekursiv arbeitenden Klassen führen dazu, dass die genaue Objektgraphentiefe nicht ermittelt werden kann. Aus diesem Grunde wird die Tiefe des Graphen, den db4o aktualisieren / aktivieren soll, erstmal bei Tiefe 22 angesetzt. Dies ist 10 Tiefenzähler mehr als errechnet.

Nach Überarbeitung des Daten-Modells fallen die `Class` und `Method` weg, so dass es leichter ist, eine Graphentiefe zu ermitteln. Mit `Description` und `Throwable` enthält das `Result` von JUnit jedoch noch zwei rekursiv arbeitende Klassen. Die errechnete Graphentiefe bleibt bei Tiefe 12.

Die Werte werden global gesetzt, denn über die Klasse `ProjectFileHandler` ist es nur möglich, mit der Klasse `Project` Datenbankoperationen vorzunehmen. Da diese Klasse die Wurzel-Klasse der Daten darstellt, ist keine lokale Konfiguration an Unter-Klassen notwendig und sinnvoll.

In Anhang D befinden sich zwei Diagramme, anhand derer die Tiefe des Objektgraphen vor und nach der Überarbeitung ermittelt wird.

Operationen

Die Operationen, die auf der Projekt-Datei möglich sind, umfassen zum einen den Standard-Satz an Datenbankoperationen und zum anderen eine generelle Datenbankabfragen:

- Alle Projektnamen lesen
- Ein Projekt speichern
- Ein Projektes lesen
- Ein Projekt aktualisieren (ist in db4o das Gleiche wie die Speicherung)
- Ein Projekt löschen

Für den Einsatz in der Testfallverwaltung ist db4o ausreichend. Ändert und erweitert sich jedoch das Anwendungsszenario, so sollte auch die Datenbanklösung entsprechend umgebaut werden. Das heißt aber nicht, das db4o ersetzt werden soll, denn es gibt auch einen Client-Server-Modus, der die Einzelplatzlösung ersetzen kann.

8.6 Oberfläche der Testfallverwaltung

Die Benutzeroberfläche als graphische Darstellung der Testfall-verwaltung muss die im Lösungsansatz aufgeführten Benutzeraktionen anbieten. Nachfolgend werden die einzelnen Bereiche der Oberfläche beschrieben.

Allgemeines

Die Oberfläche für die Testfallverwaltung wird ausschließlich in Swing, unter Benutzung einiger AWT-Klassen umgesetzt. Die Entwicklung geschieht manuell, d. h. es wird kein graphischer Editor eingesetzt.

Es gibt eine voreingestellte Fenstergröße von 1024 x 768 Pixeln. Der Autor geht davon aus, dass die meisten Benutzer, also Programmierer, mindestens eine solche Bildschirmauflösung eingestellt haben. Eine größere Auflösung bringt der Testfallverwaltung nichts, da die anzuzeigenden Elemente darin ausreichend Platz finden. Eine kleinere Auflösung führt dazu, dass zwei Tabellen nicht mehr korrekt dargestellt werden, so dass die Fenstergröße von 1024 x 768 Pixeln als ideal vom Autor angesehen wird. Der Benutzer kann die Auflösung trotzdem verändern, muss dabei mit möglichen Schwierigkeiten bei der Anzeige rechnen.

Umsetzung mit Mustern

Bei der Umsetzung müssen die Oberfläche und die Anwendungslogik miteinander verbunden werden. Die Oberfläche enthält sowohl die

Komponenten der Anzeige, als auch die Steuerungskomponenten. Die dritte Komponente ist die Anwendungslogik, sie besteht aus dem Datenmodell (siehe Kapitel 7.3), dem Klassenlader (`FileClassLoader`, siehe Kapitel 7.4) und einer Klasse, die mit der Projekt-Datei und dem Projekt umgeht (`ProjectFileHandler`, siehe Kapitel 7.5).

Die drei Komponenten werden normalerweise getrennt voneinander programmiert und lose miteinander verkoppelt. Am Anfang wird diese Strategie nur mit dem Beobachter-Muster umgesetzt. Als Hilfe wird eine eigene Ereignisstruktur mit Quelle- und Empfänger-Schnittstelle, Unterstützungs-Klasse und eigener Ereignis-Klasse erstellt.

Mit wachsender Anzahl von Oberflächen-Klassen und umgesetzten Benutzeraktionen wird es umständlicher und unübersichtlicher, die Klassen zu verbinden. Daraus wird eine große Überarbeitung, in der das Model-View-Controller-Muster umgesetzt wird. Die Überarbeitung ist nicht einfach. Das Ergebnis ist jedoch, dass die Struktur sauberer und verständlicher wird. Die Quellcode-Pakete sind entsprechend der Aufteilung nach dem MVC-Muster gestaltet.

Alle drei Komponenten werden erläutert und graphisch dargestellt:

1. Komponente: Steuerung

Die Steuerung (siehe Abbildung 3, engl. controller) fällt mit der Schnittstelle `IProjectController` und einer Implementierung klein aus. Die Schnittstelle definiert die Steuerungsmöglichkeiten. `ProjectController` übernimmt die komplette Steuerung der Oberfläche der Testfallverwaltung. Es wird nur eine Steuerung erstellt, da einige Aktionen über mehrere Wege gestartet werden können. Die Implementierung sieht dafür gleich aus, daher genügt eine steuernde Klasse.

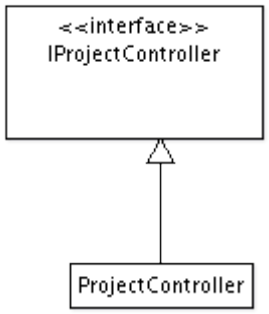


Abbildung 3: Steuerungs-Diagramm

2. Komponente: Anzeige

Die Anzeige (siehe Abbildung 4, engl. view) besteht aus seinen einzelnen Komponenten, die über eine Komposition verbunden sind (bei der Anzeige ist das Kompositum-Muster umgesetzt). Drei dieser Klassen sind Empfänger von Ereignissen aus dem Modell, sie implementieren die Schnittstelle `TcaEventListener`. Damit wird das Beobachter-Muster umgesetzt. Alle View-Klassen, die Ereignisse selber auslösen, delegieren diese über das Strategie-Muster an die Steuerung (`IprojectController`). Die zwei Chooser-Klassen und die Dialog-Klasse am rechten Rand werden als separate Fenster angezeigt und hängen daher nicht mit `TcaFrame` zusammen.

