FACHBEREICH INFORMATIK

LEHRGEBIET PROGRAMMIERSYSTEME

Prof. Dr. Friedrich Steimann

Abschlussarbeit im Studiengang

Master of Computer Science

Ein Eclipse-Framework zur Markierung von logischen Fehlern im Quellcode

Vorgelegt von

Nils Meyer
Birkengrund 30
85276 Pfaffenhofen
Matrikelnummer 6985017
nils.meyer@jcom1.com

Pfaffenhofen, den 4. April 2007

Inhaltsverzeichnis

1 Einleitung

Eine integrierte Entwicklungsumgebung (IDE), die syntaktische und semantische (Typ-) Fehler nicht in der Nähe ihres Auftretens darstellt, ist heutzutage nicht mehr vorstellbar. In Eclipse als weit verbreitete Java-IDE werden diese Fehler dem Programmierer durch Auflistungen von Problemen und Markierungen direkt im Quellcode mitgeteilt. So zeigt Abbildung 1 einen syntaktischen Fehler in der Zeile 11 des Quellcodes und einen semantischen Fehler in der Zeile 6. Die Fehler werden dabei in einer Liste (dem *Problems View*), direkt am Quellcode und in der Struktur des Projektes (im *Package Explorer* links und im *Outline View* rechts) dargestellt.

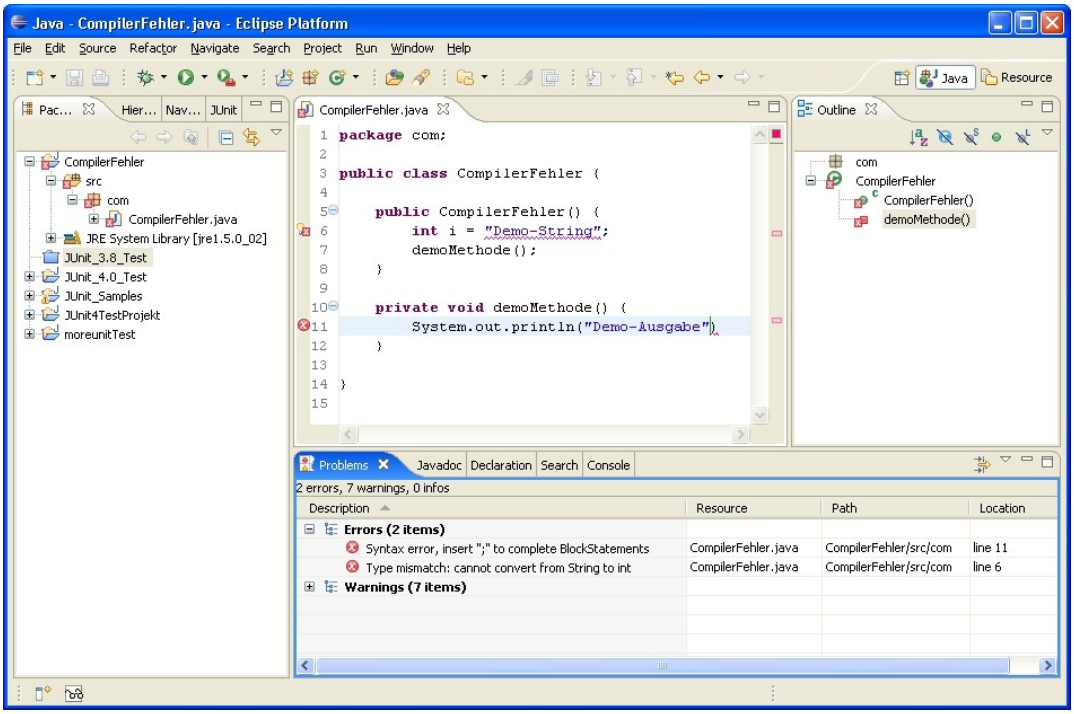

Abbildung 1: Visualisierung von Compilerfehlern in Eclipse

Unit-test-Frameworks wie JUnit probieren, weitere Fehler im Quellcode durch so genannte Modultests (engl. *unit test*) zu finden. Bei diesen Fehlern handelt es sich um so genannte logische Fehler. Bei diesen ist der Quellcode zwar syntaktisch korrekt und lauffähig, liefert aber nicht das erwartete Ergebnis. Ein Beispiel für so einen Fehler ist eine Additionsfunktion, die für die Parameter 12 und 14 nicht 26, sondern 29 zurück gibt.

Die Implementierung von Modultests erfolgt in Form von Testmethoden. In diesen beschreibt der Entwickler das Ergebnis, das er bei der Übergabe bestimmter Parameter an das zu testende Modul erwartet. Wird diese Testmethode ausgeführt und liefert das Modul nicht das erwartete Ergebnis zurück, ist ein logischer Fehler in diesem Modul sehr wahr-

scheinlich[1]. Listing 1 zeigt ein Beispiel für einen JUnit-Testmethode. Dieser überprüft, ob die Addition der Objekte die 12 beziehungsweise 14 CHF darstellen, gleich dem Objekt ist das 26 CHF darstellt.

```
1: @Test public void testSimpleAdd() {
2:         // [12 CHF] + [14 CHF] == [26 CHF]
3:         Money expected= new Money(26, "CHF");
4:         assertEquals(expected, f12CHF.add(f14CHF));
5: }
```

Listing 1: JUnit Beispiel – Eine einfache Testmethode

Schlägt die Ausführung einer Testmethode fehl, so bietet Eclipse die Navigation zu dieser fehlgeschlagenen Testmethode an, nicht aber zu der eigentlichen Ursache des Fehlschlags. Diese muss vom Entwickler aus der Testmethode ermittelt werden. Wie in Abbildung 2 ersichtlich werden logische Fehler also nicht auf eine syntaktischen und semantischen Fehlern entsprechende Art und Weise visualisiert.

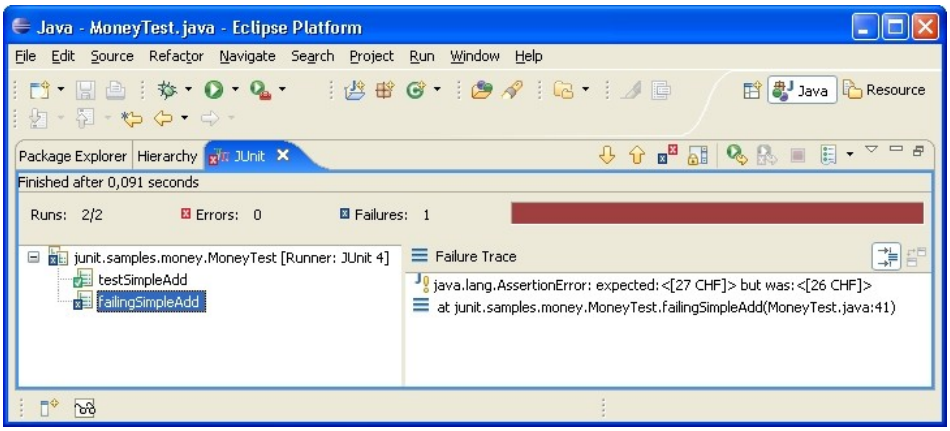

Abbildung 2: Visualisierung eines logischen Fehlers in Eclipse

Betrachtet man diesen Weg analog für das Auftreten eines syntaktischen Fehlers, würde das bedeutenden, dass der Entwickler lediglich angezeigt bekommen würde, welche syntaktische Regel verletzt wurde. Von dieser ausgehend müsste er dann die passende Stelle im Quellcode finden, an der diese Regel verletzt ist – ein Vorgehen, dass so nicht praktikabel ist.

Ziel dieser Arbeit ist die Entwicklung einer JUnit-Erweiterung und deren Einbindung in ein Eclipse-Plugin, das die Visualisierung eines logischen Fehlers auf dieselbe Art und Weise wie die eines syntaktischen Fehlers ermöglicht. Dazu werden in Kapitel 2 die nötigen Grundlagen beleuchtet. Diese reichen von Extreme Programming über den Test von Modulen mit Hilfe von JUnit bis hin zu Eclipse und der JUnit Integration in dieser Plattform. In Kapitel 3 wird der Beitrag dieser Arbeit und in Kapitel 4 dessen Implementierungen in

1 Vergleiche [Steimann et. al 2005], Seiten 81 und 82

Form eines Eclipse-Plugins erläutert. Die Diskussion der zugrunde liegenden Ideen und des Plugins in Kapitel 5 leitet zur Gegenüberstellung der selbigen mit verwandten Arbeiten über. Abgeschlossen wird diese Arbeit dann im Kapitel 6 mit einem Ausblick auf mögliche Erweiterungen des Plugins.

Im Rahmen dieser Arbeit werden dabei speziell JUnit Modultests betrachtet. Die Konzepte lassen sich aber auf andere Programmiersprachen und andere xUnit-Implementierungen übertragen.

2 Grundlagen

Zu Beginn sollen an dieser Stelle die für diese Arbeiten wesentlichen Grundlagen betrachtet werden. Diese beschränken sich vor allem im Abschnitt 2.3 hauptsächlich darauf, Querverweise zu relevanter Literatur zu liefern, die das Thema sehr ausführlich behandelt.

2.1 Extreme Programming

Extreme Programming (XP) ist ein Softwareentwicklungansatz, dessen Fokus auf Flexibilität und Teamwork liegt. Basierend auf den Ideen von Kent Beck, wurde er 1996 von ihm erstmals in einem Smalltalk-Entwicklungsprojekt bei Chrysler eingesetzt.

Die Grundlage für seinen Ansatz hat er in [Beck & Andres 2005] auf Seite 127 zusammengefasst: „an always-deployable system to which features, chosen by the customer, are added and automatically tested on a fixed heartbeat."

2.1.1 Grundlegende Ideen

Beck bindet in seinen Ansatz neben verschiedenen Vorgehensweisen (*practices*) auch die zu Grunde liegenden Werte (*values*) und Prinzipien (*principles*) mit ein. Während die Vorgehensweisen ganz konkrete Schritte und Techniken beschreiben, stellen die Werte und Prinzipien die Gründe dar, warum eine Tätigkeit nötig ist.

Diese Einbindung ermöglicht es ihm, jede Tätigkeit zu hinterfragen, auf die zu Grunde liegende Prinzipien und Werte zurück zu führen und die Tätigkeiten zu eliminieren, die sich nicht auf definierte Werte stützen und um der Tätigkeit willen durchgeführt werden. Dadurch kann der Ansatz Extreme Programming schlank gehalten werden.

Obwohl die Werte und Prinzipien von Projekt zu Projekt unterschiedlich und auch unterschiedlich stark ausgeprägt sein können, führt [Beck & Andres 2005] einige Werte auf, die für ein Projekt auf Basis des Extreme Programming Ansatzes unerlässlich sind:

- Kommunikation (*communication*)
- Einfachheit (*simplicity*)
- Feedback
- Mut (*courage*)
- Respekt (*respect*)

Bei der Betrachtung dieser Werte wird bereits offensichtlich, dass Extreme Programming sich nicht auf Methoden und Abläufe beschränkt, sondern die Gründe für den Erfolg oder Misserfolg eines Projektes viel mehr in den Menschen sieht, die am Projekt beteiligt sind. Die Definition gemeinsamer Werte ist der Grundstein für den Erfolg eines Projektes.

Die definierten Werte alleine sind allerdings zu global, um daraus direkt Vorgehensweisen abzuleiten. Um die Lücke zwischen Werten und Vorgehensweisen zu schließen, kommen bei Beck Prinzipien zum Einsatz.[2] Prinzipien sind Richtlinien für bestimmte Situationen. Was damit gemeint ist, wird an einem Beispiel klar: In der Richtlinie *Failure*[3] beschreibt Beck das Verhalten, das er von einem Entwickler erwartet, wenn nicht klar ist, welche mögliche Implementierung für ein Problem die effektivste ist: Er soll die Möglichkeiten probieren und aus den Fehlschlägen lernen. Diese Richtlinie beschreibt noch keine konkrete Vorgehensweise, aber konkretisiert das Verhalten für eine spezielle Situation. Damit wird durch das Prinzip *Failure* eine Brücke zwischen den Werten Mut und Einfachheit und einem konkreten Verhalten gebaut.

Auf die treibenden Prinzipien von Extreme Programming soll hier nicht weiter eingegangen werden. Für diese sei genau wie auch für nicht näher ausgeführte Verhaltensweisen auf [Beck & Andres 2005] verwiesen.

Abbildung 3[4] gibt eine Übersicht über die Vorgehensweisen, die im Extreme Programming Umfeld zum Einsatz kommen (können). An dieser Stelle soll lediglich die Vorgehensweise betrachtet werden, die durch die Ergebnisse dieser Arbeit unterstützt wird: *Test-First Programming*.

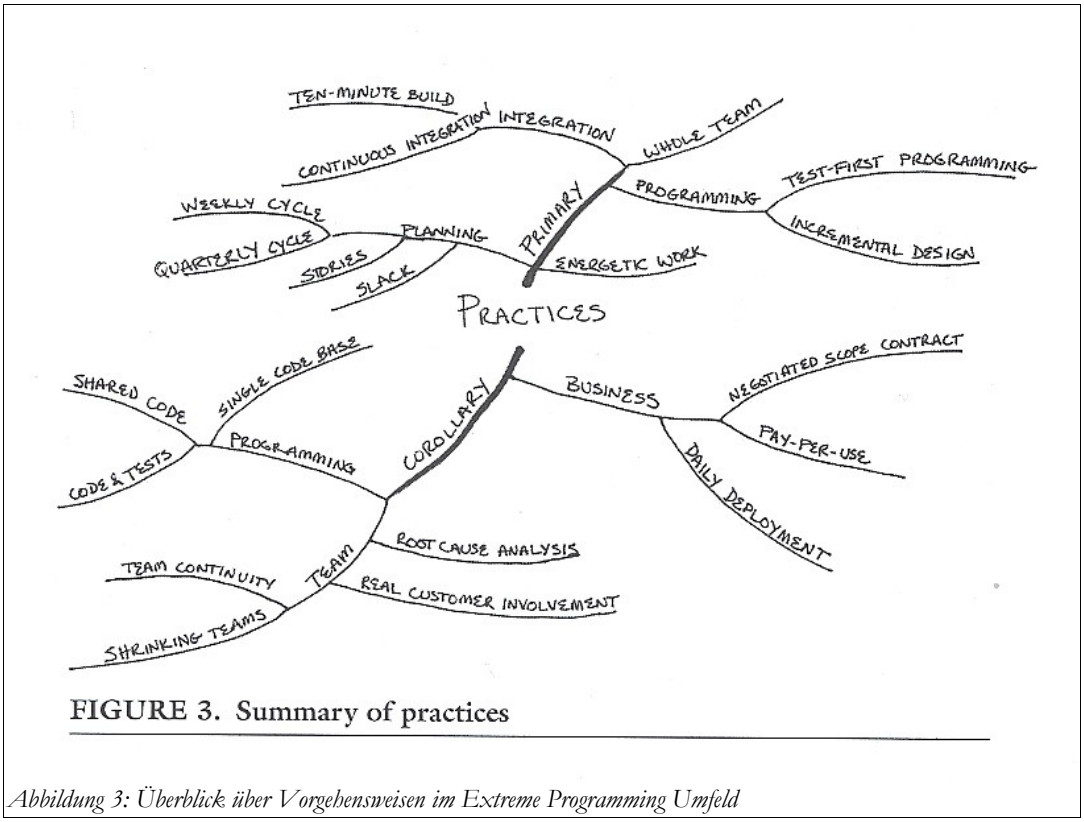

Abbildung 3: Überblick über Vorgehensweisen im Extreme Programming Umfeld

2 Vgl. [Beck & Andres 2005], Seite 14-15
3 Vgl. [Beck & Andres 2005], Seite 32
4 Vgl. [Beck & Andres 2005], Seite 36

2.1.2 Test-First Programming

Mit der Vorgehensweise *Test-First Programming*[5] schlägt Beck vor, vor der Änderung eines Stück Codes erst einen automatisierten Test zu schreiben, der so lange fehlschlägt, bis die Änderung am Code das erwartete Ergebnis liefert. Dadurch werden verschiedene positive Ergebnisse gefördert:

- Konzentration auf das Wesentliche:
  Der Entwickler ändert nur Code, der auch wirklich nötig ist, damit der Test nicht mehr fehlschlägt.
- Lose Kopplung von Funktionalitäten:
  Guter Code ist leicht alleine testbar, bei stark verwobenem Code ist es schwierig, Tests zu schreiben.
- Vertrauen im Team
  Funktionierende Test zeigen Kollegen, was der entwickelte Code tun soll, und dass er diese Intention auch wirklich erfüllt.

Jeder der so entstandenen Tests hat einen sehr begrenzten Fokus. Dies ist aber nicht negativ, sondern führt dazu, dass er in der Regel sehr wenig Ausführungszeit benötigt. Dadurch können alle Tests sehr häufig durchgeführt werden und zeigen Probleme sehr schnell auf, die durch eine Änderung (womöglich an einer ganz anderen Stelle) hervorgerufen wurden.

In [Saff & Ernst 2004] und auch in [Beck & Gamma 2004] ist dabei der Begriff „häufig" sogar dem Begriff „ständig" gewichen. Sie schlagen vor, die entstandenen Test genauso oft durchzuführen, wie der Code compiliert wird. Das Kapitel 5.2.2, „Continuous Testing", befasst sich mit diesem Thema noch ausführlicher.

Die Wichtigkeit, die Kent Beck dieser Vorgehensweise beimisst, kann man auch daran erkennen, dass er mit [Beck 2005] ein Buch geschrieben hat, das sich rein auf die test-getriebene Softwareentwicklung konzentriert.

2.2 JUnit

JUnit ist ein Framework für die Durchführung von Modultests in Java-Programmen. Es kommt daher häufig für das *Test-First Programming* in Java-Projekten zum Einsatz, die auf den Extreme-Programming-Ansatz setzen. Mit Kent Beck und Erich Gamma als Autoren wurde es federführend von dem Protagonisten des Extreme Programming (Beck) und dem Leiter der Entwicklung der Eclipse Plattform (Gamma) entwickelt. Unter [junit.org 2007] stehen neben den Quellen des Framework auch grundlegende Artikel und Referenzen zu verschiedenen Büchern bereit, die sich weitergehend mit dem Thema befassen. Dort finden

5 Vgl. [Beck & Andres 2005], Seiten 50, 51

sich auch Verweise zu anderen Seiten, die ähnliche Implementierungen für andere Programmiersprachen bereitstellen. Diese werden zusammen mit JUnit als xUnit-Framework bezeichnet.

In viele Entwicklungsumgebungen ist JUnit direkt eingebunden. Dadurch wird bereits eine enge Verzahnung von Modulprogrammierung und -test ermöglicht. Auf die Möglichkeiten und Grenzen der Integration in Eclipse wird im Kapitel 2.3.1, „Eclipse JUnit Support“, noch explizit eingegangen. Hier sollen im weiteren nur der grundsätzliche Aufbau eines JUnit Testfalls und die Ausführung von JUnit Testfällen erläutert werden. Außerdem wird das Beispiel vorgestellt, das im Rahmen dieser Arbeit verwendet wird, um die neu entwickelten Konzepte zu illustrieren.

In JUnit gibt es pro Test eine Testmethode. Mehrere Tests werden zu einer Testsuite zusammengefasst und können sich gemeinsame Hilfsmethoden teilen. Dies sind zum Beispiel Methoden für den Aufbau eines für den Test nötigen Objektgeflechts, die so genannte Test Fixture, und dessen Freigabe.[6]

Die Ausführung eines Tests kann zu zwei Ergebnissen führen: Entweder ist der Test erfolgreich oder er schlägt fehl. Im Falle eines Fehlschlags werden allerdings noch zwei Fälle unterschieden: einfache Fehlschläge (*failures*) und Ausnahmen (*exceptions*). Während erstere signalisieren, dass ein Rückgabewert nicht dem vorgegebenen Vergleichswert entspricht, treten letztere auf, wenn ein nicht erwarteter Java-Fehler geworfen wird. Wichtig ist dabei, dass der Java-Fehler nicht erwartet wurde: Ein Test kann auch durchaus definieren, dass seine Ausführung zum Wurf eines Java-Fehlers führt; somit signalisiert genau dieser Wurf, dass die Durchführung des Tests erfolgreich verlaufen ist.

2.2.1 Testdefinition

Die Markierung einer Methode als Testmethode ist von der JUnit- Version abhängig. In den Versionen vor JUnit 4 müssen Testmethoden mit dem Prefix `test` im Namen gekennzeichnet werden. Die Namen der Methoden für den Auf- und Abbau der Fixture sind mit `setUp()` und `tearDown()` festgelegt.

Im Gegensatz zu früheren Versionen geht die aktuelle JUnit Version von den Namenskonventionen weg und setzt statt dessen auf ein neues Java- Sprachkonstrukt, das mit der Java-Version 5.0 eingeführt wurde: die Annotationen. Diese sind ein Konzept, mit dessen Hilfe Anmerkungen in Java-Quellcode eingebaut werden können.[7]

Der bisher verwendete Prefix `test` wird dabei durch eine Annotation `@org.junit.Test` ersetzt. Methoden, die die Fixture eines Tests initialisieren beziehungsweise aufräumen, werden über die Annotationen `@org.junit.Before` beziehungsweise `@org.junit.After` gekennzeichnet. Auch weitergehende Informationen zu einer Testme-

6 Vgl. [Steimann et al. 2005], Seite 83-84
7 Vgl. [Nowak 2005], Seite 165

thode werden über Annotationen oder Annotationsparameter definiert. So schließt zum Beispiel die Annotation `@org.junit.Ignore` einen Test vorübergehend aus der Testausführung aus. Für eine vollständige Beschreibung der Annotationen und ihrer Parameter sei an dieser Stelle auf [JUnit API] verwiesen.

2.2.2 Testausführung

Frühere JUnit Versionen beinhalten eigene Werkzeuge für die Ausführung und graphische Übersicht über ausgeführte Tests. Der Aufruf über die Swing-, AWT- und Textschnittstelle ist in Listing 2 dargestellt.

```
1: junit.swingui.TestRunner.run(junit.samples.money.MoneyTest.cl
   ass);
2: junit.awtui.TestRunner.run(junit.samples.money.MoneyTest.clas
   s);
3: junit.textui.TestRunner.run(junit.samples.money.MoneyTest.cla
   ss);
```

Listing 2: Testausführung in JUnit 3.8 über Swing-, AWT- und Textschnittstelle

Abbildung 4 zeigt die graphische Darstellung durch die Swing GUI.

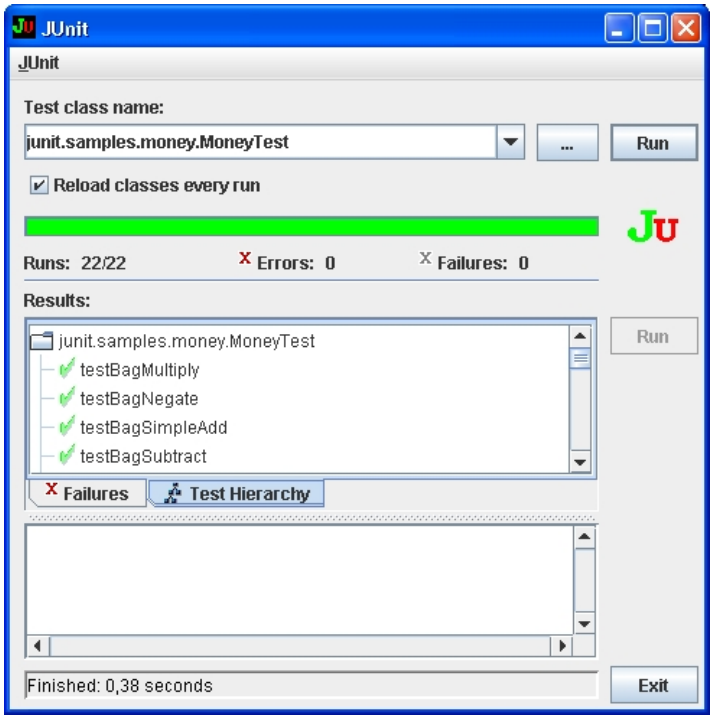

Abbildung 4: JUnit 3.8 Swing GUI Darstellung eines Testlaufs

Um die Nutzung von JUnit zu vereinfachen, ist allerdings eine Integration in die Entwicklungsumgebung von Vorteil. Diese ermöglicht es zum Beispiel, von einem fehlgeschlagenen Testfall aus der Liste der durchgeführten Tests zum Quellcode der entsprechenden Testmethode zu springen. Da eine solche Integration in die Entwicklungsumgebung sehr weit verbreitet ist, werden die graphischen Testrunner, die im wesentlichen dasselbe tun wie die IDE Einbindungen, seit der JUnit Version 4.0 nicht mehr mit ausgeliefert. Statt dessen werden von der Klasse `JUnitCore` verschiedene Methoden zur Verfügung gestellt, mit deren Hilfe eine Entwicklungsumgebung Testfälle ausführen kann, die Ergebnisse zurück geliefert bekommt und in einer integrierten Form darstellen kann.