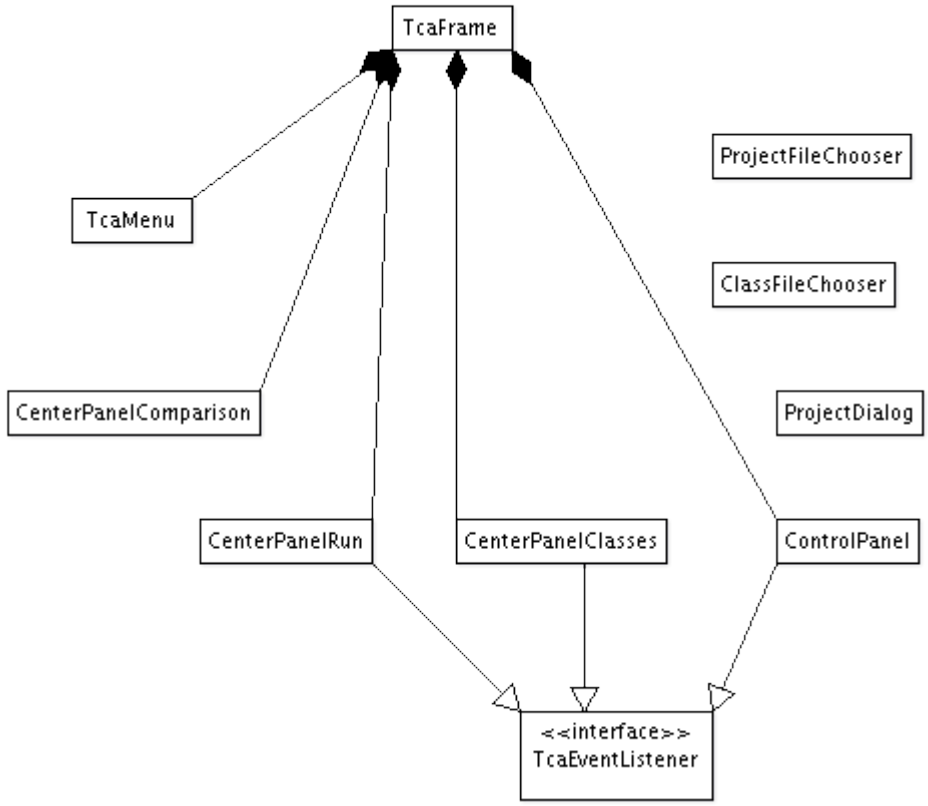


Abbildung 4: Anzeigen-Diagramm

3. Komponente: Modell

Das Modell (siehe Abbildung 5, engl. model) beherbergt alle Klassen des Daten-Modells. Dazu kommt noch `ProjectFileHandler` als Klasse, die mit der Projekt-Datei und dem Projekt umgeht. `ProjectFileHandler` implementiert `IProjectModel` und `TcaEventSource`. In `IProjectModel` werden die Methoden angeboten, um Daten vom Modell abzurufen und das Modell zu beeinflussen. Über `TcaEventSource` wird der Anzeige angeboten, sich als Beobachter zu registrieren. Bei entsprechenden Ereignissen im Modell werden sie informiert. `Project` und `ProjectFileHandler` haben Zugriff auf den eigenen Klassenlader (`FileClassLoader`).

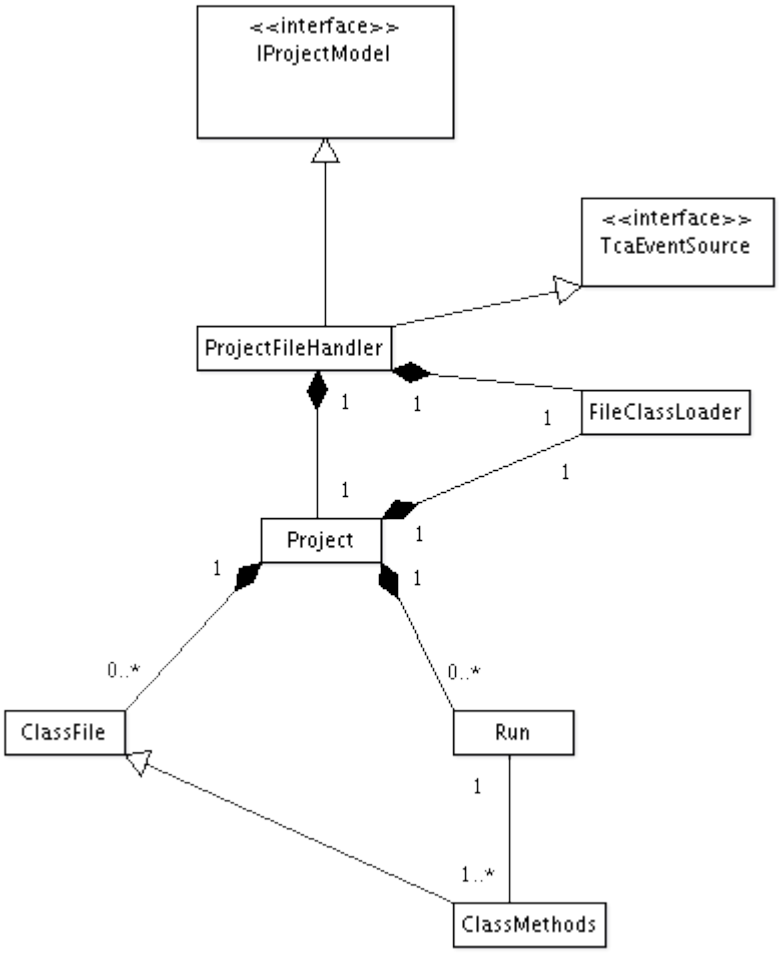


Abbildung 5: Modell-Diagramm

Erscheinungsbild der Oberfläche (look & feel)

Das Erscheinungsbild (engl. look & feel) ist gerade für Windows-Benutzer eine Frage von großer Bedeutung. Java möchte zwar vom Grundsatz her plattformunabhängig sein, stellt jedoch über die `Swing`-Bibliotheken verschiedene Erscheinungsbilder für unterschiedliche Betriebssysteme bereit. Diese lassen sich zur Laufzeit (am Besten noch vor Erstellung der Oberflächen-Objekte) setzen.

Standardmässig setzt Java das Metal-Erscheinungsbild ein. Metal ist zwar einheitlich in allen Systemen mit der Java-Laufzeitbibliothek von Sun verfügbar, wird jedoch nicht von allen Benutzern akzeptiert. Um es allen Benutzern gerecht zu machen, wird per Standard das Erscheinungsbild des Systems gesetzt, dadurch wird unter Windows das einheitliche Erscheinungsbild gewahrt.

Menü

Ein Menü erscheint am oberen Rand des Hauptfensters und ermöglicht es, mit Projekt-Dateien und Projekten Aktionen durchzuführen.

Fenster-Flächen

Das Hauptfenster wird in zwei Flächen (`JPanel`) unterteilt. Die linke Fläche enthält Elemente zur Steuerung, sie bleibt die komplette Laufzeit über sichtbar an dieser Stelle. Rechts daneben gibt es die zentrale Fläche. Die zentrale Fläche zeigt verschiedene Ansichten (die verwalteten Test-Klassen, ein Test-Ergebnis oder ein Vergleich zweier Test-Ergebnisse) an. Es wird immer nur eine Ansicht in der zentralen Fläche angezeigt.

Die Flächen werden mit dem Layout-Manager `GridBagLayout` gestaltet, um größere Gestaltungsmöglichkeiten gegenüber einfacheren Layouts (wie `BorderLayout`) zu haben. Hierzu wird die fremde Hilfsklasse (`GBC`, unter der GPLv2 stehend) genutzt, um das `GridBagLayout` einfacher nutzbar zu machen.

Die Benutzer-Aktionen, die nicht durch das Menü abgedeckt werden, sind in den Fenstern eingebaut. Zusammen mit den Menü-Aktionen sind damit alle Aktionen möglich, die im Ansatz gefordert werden.

Schließen der Oberfläche

Beim Schließen der Oberfläche über das Menü oder die Schließen-Schaltfläche wird ein Dialog angezeigt. Dieser fragt den Benutzer, ob er das aktuelle Projekt vor dem Beenden speichern oder nicht speichern will. Alternativ wird nicht beendet. Beendet der Benutzer, so wird die Datenbank-Verbindung geschlossen, bevor die Anwendung schließt.

8.7 Ausnahmen

Es werden nur geprüfte Ausnahmen abgefangen.

Die Ausnahmen entstehen alle im Modell-Paket. Von dort werden sie an die aufrufende Steuerung weitergereicht. Die Steuerung erstellt daraus aussagekräftige Nachrichten und lässt sie anzeigen.

8.8 Probleme bei der Plattformunabhängigkeit

Im Ansatz werden drei Problembereiche aufgeführt: Dateioperationen, Klassenpfad und Oberfläche. Die Problembereiche sind zu unterscheiden nach Programm- und Testcode.

Im Programmcode sind nur der Klassenpfad und die Oberfläche betroffen. Der Klassenpfad wird im Klassenlader gebraucht, es wird der plattformunabhängige Pfadtrenner von `System.getProperty („path.seperator“)` genutzt.

Die Oberfläche ist zwar angepasst, dadurch dass der System-Standard für das Erscheinungsbild benutzt wird. Die Tabellen bereiten aber dabei Probleme, wenn ihnen nicht genug Raum auf der Anzeigenfläche gegeben wird. Vergleiche ergeben, dass Windows weniger Probleme macht, Tabellen zu skalieren als Linux. Unter Linux mit Metal-Erscheinungsbild verschwinden Tabellen, wenn die Oberfläche eine bestimmte Fenster-Auflösung unterschreitet. Daher ist auch die Start-Auflösung vorgegeben.

Im Testcode wird neben dem Klassenpfad mit hart kodierten Dateinamen gearbeitet. Der Verzeichnistrenner ist bei Dateioperationen auf den Systemen unterschiedlich, so dass mit `File.seperator` als Trenner gearbeitet wird.

8.9 Auslieferung der Testfallverwaltung

Das System soll als Jar-Archiv ausgeliefert werden, welches bei Ausführung per Klick die Hauptklasse startet (dies setzt voraus, dass der Java-Ausführer mit dem Datei-Format .jar im Betriebssystem verknüpft ist).

Das Werkzeug Ant stellt einen einfachen Weg dar, das Jar-Archiv zu erstellen und jederzeit den Vorgang automatisiert zu wiederholen. Ant bietet an, eine Manifest-Datei zu erstellen. Darin wird der Verweis auf die ausführbare Hauptklasse in dem Attribut `Main-Class` eingetragen.

Es gibt dabei ein Problem mit der Benutzung der Rahmenprogramme von JUnit, db4o und log4j als externe Datei. Der Aufruf des Jar-Archivs mit `java -jar` lässt kein Setzen des Klassenpfades über die Option `-cp` oder die Auswertung der Umgebungsvariable `CLASSPATH` zu. Es gibt zwei Möglichkeiten das Problem zu lösen, damit der Klassenlader die benötigten Klassen der Rahmenprogramme findet. Entweder es werden die kompletten Klassen-Verzeichnisse mit in die eigene Jar-Datei gepackt oder sie werden als `Class-Path`-Attribut in der Manifest-Datei gesetzt.

Der Autor hält es für eine bessere Lösung, die Jar-Dateien der benutzten Rahmenprogramme vom eigenen Software-Paket zu trennen. Das System wird eine Jar-Datei mit dem eigenen Software-Paket enthalten, sowie die Jar-Dateien der benutzten Programmen in einem zusätzlichen Unterverzeichnis mit dem Namen *lib*. Die Manifest-Datei der eigenen Jar-Datei enthält dann neben dem Verweis auf die Hauptklasse einen speziellen Klassenpfad zu den drei benutzten Rahmenprogrammen.

Zur fertigen Auslieferung wird die Jar-Datei (junittca.jar) mit dem *lib*-Verzeichnis und den Lizenztexten in eine gepackte Zip-Datei gepackt. Auch hier erleichtert Ant es, eine Zip-Datei automatisiert zu erstellen, nachdem die Jar-Datei erstellt wird.

Diese Datei muss der Benutzer nur entpacken und junittca.jar ausführen.

Die Zip-Datei wird folgenden Inhalt haben:

- junittca.jar (die eigene Jar-Datei)
- Verzeichnis lib mit folgendem Inhalt:
  - db4o-6.1-java5.jar
  - junit-4.3.1.jar
  - log4j-1.2.14.jar
  - Lizenztexte der drei Rahmenprogramme (CPL 1.0, GPLv2, Apache License 2.0)
- README (enthält Informationen zum Inhalt der Zip-Datei)

9. Angaben zur Programmierung

Dokumentation

Die Dokumentation der Klassen erfolgt nach Java-Standard. Allerdings wird Wert darauf gelegt, dass die benutzten Bezeichner aussagekräftig sind, um auch ohne die Dokumentation eine Aussage über deren Zweck erkennen zu können. Als Sprache wird Englisch benutzt, das erleichtert es, den Quellcode zu veröffentlichen. Mehr in Anhang D Programmierstandards.

Konventionen

Es gelten bei den Konventionen auch die Programmierstandards aus Anhang D.

Zusätzlich gilt es so weit wie möglich, lokale Variablen und Objektvariablen `final` zu setzen.

Lizenzinformationen

Die kompletten Lizenztexte der benutzten Rahmenprogramme befinden sich im lib-Verzeichnis des Projektes.

Entwurfsmuster

Eine genaue Beschreibung des Einsatzes von Entwurfsmustern erfolgt beim jeweiligen Unterpunkt der Lösungsausführung.

Hier ist eine Übersicht der eingesetzten Muster mit ihrem Einsatzort.

- Singleton-Muster: Beim Umgang mit db4o wird bis zum Einsatz des Model-View-Controller-Muster das Singleton-Muster eingesetzt.

- Beobacher-Muster: Bei der Oberfläche wird es zuerst alleinstehend eingesetzt, um Ansicht- und Steuerungskomponente von den Kern-Programmteilen zu trennen und lose miteinander zu verknüpfen.

- Model-View-Controller-Muster (MVC): Es wird bei der Oberfläche nach einer großen Überarbeitung eingesetzt. Das MVC-Muster ist ein zusammengesetztes Muster und benutzt als wichtigste Muster Strategie, Beobachter und Kompositum. Es trennt die Programmteile Anzeige, Steuerung und Modell voneinander.

Protokollierung

Beim Programmieren hat sich gezeigt, dass Textmeldungen auf der Kommandozeile einfach auszugeben nicht die beste Art ist, den Fortschritt (während der Programmausführung) zu protokollieren. Einfache Textmeldungen sind zu ungenau in ihren Aussagen und nicht genug steuerbar.

Die Protokollierung wird deshalb geändert, um die ungesteuerten Konsolenausgaben zu vermeiden. Sie wird nun mit dem Rahmenprogramm log4j erledigt. Das hat den Vorteil, dass mehr Konfigurationsmöglichkeiten vorhanden sind, um gezielt Ausgaben zu produzieren oder zu unterdrücken. Die Ausgaben sind zudem noch aussagekräftiger.

Quellcodezeilen des Systems

Eclipse bietet es nicht an, die Zeilen an Quellcode zu zählen. Dafür wurde die Erweiterung Metrics installiert, welches die Zeilen zählt und diverse andere Statistiken erstellt.

Die Zeilenanzahl (netto, gerundet auf volle 100) beträgt zur Abgabe 3.300 in allen Paketen, davon 1.000 im Test-Paket, 700 im Modell-Paket, 1.300 im Anzeigen-Paket und 300 im Steuerungs-Paket. Daran lässt sich ablesen, dass neben der Oberfläche (Anzeige und Steuerung), das Test-Paket mit 30% einen großen Anteil am Quellcode einnimmt.

10. Bewertung

Die einzige gefundene Alternative zur Testfallverwaltung ist derzeit Eclipse, um JUnit 4-Tests graphisch auszuführen. Die Testfallverwaltung stellt im Vergleich zu Eclipse mehr Funktionalität bereit:

- Test-Klassen lassen sich als Sammlung ausführen.

- Die Historisierung der Ergebnisse wird über die Programmlaufzeit hinaus erledigt. Dadurch kann man alte Ausführungsergebnisse anzeigen und miteinander vergleichen lassen.

Ein Benutzer ist mit der Testfallverwaltung unabhängiger von Entwicklungsumgebungen wie Eclipse. Die Testfallverwaltung stellt einen eigenen Klassenlader bereit, der es ermöglicht, Klassen aus einem beliebigen definierten Klassenpfad hinzuzufügen. Die Klassen werden vor jeder Test-Ausführung erneut eingelesen, um Veränderungen während der Laufzeit zu erfassen.

Der Programmcode der Testfallverwaltung wurde mehrfach überarbeitet, um die Struktur zu verbessern. Selbst geschriebene Modultests sollen die Funktionalität absichern.

11. Rückblick auf den Verlauf der Arbeit

Die Funktionalität der Testfallverwaltung bietet derzeit kein anderes Programm. Selbst die großen Entwicklungsumgebungen unterstützen JUnit 4 entweder gar nicht oder nur teilweise.

Die Testfallverwaltung als separate Oberfläche mag seine Vorteile in der Funktionalität haben. Dabei ist es fraglich, ob Programmierer eine separate Oberfläche starten, wenn sie hauptsächlich eine der großen Entwicklungsumgebungen benutzen.

So schön die Welt des Testens und der testgetriebenen Entwicklung ist, hat sich das Testen doch noch nicht überall zur Steigerung der Code-Qualität durchgesetzt. Dies hat sicherlich viele Ursachen. Bei JUnit 4 könnte einer der Gründe in mangelnden Werkzeugen liegen, die es seit seiner Veröffentlichung im Februar 2006 unterstützen. Mit der Testfall-verwaltung würde ein weiteres Werkzeug zum Testen bereitstehen, das möglicherweise eine vorhandene Hemmschwelle für vermehrtes Testen senkt.