2.2.3 JUnit Beispiel

In dieser Arbeit wird durchgängig ein JUnit-Beispiel verwendet. Dabei handelt es sich um ein Standardbeispiel, das mit JUnit ausgeliefert und das auch in [Beck 2005] verwendet wird.

In diesem Beispiel werden Objekte vom Typ `Money` definiert, die neben einem Wert auch eine Währung besitzen. Die Klasse `Money` bietet Methoden für die Addition und Subtraktion ihrer Objekte. Der Quellcode des in den folgenden Tests verwendeten Interfaces `IMoney` und der Klassen `Money` und `MoneyBag` ist neben den oben genannten Quellen auch auf der beiliegenden CD zu finden.

```
1: @Before public void setUp() {
2:         f12CHF= new Money(12, "CHF");
3:         f14CHF= new Money(14, "CHF");
4:         f7USD= new Money( 7, "USD");
5:         f21USD= new Money(21, "USD");
6:
7:         fMB1= MoneyBag.create(f12CHF, f7USD);
8:         fMB2= MoneyBag.create(f14CHF, f21USD);
9: }
```

Listing 3: JUnit Beispiel – Erstellung der Fixture

Für den Aufbau einer Fixture für alle Tests kommt die Methode `setUp()` in Listing 3 zum Einsatz. Sie folgt von ihrem Namen her noch den JUnit 3.8 Namenskonventionen, obwohl das durch die Verwendung von JUnit 4 und der Annotation `@Before` in Zeile 1 nicht nötig wäre. In den Zeilen 2-8 werden dann die (im Kopf der Klasse definierten) Felder mit neuen `Money` und `MoneyBag` Objekten initialisiert. Sowohl die Klasse `Money` als auch die Klasse `MoneyBag` implementieren das Interface `IMoney`. Ein `Money` Objekt ist dabei durch seinen Wert und eine Währung definiert. Ein `MoneyBag` Objekt ist ein Behälter für mehrere `IMoney` Objekte.

Eine Methode, die mit `@org.junit.After` annotiert ist wird in diesem Beispiel nicht benötigt, da keine Systemressourcen belegt werden, die explizit wieder freigegeben werden müssen. Die in der Methode `setUp()` erzeugten Objekte können automatisch vom Java Garbage Collector zerstört werden, sobald der Test gelaufen ist und keine Referenz mehr auf das vom Testrunner erzeugte Testobjekt vorhanden ist.

```
1: @Test public void testSimpleAdd() {
2:         // [12 CHF] + [14 CHF] == [26 CHF]
3:         Money expected= new Money(26, "CHF");
4:         assertEquals(expected, f12CHF.add(f14CHF));
5: }
```

Listing 4: JUnit Beispiel – Eine einfache Testmethode

Listing 4 zeigt eine der Testmethoden, die in der Testklasse definiert ist. Auch sie hält sich wiederum an die Namenskonvention von JUnit 3.8, obwohl das auch hier durch die Annotation `@Test` in Zeile 1 nicht nötig wäre. Zeile 2 enthält einen Kommentar, der lediglich erklärt, dass mit dieser Testmethode überprüft werden soll, ob die Addition von 12 Schweizer Franken und 14 Schweizer Franken tatsächlich 26 Schweizer Franken ergibt. Durch die Initialisierung der Variablen `expected` in Zeile 3 mit einem `Money` Objekt, das durch den Wert 26 und die Währung CHF definiert ist, wird offensichtlich, dass dieses Objekt das erwartete Ergebnis ist. Die Methode `assertEquals` in Zeile 4 vergleicht dieses erwartete Ergebnis mit dem Ergebnis, das sich durch die Addition der beiden in der Fixture definierten Objekte ergibt. Die Methoden `assertEquals`, `assertTrue` und `assertFalse` werden von der Klasse `org.junit.Assert` bereitgestellt. Sie überprüfen, ob die zwei übergebenen Werte übereinstimmen beziehungsweise `true` oder `false` sind. Ist dies nicht der Fall, wird ein `java.lang.AssertionError` geworfen, der wiederum von einem Testrunner gefangen und in die Fehlerliste aufgenommen werden kann.

2.3 Eclipse

Was ist Eclipse? Diese Frage wird immer schwieriger zu beantworten.

In [ECL2001], der Pressemitteilung zur Gründung des Konsortiums Eclipse.org, wurde die Eclipse-Plattform noch als Umgebung für Entwicklungswerkzeuge bezeichnet. Bereits zu diesem Zeitpunkt war ein Plugin-Konzept die Basis für die Architektur von Eclipse, was es den beteiligten Firmen ermöglicht, die Eclipse-Plattform als Basis für ihre Produkte zu nutzen, wie zum Beispiel IBM für ihren Websphere Application Developer (WSAD). Spätestens mit der Version 2.1 im April 2003 ist Eclipse zumindest zu einer bedeutenden Java IDE (Integrierten Entwicklungsumgebung) aufgestiegen, bedingt auch dadurch, dass gerade im Unterprojekt Java Development Tools (JDT) viele Erweiterungen implementiert werden, die in anderen Entwicklungsumgebungen nicht ähnlich stark unterstützt werden, wie z.B. Refactoring-Werkzeuge.

Da die Möglichkeiten der Eclipse Plattform von vielen Entwicklern nicht nur für Entwicklungswerkzeuge, sondern auch für Applikationen in ganz anderen Domänen genutzt werden, entschließt sich die mittlerweile gegründete Non-profit-Organisation Eclipse Foundation im Juni 2004 dazu, mit der Version 3.0 eine Eclipse Rich Client Platform zu schaffen. Diese wird in der Pressemittleilung [ECL2004] angekündigt und dient als Basis für alle Arten von Desktop-Anwendungen – sowohl Entwicklungswerkzeuge, als auch Tools anderer Anwendungsgebiete.

Heute, Ende März 2007, stehen hinter dem Namen Eclipse 10 Top-Level Projekte, die sich in ca. 50 Sub-Projekte unterteilen. Es gibt unter dem Label Eclipse neben der bekannten Java-IDE Unterstützung für die C++, Cobol, und PHP Entwicklung, Werkzeuge für das Definieren von Datenstrukturen, für Webservices und SOA, für das Testen und das Monitoren von Anwendungen etc. Darüber hinaus führt die Seite „eclipse plugin central“ (http://www.eclipseplugincentral.com/) 829 Plugins auf, die sich ebenfalls in das Framework einbinden lassen, deren Entwicklung aber nicht innerhalb der Eclipse Foundation betrieben wird.

Für diese Arbeit ist es im ersten Schritt ausreichend, Eclipse als eine offene Java-IDE zu betrachten, die über ein ausgefeiltes Plugin-Konzept Erweiterungen an fast allen denkbaren Stellen ermöglicht. Für eine Einführung in Eclipse und eine Beschreibung des Plugin-Konzeptes sei an dieser Stelle auf folgende Literatur verwiesen:

- [Beck & Gamma 2004] stellt neben den ersten Schritten, die für die Entwicklung eigener Eclipse-Erweiterungen nötig sind, auch einige Ideen da, wie sich die Macher von Eclipse das typische Vorgehen bei den Entwicklung einer Erweiterung vorstellen.
- [Daum 2005] beschreibt an Hand von Beispielen verschiedenste Erweiterungspunkte, betrachtet aber auch den Aufbau der Plattform. Im Gegensatz zu [Beck & Gamma 2004] geht er aber auch auf die *Rich Client Platform* ein.
- Eine kurze Zusammenfassung grundlegender Aspekte findet sich auch in [Bach 2007] im Anhang der Arbeit. Bis auf den Abschnitt „Natures und Builders“ kommen alle dort beschriebenen Erweiterungspunkte auch in dieser Arbeit zum Einsatz.

2.3.1 Eclipse JUnit Support

Wie bereits im Kapitel 2.2 angedeutet, existiert bereits heute eine Integration von JUnit in Eclipse. Diese bietet sowohl Unterstützung für die Erzeugung und Ausführung von JUnit Tests, als auch eine Schnittstelle für die Verarbeitung der anfallenden Testergebnisse.

2.3.1.1 Testfallverwaltung

Eclipse bietet unter Datei > Neu > Test Case einen Wizard an, der den Entwickler bei der Erstellung von Testmethoden unterstützt.

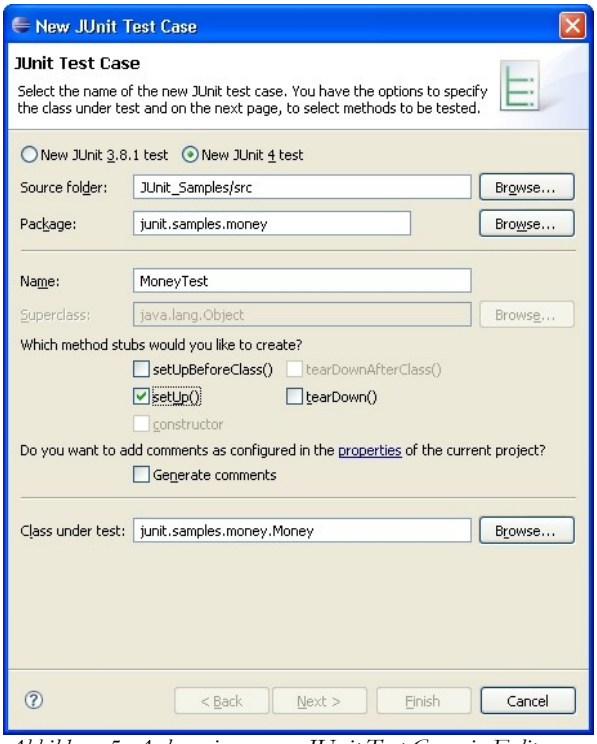

Abbildung 5: Anlage eines neuen JUnit Test Cases in Eclipse

Wie in Abbildung 5 ersichtlich ist, bietet dieser Wizard an, auf Basis einer zu testenden Klasse (in diesem Fall `junit.samples.money.Money`) sowohl eine Klasse für die Testfälle (in diesem Fall `MoneyTest`) als auch die passenden Methoden für die Fixture zu erstellen. In einem zweiten Schritt bietet der Wizard im Anschluss an für jede Methode, die in der Class under test gefunden wurde, eine Testmethode in der Testklasse zu erstellen. Mit dem Häkchen bei „Generate comments“ kann dabei angegeben werden, ob die unten beschriebenen Javadoc Kommentare erzeugt werden sollen oder nicht.

Listing 5 zeigt das Ergebnis dieses Wizards, wenn auf der zweiten Seite lediglich die Methode `add(IMoney money)` ausgewählt wurde: Eine Testklasse (Zeile 7), eine Methode zur Erstellung der Fixture (Zeilen 9-11) und eine Testmethode für die Methode `add(IMoney)` (Zeilen 13-16). Die generierte Implementierung dieser Testmethode führt beim nächsten Lauf der Tests erst einmal zu einer „roten Ampel“. Dies ist aber auch ganz im Sinne der test-getriebenen Entwicklung.

```
1: package junit.samples.money;
2:
3: import static org.junit.Assert.*;
4: import org.junit.Before;
5: import org.junit.Test;
6:
7: public class MoneyTest {
8:
9:       @Before
10:      public void setUp() throws Exception {
11:      }
12:
13:      @Test
14:      public void testAdd() {
15:              fail("Not yet implemented");
16:      }
17:}
```

Listing 5: Generierter JUnit Test Case in Eclipse

Eine Verbindung zwischen Testmethoden und getesteten Methoden, die problemlos programmatisch ausgewertet werden kann, wird von Eclipse nicht bereitgestellt. Es kann lediglich ein Javadoc Kommentar erzeugt werden, der auf eine getestete Methode verweist. Dieser enthält nach der Generierung lediglich die Referenz zu einer getesteten Methode. Eine n : m Beziehung wird von Eclipse nicht vorgesehen. Über die Eclipse-API ist ein Zugriff auf den Javadoc Kommentar nur in Form eines String-Objekts möglich. Im allgemeinen lassen sich aus einer Testmethode die getesteten Methoden durch die Betrachtung der in der Methode aufgerufenen Methoden ermitteln. Durch eine Suche nach den Aufrufstellen der getesteten Methode sind umgekehrt die Testmethoden zu finden. Dabei muss allerdings beachtet werden, dass diese Suche nicht nur Testmethoden, sondern auch Aufrufe aus dem normalen Programmkontext zurück liefert.

2.3.1.2 Runtime und Visualisierung von JUnit Testfällen

Eclipse bietet verschieden Möglichkeiten, JUnit-Tests auszuführen. Neben einer Ausführung aller Tests in einer Testklasse über das Kontextmenü existiert die Möglichkeit, Ausführungskonfigurationen (*run configurations*) für JUnit zu definieren. Diese führen Tests in einer einzelnen Testklasse oder alle Tests, zum Beispiel in einem Paket oder einem Java Projekt, aus und können vom Benutzer wiederholt angestoßen werden.

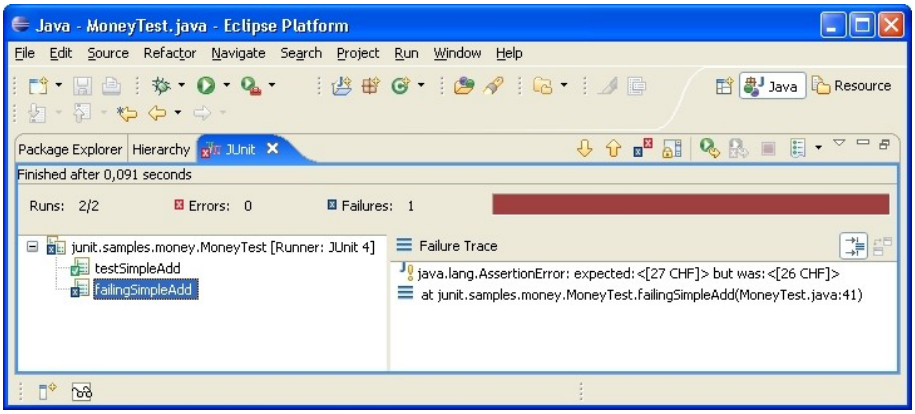

Abbildung 6: Eclipse Visualisierung von JUnit Testergebnissen

Die Ergebnisse der Ausführung eines JUnit-Tests durch Eclipse werden in einem eigenen JUnit View dargestellt. Abbildung 6 zeigt einen erfolgreichen und einen fehlgeschlagenen Test in diesem View. Schlägt ein Test fehl, wird die Aufrufkette (*stack trace*) zu dem Fehler ausgegeben. Einfache Fehlschläge und Ausnahmen werden getrennt ausgewertet. Handelt es sich um einen Fehlschlag, kann man aus der Aufrufkette in der Rechten Hälfte der Abbildung das erwartete (27 CHF) und das tatsächliche Ergebnis (26 CHF) sowie die Zeile des fehlgeschlagenen Vergleichs (Zeile 41) in der Testmethode entnehmen. Die Aufrufkette einer Ausnahme enthält die Art der Ausnahme sowie den Ort, an dem die Ausnahme aufgetreten ist. Dieser kann sowohl in der Testmethode als auch in einer (direkt oder indirekt) getesteten Methode liegen.

Durch einen Doppelklick auf einen Test kann dieser direkt im Java Editor geöffnet werden. Die gleiche Aktion auf die zweite Zeile von oben der Aufrufkette (in diesem Fall die Zeile mit dem Text „MoneyTest.java:41) des Fehlers öffnet die Klasse und springt direkt zu der Zeile, in der der Vergleich von erwartetem und tatsächlichen Wert fehlgeschlagen ist beziehungsweise in der ein Fehler geworfen wurde.

Wenn nach einem Testlauf die Fehler repariert wurden, können entweder alle fehlgeschlagenen Tests oder aber auch alle Tests nochmals durchgeführt werden.

2.3.1.3 Extension point „Test Run Listeners“

Neben der Unterstützung für die Erstellung und Ausführung von JUnit Tests bietet Eclipse auch einen Erweiterungspunkt (*extension point*) für die Registrierung zusätzlicher *Test Run Listener*. Plugins, die sich an diesen Erweiterungspunkt anhängen, werden über die Ausführung von JUnit-Tests informiert. Die Information umfasst dabei Benachrichtigungen über den Start und die Beendigung einzelner Tests und Testläufe und über fehlgeschlagene Tests.

3 Markierung von logischen Fehlern im Quellcode

In den bisherigen Kapiteln lag der Fokus auf den existierenden Möglichkeiten von Eclipse, auf JUnit und auf Tools im JUnit-Umfeld. Dieses Kapitel stellt das Konzept für die Markierung von logischen Fehlern im Quellcode vor, das im Rahmen dieser Arbeit in Form eines Eclipse-Plugins realisiert wurde. Die Implementierung des Konzepts ist in Kapitel 4 beschrieben.

3.1 Verknüpfung von getesteten und Testmethoden

Grundsätzlich lassen sich zwei mögliche Ursachen für das Fehlschlagen eines Testfalls unterscheiden:

- Der Testfall ist fehlerhaft definiert.
- Eine der getesteten Methoden funktioniert nicht korrekt.

Mit dem richtigen Vorgehen lässt sich die Wahrscheinlichkeit, dass die Testfälle korrekt definiert sind, sehr hoch halten: die Tests müssen immer verbessert werden, wenn ein Fehler gefunden wurde, und der Einbau eine Fehlers kann benutzt werden, um einen Test auf die Probe zu stellen.[8] Daher wird im weiteren Verlauf der Arbeit davon ausgegangen, dass bei einem fehlgeschlagenen Testfall eine Ursache der zweiten Kategorie vorliegt. Daher soll auch dort der Fehler gesucht werden und dem entsprechend ist das Ziel, den Fehler auch dort zu visualisieren.

Primär liegt bei einem fehlgeschlagenen Test allerdings erst einmal die Information vor, welche Testmethode nicht erfolgreich durchlaufen wurde. Um zu der eigentlichen Ursache zu kommen, muss daher eine Verknüpfung zwischen der Testmethode und der oder den getesteten Methoden vorhanden sein. Diese Verknüpfung kann entweder automatisch aus der Testmethode ermittelt oder aber manuell definiert werden.

3.1.1 Automatische Verknüpfung

Für die automatische Ermittlung der Verknüpfung von Testmethoden und zu testenden Methoden kommen wiederum verschiedene Möglichkeiten in Betracht: die dynamische Verknüpfung zur Laufzeit auf Basis des *Stack Traces* eines fehlgeschlagenen Tests, die dynamische Analyse auf Basis der tatsächlich aufgerufenen Methoden eines Tests und die statische Analyse des Aufrufbaums einer Methode.

3.1.1.1 Dynamische Analyse

Bei der dynamischen Analyse auf Basis des *Stack Traces* des Testfalls und des Quellcodes der Testmethode wird aus ersterem die Zeile der Testmethode extrahiert, in der der Testfall fehlgeschlagen ist. In Abbildung 7 ist dies die Zeile 36 in der Datei MoneyTest.java. Listing

8 Siehe [Hunt & Thomas 2004], Seiten 96 bis 99

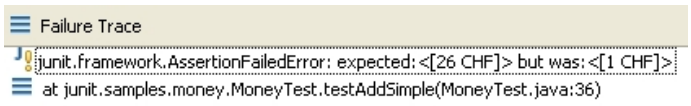

Abbildung 7: Stack Trace eines fehlgeschlagenen Testfalls

6 stellt einen Ausschnitt dieser Klasse dar. Die Zeile 36 in der Klasse entspricht der Zeile 5 im Listing. Betrachtet man diese Zeile 5, erkennt man, dass in dieser Zeile die Methode `add(IMoney)` des Objekts `f12CHF` überprüft wird.

```
1: @Test
2: public void testSimpleAdd() {
3:         // [12 CHF] + [14 CHF] == [26 CHF]
4:         Money expected= new Money(26, "CHF");
5:         assertEquals(expected, f12CHF.add(f14CHF));
6: }
```

Listing 6: Testmethode zur Überprüfung einer Additionsfunktion

Dem entsprechend wird diese Methode `add(IMoney)` in der Klasse `Money` als die potentiell fehlerverursachende Methode betrachtet. An ihr müsste die Visualisierung für diesen Fehler festgemacht werden.

Diese vereinfachte Sicht lässt allerdings einige Aspekte außer acht. Daher ist eine reine Analyse über den *Stack Trace* eher schlecht geeignet. Grundsätzlich wird in einer Testmethode nicht nur eine Methode getestet. In Listing 6 wird zum Beispiel auch der Konstruktor `Money(int, String)` mit überprüft, der somit ebenfalls die Ursache für den Fehler beinhalten könnte. Darüber hinaus werden auch in diesen Methoden wieder weitere Methoden wie zum Beispiel die Methoden `addMoney(Money)` der Klasse `Money` aufgerufen, die für die Suche nach der Ursache des Fehlers nicht außer Acht gelassen werden dürfen. Ein möglicher Ansatz, dieser Tatsache Rechnung zu tragen, ist das *Profiling* der Testausführung zum Beispiel mit Hilfe des *Java Virtual Machine Tool Interface (JVM TI)*. Dieses bietet Benachrichtigungsmechanismen, die es erlauben eine Liste aller Methoden zu erstellen, die für die Durchführung eines Testfalls tatsächlich aufgerufen wurden.[9]

Dieses *Profiling* ist allerdings aufwendig und führt damit zu erhöhten Laufzeiten der Unit-Tests. Dadurch sinkt die Akzeptanz für eine häufige Durchführung dieser Tests durch den Entwickler, wie durch den Extreme Programming Ansatz gefordert. Damit existiert zumindest solange ein gewichtiges Argument gegen den Einsatz dieser Methodik für die Ermittlung von potentiellen Fehlerquellen, bis leistungsstärkere Rechner auch eine schnelle Testdurchführung bei aktiviertem *Profiling* erlauben.

9 Vgl. [JVMTI 2005], Abschnitt MethodEntry

3.1.1.2 Statische Analyse

Die statische Analyse der Aufrufbaums setzt auf die Untersuchung des Quellcodes auf. Sie muss dem entsprechend nicht parallel mit der Durchführung der Tests ablaufen und hat damit auch keinen Einfluss auf die (kurze) Ausführungszeit der Testmethoden. Bei dieser statischen Analyse wird jeder einzelne Ausdruck der Testmethode untersucht. Die einzelnen Ausdrücke werden weiter zerlegt, bis eine Liste der tatsächlich aufgerufenen Methoden vorliegt. Alle Methoden in dieser Liste werden als potentiell fehlerverursachend betrachtet.

Um auch hier Methoden zu kennzeichnen, die indirekt von der Testmethode aufgerufen werden, müssen alle in den Ausdrücken der Testmethode gefundenen Methoden auf die selbe Art und Weise untersucht werden. Dieser Vorgang wiederholt sich rekursiv. Wichtig ist dabei das Mitführen einer Liste der bereits untersuchten Methoden, um Endlosschleifen zu vermeiden.