Die Veröffentlichung der Testfallverwaltung kann leider noch nicht geschehen, da die benutzten Rahmenprogramme db4o und JUnit unter inkompatiblen Lizenzen stehen. Da es sich bei beiden Lizenzen (GPL und CPL) jeweils um freie Lizenzen handelt, ist diese Inkompatibilität nicht akzeptabel. So gut der Funktionsumfang und die Leistung von db4o auch sein mag, gerade die Lizenz von db4o (GPL) macht die Testfallverwaltung nicht veröffentlichbar. Eine Lösung könnte darin bestehen, db4o auszutauschen.

Die Testfallverwaltung ist noch erweiterbar. Sowohl mehr Funktionalität, als auch mehr Ansichten sind vorstellbar.

Der Autor hat durch die gelesene Literatur festgestellt, das Extreme Programming eine ausgewachsene Vorgehensweise zur Software-Entwicklung ist. Besonders die aus XP stammende Programmiertechnik der testgetriebenen Entwicklung hat dem Autor Spaß bereitet. Er hat bei der Entwicklung erfahren, dass Techniken, wie ein einfaches Design, einzuhalten, nicht immer auch einfach anzuwenden sind. Der Autor sieht darin einen stetigen Lernprozess, der auch in [xp02] beschrieben wird.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, 01.08.2007

.....

(Bernd Steindorff)

Danksagungen

In Bezug auf die Diplomarbeit bedanke ich mich bei meinem betreuenden Prof. Dr. Grude. Er stand mir mit Rat und Tat zur Seite. Das „Projekt“ Diplomarbeit schien anfangs in weiter Ferne, er hat mich stets motiviert, eine erfolgreiche Abgabe zu erreichen.

Ich bedanke mich auch bei meinen Eltern und meiner Großmutter, ohne deren Unterstützung ich das Studium nicht geschafft hätte.

Den Korrekturlesern gebührt auch Dank, innerhalb kurzer Zeit die Diplomarbeit durchzuarbeiten.

Dann bedanke ich mich bei meinen noch verbleibenden Freunden, die trotz meines Zeitmangels noch zu mir gehalten haben.

Zum Gesamterfolg tragen auch einige Kommilitonen bei. Hier hebe ich Sascha Kupper hervor, er hat mich während des Studiums über begleitet und in die Programmierung eingeführt. Außerdem noch Sebastian Sauer, der mich in die Richtung freier Software erfolgreich gewiesen hat.

Anhang A Benutzte Software

Es wurde folgende Software verwendet:

Name	Beschreibung	Adresse
Kubuntu 6.10	Betriebssystem mit Linux-Kernel	http://www.kubuntu.de
OpenOffice 2.0.4	Office-Programm mit Textverarbeitung	http://de.openoffice.org
Eclipse 3.3	Entwicklungsumgebung	http://www.eclipse.org
JUnit 4.3.1	Rahmenprogramm für Modulstests	http://sourceforge.net/projects/junit
db4o 6.1	Objektorientierte Datenbank	http://www.db4o.de
subclipse	Plugin für Eclipse, Austausch von Dateien mit Subversion als System zur Versionsverwaltung	http://subclipse.tigris.org
Ant 1.7	Werkzeug zur Steuerung des Build-Vorgangs in Java	http://ant.apache.org
yEd Graph Editor 2.4.2.2	Programm zur Erstellung von Graphen	http://www.yworks.com
ArgoUML 0.24	Erstellung von UML-Diagrammen	http://argouml.tigris.org
Metrics	Plugin für Eclipse, erstellt Statistiken über den Quellcode	http://sourceforge.net/projects/metrics
Krita	Zeichen- und Bildbearbeitungsprogramm in KDE	http://www.kde.org

Anhang B Literatur

Neben den unten aufgeführten Sachbüchern und Zeitschriftenartikeln wurde die Dokumentation der verwendeten Software benutzt.

Sachbücher

we05	Westpfahl, Frank (2006): Testgetriebene Entwicklung mit JUnit & FIT. Wie Software änderbar bleibt.
be03	Beck, Kent (2003): Extreme Programming
ul07	Ullenboom, Christian (2007): Java ist auch eine Insel, 6. Auflage
vi07	Visengeriyeva, Larysa (2007): db4o, schnell + kompakt
if07	Institut für Rechtsfragen der Freien und Open Source Software (2005): Die GPL kommentiert und erklärt
hc05	Horstmann, Cay S. und Cornell, Gary (2005): Core Java 2 Band 1 + Band 2
ba05	Balzert, Heide (2005): Lehrbuch der Objektmodellierung
ed03	Edlich, Stefan (2003): Ant kurz & gut
fr06	Freeman, Eric und Freeman, Elisabeth (2006): Entwurfsmuster von Kopf bis Fuß
xp02	Lippert, Martin; Roock, Stefan und Wolf, Henning (2002): XP – Software entwickeln mit Extreme Programming

Zeitschriftenartikel

Beck, Kent (2006):	Interview, iX, Seite 66 ff., Heise Zeitschriftenverlag
Fabian Schmieder (2006):	Urheberrechtsfibel für Programmierer, c't, Seite 174 ff., Heise Zeitschriftenverlag

Anhang C Dokumentation der Oberfläche

Diese Dokumentation beschreibt die Oberfläche der Testfallverwaltung (Version 02.08.2007). Es werden alle Bereiche der Oberflächen bildhaft dargestellt und dazu schriftlich erläutert.

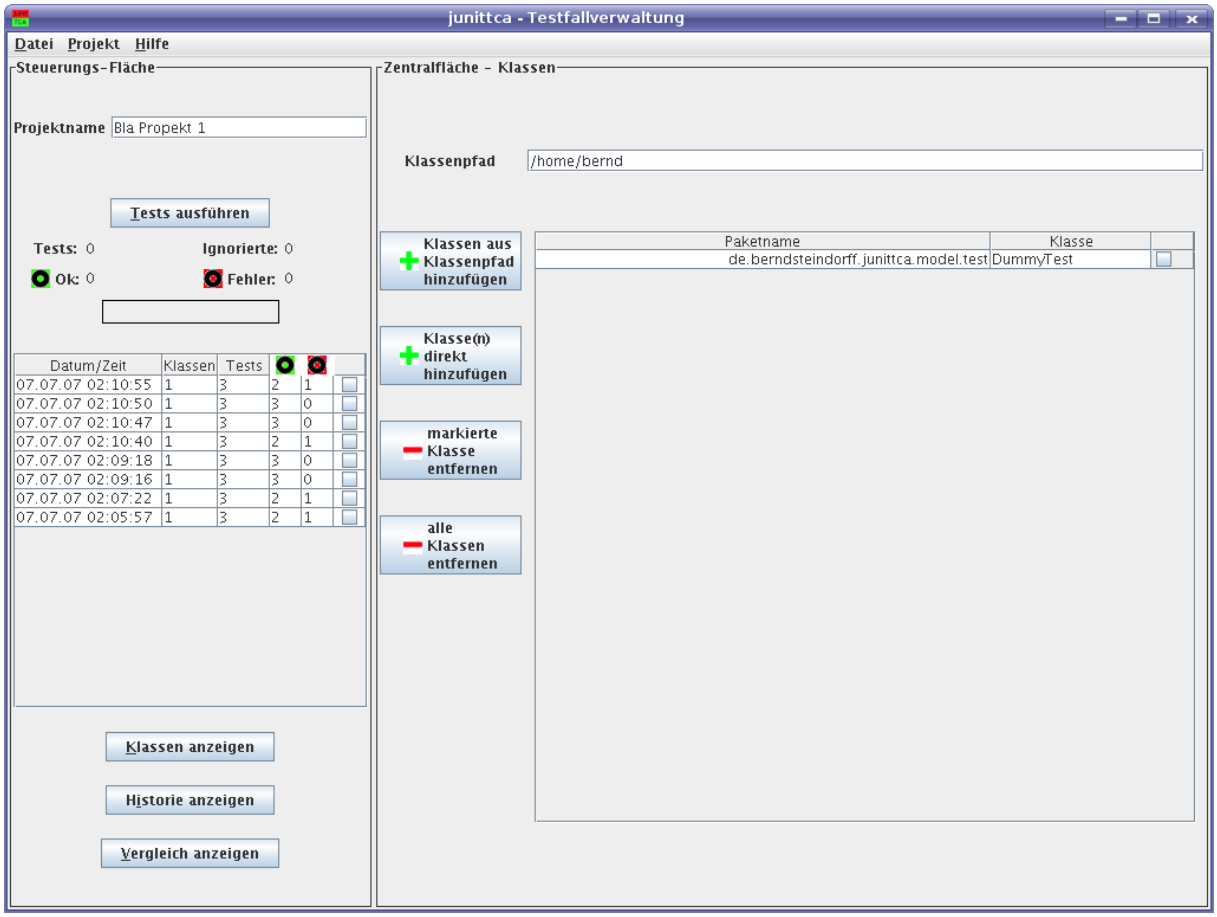
Allgemeines

Die Testfallverwaltung ist eine graphische Oberfläche, um JUnit 4-Tests auszuführen und deren Ergebnisse zu speichern.