Probleme ergeben sich bei dieser Analyse durch Vererbungsstrukturen und überschriebene Methoden. Die in Listing 6 in Zeile 22 aufgerufene Methode `Money.add(IMoney)` beinhaltet für sich betrachtet einen Aufruf der Methode `IMoney.addMoney(Money)`. Da an die Methode `Money.add(IMoney)` aber ein Objekt vom Typ `Money` übergeben wird, wird zur Laufzeit des Tests die konkrete Implementierung der Methode `addMoney(Money)` der Klasse `Money` aufgerufen. Damit können auch Methoden, die aus dieser Methode `Money.addMoney(Money)` aufgerufen werden, ein Fehlschlagen des Testfalls verursachen. Eine reine Betrachtung der deklarierten Methodenaufrufe ist bei der statischen Analyse somit nicht ausreichend. Eine Analyse der Belegung von Variablen und damit eine Ermittlung der tatsächlich aufgerufenen Methoden ist mit vertretbarem Aufwand nur zur Laufzeit möglich. Daher beschränkt sich die statische Analyse darauf, alle Methoden als potentiell fehlerverursachend zu betrachten, die in einer solchen von der Testmethode ausgehenden Aufrufstruktur direkt deklariert sind beziehungsweise die deklarierten Methoden überschreiben. Dies verlangt eine Möglichkeit, die bekannten überschreibenden Methoden zu ermitteln.

Durch diese Vorgehensweise und durch die wiederholte Verwendung von gleichen Methoden in fast allen Testmethoden für den Vergleich von erwarteten und tatsächlichen Werten ergeben sich für eine Testmethode eine sehr große Anzahl an potentiell fehlerverursachenden Methoden, die mit sehr unterschiedlichen Wahrscheinlichkeiten wirklich die Ursache für ein Fehlschlagen eines Tests sind. Eine der ermittelten Methoden wäre in dem Beispiel aus Listing 6 auch die Methode `assertEquals(Object, Object)` in Zeile 5, die mit sehr hoher Wahrscheinlichkeit nicht für den Fehlschlag des Tests verantwortlich ist.

Um den Entwickler nicht mit einer Vielzahl von falschen Fährten zu verwirren, ist es wichtig, nicht relevante potentielle Fehlerursachen auszublenden. So können Filter zum Einsatz kommen, um bestimmte Klassen oder Methoden für die Ermittlung der Fehlerursache ein- beziehungsweise auszuschließen. Werden solche Listen allerdings global für ein Projekt gepflegt, ist es nicht möglich, in einer Testmethode Methoden auszuschließen, die gezielt in anderen Methoden getestet werden.

Eine explizite Verknüpfung von Testmethoden und getesteten Methoden durch den Entwickler kann die Anzahl an falschen Fährten bei der Fehlersuche hingegen recht einfach reduzieren.

3.1.2 Die Annotation "Method under Test"

Um die Beziehung zwischen einer Testmethode und den von ihr getesteten Methoden explizit auszudrücken, wird im Rahmen dieser Arbeit die *Method under Test* Annotation (im weiteren Text als @MUT Annotation abgekürzt) definiert. Sie wird an die Testmethode gehängt und enthält eine Liste der voll qualifizierten Methodennamen, die durch die markierte Methode getestet werden.

Die bewusst getesteten Methoden, deren Beziehung zur Testmethode explizit gemacht werden soll, sind, sofern sie und die Testmethode bereits komplett ausprogrammiert sind, immer eine Untermenge aller Methoden, die direkt oder indirekt aufgerufen werden. Diese können über die oben aufgeführten Methoden zur statischen, automatischen Verknüpfung ermittelt werden, insofern es sich nicht um Methodenaufrufe über die Reflection-API handelt.[10] Damit ist es möglich, dem Benutzer direkt eine Liste der aufgerufenen Methoden zur Verfügung zu stellen, aus denen er dann explizit die Methoden auswählen kann, die der @MUT Annotation hinzugefügt werden sollen. Dadurch werden Tippfehler in der Eingabe vermieden und die Benutzung für den Programmierer vereinfacht.

Da die @MUT Annotation manuell gepflegt wird, ist allerdings die Gefahr vorhanden, dass die tatsächlich in der Testmethode aufgerufenen Methoden und die annotierten Methoden auseinander laufen. Eine mögliche Lösung wird im Kapitel 6.5 skizziert.

3.2 Visualisierung von fehlgeschlagenen Tests

Fehlgeschlagene Testfälle sind genauso Hinweise auf semantische Fehler im Quellcode wie Typinkompatibilitäten bei Zuweisungen, die vom Compiler entdeckt werden. Daher ist eine ähnliche Visualisierung nur konsequent.

10 Eine Erklärung von Methodenaufrufen über die Reflection-API findet sich in [Ullenboom 2007] im Kapitel 21.4

Eclipse stellt semantische Fehler sowohl im *Problems View*, im *Package Explorer* und im *Outline View* als auch direkt im Editor an der Stelle der fehlerhaften Zuweisung dar. Abbildung 8 zeigt einen solchen Fehler.

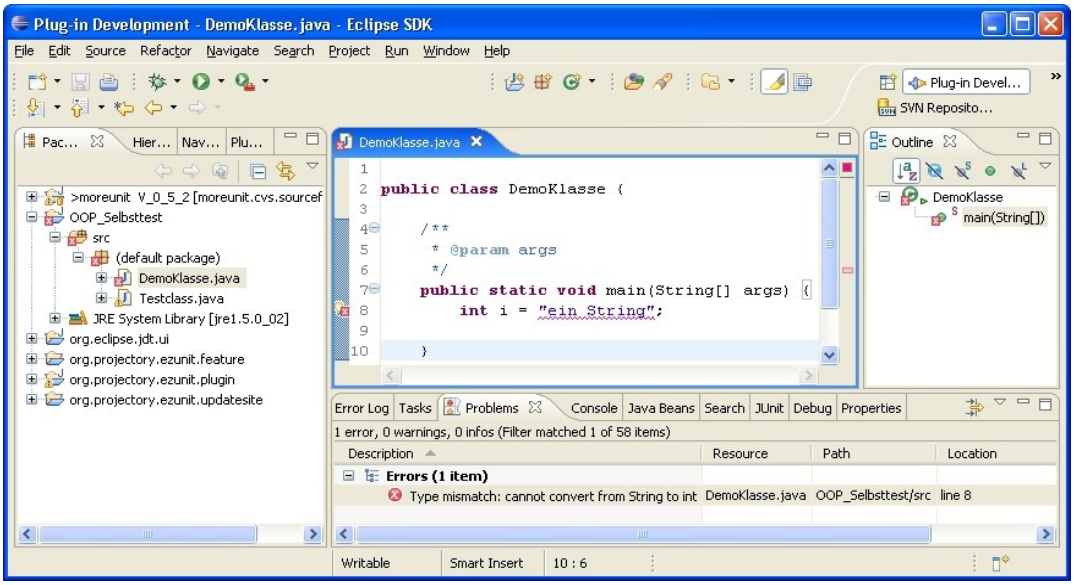

Abbildung 8: Visualisierung von semantischen Fehlern in Eclipse

Durch die Definition eines eigenen Markers[11] für fehlgeschlagene Testfälle ist es in Eclipse möglich, sich in diese Art der Visualisierung einzuhängen. Die Implementierung der dazu nötigen Erweiterung wird im Kapitel 4.10, „Testrunlistener und Marker“, beschrieben.

Die Erzeugung eines Markers zu Kennzeichnung eines fehlgeschlagenen Testfalls ist nicht neu. In [Beck & Gamma 2004] wird ein Marker für jede Testmethode erzeugt, die nicht erfolgreich ausgeführt wurde, und auch an dieser befestigt. Damit wird aber in der Regel nicht der eigentliche Fehler gekennzeichnet, sondern ein Symptom des Fehlers. Durch die Verknüpfung der Testmethoden mit denen durch sie getesteten Methoden ist es hingegen möglich, die Visualisierung näher an den tatsächlichen Fehler zu rücken: der Marker wird statt an der Testmethode an der getesteten Methode befestigt. Diese lassen sich über die im Kapitel 3.1.2 beschriebe Annotation ermitteln. Dem Marker wird dabei eine Beschreibung hinzugefügt, die die Information enthält, welche Testmethode fehlgeschlagen ist.

11 Eine Erklärung von Eclipse Markern findet sich zum Beispiel auf Seite 106 in [Bach 2007] oder auf Seite 130 in [Beck & Gamma 2004]

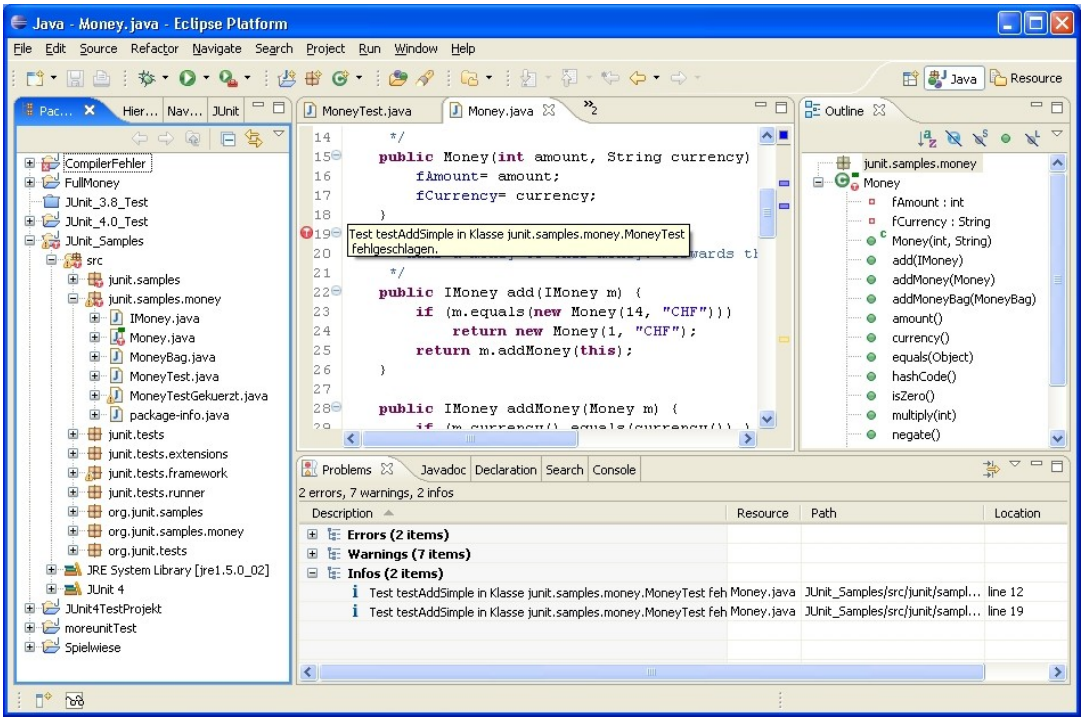

Abbildung 9: Markierung einer nicht erfolgreich getesteten Methode

Abbildung 9 zeigt die Visualisierung eines fehlgeschlagenen Testfalls direkt an der getesteten Methode mit Hilfe des im Rahmen dieser Arbeit entwickelten Plugins. Im Unterschied zu Compilerfehlern werden fehlgeschlagene Testfälle mit einem kleinen „T"-Symbol markiert.

3.3 Navigation zwischen getesteten und Testmethoden

Ist ein Test fehlgeschlagen, beginnt die Suche nach der Ursache. Dabei ist die getestete Methode der richtige Ausgangspunkt, da die Ursache in der Regel dort zu finden sein wird. Es kann allerdings helfen, den Aufrufkontext der nicht erfolgreich getesteten Methode zu betrachten. Dieser wird aus der Testmethode ersichtlich. Eine Unterstützung für die schnelle Navigation zwischen getesteter und Testmethode ist somit hilfreich und kann über eine *Marker Resolution*[12] realisiert werden. Abbildung 10 zeigt die *Marker Resolution*, die nach einem Klick auf das „T"-Symbol zur Verfügung steht, um direkt zu dem fehlgeschlagenen Testfall zu springen. Die Methode, auf die der Test fehlgeschlagen ist, ist blau markiert.

Diese Navigation zwischen getesteter und Testmethode ist aber nicht nur in diese Richtung und nicht nur nach einem fehlgeschlagenen Testfall interessant. Über die Verknüpfungen, die durch die in Kapitel 3.1.2 beschriebene Annotation beschrieben werden, ist zusätzlich auch eine dauerhafte Navigationsmöglichkeit zwischen getesteten und Testmethoden

12 Eine Erklärung des Begriffes *Marker Resolution* findet sich auf Seite 108 in [Bach 2007] und auf Seite 185 in [Beck & Gamma 2004]

Abbildung 10: Navigation zwischen nicht erfolgreich getester Methode und Testmethode

implementiert. Im Gegensatz zu anderen Werkzeugen, die auch eine entsprechende Navigation anbieten (wie zum Beispiel das im Abschnitt 5.2.3, „moreUnit“, beschriebene Eclipse-Plugin), kann mit einer Navigation auf der Basis der @MUT Annotation sowohl von einer Testmethode zu allen von ihr getesteten Methoden als auch von einer getesteten Methode zu all ihren Testmethoden navigiert werden – ein Vorteil der abgebildeten n : m Beziehung.

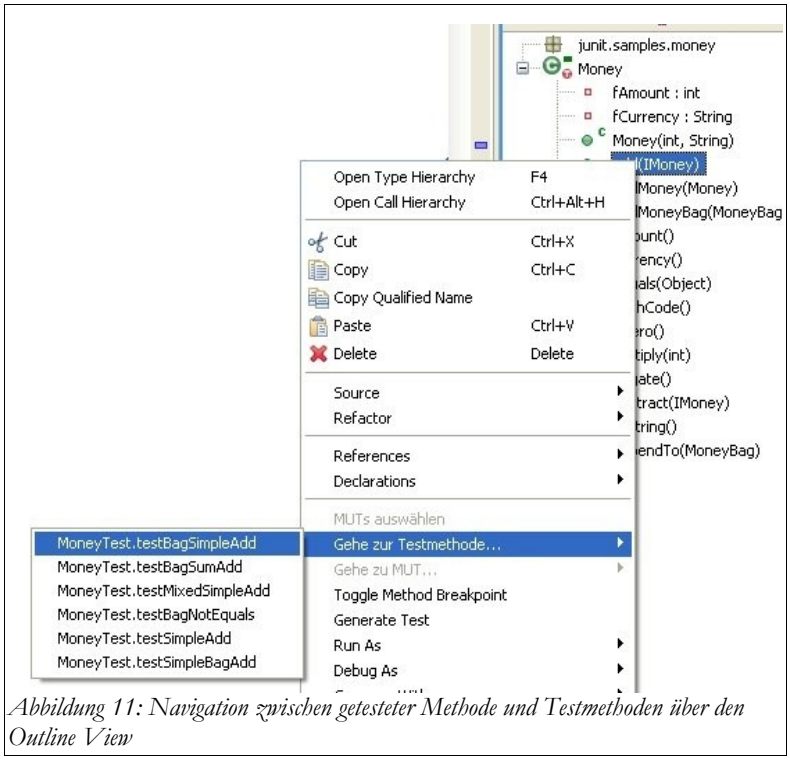

Abbildung 11: Navigation zwischen getesteter Methode und Testmethoden über den Outline View

Das im Rahmen dieser Arbeit entwickelte Plugin realisiert diese Navigation über die beiden in Abbildung 11 sichtbaren Menüpunkte „Gehe zur Testmethode...“ und „Gehe zur MUT...“. Diese beiden Menüpunkte werden wahlweise auf Basis der @MUT Annotationen oder der statisch ermittelten Methodenaufrufe der Testmethoden befüllt.

3.4 Voreinstellungen des Plugins

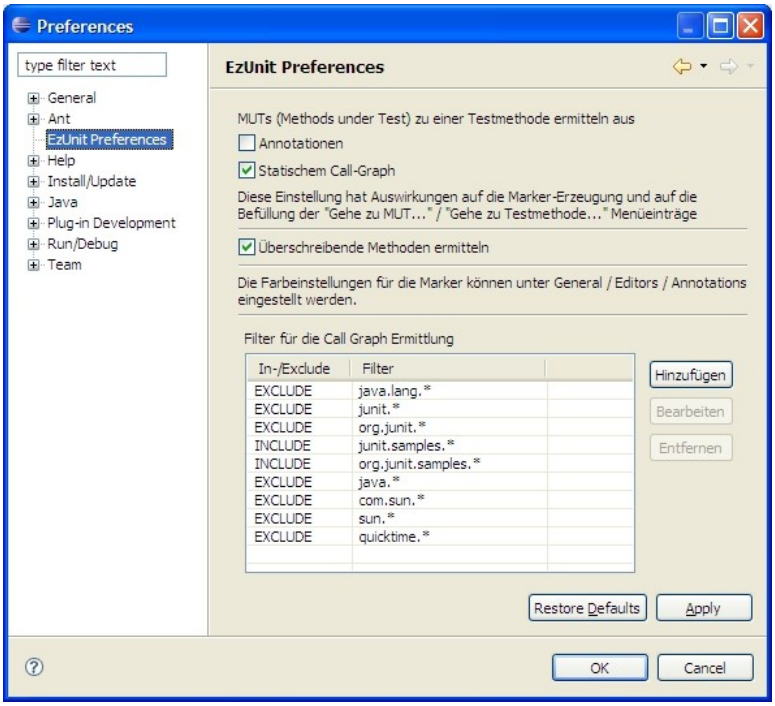

Abbildung 12: Die Voreinstellungen des Plugins

Um zwischen den verschiedenen Varianten für die Visualisierung der logischen Fehler im Quellcode auswählen zu können, wird von dem im Rahmen dieser Arbeit entwickelten Plugin eine Seite mit Voreinstellungen mitgeliefert. Diese ist in Abbildung 12 zu sehen.

Dort kann ausgewählt werden, welche Mechanismen verwendet werden sollen, um die getesteten Methoden zu ermitteln. Zur Auswahl stehen dort momentan die Ermittlung auf Basis der @MUT Annotation und auf Basis des statischen Aufrufbaums. Diese Auswahl hat sowohl Auswirkungen auf die Markierung, als auch auf die Navigation zwischen Testmethoden und getesteten Methoden. Darüber hinaus kann das (zeitaufwendige) untersuchen der überschreibenden Methoden aktiviert beziehungsweise deaktiviert werden.

Durch das Angeben von Filtern für die Einbindung beziehungsweise Ignorierung von Methoden oder Klassen bei der Untersuchung des Aufrufbaums kann diese beschleunigt werden. Zweitens existiert damit eine Möglichkeit, als korrekt betrachtete Klassen, wie zum Beispiel Java-Bibliotheken aus der Untersuchung auszuschließen und damit die Liste der potentiellen Fehlerquellen übersichtlich zu halten. Die angegebenen Filterkriterien werden von oben nach unten ausgewertet.

4 Das Plugin

Während das Kapitel 3 die Ideen und ihre Umsetzung im Rahmen dieser Arbeit auf der konzeptionellen Ebene beschreibt, ist der Inhalt dieses Kapitels die Erläuterung der tatsächlich entwickelten Bausteine, um die im vorigen Kapitel vorgestellten Funktionalitäten anbieten zu können.

In diesem Kapitel werden nur die wichtigsten Quellcode-Ausschnitte aufgegriffen werden, die nötig sind, um einzelne Aspekte zu erklären. Für den kompletten Quellcode sei an dieser Stelle auf die entsprechenden Eclipse-Projekte für Plugin, Feature und Update-Site verwiesen. Diese sind auf der dieser Arbeit beiliegenden CD zu finden.

4.1 Packages des Plugins

Das Plugin ist nach Funktionalitäten in verschiedene Java-Packages aufgeteilt. Jedes Package beginnt dabei mit dem Prefix `org.projectory.ezunit`. Klassen, die für andere Plugins nicht sichtbar sein sollen, tragen den Prefix `org.projectory.ezunit.internal`. Die Funktionalitäten, die durch die Klassen in den Paketen erfüllt werden, sind in der folgenden Tabelle beschrieben.

Paketname	***Beschreibung***
org.projectory.ezunit	Enthält die Plugin-Hauptklasse sowie die Klassen, die für andere Plugins zur Verfügung stehen sollen.
org.projectory.ezunit.internal	Basispaket für alle internen Klassen; beinhaltet selber Klassen, die von verschiedenen Unterpaketen benötigt werden.
org.projectory.ezunit.internal.buildPath	Enthält die Klassen, die für das Bereitstellen der @MUT Annotation im Erstellungspfad für andere Projekte nötig sind.
org.projectory.ezunit.internal.completion	Enthält den *Completion Proposal Computer*, der für die Ermittlung der Code-Assists innerhalb der @MUT Annotation zuständig ist
org.projectory.ezunit.internal.contribution	Enthält die Klassen, die für das Bereitstellen der Popup-Menüpunkte und der dahinter stehende Dialoge nötig sind.
org.projectory.ezunit.internal.decorator	Enthält die Klasse für die Markierung von fehlgeschlagenen Testfällen im Package Explorer.
org.projectory.ezunit.internal.failuremarker	Enthält die Klassen, die bei einem JUnit-Testlauf auftretende Test-Fehlschläge in entsprechende Markierungen an den getesteten Methoden umsetzt.
org.projectory.ezunit.internal.icons	Enthält intern benötigte Icons.
org.projectory.ezunit.internal.methodwrapper	Enthält Klassen, die den Zugriff auf die MUT Annotation, die Testmethoden einer Methode beziehungsweise die getesteten Methoden ermöglichen.
org.projectory.ezunit.internal.multilanguage	Enthält die Klassen für die Internationalisierung des Plugins
org.projectory.ezunit.internal.preferences	Enthält die Klassen, die für die Bearbeitung der Voreinstellungen dieses Plugins nötig sind.

4.2 Die MUT Annotation

Kern der Verknüpfung von getesteten und Testmethoden ist die „Method under Test“ Annotation, kurz *@MUT Annotation*. Sie ist im Paket `org.projectory.ezunit` wie in Listing 7 ersichtlich definiert:

```
1: public @interface MUT {
2:         String[] methods();
3: }
```

Listing 7: Die @MUT Annotation

Das String-Array `methods` in dieser Annotation enthält die voll qualifizierten Methodennamen, die durch die annotierte Testmethode überprüft werden und die sich wie folgt zusammensetzen:

`<Paket>.<Klasse>.<Methode>(<Typ 1>, ..., <Typ n>)`

- <Paket> ist der voll qualifizierte Name des Pakets, in dem die Klasse zu finden ist, in der eine Methode getestet wird.
  Beispiel: `junit.samples.money`
- <Klasse> ist der Name der Klasse, in der sich die getestete Methode befindet.
  Beispiel: `Money`
- <Methode> ist der Name der Methode, die getestet wird.
  Beispiel: `add`
- <Typ 1>, ..., <Typ n> sind die Typen der Parameter der getesteten Methode. Die Angabe der Typen wird in der Form erwartet, wie sie von der Methode `org.eclipse.jdt.core.IMethod.getParameterTypes()` zurück gegeben wird[13]
  Beispiel: `QIMoney;`

Somit würde sich für eine Testmethode `testAddSimple`, die auf ein Objekt vom Typ `Money` die Methode `add` mit einem Parameter vom Typ `IMoney` aufruft, der in Listing 8 gezeigte Methodenkopf ergeben:

Um die Annotation in der in diesem Listing gezeigten Form nutzen zu können, muss die Klasse eine passende Import-Anweisung beinhalten:

13 Erläuterungen zu dieser internen Form finden sich im Anhang „Methodensignaturen“

```
1: @MUT (
2:        methods = {
3:           "junit.samples.money.Money.add(QIMoney;)"
4:        }
5: )
6: @Test
7: public void testSimpleAdd() {
8:        ...
9: }
```

Listing 8: Mit @MUT Annotation versehender Methodenkopf

```
import org.projectory.ezunit.MUT;
```

Alternativ kann wie in Listing 9 bei der Annotation der Methode `testAddSimple` auch der voll qualifizierte Name der @MUT Annotation verwendet werden.

```
10:@org.projectory.ezunit.MUT (
11:       methods = {
12:          "junit.samples.money.Money.add(QIMoney;)"
13:       }
14:)
15:@Test
16:public void testSimpleAdd() {
17:       ...
18:}
```

Listing 9: Mit voll qualifizierter @MUT Annotation versehender Methodenkopf

4.3 Der MUT Classpath Container

Um die @MUT Annotation auflösen zu können, muss diese im Erstellungspfad des Eclipse-Java-Projekts, in dem die Testmethode liegt, vorhanden sein. Dem Eclipse-Projekt muss also ein *Classpath Entry* für die JAR-Bibliothek hinzugefügt werden, der die Annotation enthält. Die verschiedenen Typen von *Classpath Entries* sind im Anhang A1 beschrieben.

Für eine Einbindung ohne das Hantieren mit Pfaden zu JAR-Dateien innerhalb des Eclipse-Plugins stehen zwei Typen von *Classpath Entries* zur Auswahl[14]: *Variable Classpath Entries* und *Classpath Container Entries*. Obwohl ersterer Typ für die Einbindung einer einzelnen JAR-Bibliothek ausreichend wäre[15], kommt in diesem Plugin ein *Classpath Container* zum Einsatz. Die Nutzung von *Classpath Containern* ist in Eclipse weiter verbreitet und kommt auch für das Hinzufügen der Bibliotheken für JUnit zum Einsatz. Daher ist durch die Erstellung eines *Classpath Containers* eine konsistente Bedienung gewährleistet, die bei

14 Das Kapitel „Classpath Repräsentation in Eclipse“ im Anhang stellt die unterschiedlichen *Classpath Entries* dar.

15 Das Plugin enthält in der Klasse `org.projectory.ezunit.buildPath.MUTClasspathVariableInitializer` auch das Beispiel, das in den Listings 36 und 37 beschrieben wird

Interesse sogar in der Form erweitert werden kann, dass der MUT-Container neben der Annotation auch gleich die JUnit *Classpath Entries* mitbringen kann – alternativ wäre natürlich auch denkbar die Annotation zum Junit-Classpath-Container hinzuzufügen.

Ein *Classpath Container* kann einem Java-Projekt über die Erstellungspfad-Seite des Eigenschaftsdialogs des Projekts hinzugefügt werden. Abbildung 13 zeigt das Hinzufügen des *Containers* MUT, der von diesem Plugin bereit gestellt wird. Was für diese Bereitstellung konkret benötigt wird, ist im folgenden beschrieben.

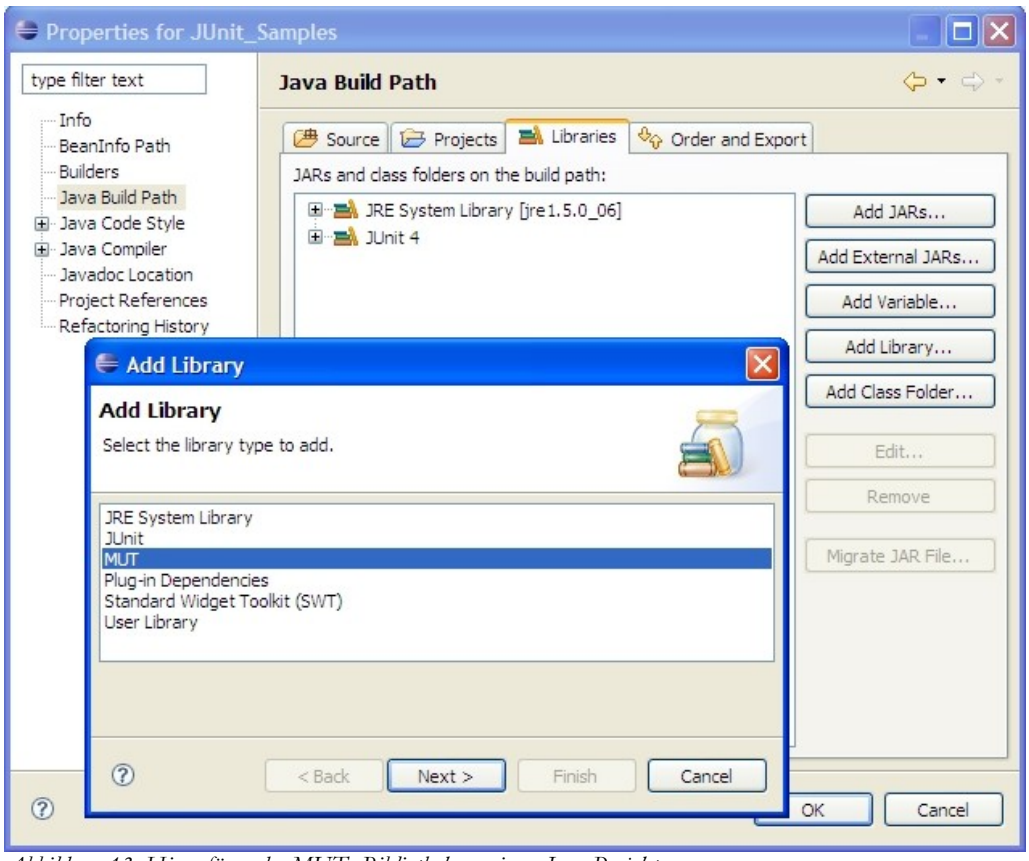

Abbildung 13: Hinzufügen der MUT- Bibliothek zu einem Java Projekt

Die Definition eines *Classpath Containers* setzt sich aus zwei Teilen zusammen: der Implementierung der Erweiterungspunkte

- `org.eclipse.jdt.ui.classpathContainerPage` und
- `org.eclipse.jdt.core.classpathContainerInitializer`

Listing 10 zeigt die Definition der *Classpath Container Page* für die Bibliothek MUT. In den Zeilen 1 und 2 ist angegeben, dass es sich um eine Erweiterung für den Erweiterungspunkt `org.eclipse.jdt.ui.classpathContainerPage` handelt. In Zeile 7 wird der Name angegeben, der in der Liste aus Abbildung 13 zu sehen ist. Die Zeilen 4 und 5 ent-

```
1: <extension point=
2:       "org.eclipse.jdt.ui.classpathContainerPage">
3:    <classpathContainerPage
4:       class="org.projectory.ezunit.internal.
5:              BuildPath.MUTClasspathContainerPage"
6:       id="org.projectory.ezunit.MUT_CONTAINER"
7:       name="MUT"/>
8:  </extension>
```

Listing 10: Defintion der Classpath Container Page für die Bibliothek MUT

halten den voll qualifizierten Namen einer Klasse, die das Interface `org.eclipse.jdt.ui.wizards.IClasspathContainerPage` implementiert. Sie stellt eine Seite für den Wizard aus Abbildung 13 bereit, auf der weitere Informationen zu dem hinzugefügten *Container* eingegeben werden können. Für den von Eclipse bereitgestellten *Container* „JRE System Library“ wird auf dieser Seite zum Beispiel die *Java Runtime Environment* ausgewählt, deren Bibliotheken dem Erstellungspfad des Projekts hinzugefügt werden sollen. Da für die @MUT Annotation keine weiteren Einstellungen nötig sind, fällt diese Seite, wie in Abbildung 14 ersichtlich, optisch sehr einfach aus.

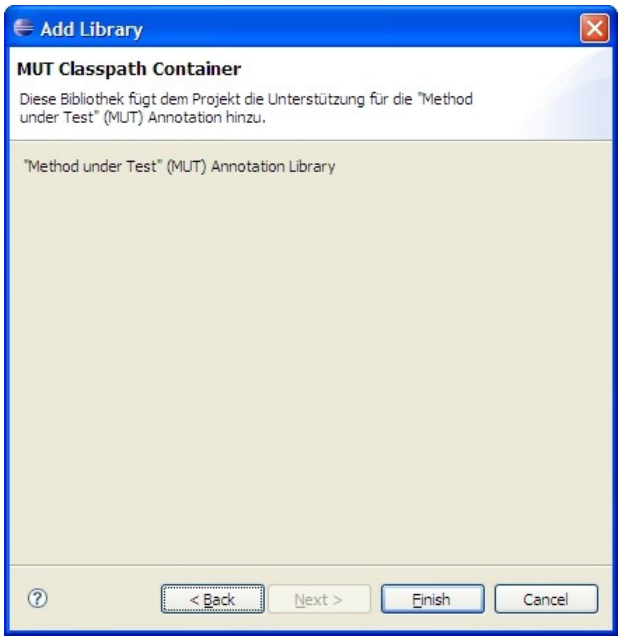

Abbildung 14: Wizard Seite für das Hinzufügen des Bibliothek für die @MUT Annotation

In der Zeilen 6 findet sich letztendlich die ID des *Containers*, der dem Projekt durch diese Wizard-Seite hinzugefügt werden soll. Diese ID muss identisch mit der ID sein, die für den `classpathContainerInitializer` angegeben ist, der weiter unten beschrieben wird.

Da die Klasse `MUTClassPathContainerPage` als zweites Interface auch noch `org.eclipse.core.runtime.IExecutableExtension` implementiert, kann die so angegebene ID in der Methode `setInitializationData(...)` ausgelesen werden. Auf Basis dieser ID kann dann ein `org.eclipse.jdt.core.IClasspathEntry` erzeugt werden, der von der Methode `getSelection()` als Ergebnis der Wizard-Seite zurück gegeben wird. Der übliche Typ für solch eine Wizard-Seite ist ein *Container Classpath Entry*.

Die tatsächliche Auflösung dieses `IClasspathEntry` Objekts in eine Menge von JAR-Bibliotheken, Quellcode-Ordnern und Projektreferenzen wird von der Klasse durchgeführt, die in der Erweiterung des Erweiterungspunkts `org.eclipse.jdt.core.ClasspathContainerInitializer` angegeben ist. In Listing 11 wird so die Klasse `MUTClasspathContainerInitializer` als die Klasse angegeben, die die Auflösung für *Classpath Container Entries* mit der ID `org.projectory.ezunit.MUT_CONTAINER` übernimmt.

```
1: <extension point="org.eclipse.jdt.core.
2:                   classpathContainerInitializer">
3:    <classpathContainerInitializer
4:      class="org.projectory.ezunit.internal.
5:        buildPath.MUTClasspathContainerInitializer"
6:      id="org.projectory.ezunit.MUT_CONTAINER"/>
7: </extension>
```

Listing 11: Defintion der Classpath Container Initializers für die Bibliothek MUT

Die so spezifizierte Klassse muss dabei eine Subklasse der Klasse `org.eclipse.jdt.core.ClasspathContainerInitializer` sein und deren abstrakte Methode `initialize(...)` implementieren. Aufgabe dieser Methode ist es, dem übergebenen `IJavaProject` einen `IClasspathContainer` hinzuzufügen. Konkret wird dem übergebenen Projekt von der Klasse `MUTClasspathContainerInitializer` eine Instanz der internen Klasse `MUTContainer` hinzugefügt, die im wesentlichen nur den Namen des *Classpath Containers* und einen *Library Classpath Entry* zurück gibt, der die JAR-Bibiothek `lib/MUT.jar` im Plugin referenziert. Da die weitere Implementierung dieser Klasse selbst erklärend ist, soll an dieser Stelle nicht näher darauf eingegangen werden.

Über den so hinzugefügten *Classpath Container* steht jetzt allen Java Klassen in diesem Projekt die Annotation `@MUT` zur Verfügung. Damit kann sie verwendet werden, um Testmethoden mit denen von ihr getesteten Methoden zu verknüpfen.

4.4 Ermittlung der getesteten Methoden

Für die Ermittlung der getesteten Methoden aus einer Testmethode stehen im Plugin drei mögliche Varianten zur Verfügung:

- die Untersuchung des statischen Aufrufbaums über den AST der Testmethode,

- die Untersuchung der @MUT Annotation über den AST der Testmethode sowie
- die Untersuchung der @MUT Annotation mit Hilfe einer *Regular Expression*, die auf den Quellcode der Testmethode angewendet wird.

Andere Varianten als die erste Variante der automatischen Ermittlung von getesteten Methoden ohne @MUT Annotation, die im Kapitel 3.1, „Verknüpfung von getesteten und Testmethoden“, beschrieben sind, sind in dem Plugin nicht implementiert.

Während bei der ersten Variante eine explizite Verknüpfung der Testmethoden mit den getesteten Methoden nicht nötig ist, setzen die beiden anderen Varianten eine vorhandene @MUT Annotation voraus.

Die erste Variante kann daher zum Beispiel zum Einsatz kommen, um den Entwickler beim Anlegen der @MUT Annotation zu unterstützen, oder aber auch um Fehlermarkierungen ohne explizite @MUT Annotation an Methoden anbringen zu können, auf die ein Testfall fehlgeschlagen ist.

Die Implementierung der ersten Variante befindet sich in der Klasse `org.projectory.ezunit.internal.methodwrapper.TestIMethodWrapper`[16] in der Methode `extractTestedMethodsFromBody()`. Diese Methoden greift für die Untersuchung der Struktur der Testmethode auf den von Eclipse erstellten *Abstract Syntax Tree* (im weiteren als AST bezeichnet) der Testmethode zurück. Ein AST stellt dabei das Ergebnis des Parsens und Analysierens einer Kompilierungseinheit dar.[17] Dieser kann mit Hilfe einer Kindklasse der abstrakten Klasse `org.eclipse.jdt.core.dom.ASTVisitor` untersucht werden. Dies ist eine Anwendung des Visitor-Patterns, das in [Gamma et. al 1997] auf den Seiten 331 bis 344 näher beschrieben ist. Für Details zum AST und zum ASTVisitor sei an dieser Stelle auf [Beck & Gamma 2004], Seite 319 bis 323 und auf [Bach 2007], Seite 104 bis 106 verwiesen.

In dieser Methode kommt eine Instanz der Klasse `org.projectory.ezunit.internal.methodwrapper.CalledMethodExtractorASTVisitor` zum Einsatz. Dieser *Visitor* sammelt in zwei Vektoren alle aufgerufenen Methoden und alle untersuchten Methoden. Die aufgerufenen Methoden sind dabei eine Untermenge der untersuchten Methoden, die bereits um die Methoden bereinigt wurden, die durch entsprechende Filter in den Voreinstellungen des Plugins (siehe dazu Kapitel 4.11.3) von der weiteren Untersu-

16 Wrapper ist hier nicht als Synonym für einen Adapter im Sinne des Adapter-Design-Pattern wie in [Gamma et. al 1997] auf Seite 139 beschrieben zu verstehen. Die Klasse vermittelt nicht zwischen zwei inkompatiblen Interfaces, sondern kapselt Funktionalitäten, die den Testmethoden spezifischen Zugriff auf ein IMethod Objekt erleichtern. Damit kommt sie eher dem Design-Pattern Facade auf Seite 185 näher, zu dem sie allerdings auch einige Unterschiede aufweist, auf die hier nicht näher eingegangen werden soll. Daher wurde hier der allgemeinere Begriff Wrapper für die Benennung der Klasse verwendet.

17 Vergleiche [Beck & Gamma 2004], Seite 319

chung ausgeschlossen worden sind. Für das Sammeln der aufgerufenen Methoden sind die Methoden `visit(ClassInstanceCreation)` und `visit(MethodInvocation)` überschrieben. Sie prüfen jeweils, ob die übergebene Methode beziehungsweise der übergebene Konstruktor bereits untersucht wurde, und wenn nein, ob sie laut den voreingestellten Filtern weiter untersucht werden soll.

Wurde die Methode noch nicht untersucht, ist sie jedoch nicht im Filter enthalten, wird sie einfach zu den untersuchten Methoden hinzugefügt und die Untersuchung des AST fortgesetzt.

Wurde die Methode noch nicht untersucht und ist sie hingegen aber im Methodenfilter enthalten, wird sie sowohl zu der Liste der aufgerufenen Methoden als auch zu der Liste der untersuchten Methoden hinzugefügt. Wie im Kapitel 3.1.1 beschrieben, müssen die so gefunden Methoden und auch die Methoden, die die statisch deklarierten Methoden überschreiben, weiter untersucht werden. Dazu werden mit Hilfe der privaten Methode getOverwritingMethods(IMethod) über die Eclipse-Typ-Hierarchie[18] alle überschreibenden Klassen und aus diesen die überschreibenden Methoden für die gefundene Methode extrahiert. Der Rumpf der so gefunden Methoden wird dann (falls noch nicht geschehen) mit der selben Instanz des `CalledMethodExtractorASTVisitor` untersucht, in dem dieser an die Methode `MethodDeclaration.accept(ASTVisitor)` übergeben wird. Das `MethodDeclaration` Objekt wird dabei aus der gefunden `IMethod` Instanz mit Hilfe der Methoden `ASTHelperMethods.getMethodDeclarationFromIMethod(IMethod)` ermittelt, die wiederum indirekt auf einen ASTParser aufsetzt.

Hat der ASTVisitor den gesamten Baum durchlaufen, kehrt er zum Aufrufpunkt in der Methode `extractTestedMethodsFromBody()` zurück, die dann den Vektor der gefundenen Methoden zurück gibt.

Im Gegensatz zu der Ermittlung auf der Basis des Aufrufbaums setzen die beiden anderen Varianten auf die @MUT Annotation auf. Sie unterscheiden sich nur in der internen Implementierung des Zugriffs auf die @MUT Annotation und stehen nur intern in der Klasse `TestIMethodWrapper` frei zur Auswahl. Für den Zugriff von außen über die Methode `extractIMethodsUnderTestFromAnnotation()` ist es transparent, welche Implementierung zum Einsatz kommt. Intern delegiert diese Methode Anfragen abhängig vom Wert der internen Variable `mutTestMethodImpl` über die Methode `extractMUTNamesFromAnnotation()` entweder an die Methode `extractMUTNamesFromAnnotationByAST()` oder an die Methode `extractMUTNamesFromAnnotationByRegex()`. Standardmäßig ist die *Regular-Expression*-Variante aktiviert. Die AST-Variante kann durch eine Zuweisung des Wertes der Konstanten `AST_IMPL` an die Variable `mutTestMethodImpl` erfolgen. Eine Konfiguration über die Voreinstellungen ist nicht vorgesehen.

18 Eine Beschreibung der Verwendung der Eclipse-Typ-Hierachie findet sich im Anhang A2, „Typ-Hierachie“

In der AST-Variante kommt eine Instanz der Klasse `TestedMethodNameFromAnnotationExtractor` zum Einsatz, die nur diejenigen Knotentypen näher untersucht, die zwischen dem Quellcode der Testmethode und dem Inhalt der Annotation vorkommen können. Die Details ergeben sich direkt aus der Implementierung, die dieser Arbeit auf einer CD beiliegt.

Die *Regular-Expression*-Variante erfordert hingegen nicht, dass aus der Methode ein AST geparsed wird, sondern analysiert den Quellcode der Methode direkt mit Hilfe der beiden *Regular-Expressions*, die in Listing 12 dargestellt sind.

```
1:  private static final Pattern annotationFromMethodSourcePattern =
2:      Pattern.compile("(?s)(?:@MUT|@" + MUT_FQN + ")\\s*\\((.*})\\s*\\)");
3:
4:  private static final Pattern methodNamesFromAnnotationContentPattern =
5:      Pattern.compile("(?s)\"([^\"]*)\"");
6:
```

Listing 12: Regular-Expression-Patterns für die Extrahierung von den Namen der getetsten Methoden aus der @MUT Annotation

Während das erste Pattern den Inhalt der @MUT Annotation aus dem Quelltext der Methode raus zieht, ist das zweite Pattern dafür verantwortlich aus dem so gewonnen Inhalt die voll Methodennamen zu ermitteln. Eine Erklärung zum Aufbau einer *Regular-Expression* findet sich in der Java-API-Doku zu der Klasse `java.util.regex.Pattern`.

4.5 Kontributor für die Bearbeitung der MUT Annotation

Die Methoden, die über das in Kapitel 4.4 beschriebene Verfahren ermittelt wurden, werden dem Entwickler für das Anlegen der @MUT Annotation über einen Dialog an die Hand gegeben, um diese Arbeit zu erleichtern und Tippfehler zu minimieren. Abbildung 15 zeigt diesen Dialog.

Um diesen Dialog über das Kontextmenü eine Methode im *Outline View* zur Verfügung zu stellen, ist in dem Plugin der in Listing 13 dargestellte Kontributor definiert. Der Erweiterungspunkt `org.eclipse.ui.popupMenus` ist im Anhang A3, „Hinzufügen von Menüpunkten zu einem Kontextmenü“, beschrieben. In Zeile 5 ist angegeben, dass der folgende Menüpunkt immer zur Verfügung stehen sollen, wenn es sich bei dem momentan selektierten Objekt um ein `IMethod` Objekt handelt. Auf die fehlenden Menüpunkte in Zeile 6 wird im folgenden Kapitel eingegangen. Ab Zeile 7 wird der Menüpunkt für den oben genannten Dialog definiert. Wichtig sind an dieser Stelle das Attribut `label` in Zeile 11, das auf einen Schlüssel in der Datei `plugin.properties` verweist, das Attibut `menubarPath` in Zeile 12, das angibt, an welcher Stelle im Kontextmenü der Menüpunkt auftauchen soll (der Anker `additions` sollte standardmäßig in jedem Kontextmenü definiert sein) und die Referenz auf die Klasse `MUTDialogAction` in den Zeilen 8 bis 9. Zeile 10

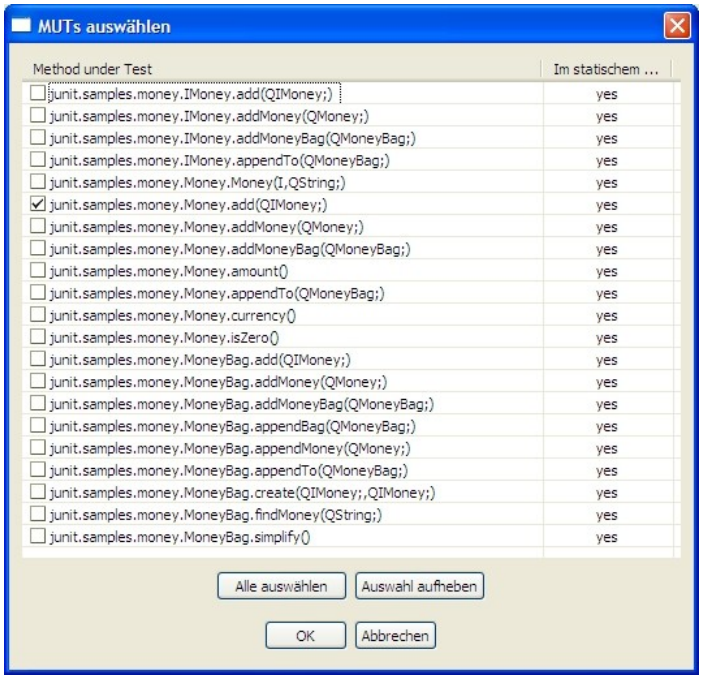

Abbildung 15: Dialog zum Hinzufügen direkt oder indirekt aufgerufener Methoden zu einer MUT Annotation

definiert eine ID, über die diese Action referenziert werden kann. Diese ID wird zwar im Rahmen dieses Plugins nicht verwendet, muss bei der Definition der Erweiterung aber trotzdem mit angegeben werden.

```
1: <extension
2:     point="org.eclipse.ui.popupMenus">
3:  <objectContribution
4:     id="org.projectory.ezunit.iMethodContribution"
5:     objectClass="org.eclipse.jdt.core.IMethod">
6:    ...
7:    <action
8:       class="org.projectory.ezunit.internal.
9:              contribution.MUTDialogAction"
10:     id="org.projectory.ezunit.mutDialog"
11:     label="%editMUTs"
12:     menubarPath="additions"/>
13: </objectContribution>
14:</extension>
```

Listing 13: Defintion des Kontributors für die Bearbeitung der Method under Test

Diese Action implementiert das Interface `org.eclipse.ui.IObjectActionDelegate`, dem über die Methode `selectionChanged(IAction, ISelection)` stets die aktuelle Auswahl übermittelt wird, und dem über die Methode `run(IAction)` mitgeteilt wird, dass der zugehörige Menüpunkt angeklickt wurde.