Fensteraufbau

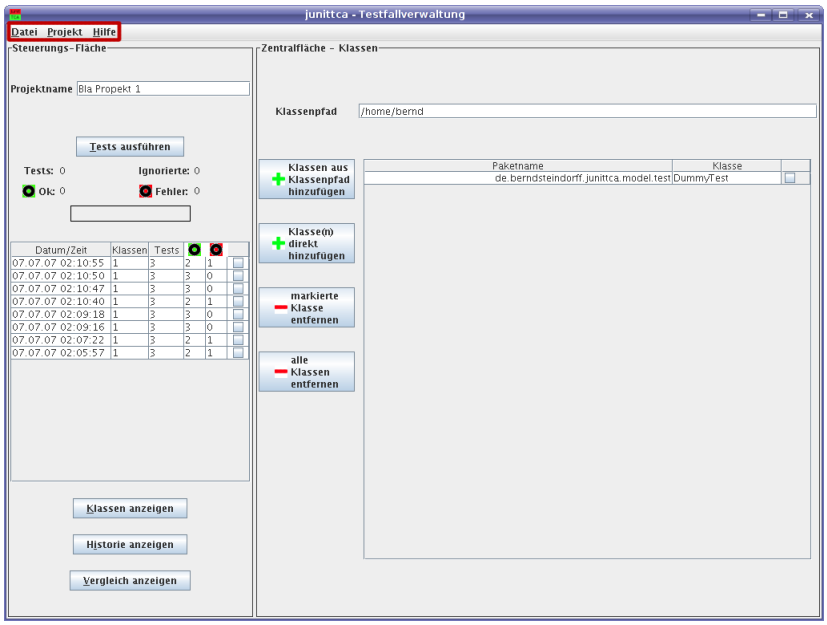
Hier werden die Fenster mit ihren einzelnen Bereichen erläutert.

Grundfenster



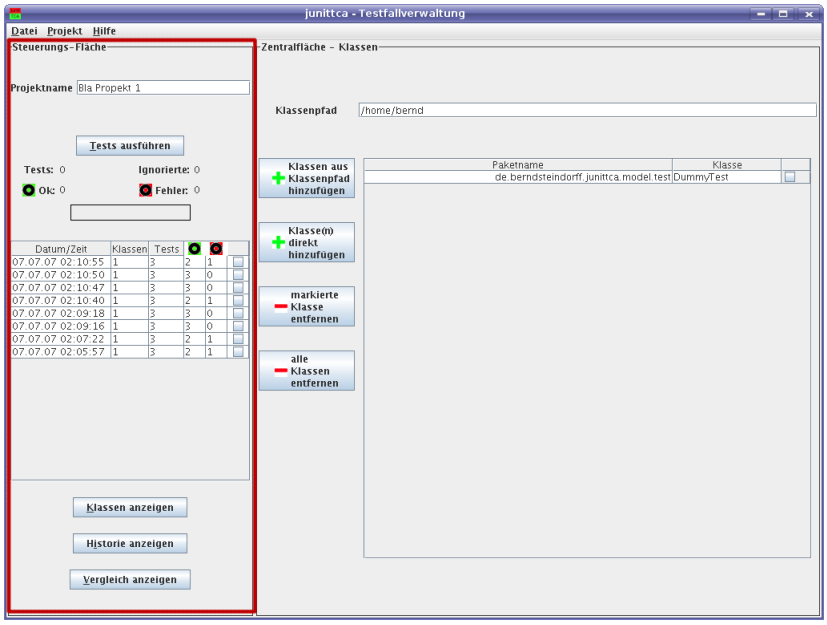
Hier sieht man die Grundfenster der Testfallverwaltung.

Menü



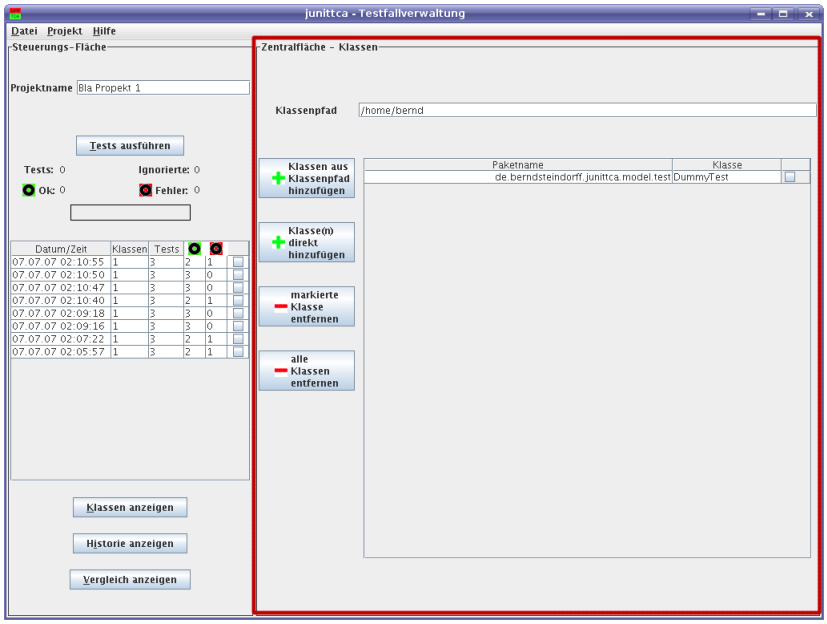
Das Menü (rot umrandet) ermöglicht es, Aktionen mit Projekt-Dateien und Projekten zu machen.

Steuerungsfläche



Die Steuerungsfläche ist der linke Bereich im Fenster (rot umrandet). Die Ansicht bleibt die Programmlaufzeit über bestehen. Neben vier Schaltflächen gibt es ein Textfeld für den Projektnamen und eine Historien-Übersicht im mittleren Teil der Anzeige.

Zentralfläche



Die Zentralfläche (rot umrandet) befindet sich rechts neben der Steuerungsfläche und nimmt die größte Fläche ein. Hier werden verschiedene Ansichten angezeigt, sie werden oben links in der Umrandung mit einem Namen versehen. Es gibt die Ansichten:

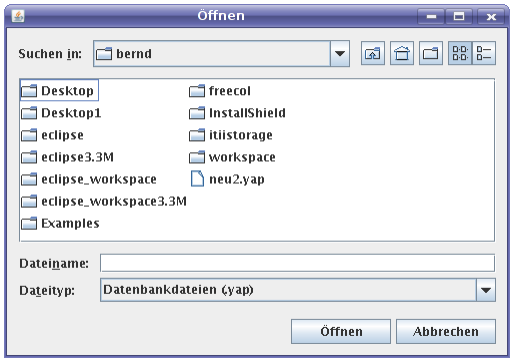
- Zentralfläche – Klassen
- Zentralfläche – Ergebnis eines Testlaufes
- Zentralfläche – Vergleich zweier Testläufe

Aktionen

Hier werden die möglichen Aktionen erklärt, die die Testfallverwaltung anbietet.

Start

Beim Start der Testfallverwaltung erscheint ein Dialog, an dem Sie eine Projekt-Datei auswählen (Endung *.yap*) oder einen neuen Dateinamen eingeben müssen. Das Dialog sieht so aus:



Wir eine vorhandene Projekt-Datei ausgewählt, die Projekte enthält, so wird gefragt, ob Sie ein Projekt auswählen (siehe *Projekt öffnen*) oder ein neue Projekt anlegen möchten (siehe *Neues Projekt anlegen*).

Alternativ wird man aufgefordert, ein neues Projekt anzulegen (siehe *Neues Projekt anlegen*).

Neue Projekt-Datei anlegen

Soll eine neue Projekt-Datei angelegt werden, so müssen Sie im Menü / *Datei* / *Neue Projektdatei* auswählen.

Danach erscheint der bereits in *Start* gesehenen Dialog.

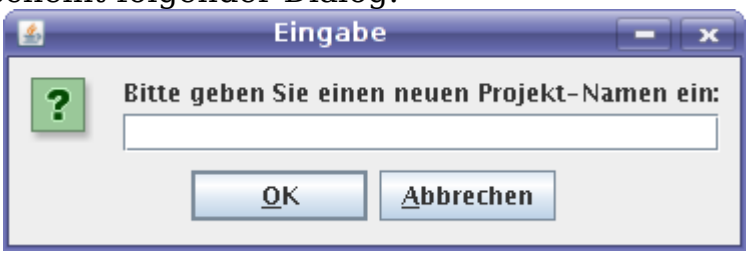
Projekt-Datei öffnen

Soll eine Projekt-Datei geöffnet werden, so müssen Sie im Menü / *Datei* / *Öffnen* auswählen.

Danach erscheint der bereits in *Start* gesehenen Dialog.

Neues Projekt anlegen

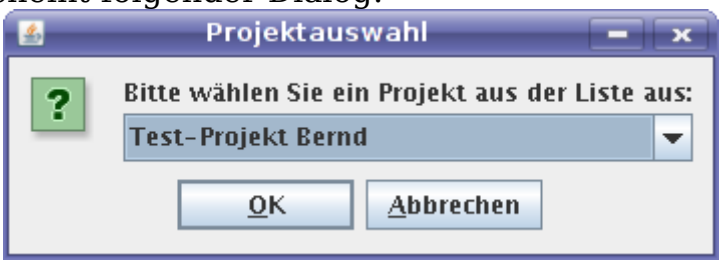
Soll ein neues Projekt angelegt werden, so wählen Sie im Menü / *Projekt* / *Neu*. Dann erscheint folgender Dialog:



Danach öffnet wird das Grundfenster aktualisiert, bis auf den Projekt-Namen sind alle Daten-Felder ohne Inhalt.

Projekt öffnen

Soll ein Projekt geöffnet werden, so wählen Sie im Menü / *Projekt* / *Öffnen*. Es erscheint folgender Dialog:



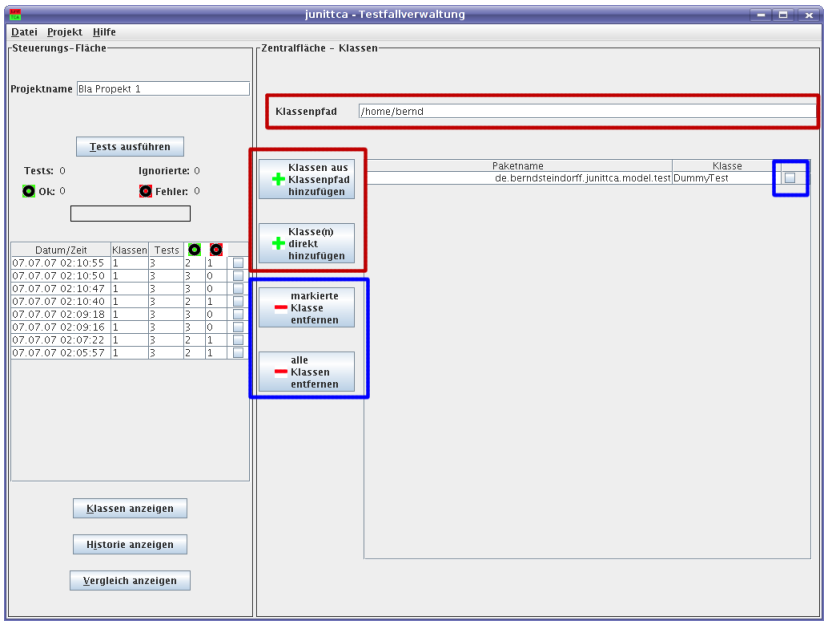
Danach wird das Grundfenster mit den Daten-Inhalten des Projektes aktualisiert.

Projekt Speichern

Soll ein Projekt geöffnet werden, so wählen Sie im Menü / *Projekt* / *Speichern*. Danach sind alle Projekt-Daten gespeichert.

Klassen hinzufügen / entfernen

Klassen dem Projekt hinzufügen und entfernen können Sie über entsprechende Schaltfläche in der *Zentrallfläche – Klassen*. Wird diese nicht angezeigt so sehen Sie bitte unter Punkt *Klassen / Historie / Vergleich anzeigen* nach. Die Aktionen können hier ausgeführt werden:



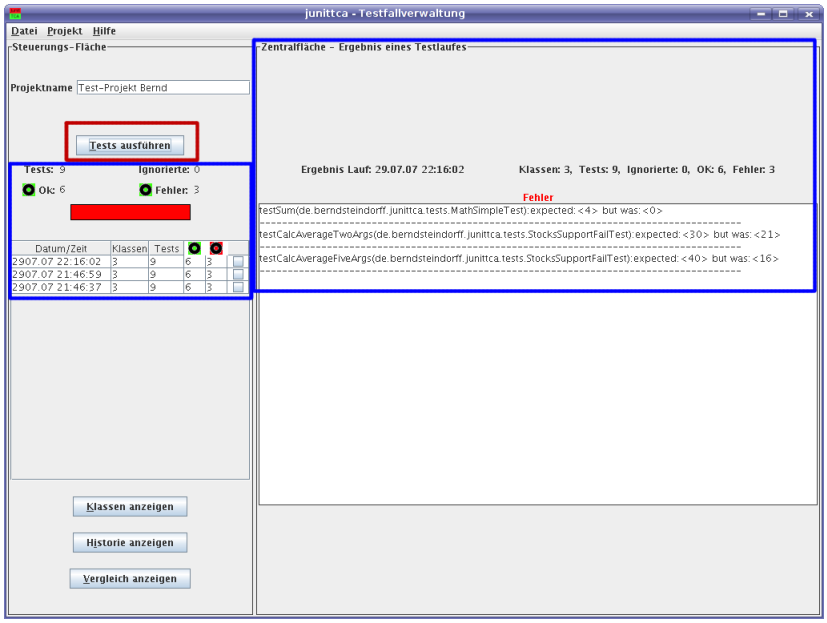
Um Klassen hinzuzufügen muss zuerst ein existierendes Verzeichnis als Klassenpfad eingegebenen werden (obere rote Umrandung). Der Klassenpfad muss in der Art eingegeben sein, die ihr Betriebssystem fordert, z. B. für Windows *C:\work\projekt1\bin; C:\work\projekt2\bin*. Die Klassen müssen dort in übersetzter Form mit normaler Paketstruktur vorliegen.

In der unteren roten Umrandung werden über die Schaltfläche *Klassen aus Klassenpfad hinzufügen* alle Test-Klassen dem Projekt hinzugefügt, die sich im definierten Klassenpfad befinden. Die Schaltfläche darunter, *Klasse(n) direkt hinzufügen*, zeigt Ihnen einen Dateidialog an, in dem Sie Klassen-Dateien einzeln oder zusammen auswählen können. Bitte beachten Sie, dass sich die Klassen-Dateien im definierten Klassenpfad befinden müssen.

Um Klassen zu entfernen haben Sie auch zwei Möglichkeiten. Sie können mit der Schaltfläche *alle Klassen entfernen* (blaue Umrandung links) alle Test-Klassen aus dem Projekt entfernen. Wenn Sie die Schaltfläche *markierte Klasse entfernen* betätigen, können Sie einzelne Klassen in der Übersicht auswählen (blaue Umrandung rechts), diese werden dann aus dem Projekt entfernt.

Tests ausführen

Um Test-Klassen auszuführen, müssen Sie in der Steuerungsfläche die Schaltfläche *Tests ausführen* (rot umrandet, siehe Schaubild unten) betätigen. Das Ergebnis wird im blauen Kasten darunter (in der Steuerungsfläche) als Übersicht angezeigt, dazu gibt es den klassischen grünen oder roten Balken von JUnit, der den Erfolg der Tests anzeigt. Desweiteren erfolgt ein Eintrag des Test-Laufes in der Historien-Übersicht. Rechts wird die Anzeige *Zentralfläche – Ergebnis eines Testlaufes* angezeigt.