In der letztgenannten Methode wird eine Instanz des Objekts `org.projectory.ezunit.internal.contribution.MUTDialogAction.MUTDialog` erzeugt; diese Instanz stellt sich als der Dialog aus Abbildung 15 dar. Dieser Dialog nutzt indirekt durch einige weitere Klassen für die Darstellung der Tabelle[19] die Methode `extractTestedMethodsFromBody()` aus dem vorangegangenen Kapitel, um alle Testmethoden zu der im *Outline View* selektierten Methode zu ermitteln und darzustellen.

Nachdem der Benutzer durch die Auswahl der Checkboxen die Methoden ausgewählt hat, die er in die MUT Annotation aufnehmen möchte, stößt er durch einen Klick auf die OK-Schaltfläche die Methode an, die sich um die Änderung der Annotation an der Testmethode kümmert. Dazu stehen in der aktuellen Version des Plugins zwei Implementierungen zur Verfügung.

Die erste findet sich in der Methode `MUTDialog.updateMUTAnnotationViaSource()`, die wiederum auf die Methode `TestIMethodWrapper.UpdateMUTAnnotationViaSource (Vector<IMethod>)` zugreift. Die zweite Implementierung verwendet über die Methode `MUTDialog.updateMUTAnnotationViaAST()` die Methode `TestIMethodWrapper.createMUTAnnotation(Vector<IMethod>)`. Die beiden Methoden der Klasse `TestIMethodWrapper` werden im nächsten Kapitel näher beschrieben.

4.6 Erstellung der MUT Annotation

Für die Erstellung der @MUT Annotation aus einer gegebenen Liste von `org.eclipse.jdt.core.IMethod` Objekten stehen in der Klasse `TestIMethodWrapper` zwei Implementierungen zur Verfügung.

Die erste findet sich in der Methode `TestIMethodWrapper. UpdateMUTAnnotationViaSource(Vector<IMethod>)`. Sie greift über das Eclipse-Java-Model direkt auf den Quellcode der Methode zu und manipuliert ihn durch das Ersetzen von Textabschnitten. Da der Zusammenbau der Annotation so nicht ganz einfach zu verstehen ist und der zweite Ansatz in den Augen des Autors sowohl der elegantere als auch der zukunftsträchtigere ist, wird auf diesen ersten Ansatz nicht weiter eingegangen. Der Quellcode ist im Plugin nicht desto trotz enthalten, da dieses Verfahren den Vorteil besitzt, dass die eingefügte Annotation formatiert werden kann. Die geht mit der zweiten Implementierung nicht.

In dieser zweiten Implementierung in der Methode `TestIMethodWrapper. createMUTAnnotation(Vector<IMethod>)` wird für die Manipulation der @MUT Annotation auf den *Abstract Syntax Tree* der Testmethode zugegriffen. [Kuhn & Thomann 2006] beschreibt die Manipulation einer Java-Klasse über ihren AST allgemein, daher soll an dieser Stelle nur auf die konkrete Implementierung für das Erstellen der @MUT Annotation eingegangen werden.

19 [Gauthier 2003] ist ein Artikel, der die Verwendung der eingesetzten Klassen und Interfaces ICellModifier, IStructuredContentProvider und LabelProvider beschreibt.

Nachdem mit Hilfe eines `ASTParser` Objekts in der Methode `ASTHelperMethods.createASTCompilationUnit(ICompilationUnit)` ein `org.eclipse.jdt.core.dom.CompilationUnit` Objekt erzeugt wurde, muss dieses Objekt angewiesen werden, Änderungen am AST aufzunehmen. Dies geschieht durch einen Aufruf der Methode `CompilationUnit.recordModifications()`. Im Anschluss daran entfernt die Methode `removeMUTAnnotation(MethodDeclaration)` über eine anonyme `ASTVisitor` Klasse eine eventuell bereits bestehende @MUT Annotation.

Listing 14 zeigt den Ausschnitt aus der Methode, in dem die @MUT Annotation neu angelegt wird. Die Annotation ist ein Objekt vom Typ `NormalAnnotation` (Zeilen 2 bis 3), dem über das Attribut `TypeName` ein `Name` Objekt (Zeilen 4 bis 5) zugwiesen wird, (Zeile 6) das den Typ der Annotation angibt. In die Liste der `values` der Annotation wird ein Objekt vom Typ `MemberValuePair` eingefügt (Zeile 11), das den Namen „methods“ erhält (Zeile 14 bis 17) und als Wert ein Objekt, das ein Array initialisiert. Dieses wird in den Zeilen 12 und 13 erstellt und dem `MemberValuePair` Objekt in der Zeile 18 hinzugefügt. Dem zu initialisierenden Array wird im Anschluss für jede Methode ein `StringLiteral` Objekt hinzugefügt (Zeilen 20 bis 30), bevor die Annotation als ganzes der Testmethode an erster Stelle hinzugefügt wird (Zeile 33).

```
1:  // MUT Annotation anlegen
2:  NormalAnnotation mutAnnotation =
3:      testMethodDeclaration.getAST().newNormalAnnotation();
4:  Name mutTypeName =
5:      mutAnnotation.getAST().newName(MUT.class.getName());
6:  mutAnnotation.setTypeName(mutTypeName);
7:
8:  // "methods" Attribut anlegen
9:  MemberValuePair methodsMemberValuePair =
10:     mutAnnotation.getAST().newMemberValuePair();
11: mutAnnotation.values().add(methodsMemberValuePair);
12: ArrayInitializer arrayInitializer =
13:     methodsMemberValuePair.getAST().newArrayInitializer();
14: SimpleName simpleName =
15:     methodsMemberValuePair.getAST().
16:     newSimpleName("methods");
17: methodsMemberValuePair.setName(simpleName);
18: methodsMemberValuePair.setValue(arrayInitializer);
19:
20: // Methodennamen zum "methods" Attibut hinzufügen
21: for (IMethod method : methodsUnderTest) {
22:         IMethodUnderTestWrapper methodUnderTestWrapper =
23:       new IMethodUnderTestWrapper(     method);
24:
25:         StringLiteral stringLiteral = arrayInitializer.getAST()
26:                   .newStringLiteral();
27:         stringLiteral.setLiteralValue(methodUnderTestWrapper
28:                   .getAnnotationContent());
29:         arrayInitializer.expressions().add(stringLiteral);
30: }
31:
32: // MUT Annotation an die Methode anhängen
33: testMethodDeclaration.modifiers().add(0, mutAnnotation);
```

Listing 14: Erstellen der MUT Annotation über den AST

Im Anschluss an die Erstellung der @MUT Annotation werden die aufgenommen Änderungen durch die Methode `applyASTChanges(CompilationUnit)` mit Hilfe eines `org.eclipse.text.edits.TextEdit` Objekts, das von der Methode `CompilationUnit.rewrite(IDocument, Map)` zurückgeliefert wird, in den aktuell geöffneten Editor übertragen.

4.7 Completion Proposal Computer für die MUT Annotation

Seit der Eclipse Version 3.2 ist es über den Erweiterungspunkt `org.eclipse.jdt.ui.javaCompletionProposalComputer` möglich, den existierenden Java-Editor um eigene *Completion Proposal Computer* zu erweitern. In vorhergehenden Eclipse-Versionen war das Hinzufügen von eigenen *Completion Proposals* zu bestehenden Editoren nicht möglich. Stattdessen musste auf Basis der bestehenden Klassen ein neuer Editor definiert werden, der durch das Überschreiben einiger Methoden an den entsprechenden Stellen eigene *Completion Proposals* hinzufügen konnte. Dies war in den ersten Plugin-Versionen auch so implementiert.

```
1: <extension
2:    id="org.projectory.ezunit.
3:        mutCompletionProposalComputer"
4:    name="MUT Completion Proposal Computer"
5:    point="org.eclipse.jdt.ui.
6:           javaCompletionProposalComputer">
7:  <javaCompletionProposalComputer
8:      activate="true"
9:      categoryId="org.eclipse.jdt.ui.
10:                 defaultProposalCategory"
11:     class="org.projectory.ezunit.internal.completion.
12:            MUTMethodNameCompletionProposalComputer">
13:  <partition type="__java_string"/>
14: </javaCompletionProposalComputer>
15:</extension>
```

Listing 15: Defintion des Completion Proposal Computers für den Inhalt der MUT Annotation

In Listing 15 wird der *Completion Proposal Computer* in den Zeilen 2 bis 4 mit einer ID und einem Namen versehen. Das Attribut in Zeile 8 sorgt dafür, dass diesen Plugin spätestens dann geladen wird, wenn in einem Java-Editor *Content Assist* angefordert wird. In den Zeilen 9 und 10 wird die Zugehörigkeit dieses *Computers* zu der Standardkategorie definiert und in den Zeilen 11 und 12 die Klasse referenziert, die die Erstellung der Vorschläge übernimmt. Die dort angegebene Klasse muss dabei das Interface `org.eclipse.jdt.ui.text.java.IJavaCompletionProposalComputer` implementieren. Die Zeile 13 schränkt die Gültigkeit für diese Vervollständigung auf die Definition von Strings ein.

In der Klasse `MUTMethodNameCompletionProposalComputer` ist `computeCompletionProposals (ContentAssistInvocationContext, IprogressMonitor)` die für diese Vervollständigung interessante Methode.

In Listing 16 sind nur die wichtigsten Ausschnitte aus dieser Methode abgebildet. In den Zeilen 5 bis 7 wird aus dem übergebenen Aufrufkontext zuerst die Methoden ermittelt, in der sich der Cursor zum Zeitpunkt der Anforderung der Codevervollständigung befindet. Die Zeilen 9 bis 13 dienen der Ermittlung des innerhalb des String-Literals bereits eingegebenen Textes, das heißt des Strings zwischen dem öffnenden Anführungszeichen und der aktuellen Position. Aus der am Anfang ermittelten Methode werden alle aufgerufenen Methoden (Zeilen 15 und 16) sowie alle bereits in der Annotation vorkommenden Methoden (Zeilen 18 und 19) bestimmt. Auf Basis all dieser Informationen werden in einer Schleife (die im Listing nicht dargestellt ist) alle aufgerufenen Methoden durchlaufen. Kommt eine Methode noch nicht in der @MUT Annotation vor und beginnt sie mit dem bereits eingegebenen Text, so wird ein neuer Vervollständigungsvorschlag erstellt (Zeilen 21 bis 23). Diesem werden der neu einzufügende Text (`methodName`) sowie die Startposition und Länge des zu ersetzenden Textes übergeben.

```
1:  public List computeCompletionProposals(
2:          ContentAssistInvocationContext context,
3:          IProgressMonitor monitor) {
4:  ...
5:  IJavaElement javaElement =
6:          ((JavaContentAssistInvocationContext)context).
7:          getCompilationUnit().getElementAt(offset);
8:  ...
9:  CompletionContext coreContext =
10:         ((JavaContentAssistInvocationContext)context).
11:         getCoreContext();
12: ...
13: String startLetters = new String(coreContext.getToken());
14:
15: Vector<IMethod> methods = new TestIMethodWrapper(method).
16:         extractTestedMethodsFromBody();
17: ...
18: existingAnnotations = new TestIMethodWrapper(method).
19:         extractMUTNamesFromAnnotation();
20: ...
21: completionProposals.add(new MUTAnnotationCompletionProposal(
22:         methodName, offset - startLetters.length(),
23:         startLetters.length()));
```

Listing 16: Ausschnitte aus der Methode computeCompletionProposals

Die Klasse `MUTAnnotationCompletionProposal`, die das Interface `org.eclipse.jface.text.contentassist.ICompletionProposal` implementiert, ist von der Implementierung dann recht einfach. Sie kümmert sich in der in Listing 19 abgebildeten Methoden `apply(IDocument)` um den tatsächlichen Austausch des bereits eingegebenen Textes gegen den kompletten Methodennamen, falls der entsprechende Vorschlag ausgewählt wird.

```
1: public void apply(IDocument document) {
2:   try {
3:    document.replace(offset, length, annotationContent);
4:   } catch (BadLocationException e) {
5:    ...
6:   }
7: }
```

Listing 17: Methode Apply der Klasse MUTAnnotationCompletionProposal

4.8 Ermittlung der Testmethoden

Die Ermittlung der Testmethoden zu einer getesteten Methode ist aufwendiger als der umgekehrte Weg. Dies liegt daran, dass die Testmethode direkt über ihre @MUT Annotation beziehungsweise über ihren statischen Aufrufbaum eine Referenz zu der getesteten Methode besitzt. Die getestete Methode hingegen besitzt keine Referenzen zu ihren Testmethoden. Daher müssen alle Testmethoden untersucht werden, ob sie die getestete Methode referenzieren.

Für die Untersuchung der Testmethoden stehen, wie im Kapitel 4.4, „Ermittlung der getesteten Methoden“, beschrieben, drei mögliche Varianten zur Verfügung:

- die Untersuchung des statischen Aufrufbaums über den AST der Testmethode,
- die Untersuchung der @MUT Annotation über den AST der Testmethode sowie
- die Untersuchung der @MUT Annotation mit Hilfe einer *Regular Expression*, die auf den Quellcode der Testmethode angewendet wird.

Um ein Gefühl für den Rechenaufwand der verschiedenen Varianten zu bekommen, ist es in der Methode `org.projectory.ezunit.internal.methodwrapper.IMethodUnderTestWrapper.getTestMethods()` möglich, über den Schalter `IMPLEMTATION_COMPARISON` zwischen der Ausführung von nur einer der drei Varianten und einem Vergleich des Zeitbedarfs aller Varianten zu wechseln.

Auf Basis des Eclipse-Projektes „EzUnit_Sample“, das auf der CD zu dieser Arbeit enthalten ist, kann ein kleines Testszenario aufgebaut werden. In einem Workspace, in dem lediglich dieses Projekt geöffnet ist, ergeben sich durch das Öffnen des Menüpunktes „Gehe zur Testmethode...“ für die Methode `junit.samples.money.IMoney.add(IMoney)` die folgenden Zeiten.

	Testläufe (Dauer in Millisekunden)						
Variante	***1. Lauf***	***2. Lauf***	***3. Lauf***	***4. Lauf***	***5. Lauf***	***6. Lauf***	***Ø 2 - 6***
Analyse des Aufrufbaums	85641	84156	83031	83046	83297	82969	83299,8
AST-Analyse der @MUT Annotation	156	125	125	141	110	110	122,2
Regular Expression Analyse der @MUT Annotation	203	63	62	47	62	47	56,2

Auch wenn diese Zahlen nicht über viele Läufe gemittelt sind und von Rechner zu Rechner recht unterschiedlich ausfallen können, helfen sie dennoch, gewisse Aussagen über die verschiedenen Varianten zu machen.

Durch die extrem lange Laufzeit von über einer Minute für die Analyse des Aufrufbaums schon bei diesem kleinen Projekt (64 Methoden in 4 Klassen) scheidet die erste Variante für den praktischen Gebrauch aus. Um von einer getesteten Methode zu einer Testmethode navigieren zu können, ist also eine explizite Annotation der Beziehung nötig.

Die Divergenz der Zeiten bei der dritten Variante beruht auf der Tatsache, dass bei der ersten Ausführung die *Regluar Expression Patterns* für die Suche kompiliert werden müssen. Bei jeder weiteren Ausführung werden die bereits kompilierten *Patterns* erneut verwendet, wodurch die recht ähnlichen Zeiten erst ab dem zweiten Lauf zu Stande kommen.

Eine klare Aussage darüber, ob die zweite oder die dritte Variante besser für größere Projekte geeignet ist, lässt sich aus diesen Zahlen nicht ableiten. Während bei einigen Stichproben mit mehreren unabhängigen Projekten die *Regular-Expression*-Variante die besseren Ergebnisse lieferte, skalierte die AST-Variante bei dem Test mit einer zusätzlichen umbenannten Kopie der „EzUnit_Sample“ Projekts besser.

4.9 Kontributoren für die Navigation

Auf Basis der über die @MUT Annotation dargestellten Beziehung zwischen Testmethoden und getesteten Methoden bieten zwei weitere Menüpunkte die Möglichkeit, zwischen diesen Methoden zu navigieren. Listing 18 zeigt die Definition der dazu nötigen Erweiterungen.

Die Bedeutung der Attribute `class`, `id`, `label` und `menubarPath` ist bereits im Kapitel 4.5 beschrieben worden und soll deshalb hier nicht weiter ausgeführt werden. Das Attribut `style` hingegen ist dort noch nicht betrachtet worden. Die Beschreibung des Erweiterungspunkts `org.eclipse.ui.popupMenus` in der Eclipse Hilfe führt für dieses Attribut vier mögliche Werte auf:

- `push` (als Standardwert) für normale Menüpunkte
- `radio` für eine Gruppe von Menüpunkten von denen einer selektiert sein kann
- `toggle` für einen Menüpunkt, der entweder selektiert oder nicht selektiert sein kann
- `pulldown` für einen Menüpunkt, der ein weiteres Untermenü öffnet

Für die Navigation ist ein Menüpunkt mit einem Untermenü vorgesehen, da in diesem alle ermittelten Test- beziehungsweise getesteten Methoden zur Auswahl zur Verfügung gestellt werden können.

```
1: <extension
2:    point="org.eclipse.ui.popupMenus">
3:  <objectContribution
4:     id="org.projectory.ezunit.iMethodContribution"
5:     objectClass="org.eclipse.jdt.core.IMethod">
6:   <action
7:      class="org.projectory.ezunit.internal.
8:             contribution.GotoMUTMenuAction"
9:      id="org.projectory.ezunit.gotoMUTMenu"
10:     label="%gotoMUT"
11:     menubarPath="additions"
12:     style="pulldown"/>
13:  <action
14:     class="org.projectory.ezunit.internal.
15:            contribution.GotoTestMethodMenuAction"
16:     id="org.projectory.ezunit.gotoTestMethodMenu"
17:     label="%gotoTestMethod"
18:     menubarPath="additions"
19:     style="pulldown"/>
20:  ...
21: </objectContribution>
22:</extension>
```

Listing 18: Defintion des Kontributoren für die Navigation zwischen Testmethoden und getesteten Methoden

Für die Generierung des Untermenüs ist ein Objekt vom Typ `org.eclipse.jface.action.IMenuCreator` zuständig. Eclipse ermittelt das konkret zuständige Objekt für die Darstellung eines Untermenüs einer Action direkt durch eine Anfrage an die Methode `getMenuCreator()` der Action. Die beiden Klassen `GotoMUTMenuAction` und `GotoTestMethodMenuAction`, die hinter den beiden in Listing 18 definierten Menüpunkten liegen, implementieren neben dem Interface `org.eclipse.ui.IObjectActionDelegate` auch das Interface `IMenuCreator` und können sich somit selber bei der Action als Generator für den Inhalt des Untermenüs registrieren.

Die durch letzteres Interface eingebrachte und implementierte Methode `getMenu(Menu)` wird allerdings lediglich bei der ersten Erstellung des Untermenüs aufgerufen. Da sich der Inhalt aber jedes Mal ändern kann (es kann ja eine andere Methode selektiert worden sein), kann der Inhalt des Untermenüs nicht direkt in dieser Methode ermittelt werden. Stattdessen wird auf der Basis des übergebenen `Menu` Objekts lediglich ein neues Menü erzeugt, dem ein `org.eclipse.swt.events.MenuListener` hinzugefügt wird. Diese abstrakte Klasse definiert eine Methode `menuShown(MenuEvent)`, die vor jedem Anzeigen des Menüs aufgerufen wird. Hier können jetzt alle existierenden Menüpunkte in dem Untermenü entfernt und der neue Inhalt des Menüs ermittelt werden.

Abhängig davon ob es sich bei der momentan selektierten Methode um eine Test- oder um eine getestete Methode handelt, kommen für die Ermittlung der Untermenüpunkt unterschiedliche Methoden zum Einsatz. Die Unterscheidung zwischen Test- und getesteten

Methoden wird aus Performance-Gründen auf eine Überprüfung des Vorkommens des Strings „@Test“ im Quellcode der selektierten Methode beschränkt, anstatt den AST der Methode zu erzeugen und in ihm auf die Existenz der Annotation zu prüfen.

Wenn es sich bei der selektierten Methode um eine Testmethode handelt, werden über die Methode `TestIMethodWrapper.extractTestedMethods()` alle getesteten Methoden ermittelt. Abhängig von den Voreinstellungen, die im Plugin getroffen sind, wird dabei nur die @MUT Annotation ausgewertet oder aber der statische Aufrufgraph der Testmethode, wie im Kapitel 4.4 beschrieben. Handelt es sich bei der selektierten Methode um eine getestete Methode, kommt die Methode `IMethodUnderTestWrapper.getTestMethods()` zum Einsatz. Wie in Kaptitel 4.8 beschrieben ermittelt diese Methode die Testmethoden, die in ihrer @MUT Annotation die selektierte Methode enthalten.

Für jede ermittelte Methode wird in beiden Fällen ein Instanz der Klasse `org.projectory.ezunit.internal.contribution.GotoMethodAction` erzeugt, die den Namen der Methode als Bezeichnung des Menüpunktes zurück gibt und die entsprechende Methode im EzUnit-Java-Editor öffnet, wenn der generierte Menüpunkt ausgewählt wird. Dieses Objekt wird dann dem entsprechenden Menü zum Listener hinzugefügt.

Konnte keine Methode ermittelt werden, wird dem Menü lediglich ein einfaches `ActionContributionItem` Objekt hinzugefügt, das nicht anklickbar ist, da es nicht aktiviert ist und lediglich die Information ausgibt, dass keine Methode gefunden werden konnte.

4.10 Testrunlistener und Marker

Kern dieser Arbeit ist die Markierung logischer Fehler im Quellcode. Um diese Fehler wie im Kapitel 3.2, „Visualisierung von fehlgeschlagenen Tests“, beschrieben zu kennzeichnen, greift das Plugin auf die Erweiterungspunkte `org.eclipse.jdt.junit.testRunListeners` und `org.eclipse.core.resources.markers` zurück. Dazu kommen noch drei weitere Erweiterungspunkte für die Visualisierung der Marker, die weiter unten in diesem Kapitel beschrieben werden.

4.10.1 Testrunlistener-Definition und -Implementierung

Wie aus Listing 19 ersichtlich ist die Definition eines Testrunlisteners recht einfach. Es wird lediglich eine Klasse angegeben, die das Interface `org.eclipse.jdt.junit.ITestRunListener` implementieren muss. Dieses Interface stellt Methoden zur Verfügung, mit denen sich Listener über den Beginn und das Ende von einzelnen Tests oder des kompletten Testlaufs, einzelne wiederholte Tests oder fehlgeschlagene Tests informieren lassen können. Die Klasse `org.projectory.ezunit.internal.failuremarker.MarkerCreatorTestRunListener` verarbeitet nur zwei dieser Ereignisse: den Start eines Testlaufs und das Fehlschlagen eines Testfalls.

```
1: <extension
2:    point="org.eclipse.jdt.junit.testRunListeners">
3:  <testRunListener
4:     class="org.projectory.ezunit.
5:            internal.Failuremarker.
6:            MarkerCreatorTestRunListener"/>
7: </extension>
```

Listing 19: Defintion des Testrunlisteners, der sich um die Erzeugung der Marker kümmert

Beim Auftreten des ersten Ereignisses werden in der Methode `testRunStarted(int)` alle geöffneten Projekte durchlaufen, um vorhandene `org.projectory.ezunit.failedTestMarker` zu entfernen. Listing 20 zeigt den zugehörigen Codeausschnitt. Hinter der Konstanten `MarkerCreatorTestRunListener.MARKER_ID` verbirgt sie die ID des genannten `failedTestMarker`.

```
1: public void testRunStarted(int testCount) {
2:   IProject[] projects = ResourcesPlugin.
3:     getWorkspace().getRoot().getProjects();
4:   for (IProject project : projects) {
5:     try {
6:       if (project.isOpen())
7:         project.deleteMarkers(
8:            MarkerCreatorTestRunListener.MARKER_ID,
9:            false, IResource.DEPTH_INFINITE);
10:    } catch (CoreException e) {
11:      ...
12:    }
13:  }
14:}
```

Listing 20: Entfernen der failedTestMarker von allen geöffneten Projekten

Bevor der `failedTestMarker` näher betrachtet wird, soll noch ein kurzer Blick auf die zweite implementierte Methode geworfen werden. Die in Listing 21 dargestellte Methode `testFailed` wird beim Fehlschlag eines Testfalls aufgerufen. Um aus der fehlgeschlagenen Testmethode die @MUT Annotation zu extrahieren beziehungsweise ihren Aufruf-

```
1: public void testFailed(int status, String testId,
2:       String testName, String trace) {
3:   IMethod testMethod = getTestMethod(testName);
4:   try {
5:     Collection<IMethod> testedMethods = new
6:          TestIMethodWrapper(testMethod).
7:          extractTestedMethods();
8:     for (IMethod method : testedMethods) {
9:       createMarker(method, testMethod);
10:    }
11:  } catch (JavaModelException e) {
12:    ...
13:  }
14:}
```

Listing 21: Behandlung eines fehlgeschlagenen Tests

baum zu betrachten, muss sie als `IMethod` Objekt zur Verfügung stehen. Da keine entsprechende Referenz an die Methode `testFailed` übergeben wird, muss sie aus den übergebenen Daten ermittelt werden. Dazu bietet sich der Parameter `testName` an, der sich aus dem Namen des Testmethode und der Klasse, in der diese Methode zu finden ist, zusammensetzt (zum Beispiel „testMethode(meine.Testklasse)“). Da eine Testmethode keine Parameter erwartet, reichen diese Information der Methode `getTestMethod(String)`, die in Zeile 3 aufgerufen wird, um über eine *Regular Expression* diese beiden Bestandteile zu extrahieren und innerhalb der Java-Projekte nach einer passenden Methode zu suchen. Das entsprechende `IMethod` Objekt wird dann zurück geliefert.