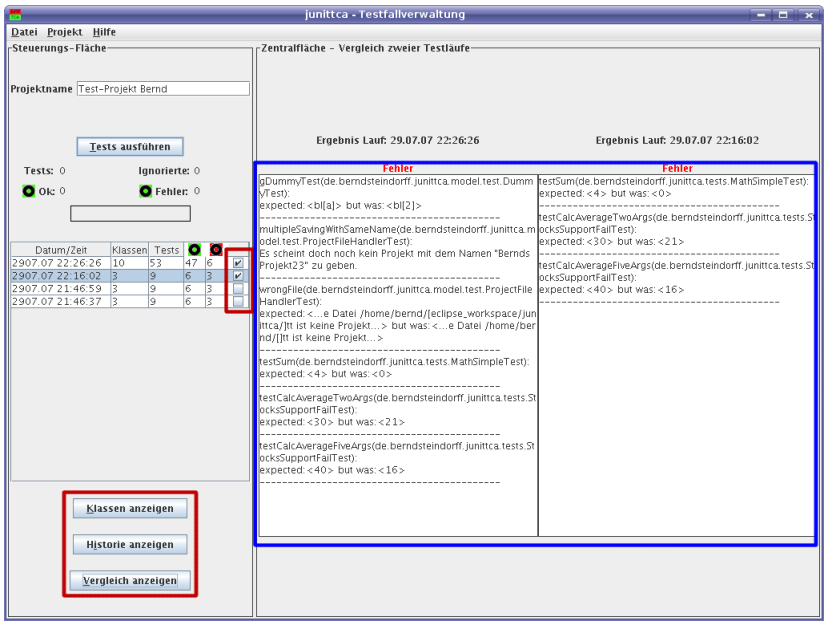
Klassen / Historie / Vergleich anzeigen

In der Zentralfläche gibt es drei Anzeigen.

Die erste Anzeige *Zentralfläche – Klassen* haben Sie bereits in Punkt *Klassen hinzufügen / entfernen* kennen gelernt. Die dazu gehörende Schaltfläche heisst *Klassen anzeigen* (rote Umrandung unten).

Die zweite Anzeige *Zentralfläche – Ergebnis eines Testlaufes* haben bereits in Punkt *Tests ausführen* kennen gelernt. Das Ergebnis eines Testlaufes lässt sich auch unabhängig von einer Ausführung anzeigen, dazu muss ein Testlauf in der Historien-Übersicht (rote Umrandung oben) ausgewählt werden. Danach wird die Schaltfläche *Historie anzeigen* (rote Umrandung unten) betätigt.

Die dritte Anzeige *Zentralfläche – Vergleich zweier Testläufe* vergleicht zwei Test-Läufe miteinander. Dazu müssen in der Historien-Übersicht zwei Test-Läufe ausgewählt werden (rote Umrandung oben). Danach wird die Schaltfläche *Vergleich anzeigen* (rote Umrandung unten) betätigt. Rechts daneben (blaue Umrandung) erscheint dann die dazu gehörende Anzeige.



Anhang D Sonstige Anhänge

Hier sind sonstige Anhänge aufgeführt, auf die in der Diplomarbeit verwiesen wird.

Diagramme zur Objektgraphentiefe

Die Diagramme (auf den folgenden Seiten) sollen die Tiefe des Objektgraphen vor (Abbildung 6) und nach (Abbildung 7) der Überarbeitung des Daten-Modells aufzeigen. Ausgangspunkt ist die Klasse `Project` ganz links. Nach rechts abgehend, werden dann die Typen der Unterelemente aufgeführt. Je Schritt nach rechts, erhöht sich die Tiefe um einen Zähler.

Die Farbgestaltung sieht drei Hintergrundfarben vor:

- weiß: Klassen aus dem eigenen Daten-Modell
- gelb: Klassen aus der Java-Standardbibliothek (inkl. primitiver Datentypen) oder dem Rahmenprogramm JUnit
- rot: Klasse `Class` oder rekursiv arbeitende Klassen (`Class`, `Method`, `Description` und `Throwable`)

Einige Anmerkungen zum Diagramm:

- Reihungen werden nur weiter in ihre Unterelemente aufgelöst, wenn die Typen nicht primitiv oder vom Typ `String` sind.
- Die Klasse `Method` des Paketes `java.lang` wurde nur eine Ebene nach unten dargestellt, da `Method` weit verzweigt.
- Die Klasse `Class` wird nur eine Ebene nach unten dargestellt. Sie wird außerdem in einem gesonderten Graphen (rechts neben dem normalen Objektgraphen) dargestellt, da Class von mehreren anderen Typen referenziert wird. Ebenso wie Method verzweigt Class auch stark, daher wird nur die erste untere Ebene dargestellt. Außerdem geht deckt der Java-Quellcode einige Bereiche nicht ab, so dass das nachvollziehen behindert wird.