Auf Basis dieses Objekts kann eine Instanz der Klasse `TestIMethodWrapper` erzeugt werden. Diese bietet neben den im Kapitel 4.4 beschriebenen Methoden für den Zugriff auf die Methodennamen aus der @MUT Annotation und die Methoden aus dem Aufrufbaum noch eine weitere Methode `extractTestedMethods()`, die in Abhängigkeit von den Voreinstellungen des Plugins intern entweder auf den Aufrufbaum oder die @MUT Annotation zugreift und die getesteten Methoden zurück gibt. Der Testrunlistener nutzt diese Methode (Zeile 7 in Listing 21), um die Methoden zu ermitteln, die mit einem Marker versehen werden sollen (Zeile 9).

4.10.2 Marker-Definition und -Erzeugung

Listing 22 zeigt die Definition des Markers, mit dem eine Methoden markiert wird, auf die ein Testfall fehlgeschlagen ist. Es handelt sich um eine Erweiterung am Erweiterungspunkt `org.eclipse.core.resources.markers`. Eine Beschreibung des Erweiterungspunktes findet sich neben der Eclipse-Hilfe auch in [Beck & Gamma 2004] auf den Seiten 136 bis 139.

```
1: <extension
2:     id="failedTestMarker"
3:     name="%marker.name"
4:     point="org.eclipse.core.resources.markers">
5:  <persistent value="true"/>
6:  <attribute name="testIFile"/>
7:  <attribute name="testproject"/>
8:  <attribute name="testmethodOffset"/>
9:  <attribute name="testmethodLength"/>
10: <super type=
11:       "org.eclipse.core.resources.textmarker"/>
12: <super type=
13:       "org.eclipse.core.resources.problemmarker"/>
14:</extension>
```

Listing 22: Definition des failedTestMarkers

Die `id`, die dem Marker in Zeile 2 zugewiesen wird, bildet zusammen mit der ID des Plugins die eindeutige Identifikation, mit der auf den Marker zugegriffen werden kann. Auf diesen Marker kann also mit `org.projectory.ezunit.failedTestMarker` zugegriffen werden. Das Attribut `name` verweist auf einen Schlüssel in der Datei „plugin.properties“. In Zeile 5 wird angegeben, dass die Marker von diesem Typ, die an Dateien angehängt wurden, gespeichert und beim Neustart von Eclipse immer noch vorhanden sein sollen. In den Zeilen 6 bis 9 werden Attribute definiert, die bei der Erzeugung des Markers angegeben und später wieder verwendet werden können. In den Zeilen 10 bis 13 werden die Typen von Markern angegeben, von denen dieser Marker erben soll. Im Gegensatz zu Java ist bei dem Marker-Erweiterungspunkt Mehrfachvererbung möglich. Kinder des Typs `textmarker` werden in jedem Texteditor mit angezeigt (zu denen der Java-Editor genauso gehört wie der später noch vorgestellte EzUnit-Editor), Kinder des Typs `problemmarker` im *Problems View* von Eclipse.

Betrachten wir jetzt die in Listing 21 in Zeile 9 aufgerufene Methode `createMarker(IMethod, IMethod)` näher, sehen wir, dass dort ein Marker des oben definierten Typs erzeugt wird. Listing 23 zeigt einen Ausschnitt dieser Methode.

```
1: void createMarker(IMethod methodToMark, IMethod failedTest) {
2:  try {
3:   IResource resource = methodToMark.getUnderlyingResource();
4:   if (resource != null) {
5:    IMarker marker = resource.createMarker(MARKER_ID);
6:    ...
7:    HashMap<Object, Object> markerAttributes = new
8:        HashMap<Object, Object>();
9:    ...
10:   marker.setAttributes(markerAttributes);
11:  }
12: } catch (JavaModelException e) {
13:    ...
14: } catch (CoreException e) {
15:    ...
16: }
17: }
```

Listing 23: Erzeugung eines failedTestMarkers

Ein Marker ist grundsätzlich mit einer IResource verknüpft. Dies können auch Projekte oder Verzeichnisse sein, in den meisten Fällen sind es aber Dateien. Daher wird in der Zeile 3 die Datei ermittelt, in der die getestete Methode zu finden ist. An dieser Datei wird dann in Zeile 5 ein Marker erzeugt. Hinter der Konstante `MARKER_ID` steht der String `org.projectory. ezunit.failedTestMarker`, der als Identifikator für den durch dieses Plugin definierten Markertyp dient.

Der in Zeile 7 definierte HashMap werden die Attribute des Markers hinzugefügt. Die folgende Tabelle führt die Attribute, ihre Werte und ihre Bedeutung auf. Statt den Namen als String anzugeben, ist jeweils die Konstante genannt. Konstanten ohne vorangestellten Klassennamen befinden sich in der Klasse `MarkerCreatorTestRunListener`.

Attributname	***Attributwert***	***Bedeutung / Verwendung***
IMarker.CHAR_START	Startposition der Methode, die markiert werden soll	Beginn des Textbereichs, der durch den Marker gekennzeichnet wird
IMarker.CHAR_END	Endposition der Methode, die markiert werden soll	Ende des Textbereichs, der durch den Marker gekennzeichnet wird
IMarker.LOCATION	Konstante „Zeile“ und Nummer der Zeile im Quellcode, in der die Methode beginnt, die markiert werden soll	Wert, der im *Problems View* in der Spalte *Location* angezeigt wird
IMarker.MESSAGE	Beschreibender Text, der sich aus dem Methoden- und dem Klassennamen des Testfalls zusammensetzt, der fehlgeschlagen ist	Beschreibender Text zu dem Marker, der im *Problems View* und beim überfahren des Markers mit der Maus angezeigt wird
TESTPROJECT_NAME	Name des Projekts in dem die Testmethode liegt	Wird für den Quick-Fix Sprung zur Testmethode verwendet
TEST_IFILE_RELATIVE_	Projektrelativer Pfad der Klasse	Wird für den Quick-Fix Sprung

Attributname	*Attributwert*	*Bedeutung / Verwendung*
PATH	in der die Testmethode liegt	zur Testmethode verwendet
TESTCLASS_NAME	Name der Klasse, in der die Testmethode liegt	Wird für den Quick-Fix Sprung zur Testmethode verwendet
TESTMETHOD_OFFSET	Startposition der Testmethode	Wird für den Quick-Fix Sprung zur Testmethode verwendet
TESTMETHOD_LENGTH	Differenz aus End- und Startposition der Testmethode	Wird für den Quick-Fix Sprung zur Testmethode verwendet

Das in den meisten Beispielen gesetzte Attribut `IMarker.SEVERITY` wird in diesem Plugin bewusst nicht gesetzt. Die möglichen Werte `IMarker.SEVERITY_ERROR`, `IMarker.SEVERITY_WARNING` und `IMarker.SEVERITY_INFO` dienen dem *Problems View* für die Einordnung in eine der drei entsprechenden Kategorien, führen aber zumindest bei den ersten beiden Werten dazu, dass *Package Explorer* und *Outline View* unabhängig von bereitgestellten Icons stets (zusätzlich) das rote X für die Markierung eines Java-Fehlers anzeigen. Abbildung 16 zeigt sehr deutlich, wie unschön diese doppelte Markierung für ein und den selben Fehler ist. Daher kommen die beiden ersten Werte nicht in Frage. Der Wert `IMarker.SEVERITY_INFO` führt zwar nicht zu doppelten Symbolen, bewirkt aber, dass ein Eintrag im *Problems View* in der Kategorie „Infos“ erscheint. Wird das Attribut nicht gesetzt, taucht der Eintrag hingegen unter „Other Problems“ auf, was von der Bezeichnung wesentlich besser passt. Daher ist für dieses Plugin die Entscheidung getroffen worden, das Attribut `IMarker.SEVERITY` nicht zu setzen.

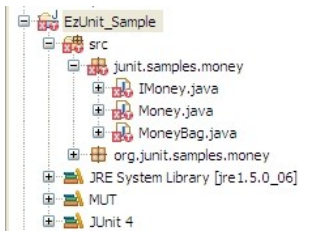

Abbildung 16: Package Explorer mit fehlerhafter Markierung von Testfehlern

4.10.3 Marker-Visualisierung

Der wesentliche Nutzen durch Marker ergibt sich erst durch eine passenden Visualisierung derselben. Marker selber markieren lediglich Objekte im Datenmodell, die Visualisierung muss von anderen Erweiterungen vorgenommen werden. In diesem Plugin sind dazu zwei Punkte recht unabhängig voneinander implementiert worden:

- Die Visualisierung im *Navigator View*, im *Package Explorer* und im *Outline View*
- Die Visualisierung im Java-Editor

Für die Darstellung in den zuerst genannten *Views* kommt der Erweiterungspunkt `org.eclipse.ui.decorators` zum Einsatz. Listing 24 zeigt die Definition des *Decorators*.[20]

```
1: <extension
2:     point="org.eclipse.ui.decorators">
3:  <decorator
4:      adaptable="true"
5:      class="org.projectory.ezunit.internal.decorator.
6:             LabelDecorator"
7:      id="org.projectory.ezunit.decorator"
8:      label="%decorator.label"
9:      lightweight="true">
10:  <enablement>
11:   <or>
12:    <objectClass
13:         name="org.eclipse.core.resources.IResource"/>
14:    <objectClass name="org.eclipse.jdt.core.IMethod"/>
15:   </or>
16:  </enablement>
17: </decorator>
18:</extension>
```

Listing 24: Definition des Decorators für den failedTestMarker

Dieser *Decorator* erzeugt das kleine „T“, das zum Beispiel in Abbildung 16 auf Seite 51 zu sehen ist. Dies passiert potentiell für alle `IResource` und `IMethod` Objekte (Zeilen 10-16), die in einem *View* dargestellt werden, der sich bei einem *Decorator Manager* registriert hat. Dies tun zum Beispiel die oben genannten *Navigator View*, im *Package Explorer* und im *Outline View*. In der Zeile 4 wird darüber hinaus aber auch angegeben, dass nicht nur Klassen, die eines der beiden genannten Interfaces implementieren, dekoriert werden sollen, sondern auch Klassen, für die ein Adapter auf eins der beiden Interfaces existiert. Dabei wird auf den Eclipse-Typerweiterungsmechanismus aufgesetzt, der in [Beck & Gamma 2004] im Kapitel 31 beschrieben ist.

20 Grundlegende Informationen zu diesem Erweiterungspunkt finden sich in der Eclipse Hilfe unter Platform Plug-in Developer Guide – Programmer's Guide – Advanced workbench concepts – Decorators und in der Beschreibung des Erweiterungspunktes

Durch den Wert des Attributs `lightweight` in Zeile 9 wird angegeben, dass die in den Zeilen 5 und 6 definierten Klasse das Interface `org.eclipse.jface.viewers.ILightweightLabelDecorator` und nicht das Interface `org.eclipse.jface.viewers.ILabelDecorator` implementiert. Während sich die Klasse im zweiten Fall um die komplette Darstellung des Icons kümmern müsste, bietet die leicht gewichtige Variante den Vorteil, dass es möglich ist, lediglich das kleine „T“ als Icon zu definieren, das dann an einer definierten Position über das normale Symbol gelegt wird.

```
1: public void decorate(Object element,
2:                      IDecoration decoration) {
3:   ...
4:   IMarker[] markers = objectResource.findMarkers(
5:         MarkerCreatorTestRunListener.MARKER_ID, true,
6:         IResource.DEPTH_INFINITE);
7:   if (markers.length > 0) {
8:     // Decorator hinzufügen
9:     ImageDescriptor imageDescriptor = EzUnitPlugin
10:        .imageDescriptorFromPlugin(EzUnitPlugin.ID,
11:        "icons/testerr_klein.gif"); //$NON-NLS-1$
12:    decoration.addOverlay(imageDescriptor,
13:        IDecoration.BOTTOM_RIGHT);
14:  }
15:  ...
16:}
```

Listing 25: Ausschnitt aus der Decorator Klasse

Ein Ausschnitt aus der Implementierung der Klasse `org.projectory.ezunit.internal.decorator.LabelDecorator` ist in Listing 25 abgebildet. Die Methode `decorate(Object, IDecoration)` wird vom Framework für die Objekte aufgerufen, die laut Definition ein Icon erhalten sollen. In diesem Plugin wird in dieser Methode überprüft, ob die übergebene Datei, eine Datei innerhalb des übergebenen `IContainer` Objekts beziehungsweise die übergebene Methode markiert ist. (Zeilen 4 bis 7). Wenn ja, wird ein Bild aus dem Plugin ermittelt (Zeilen 9 bis 11) und der Dekoration als überlagerndes Bild unten rechts hinzugefügt (Zeilen 12 und 13).

In einem Textdokument, wie es der Java-Editor anzeigt, werden Marker nicht direkt angezeigt. Stattdessen kommen dort so genannte Annotationen zum Einsatz, um Textbereiche zu kennzeichnen. Um diese Annotationen klar von den Java-Annotationen zu trennen, werden sie im weiteren immer als Text-Annotationen bezeichnen.

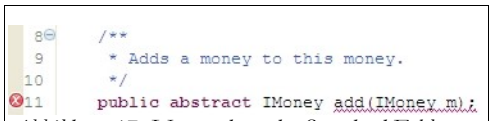

Abbildung 17: Verwendung der Standard-Fehler-Markierung in Texteditoren

```
1: <extension
2:         point="org.eclipse.ui.editors.annotationTypes">
3:  <type
4:     markerType="org.projectory.ezunit.failedTestMarker"
5:     name="org.projectory.ezunit.
6:          junitFailureAnnotationType"/>
7: </extension>
```

Listing 26: Deklaration des Text-Annotationstyps zu einem failedTestMarker

Wie aus Listing 26 ersichtlich, kann mit Hilfe des Erweiterungspunktes `org.eclipse.ui.editors.AnnotationTypes` eine Beziehung zwischen einem Test-Annotationstyp und dem bestehenden `failedTestMarker` geschaffen werden. Wird keine spezielle Visualisierung für einen bestimmten Marker benötigt, können bestehende Visualisierungen verwendet werden. So würde zum Beispiel das zusätzliche Attribut `super="org.eclipse.ui.workbench.texteditor.error"` im `type` Element dazu führen, dass Methoden, auf die ein Test fehlgeschlagen ist, genauso markiert werden, wie zum Beispiel auch Fehler im Java-Quellcode einer Datei. Abbildung 17 zeigt ein entsprechendes Beispiel.

Soll aber ein eigenes Symbol für die Visualisierung eines Markers in einem Texteditor zum Einsatz kommen, ist die Nutzung des Erweiterungspunktes `org.eclipse.ui.editors.markerAnnotationSpecification` nötig. Er bietet neben der Angabe eines eigenen Icons für einen Marker die Möglichkeit, alle weiteren Voreinstellungen einer Text-Annotation, wie zum Beispiel die Farbe der Markierung im Text und in der Leiste auf der rechten Seite des Editors, in den Voreinstellungen unter General – Editors – Text Editors – Annotations zu ändern.

Dem entsprechend dreht sich in Listing 27 auch fast alles um die Vorgaben für diese Voreinstellungsseite. Für die Beschreibung der hier nicht beschriebenen Attribute sei an dieser Stelle auf die Beschreibung des Erweiterungspunktes in der Eclipse-Hilfe verwiesen.

Das Augenmerk soll hier nur auf die Zeilen 3 und 4 sowie auf die Zeile 12 gerichtet werden. Die ersten beiden genannten Zeilen geben an, dass die Text-Annotationen des weiter oben definierten Typs `junitFailureAnnotationType` mit den in dieser Spezifikation definierten Einstellungen anzuzeigen sind. Diese Einstellungen umfassen unter anderem das Icon, das zum Einsatz kommen soll: Zeile 12 verweist auf das Bild mit dem kleinen, roten „T“.

```
1: <extension point="org.eclipse.ui.editors.
2:                   markerAnnotationSpecification">
3:  <specification
4:    annotationType="org.projectory.ezunit.
5:                    junitFailureAnnotationType"
6:    colorPreferenceKey="org.projectory.ezunit.prefColor"
7:    colorPreferenceValue="0,0,255"
8:    contributesToHeader="true"
9:    highlightPreferenceKey="org.projectory.ezunit.
10:                           prefHighlight"
11:   highlightPreferenceValue="false"
12:   icon="icons/testerr.gif"
13:   includeOnPreferencePage="true"
14:   label="JUnit Test Failure"
15:   overviewRulerPreferenceKey="org.projectory.ezunit.
16:                               prefOverviewRuler"
17:   overviewRulerPreferenceValue="true"
18:   presentationLayer="3"
19:   textPreferenceKey="org.projectory.ezunit.prefText"
20:   textPreferenceValue="false"
21:   verticalRulerPreferenceKey="org.projectory.ezunit.
22:                               prefVertRuler"
23:   verticalRulerPreferenceValue="true"/>
24:</extension>
```

Listing 27: Deklaration der Text-Annotationsspezifikation zu einem failedTestMarker

4.10.4 Marker Resolution Generator

Im Kapitel 3.3, „Navigation zwischen getesteten und Testmethoden“, wurde bereits erläutert, dass der Sprung von einer Methode, auf die ein Test fehlgeschlagen ist, zu dem fehlgeschlagenen Test hilfreich ist. Abbildung 10 zeigt die durch die Eclipse-Mechanismen nahe liegende Möglichkeit über den generierten Failed-Test-Marker. Diese wird über eine Implementierung des Erweiterungspunktes `org.eclipse.ui.ide.markerResolution` realisiert. Listing 28 zeigt die zugehörige Deklaration im Plugin.

```
1: <extension
2:   point="org.eclipse.ui.ide.markerResolution">
3:  <markerResolutionGenerator
4:    class="org.projectory.ezunit.internal.failuremarker.
5:           EzUnitFailureMarkerResolutionGenerator"
6:    markerType="org.projectory.ezunit.
7:                failedTestMarker"/>
8: </extension>
```

Listing 28: Deklaration des markerResolutionGeneratirs zu einem failedTestMarker

Im wesentlichen besagt diese Deklaration, dass für Marker des Typs `org.projectory.ezunit.failedTestMarker` (Zeilen 6 und 7) die Klasse `EzUnitFailureMarkerResolutionGenerator` (Zeilen 4 und 5) für die Ermittlung von passenden *Marker Resolutions* verwendet werden soll.

```
1: public IMarkerResolution[] getResolutions
2:     (IMarker marker) {
3:   IMarkerResolution jUnitFailureMarkerResolution =
4:       new IMarkerResolution() {
5:           ...
6:       };
7:   IMarkerResolution[] resolutions = new IMarkerResolution[]
8:       { jUnitFailureMarkerResolution };
9:   return resolutions;
10:}
```

Listing 29: Ausschnitt aus der Methode getResolutions(IMarker) der Klasse EzUnitFailureMarkerResolutionGenerator

Wie aus Listing 29 ersichtlicht, beschränkt sich diese Klasse wiederum darauf, eine Instanz einer anonymen Klasse zurück zu geben, die das Interface `org.eclipse.ui.IMarkerResolution` implementiert. Dieses Interface besitzt eine Methode `getLabel()`, die den Text zurück gibt, der angezeigt werden soll und eine Methode `run(IMarker)`, die ausgeführt wird, wenn auf den Lösungsvorschlag geklickt wird.

Die Implementierung dieser Methoden durch die anonyme Klasse ist in Listing 30 abgebildet. In ihr werden die verschiedenen Attribute des Failed-Test-Markers verwendet, die ihm bei der Erzeugung zugeordnet wurden (vergleiche Kapitel 4.10.2), um die Datei, die die Testmethode enthält, zu ermitteln (Zeilen 9 bis 19). Diese wird geöffnet (Zeilen 21 und 23) und der Abschnitt, in dem sich die Testmethode befindet, wird selektiert (Zeilen 24 bis 32).

```
1:  public void run(IMarker marker) {
2:   try {
3:    // Handelt es sich um einen JUnit Failure Marker?
4:    if (MarkerCreatorTestRunListener.MARKER_ID.equals(
5:        marker.getType())) {
6:     // Ja - IFile bestimmen
7:     IProject testProject = null;
8:     IFile testFile = null;
9:     String projectName = marker.getAttribute(
10:         MarkerCreatorTestRunListener.TESTPROJECT_NAME, null);
11:    if (projectName != null)
12:     testProject = ResourcesPlugin.getWorkspace().getRoot().
13:                   getProject(projectName);
14:    String fileName = marker.getAttribute(
15:         MarkerCreatorTestRunListener.
16:         TEST_IFILE_RELATIVE_PATH, null);
17:    if (testProject != null && testProject.exists()&&
18:        fileName != null)
19:     testFile = testProject.getFile(fileName);
20:    if (testFile != null && testFile.exists()) {
21:     IEditorPart javaEditor = IDE.openEditor(PlatformUI.
22:         getWorkbench().getActiveWorkbenchWindow().
23:         getActivePage(), testFile);
24:     if (javaEditor instanceof ITextEditor) {
25:      ITextEditor textEditor = (ITextEditor) javaEditor;
26:      int offset = marker.getAttribute(
27:         MarkerCreatorTestRunListener.TESTMETHOD_OFFSET, -1);
28:      int length = marker.getAttribute(
29:         MarkerCreatorTestRunListener.TESTMETHOD_LENGTH, -1);
30:      if (offset >= 0 && length >= 0)
31:       textEditor.selectAndReveal(offset, length);
32:     }
33:    }
34:   }
35:  } catch (PartInitException e) {
36:   ...
37:  } catch (CoreException e) {
38:   ...
39:  }
40: }
```

Listing 30: Methode IMarkerResolution.run(IMarker) in der Klasse EzUnitFailureMarkerResolutionGenerator

4.11 Unterstützende Bestandteile

Neben den bisher beschriebenen Bestandteilen des Plugins, die die Kernfunktionalität darstellen, sind in ihm noch weitere Klassen, Erweiterungen und Dateien enthalten. Diese sind für das Verständnis der Funktionalität nicht direkt ausschlaggebend, erhöhen den Nutzen des Plugins als ganzes allerdings, in dem sie es konfigurierbar und internationalisierbar machen und Hilfe oder Unterstützung für das Update von einer Plugin-Version auf die nächste anbieten. Diese Bestandteile sollen in diesem Kapitel kurz skizziert werden.