Abbildung 6: Diagramm zur Objektgraphentiefe vor der Überarbeitung

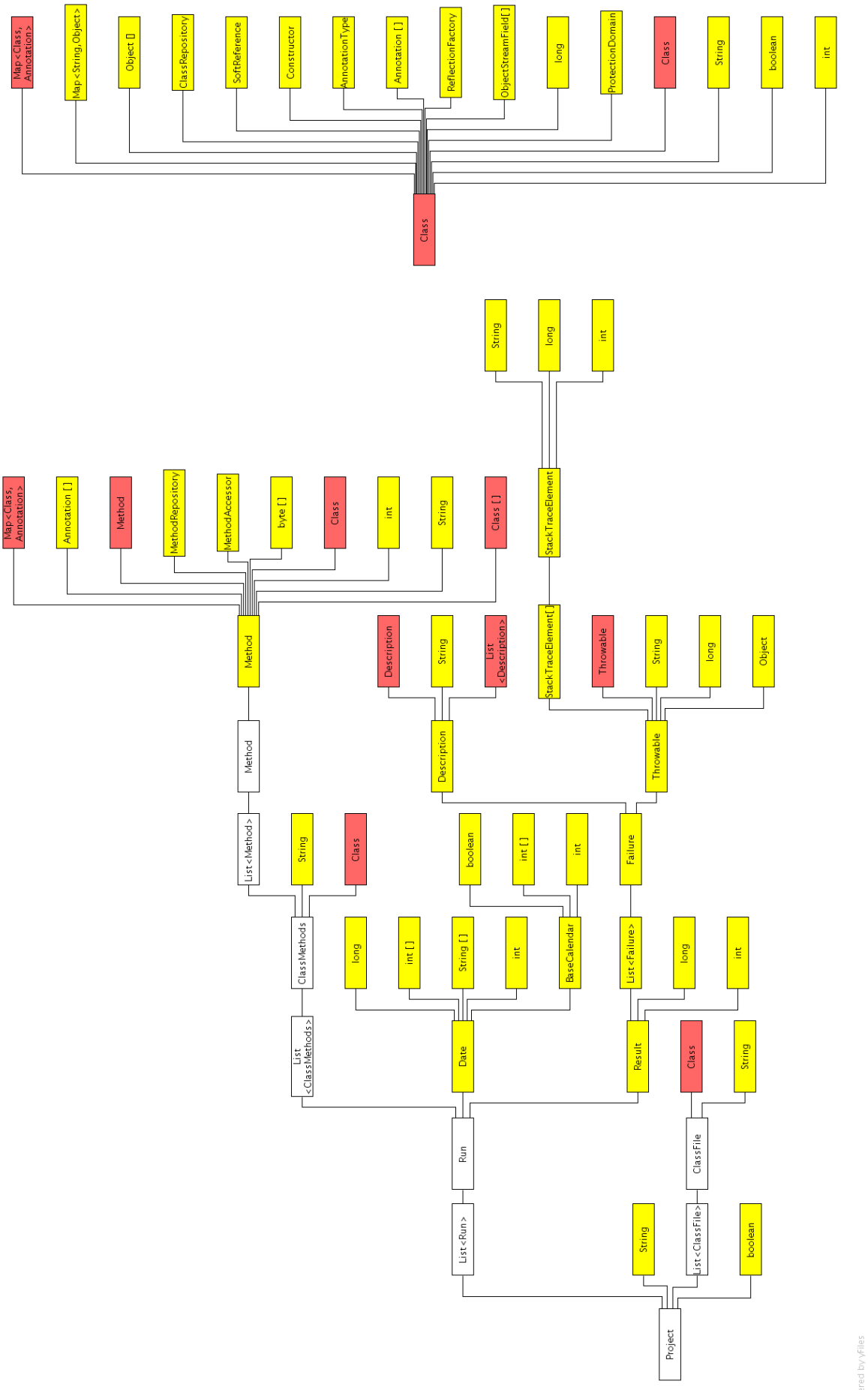
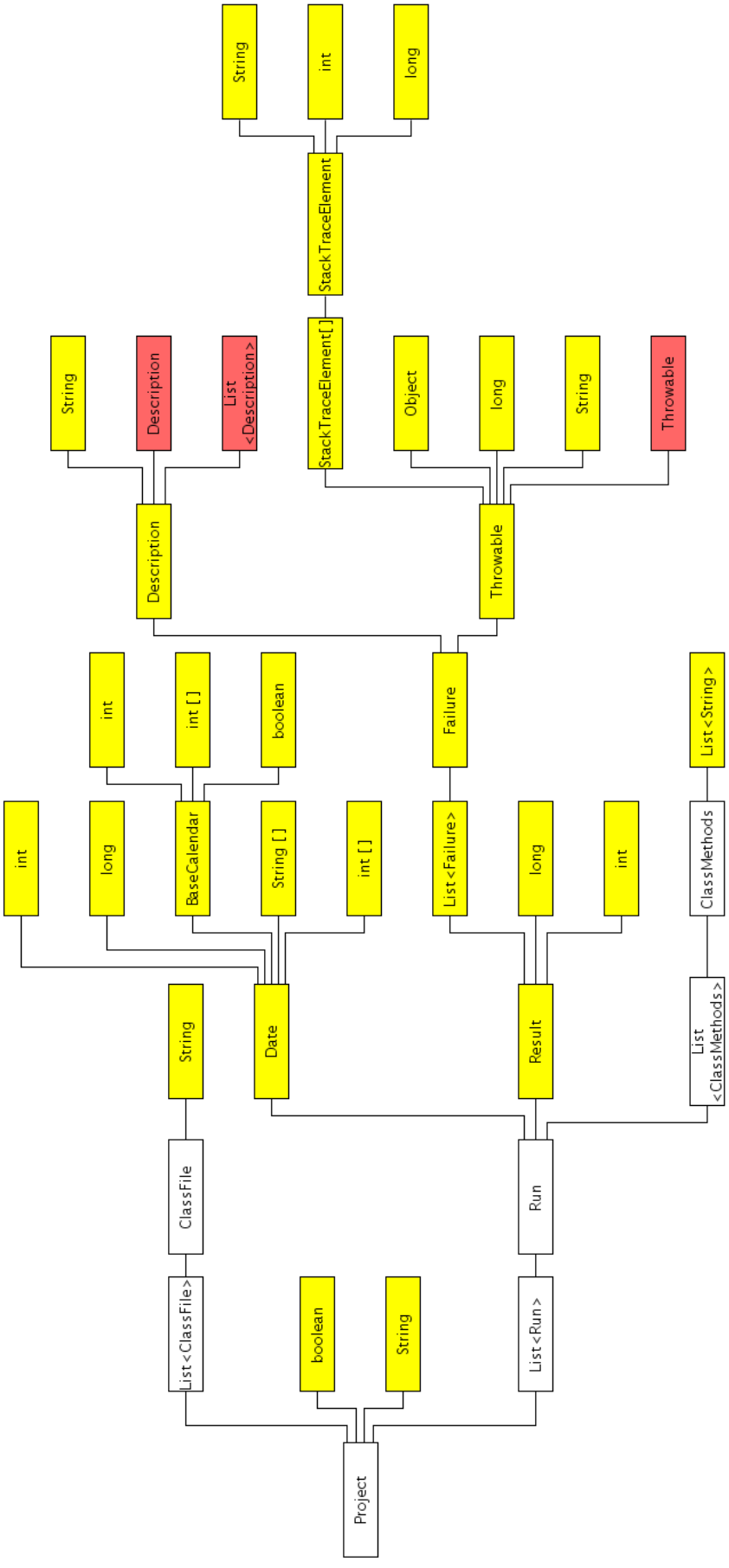


Abbildung 7: Diagramm zur Objektgraphentiefe nach der Überarbeitung



Erläuterung der Verzeichnis-Struktur auf der beiligenden CD

- Diplomarbeit als PDF-Dokument
- Zip-Datei mit der Testfallverwaltung *junittca0.8.zip* (enthält das Programm als Jar-Datei *junittca.jar*, sowie ein lib-Verzeichnis mit benutzten Rahmenprogrammen)
- Zip-Datei mit dem Quellcode und den benötigten Rahmenprogrammen *source20070801.zip*

Programmierstandards

Hier werden die erarbeiteten Programmierstandards aufgeführt,die bei der Testfallverwaltung angewendet wurden. Sie basieren auf den Java Code Conventions. Die Standards werden jeweils nur kurz erläutert, um der XP-Technik folgend daraus kein Buch zu machen:

- Namen:
  - Klassenname beginnen mit einem Großbuchstaben, z. B.
    `public class Project....`
  - Methoden- und Variablennamen mit einem kleinen Buchstaben, z. B.
    `private String classPath....`
  - Klassenvariablen werden komplett groß geschrieben, z. B.
    `private static Logger LOGGER....`
  - Schnittstellen beginnen mit einem großen I, z. B.
    `public interface IProjectModel....`
- Klassenaufbau, bis auf die Kommentare sind die Bereiche durch eine Leerzeile zu trennen:
  - Paket-Zuordnung
  - Import-Anweisungen
  - Klassenkommentar
  - Klassenname mit `extends`- und `implements`-Anweisungen
  - Klassenvariablen
  - Objektvariablen

- Konstruktoren
- Methoden

  Beispiel:

```
package de.berndsteindorff.junittca.model;

import org.apache.log4j.Logger;

/**
* Dies ist die Klasse ClassFile und hat die Aufgabe...
* @author bernd.steindorff
*/
public class ClassFile {
    private static Logger LOGGER = Logger.getLogger...
    private String classPath = „“;
    public ClassFile (String classPath) {
        this.classPath = classPath;
    }

    public String getClassPath () {
        return classPath;
    }
}
```

- Einrückungen:
  - Zeilen sind maximal 80 Zeichen lang.
  - Umbrüche sind an Trenn-Zeichen wie `. + , & |` zu machen. Dabei ist der umgebrochene Code mit 8 Leerzeichen einzurücken.

    Beispiel:

```
    if ( (bedingung1 && bedingung2)
            || (bedingung3 && bedingung4)
            || (bedingung5 && bedingung6) ) {
    }
```

- Initialisierungen: Jede Variable soll initialisiert werden, selbst wenn es der Wert `null` ist. Dies gilt nicht für finale Variablen, die im Konstruktor initialisiert werden.
- Innere Klassen: Sollen unter der letzten Methode eingefügt werden.

- Kommentare:
  - Dokumentationskommentare /** */ sind den einfachen Blockkommentaren /* */ vorzuziehen.

    Beispiel:

    ```
    /**
     * This is a javadoc comment.
     */
    ```

  - Klassen, Konstruktoren und Methoden (nur nicht-triviale) sollen mit Kommentaren versehen werden. Dabei ist Englisch als Sprache zu benutzen, um das Programm bei einer möglichen Verbreitung nicht unnötig umschreiben zu müssen.
- Variablendeklaration: Eine Variable pro Zeile, z. B.

  ```
  int bla1;
  int bla2;
  ```

- Anweisungen: Eine Anweisung pro Zeile, z. B.

  ```
  i++;
  j++;
  ```

- Klammersetzungen: Zur besseren Ansicht sollen geschweifte Klammern (selbst bei nur einer Anweisung): öffnend gleiche Zeile, schließend separat, z. B.

  ```
  if ( zahl == 1 ) {
        macheEtwas();
  } else if ( zahl == 2 ) {
        macheNochEtwas();
  }
  ```

  ```
  oder
  ```

  ```
  if ( (zahl == 1) || (zahl == 2) ) ...
  ```

  Dies gilt auch für Schleifen!

- `If`-Anweisungen: Alle `else` und `if else`-Anweisungen beginnen direkt nach der schließenden Klammer
- Sichtbarkeiten: Es ist so privat wie möglich zu programmieren. Das gilt für Variablen und Methoden.