4.11.1 Plugin-Klasse

Die Plugin-Klasse `org.projectory.ezunit.EzUnitPlugin` ist im Manifest des Plugins als so genannter *Activator* eingetragen. Von ihr wird eine Instanz erzeugt, wenn vom Eclipse-Framework auf das Plugin zugegriffen wird. Mit der statischen Methode `getDefault()` bietet sie einen Zugriff auf diese Instanz. Über diese Instanz ist der Aufruf der geerbten Methoden zum Zugriff auf das *Log* des Plugins, auf den Voreinstellungsspeicher und auf die physikalische Position der mit diesem Plugin ausgelieferten Dateien, zum Beispiel der JAR-Bibliothek für die @MUT Annotation, möglich.

Die Bedeutung des Manifests eines Plugins und des *Activators* beschreibt [Bach 2007] auf den Seiten 32 bis 34 und 83 bis 85 und [Daum 2005] in den Kapiteln 13.4 und 13.5.

4.11.2 Install Handler

Die Klasse `org.projectory.ezunit.InstallHandler` hat nur noch eine Funktionalität, wenn diese Version des Plugins über eine ältere Version des Plugins installiert wird. In vorhergehenden Versionen wurde durch den *Install Handler* der in früheren Versionen mitgelieferte EzUnit-Java-Editor als Standardeditor für *.java* Dateien gesetzt. Dieser spezielle Editor wird in der aktuellen Version allerdings nicht mehr benötigt. Seine Aufgabe war das hinzufügen von *Completion Proposals* bei der Bearbeitung der @MUT Annotation. Dies ist inzwischen über einen Extension Point möglich, wie er in Kapitel 4.7, „Completion Proposal Computer für die MUT Annotation“, beschrieben wird. Daher wird dieser Editor in der aktuellen Version nicht mehr mitgeliefert und der mitgelieferte *Install Handler* macht die Einstellung dieses EzUnit-Java-Editors als Standardeditor wieder rückgängig.

Aufgerufen wird der *Install Handler* vom weiter unten beschriebenen EzUnit Feature. Das Ant-Script `build_installhandler_jar.xml` kopiert dazu den *Install Handler* in die JAR-Bibliothek `org.projectory.ezunit.installhandler.jar` innerhalb des Features.

4.11.3 Voreinstellungen

Für die Konfiguration verschiedener Aspekte des Plugins, wie zum Beispiel die Einstellung, ob nur annotierte oder auch aufgerufene Methoden mit Failed-Test-Markern versehen werden sollen, kommt eine Seite mit Voreinstellungen für dieses Plugin zum Einsatz. Sie ist, wie in Listing 31 ersichtlich, mit Hilfe des Erweiterungspunktes `org.eclipse.ui.preferencePages` definiert.

```
1: <extension point="org.eclipse.ui.preferencePages">
2:   <page class="org.projectory.ezunit.internal.
3:                preferences.MarkerPreferencePage"
4:         id="org.projectory.ezunit.prefPage"
5:         name="%preferencepage.name"/>
6: </extension>
```

Listing 31: Definition der Voreinstellungsseite für das Plugin

In dieser Erweiterung ist neben einer Referenz zum Namen der Seite in der Datei `plugin.properties` in der Zeile 5, in den Zeilen 2 und 3 der Namen der Klasse angegeben, die das Interface `org.eclipse.ui.IWorkbechPreferencePage` implementiert und für die Darstellung der Voreinstellungsseite verantwortlich ist, die in Abbildung 18 zu sehen ist.

Die Klasse `MarkerPreferencePage` überlässt die Darstellung dabei der mit Hilfe des Eclipse-Visual-Editors erzeugten Komponente `org.projectory.ezunit.internal.preferences.MarkerPreferencePageContent` und beschränkt sich stattdessen darauf, die Voreinstellungen aus dem Speicher des Plugins beziehungsweise die Standardwerte der Voreinstellungen zu laden und darzustellen, die eingegebenen Werte zu überprüfen und gegebenenfalls zu speichern.

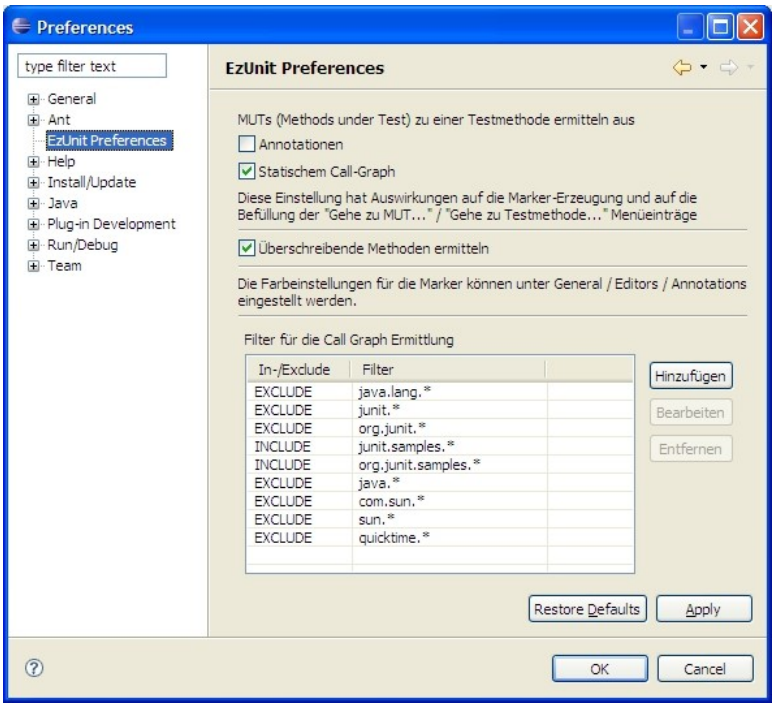

Abbildung 18: Voreinstellungsseite des EzUnit Plugins

Für das Auslesen und Speichern der Werte kommen neben der Methode `getPreferenceStore()` der Klasse `EzUnitPlugin` die Konstanten für die einzelnen Voreinstellungswerte zum Einsatz, die in der Klasse `org.projectory.ezunit.internal.preferences.PreferenceKeys` definiert sind.

Die Überprüfung der eingegebenen Werte durch die Methode `isValid()` in der Klasse `MarkerPreferencePage` setzt lediglich eine Warnung, wenn keine der auswählbaren Verfahren für die Ermittlung der getesteten Methoden ausgewählt ist. Weitere Überprüfungen werden nicht vorgenommen.

Die Standardwerte der Voreinstellungen werden nicht von der Seite direkt gesetzt, sondern durch die Implementierung des Erweiterungspunktes `org.eclipse.core.runtime.preferences`. Dieser definiert in den Zeilen 3 bis 6 von Listing 32 einen *Initializer*, der die Standardwerte der Variablen setzt. Dies geschieht in der Methode `initializeDefaultPreferences()` in der Klasse `PreferenceInitializer`.

```
1: <extension
2:     point="org.eclipse.core.runtime.preferences">
3:   <initializer
4:       class="org.projectory.ezunit.internal.
5:              preferences.PreferenceInitializer"/>
6: </extension>
```

Listing 32: Definition des Initializers für die Voreinstellungen des Plugins

Drei weitere Klassen des Plugins sind für die Behandlung der Filter nötig, die definiert werden können, um einzelne Klassen aus der Ermittlung der getesteten Methoden ein- oder auszuschließen. Dies sind auf der einen Seite die Klassen `org.projectory.ezunit.internal.preferences.FilterModifyShell` und die Klasse `FilterModifyComposite` im selben Package sowie die Klasse `org.projectory.ezunit.internal.MarkerFilter`. Die beiden ersten Klassen sind dabei wiederum mit Hilfe des Eclipse-Visual-Editors definiert und stellen die graphischen Komponenten für die Bearbeitung der Filter dar. Die Klasse `MarkerFilter` hingegen stellt einen Filter dar, der entweder eine Gruppe von Klassen ein- oder ausschließt. Darüber hinaus bietet sie zwei statische Methoden für die Konvertierung eines Strings in einem Vektor von MarkerFiltern (`stringToMarkerFilterVector(String)`) und umgekehrt (`markerFilterVectorToString(Vector<MarkerFilter>)`). Diese Methoden werden benötigt, da im Voreinstellungsspeicher eines Plugins keine Objekte gespeichert werden können.

4.11.4 Internationalisierung

Die Internationalisierung des Plugins setzt auf den Standard-Eclipse-Mechanismen auf. Diese sind in [Kehn et. al 2002] und [Daum 2005], Kapitel 12.6, ausführlich beschrieben.

Ergebnis der Anwendung der Eclipse-Mechanismen ist die Klasse `org.projectory.ezunit.internal.multilanguage.Messages`, sowie die Dateien `messages.properties` und `messages_en.properties` im selben Paket und die Dateien `plugin.properties` und `plugin_en.properties` im Wurzelverzeichnis des Plugins.

4.11.5 Hilfe

Die Benutzerhilfe des Plugins ist über eigene Einträge im Eclipse-Hilfe-Inhaltsverzeichnis realisiert. Der grundsätzliche Umgang mit dem Eclipse-Hilfe-System ist in [Daum 2005] im Kapitel 11.6.1 beschrieben.

Die Einträge in das Inhaltsverzeichnis werden in der Datei `help/toc.xml` definiert. Diese ist, wie im Listing 33 zu erkennen, im Plugin-Manifest in einer Erweiterung vom Typ `org.eclipse.help.toc` angegeben.

```
1: <extension point="org.eclipse.help.toc">
2:   <toc
3:       file="help/toc.xml"
4:       primary="true"/>
5: </extension>
```

Listing 33: Definition des Hilfe-Beitrags des Plugins

Das Attribut `primary` gibt dabei an, dass dieser Inhalt auf oberster Ebene eingeordnet werden soll. Für den Aufbau der Datei `toc.xml` und die Beziehung zu den Inhalten der Hilfe in den Verzeichnissen `help/html/` und `help/images/` sei an dieser Stelle auf die oben genannte Literatur verwiesen.

4.11.6 EzUnit Feature

Ein Eclipse-Feature fasst ein oder mehrere Plugins zu einer abgeschlossenen Funktionsgruppe zusammen, die als Ganzes verteilt werden kann. Ein Feature enthält dabei weitere Informationen wie eine Beschreibung, Lizenz- und Copyright-Informationen. Des weiteren ist ein Feature Grundlage für die Verteilung über eine Update-Site. [Daum 2005] beschreibt im Kapitel 12.3 die Grundlagen eines Eclipse-Features ausführlicher.

Ein Feature wird analog zu einem Plugin als eigenes Eclipse-Projekt angelegt. Es enthält als zentrale Definition eine Datei `feature.xml`. Diese ist für das Feature `org.projectory.ezunit.feature` in Listing 34 dargestellt. Die folgende Tabelle enthält eine kurze Beschreibung der verschiedenen Elemente.

Zeile	***Element / Attribut***	***Beschreibung***
2	id	Die ID des Features
3	label	Der Name des Features

Zeile	*Element / Attribut*	*Beschreibung*
4	version	Die dreistellige Version des Features. Wird zum Beispiel für die Erkennung durch eine Update-Site benötigt
5	provider	Der Anbieter des Features
6 – 8	install-handler	Angabe einer Klasse, die das Interface `org.eclipse.update.core.IInstallHandler` implementiert (Attribut `handler`) und in der JAR-Bibliothek zu finden ist, die im Attribut `library` angegeben ist. Beinhaltet Methoden, die bei der Installation oder Deinstallation des Features ausgeführt werden. Die unter `library` angegebene JAR-Bibliothek muss im Feature-Projekt enthalten und auch in der Datei `build.properties` aufgeführt werden.
9 – 17	description, copyright, license	Texte, die die Funktionalität, das Copyright und die Lizenzbedingungen des Features beschreiben. Diese Informationen werden bei der Installation des Features angezeigt.
18 – 21	url	Fügt der Liste der Update-Sites der Eclipse Installation eine neu Site mit der Bezeichnung im Attribut `label` und der URL im Attribut `url` hinzu.
22 – 26	requires	Drückt die Abhängigkeit des Features von den Features aus, deren IDs in den `import` Elementen im Attribut `feature` angegebenen sind. Die Version muss dabei größer oder gleich der angegebenen dreistelligen `version` sein.
27 – 30	plugin	Das Element `plugin` darf mehrfach vorkommen und definiert aus welchen Plugins sich das Feature zusammensetzt. Das Attribut `id` entspricht der ID des Plugins, das Attribut `version` kann eine spezielle oder jede verfügbare Version (0.0.0) spezifizieren und das Attribut `unpack` gibt an, ob das Plugin nach der Installation extrahiert werden muss oder nicht.

```
1:  <feature
2:      id="org.projectory.ezunit.feature"
3:      label="EzUnit"
4:      version="1.0.0"
5:      provider-name="www.projectory.org">
6:   <install-handler
7:          library="org.projectory.ezunit.installhandler.jar"
8:          handler="org.projectory.ezunit.InstallHandler"/>
9:   <description>
10:     EzUnit Feature Description
11:  </description>
12:  <copyright>
13:     EzUnit Copyright Notice
14:  </copyright>
15:  <license>
16:     EzUnit License Agreement
17:  </license>
18:  <url>
19:     <discovery label="EzUnit Update Site"
20:       url="http://www.fernuni-hagen.de/ps/prjs/EzUnit/update/"/>
21:  </url>
22:  <requires>
23:     <import feature="org.eclipse.jdt" version="3.2.0"/>
24:     <import feature="org.eclipse.platform" version="3.2.0"/>
25:     <import feature="org.eclipse.sdk" version="3.2.0"/>
26:  </requires>
27:  <plugin
28:         id="org.projectory.ezunit"
29:         version="0.0.0"
30:         unpack="false"/>
31: </feature>
```

Listing 34: Definition des EzUnit-Features

4.11.7 EzUnit Update Site

Die Definition einer Update-Site bietet die Möglichkeit, ein neues Feature einfach über den Eclipse-Mechanismus unter *Help – Software Updates – Find and Install...* zu installieren. Dabei können direkt Abhängigkeiten zu anderen Features geprüft werden und Benutzer zur Annahme von Lizenzbedingungen aufgefordert werden. [Daum 2005] beschreibt dies ausführlich in Kapitel 12.4.5.

Eine Update-Site wird analog zu Plugins und Features als eigenes Eclipse-Projekt angelegt. Sie enthält neben den Ordnern für gepackte Features und Plugins die Datei `site.xml`, die diesem Projekttyp als Definition dient. Listing 35 zeigt die Datei `site.xml` des Projektes `org.projectory.ezunit.updatesite`.

Zeile 3 besagt dabei, dass unter der dort angegebenen URL die Update-Site mit dem Namen aus Zeile 4 zu finden ist. Diese enthält lediglich ein Feature in einer Version: das in den Zeilen 6 bis 9 aufgeführte Feature `org.projectory.ezunit.feature` in der Version `1.0.0`. Der Ort, an dem die gepackte Version dieses Features zu finden ist, ist in Zeile 7 angegeben. Die von dem dort liegende Feature referenzierten Plugins sind im Ordner `plugins` zu finden.

```
1:  <site>
2:    <description
3:         url="http://www.fernuni-hagen.de/ps/prjs/EzUnit/update/">
4:       EzUnit Update Site
5:    </description>
6:    <feature
7:         url="features/org.projectory.ezunit.feature_1.0.0.jar"
8:         id="org.projectory.ezunit.feature"
9:         version="1.0.0"/>
10: </site>
```

Listing 35: Definition der Update-Site für das EzUnit-Feature

Diese Strukturen müssen nicht von Hand aufgebaut werden. Eclipse bietet nach einem Doppelklick auf die Datei `site.xml` dafür einen komfortablen Editor. Die Struktur ist hier nur für das bessere Verständnis der Einstellungsmöglichkeiten beschrieben.

5 Diskussion und verwandte Arbeiten

In diesem Kapitel werden die Ideen dieser Arbeit und die Implementierung durch das Plugin kritisch bewertet und in Beziehung zu anderen Arbeiten im gleichen Umfeld gestellt.

5.1 Bewertung der @MUT Annotation und des Plugins

Durch die explizite Verknüpfung von Testmethoden mit den getesteten Methoden über die @MUT Annotation wird eine einfache Möglichkeit geschaffen, um nach dem Fehlschlagen eines Testfalls die Markierung für einen Fehler an der potentiell fehlerverursachenden Stelle anzubringen, anstatt an dem „Symptom“ scheiternder Testfall. Dadurch werden logische Fehler von der Visualisierung her auf die selbe Stufe wie syntaktische und semantische (Typ-)Fehler gehoben.

Dazu wird dem Entwickler allerdings der Aufwand für die Erstellung der @MUT Annotation abverlangt. Durch die statische Analyse des Aufrufbaums werden ihm dabei zwar alle potentiell aufgerufenen Methoden zur Verfügung gestellt, die Einschränkung auf die Methoden, die wirklich einen Fehler verursachen können, muss aber doch noch vom Entwickler selber vorgenommen werden, um eine zu große Anzahl an potentiellen Fehlerursachen beim Fehlschlag des Testfalls zu vermeiden. Diese Eingrenzung muss allerdings nur einmal erfolgen und nicht nach unter Umständen langer Zeit bei jedem Fehlschlag der Testmethode auf der Suche nach der Ursache. Da die Zuordnung zur Annotation manuell erfolgt, besteht allerdings stets die Gefahr, dass @MUT Annotation und tatsächliche Implementierung divergieren.

Die Navigation zwischen Testmethode und getesteten Methoden kann sowohl bei der Fehlersuche als auch schon bei der Implementierung hilfreich sein. Der Zeitaufwand für die Anzeige der entsprechenden Menüpunkte darf dabei aber nicht zu hoch werden. Ist das Plugin so eingerichtet, dass es die getesteten Methoden aus dem statischen Aufrufbaum ermittelt, ist an eine Verwendung des Menüpunktes für die Navigation hin zur Testmethode nicht mehr zu denken. Dazu ist der Zeitaufwand für die Analyse zu hoch.

Die enge Integration in die Eclipse-Umgebung ist für die Akzeptanz durch den Entwickler sehr wichtig. Wird durch die Einbindung der Aufwand für die Erstellung der @MUT Annotation gering gehalten, wird er der Pflege weniger Widerstand entgegen bringen. Alle implementierten Erweiterungspunkte helfen diesen Aufwand gering zu halten.

5.2 Vergleich mit verwandten Arbeiten

Wie bereits im Kapitel über Eclipse angedeutet, gibt es sehr viele Erweiterungen für Eclipse. Einige beschäftigen sich auch mit einer besseren Integration von JUnit. Die wichtigsten sollen in diesem Kapitel beleuchtet werden.

5.2.1 Eclipse Test and Performance Tools Platform

Die Eclipse Test and Performance Tools Platform (TPTP) ist eines der Eclipse Top-Level Projekte. Im Rahmen dieses Projektes soll eine Plattform für die Einbindung von verschiedenen Test- und Performancewerkzeugen in die Eclipse Umgebung geschaffen werden. Dabei nennt [TPTP 2007] den gesamten Test- und Performance-Zyklus, vom Entwicklertest bis hin zur Analyse von Logdateien produktiver Anwendungen, als Projektumfang. Dieser wird dabei in vier Unterprojekte aufgebrochen:

- *TPTP Platform*[21]
- *Monitoring Tools*
- *Testing Tools*
- *Tracing and Profiling Tools*

Im Rahmen des Unterprojekts *Testing Tools* wird ein gemeinsamer Rahmen für verschiedene Testwerkzeuge geschaffen. Es ermöglicht die Vereinheitlichung der Ausführung und Auswertung von Tests mit Hilfe von verschiedenen Werkzeugen. Exemplarisch werden Werkzeuge für folgende Tests mitgeliefert

- Performance Test von Web-Anwendungen
- Automatisierte GUI Tests
- Manuelle Tests
- JUnit Tests

Die JUnit Werkzeuge im TPTP unterscheiden sich dabei von denen, die im Standard-Eclipse ausgeliefert werden im wesentlichen durch die standardisierte Anlage, Ausführung und Auswertung der Tests. In Abbildung 19 werden zum Beispiel das Ergebnis eines JUnit Tests und eines manuellen Tests nebeneinander dargestellt. Ein Erweiterungspunkt für *Test Run Listener* existiert in der TPTP nicht.

Auch wenn die durchgängige Darstellung von verschiedenen Arten von Tests gegenüber dem Benutzer sehr interessant ist und in Zukunft mit Sicherheit auch noch mehr Bedeutung gewinnen wird, macht es der fehlende Erweiterungspunkt für das Beobachten von ablaufenden Testfällen schwierig, direkt auf fehlgeschlagene Testfälle zu reagieren. Darüber hinaus bietet die TPTP keine Unterstützung für JUnit 4 Testfälle[22]. Aus diesem Grund ist ein integrierter Einsatz der @MUT Annotation und der TPTP im Moment nicht möglich. Ist die JUnit 4 Unterstützung gegeben, sollten die Möglichkeiten der Integration untersucht werden.

21 Bei der Benennung des Unterprojektes scheint den Verantwortlichen ein Fehler unterlaufen zu sein: Durch die Abkürzung und durch das ausgeschriebene Wort *Platform* ist dieses zwei mal im Namen enthalten.

22 Siehe Bug #154844 unter https://bugs.eclipse.org/bugs/

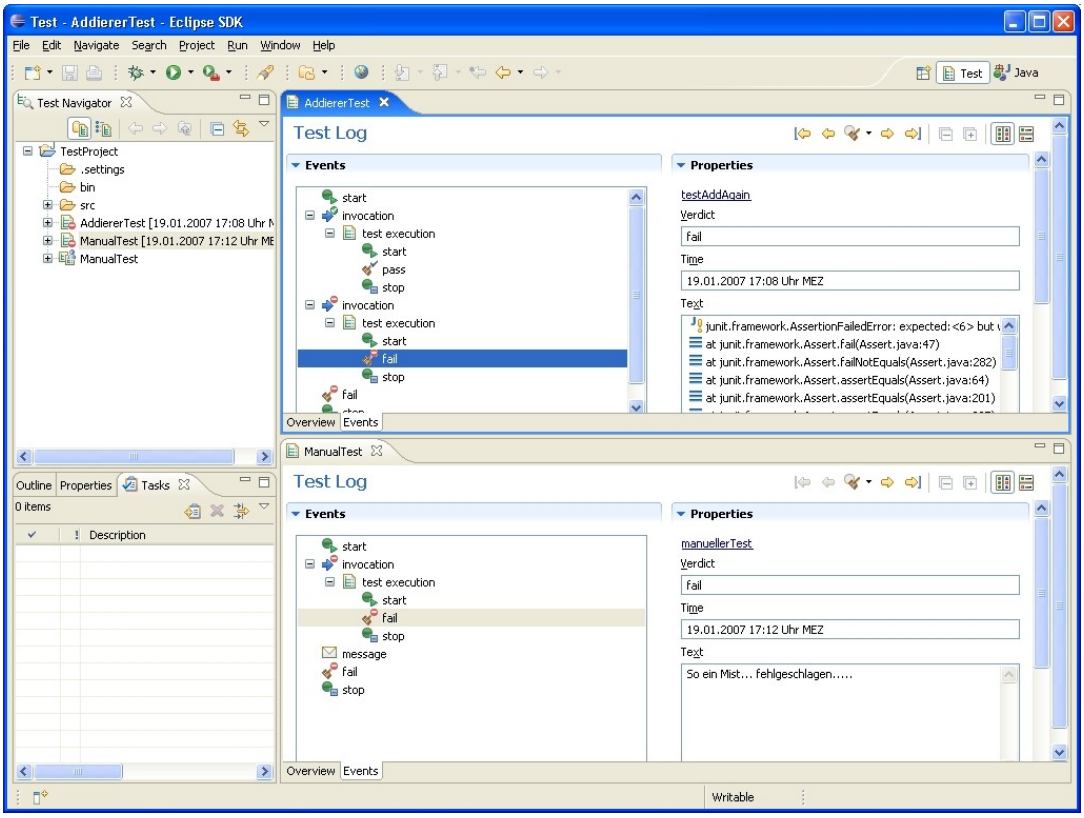

Abbildung 19: Eclipse Test and Performance Tools Platform - Ergebnisse eines manuellen und eines JUnit Tests

5.2.2 Continuous Testing

Die Idee des *Continuous Testing* wird unter diesem Namen von David Saff und Michael D. Ernst in [Saff & Ernst 2003] vorgestellt. Inspiriert von dem mittlerweile alltäglichen ständigen Kompilieren schlagen Saff und Ernst vor, Regressionstests ständig asynchron durchzuführen und damit die freien Kapazitäten des Prozessors zu nutzen, während der Programmierer entwickelt. Dadurch wird es möglich, den Entwickler sehr schnell auf logische Probleme im Quellcode aufmerksam zu machen und den Aufwand für deren Korrektur gering zu halten. Ursächlich für den geringen Aufwand ist dabei hauptsächlich die zeitliche Nähe zu den Änderungen, die das Problem wahrscheinlich verursachen.

Eine ähnliche Idee haben Erich Gamma und Kent Beck in [Beck & Gamma 2004] für ihre Einführung in die Eclipse-Plugin-Entwicklung auserkoren. Dort schildern sie in den Grundlagen, wie ein Plugin entsteht, das alle JUnit Tests in einem Projekt ausführt, so bald eine Datei in diesem Projekt editiert wird.

Im Rahmen statistischer Untersuchungen haben Saff und Ernst festgestellt, dass durch ständiges Kompilieren die Erfolgsquote für den erfolgreichen Abschluss einer Programmieraufgabe verdoppelt werden kann. *Contiuous Testing* kann diese Erfolgsquote zumindest im studentischen Umfeld verdreifachen.[23] Auch Beck bezeichnet *Continuous Testing* in [Beck & Andres 2005] auf Seite 51 als Verfeinerung für das *Test-First Programming*.

Für die Untersuchungen von Saff und Ernst kam ein Plugin für Eclipse zum Einsatz, das federführend von David Saff entwickelt wird. Es ist momentan in der Version 1.2.1 für Eclipse 3.1 unter [Saff 2006] verfügbar. Eine angekündigte Version 2.0 für Eclipse 3.2 ist bis Januar 2007 noch nicht verfügbar. Diese soll neben JUnit 3.8 Tests auch JUnit 4 Tests über die entsprechende Integration in Eclipse 3.2 unterstützen.

Wie auch in [Steimann et. al 2007] beschrieben, betrachtet Saff analog zu dieser Arbeit Unit Tests auf dem selben Level wie zum Beispiel die semantischen Prüfungen des Quellcodes im Sinne einer Typprüfung. Die beiden Ansätze ergänzen sich dabei, da sich *Continuous Testing* mit der Frage beschäftigt, wann Testfälle ausgeführt werden und diese Arbeit mit der Frage wie die Ergebnisse visualisiert werden. Auf Grund der fehlenden Verfügbarkeit der Version 2.0 des Eclipse Plugins für *Continuous Testing* wurde die Kombination der beiden Ansätze in dieser Arbeit nicht näher betrachtet.

5.2.3 moreUnit

Vera Wahler [Wahler 2006] beschreibt moreUnit als ein Eclipse-Plugin, das den Benutzer dabei unterstützen soll, mehr Unit Tests zu schreiben. Dies geschieht im wesentlichen durch die Markierung von Methoden, für die mindestens eine Testmethode existiert, und durch Navigationshilfen zwischen Testmethoden und getesteten Methoden.

23 Vgl. [Saff & Ernst 2004]

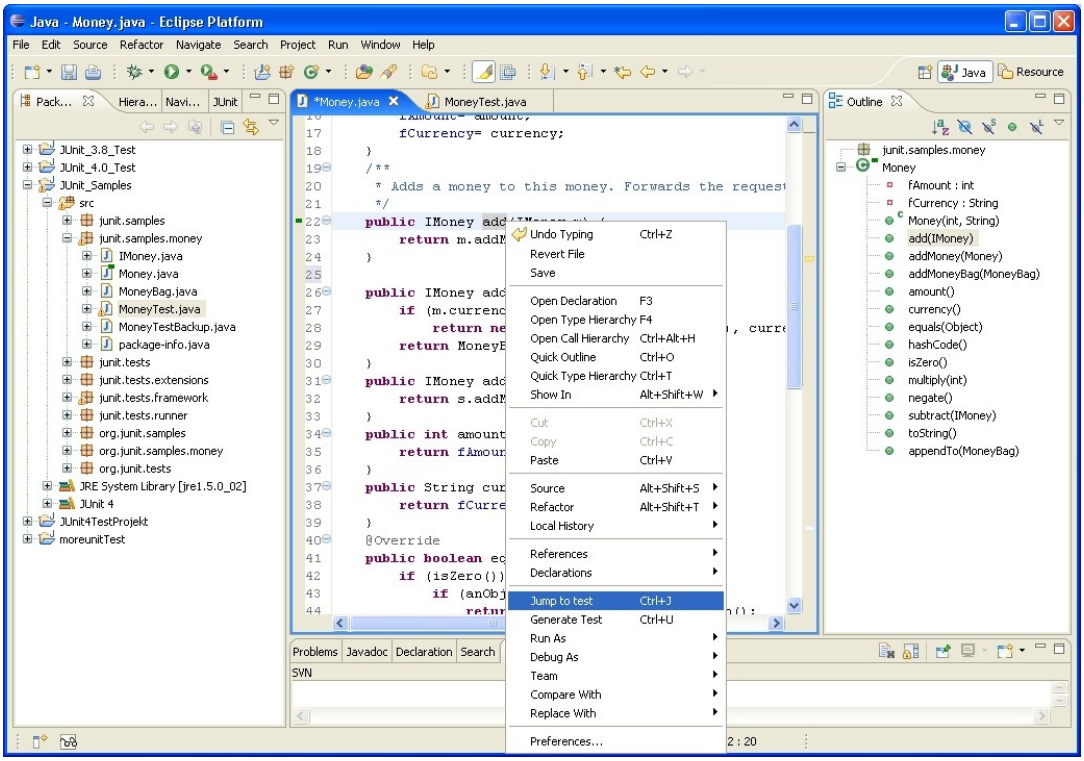

Abbildung 20: Markierung einer getesteten Methode durch moreUnit

So zeigt Abbildung 20 einen kleinen grünen Marker links im *Package Explorer* an der Klasse `Money` und links von der Zeile 22. Dieser Marker besagt, dass für diese Methode ein Test existiert. Zu diesem kann mit Hilfe des selektierten Menüpunktes *Jump to test* navigiert werden. Existiert noch kein Test, kann alternativ einer angelegt werden.

Die Verknüpfung zwischen Testmethoden und getesteten Methoden erfolgt bei moreUnit über Namenskonvention. Die Testmethoden zu einer Klasse werden stets in einer Klasse mit dem selben Namen und der (konfigurierbaren) zusätzlichen Erweiterung „Test“ gesucht. Der Name eine Testmethode setzt sich aus dem (ebenfalls konfigurierbaren) Prefix „test“, dem Namen der getesteten Methode und einer frei wählbaren zusätzlichen Zeichenfolge zusammen. Hält man sich nicht an diese Konventionen, kann moreUnit keinen Zusammenhang zwischen Test- und getesteten Methoden herstellen. Die entsprechenden Markierungen sind dann nicht vorhanden und eine Navigation über das Kontextmenü ist nicht möglich.

Eine Markierung von Methoden im *Package Explorer*, die bereits oder noch nicht getestet werden findet in dem Plugin dieser Arbeit nicht direkt statt. Lediglich über die Navigation zur Testmethode kann festgestellt werden, dass noch keine Testmethode zu einer Methode existiert. Auch eine Erstellung eines Testmethodenrumpfs ist über dieses Plugin nicht vorgesehen.

Durch die Verknüpfung über Namenskonventionen hat moreUnit den Nachteil, dass nur 1 : n Beziehungen zwischen getesteten und Testmethoden ausgedrückt werden können, das heißt es kann nicht ausgedrückt werden, dass eine Testmethode mehrere Methoden auf ihre korrekte Funktion überprüft. Dies ist in der Realität aber quasi immer der Fall und kann hingegen mit der @MUT Annotation dieser Arbeit dargestellt werden. Da moreUnit gar nicht vorsieht, dass eine Testmethode mehrere Methoden testet, ist dort auch keine Untersuchung der aufgerufenen Methoden nötig. Über den Namen der Methode ist bereits definiert, welche Methode durch sie getestet wird.

Eine Kombination von moreUnit und dem Plugin dieser Arbeit ist schwierig. moreUnit setzt auf JUnit 3.8 auf, diese Arbeit auf JUnit 4.

5.2.4 Testabdeckungen

Die Testabdeckung (*test coverage*) wird häufg auch als *Code Coverage* bezeichnet. Ihr Ziel ist das Auffinden von Programmteilen, die nicht getestet werden, das Schreiben neuer Testfälle und das Bestimmen des Anteils am Quellcode, der von Testfällen überprüft wird. Letzteres ist ein indirektes Merkmal für die Code-Qualität.[24]

Um die Abdeckung automatisch zu bestimmen, greifen Werkzeuge für die Ermittlung der Testabdeckung, genauso wie das in dieser Arbeit entwickelte Plugin, auf die dynamische oder statische Untersuchung von Aufrufstrukturen zurück. So setzt zum Beispiel [Dmitriev 2006] auf die `java.lang.instrument` API, um den Bytecode einer Klasse beim Laden zu modifizieren und Methodenaufrufe an das Werkzeug zu übermitteln.

Einige Werkzeuge wie zum Beispiel Clover bieten auch an, Methoden zu kennzeichnen, die momentan nicht von Testmethoden aufgerufen werden. Damit existiert eine Ähnlichkeit zu dem in dieser Arbeit entwickelten Plugin. Ist die Navigation „Gehe zur Testmethode...“ für eine Methode leer, wurde damit ebenfalls festgestellt, dass eine Methode nicht getestet wird.

Während der Fokus der Testabdeckung auf der Betrachtung der allgemeinen Testsituation liegt, befasst sich diese Arbeit eher mit der Visualisierung einzelner, fehlgeschlagener Tests. Allerdings kommen für die automatische Ermittlung der getesteten Methoden ähnliche Konzepte zum Einsatz, so dass es sich lohnt, verschiedene Techniken, die im Rahmen der Testabdeckung eingesetzt werden, genauer zu betrachten und auf ihre Tauglichkeit für dieses Plugin zu untersuchen.

24 Vgl. [Cornett 2007]

6 Ausblick und Schlußbetrachtungen

Bevor diese Arbeit mit einem kurzen Fazit zusammengefasst wird, sollen in diesem Kapitel verschiedene Aspekte skizziert werden, deren weitere Untersuchung spannend zu sein verspricht.

6.1 Erweiterungspunkt für die Bestimmung der getesteten Methoden

In dieser Arbeit sind im wesentlichen zwei Ansätze[25] für die Ermittlung der durch eine Testmethoden getesteten Methoden angewendet worden: die statische Analyse des Aufrufbaums der Testmethode und die Untersuchung der @MUT Annotation. Darüber hinaus gehend sind aber weitere Ansätze denkbar, so zum Beispiel ein Tracing der aufgerufenen Methoden zur Laufzeit eines Testfalls oder die Einbeziehung von Wahrscheinlichkeiten für Fehler (zum Beispiel über die Länge einer Methode) in statisch ermittelten Methoden bei der Analyse des Aufrufbaums.

Ein Teil der Ansätze ist eher für die Bestimmung der Methoden geeignet, die nach einem fehlgeschlagenem Testfall mit einem Marker versehen werden müssen (zum Beispiel der Tracing-Ansatz) während andere Ansätze besser für die Unterstützung bei der Annotationserstellung zu gebrauchen sind (zum Beispiel die Analyse des Aufrufbaums). Eine Kombination von verschiedenen Ansätzen verspricht dabei die genaueste Eingrenzung der tatsächlichen Problemstelle.

Um die einfache Einbindung der verschiedenen Ansätze zu ermöglichen, ist die Definition eines Erweiterungspunktes für die Ermittlung von getesteten Methoden sinnvoll. In den Voreinstellungen des Plugins sollten dann jede einzelne Implementierung des Erweiterungspunktes sowohl für die Markierung der Methoden auf die ein Testfall fehlgeschlagen ist, als auch für die Unterstützung zur @MUT Annotationserstellung aktivierbar beziehungsweise deaktivierbar sein. Die entsprechenden Zugriffe für die Erzeugung der Marker beziehungsweise für die Auflistung der getesteten Methoden im Dialog zur Bearbeitung der Annotation und in den *Completion Proposals* müssten diese Einstellungen berücksichtigen.

6.2 Verknüpfung mit ständiger Testausführung

Die im Kapitel 5.2.2, „Continuous Testing“, beschriebenen Ideen von David Saff ergänzen sich mit den Ideen dieser Arbeit in sofern, dass beide Arbeiten logische Fehler im Quellcode auf die selbe Stufe wie syntaktische und semantische (Typ-)Fehler heben, allerdings in unterschiedlichen Dimensionen. Saff beschäftigt sich mit der Frage, wann Tests ausgeführt werden sollen, diese Arbeit mit der Visualisierung der fehlgeschlagenen Tests. In [Bouillon et. al 2007] wird dabei noch ein weiterer Aspekt herausgestellt, der eine nähere Untersuchung interessant erscheinen lässt. Um nach einer Änderung am Quellcode einer

25 Die Einschränkung „im wesentlichen“ bezieht sich auf die alternative Nutzung des AST und von *Regular Expressions* für die Untersuchung der @MUT Annotation

Methode erkennen zu können, welches die minimale Menge an Testfällen ist, die erneut ausgeführt werden muss, um sicherzustellen, dass die Änderung zu keinem Fehler geführt hat, muss auch hier eine Beziehung zwischen getesteter und zugehörigen Testmethoden ermittelt werden. Dazu können sowohl die bereits implementierten Ansätze über die @MUT Annotation und den statischen Aufrufbaum der Testmethode, als auch die weiter oben skizzierten alternativen Möglichkeiten zum Einsatz kommen.

6.3 Eclipse Integration

Durch die Nutzung der Eclipse-Mechanismen der Marker, deren Visualisierung und der zugehörigen *Resoultion Provider*, des AST, der Erzeugung von Typhierarchien, der Bereitstellung von *Completion Proposals* und Kontextmenü-Ergänzungen für die Navigation ist dieses Plugin eng in die Plattform integriert.

Mit der Schaffung der *Test and Performance Tools Platform* (TPTP) entsteht allerdings ein neuer Rahmen, in dem auch die JUnit Integration voraussichtlich in späteren Versionen aufgehen wird und die noch mehr Möglichkeiten für die Integration bieten könnte. So muss bei der existierenden Form des Testrun-Listeners die Eclipse-Repräsentation der Testmethode zum Beispiel mühevoll aus String-Werten extrahiert werden. Möglicherweise bieten da kommende Erweiterungspunkte der TPTP einfachere Möglichkeiten.

Aber auch die Integration in die Standard-Java-Development-Tools bietet noch viel Potential für Verbesserungen. Es findet keine automatische Erzeugung der passenden @MUT Annotation bei der Erstellung eines Testklasse zu einer Methode über die entsprechenden Eclipse-Wizards statt (stattdessen wird nur ein Javadoc-Kommentar erzeugt, der auf die getestete Methode verweist, vgl. Kapitel 2.3.1), JUnit 3.8 Tests, die in Eclipse weiterhin erstellt werden können, werden gar nicht unterstützt und die Untersuchung der Aufrufstrukturen einer Testmethode inklusive der Analyse der Klassenhierarchien führt bei großen Projekten zu Performance-Problemen. Weitere Ansätze für Verbesserungen ergeben sich mit Sicherheit aus der praktischen Anwendung des Plugins.

6.4 Visualisierung in einem Side-by-Side-Editor

Die Navigation zwischen getesteten Methode und Testmethode vereinfacht die Fehlerbehebung bereits. Noch einfacher wäre es aber, wenn gar keine Navigation nötig wäre, sondern Test- und getestete Methode in einem Editor ähnlich wie in Abbildung 21 dargestellt würden.

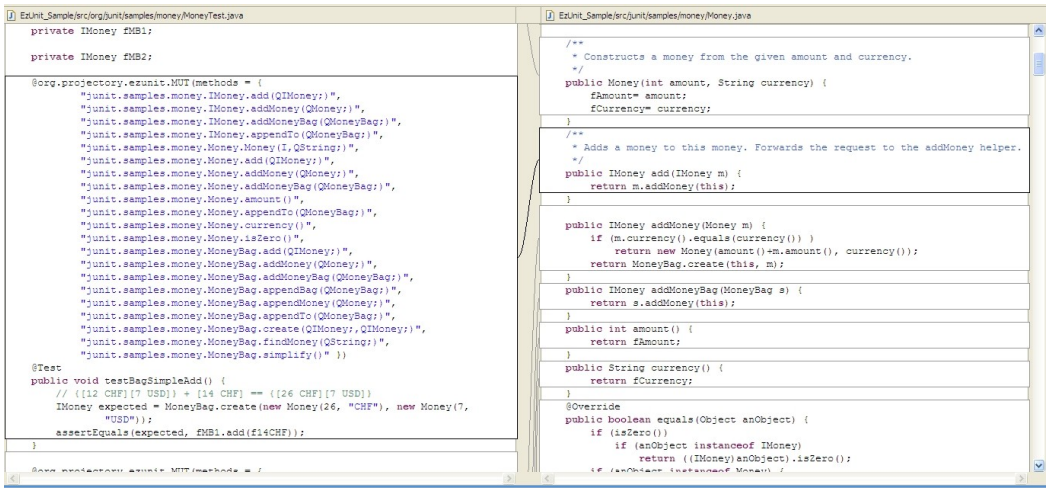

Abbildung 21: Side-by-Side Darstellung von Testmethode und getesteter Methode

Eine Darstellung für zwei Klassen gleichzeitig existiert in Eclipse über den *Compare View* zwar bereits, die Unterstützung für die Programmierung (*Code Completion*, Javadoc, etc.) ist aber für diese Editoren nicht verfügbar. Die dazu nötigen Klassen sind allerdings sehr stark mit dem Eclipse-Java-Editor verwoben und nicht für eine Wiederverwendung gedacht. Auch bietet dieser keine Möglichkeit sich mehrfach in einen Editor einbinden zu lassen.

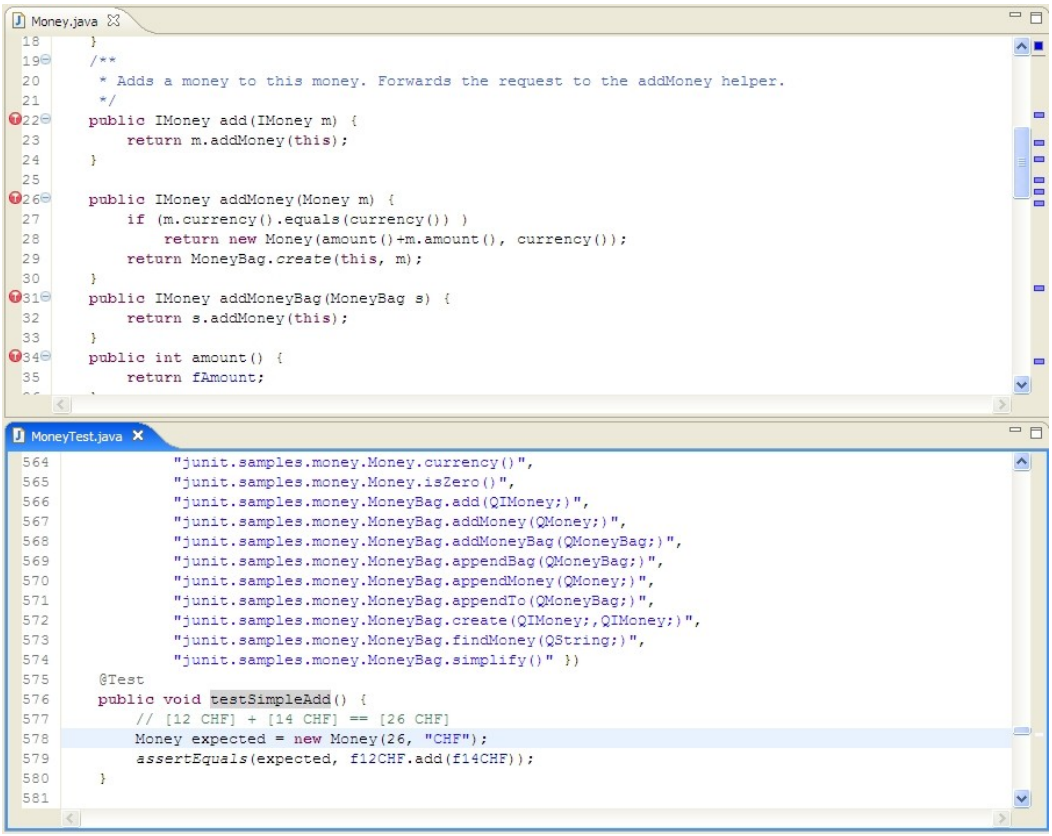

Abbildung 22: Zwei Editoren untereinander in Eclipse

Einfacher denkbar wären aber zum Beispiel die vom Programm gesteuerte Darstellung zweier mehr oder weniger unabhängigen Java-Editoren über- beziehungsweise nebeneinander, wie sie zum Beispiel in Abbildung 22 zu sehen ist.

Eine dritte Möglichkeit könnte auf einen Eclipse *View* aufsetzen, der die Testmethoden, beziehungsweise die getesteten Methoden zur momentan selektierten Methode darstellt. Welche Möglichkeit dabei die komfortabelste ist wird im Rahmen dieser Arbeit nicht mehr betrachtet.

6.5 Überprüfung der @MUT Annotationen

Auch wenn @MUT Annotationen ausschließlich über die Unterstützung durch *Completion Proposals* und den Dialog zum Bearbeiten einer @MUT Annotation erzeugt wurden, können sich durch das Ändern von Testmethoden und getesteten Methoden Fehler in die Annotation einschleichen. So können dort Methoden auftauchen, die entweder gar nicht mehr existieren, oder aus dem Aufrufbaum der Testmethode entfernt wurden.

Ein *Builder* könnte bei der Erzeugung eines Projektes solche Fälle für existierende @MUT Annotation überprüfen. Existiert eine Methode, die in der Annotation vorkommt, nicht mehr im Aufrufbaum, wird eine Warnung angegeben. Als *Quick Fix* zu dieser Warnung wird angeboten, den entsprechenden Eintrag aus der Annotation zu entfernen.

6.6 Akzeptanz in der praktischen Anwendung

Die Basis für die praktische Anwendung des in dieser Arbeit entwickelten Ansatzes für die Markierung der logischen Fehler im Quellcode ist mit dem entwickelten Plugin vorhanden. Auf dieser Basis können Untersuchungen zur Akzeptanz der zusätzlichen Arbeit für die Erstellung der @MUT Annotation durch den Entwickler und Untersuchungen zum Aufwand für die Fehlersuche mit und ohne der passenden Markierung vorgenommen werden. Die Ergebnisse dieser Untersuchungen können sowohl in die Methoden zur Eingrenzung der Fehlerursache, als auch in die Art und Weise der Visualisierung der Fehler zurückfließen.

6.7 Fazit

Der in dieser Arbeit beschriebene Ansatz zur Visualisierung von logischen Fehlern im Quellcode geht weg von der Markierung von Symptomen der logischen Fehler, wie sie durch ein Auflisten der fehlgeschlagenen Testfälle üblich ist. Stattdessen wird versucht, die tatsächliche Ursache für einen fehlgeschlagenen Fehler in der Art mit einem Marker zu versehen, wie auch syntaktische und semantische (Typ-)Fehler markiert werden. Dabei wird mit der @MUT Annotation auf eine explizite Verknüpfung von Testmethoden und getes-

teten Methoden gesetzt, die eine entsprechende Markierung ermöglicht. Durch eine statische Analyse der Aufrufbaums einer Testmethode wird der Entwickler bei der Erstellung der @MUT Annotation unterstützt.

Das im Rahmen dieser Arbeit entwickelte Plugin unterstützt diese Annotation und die Visualisierung der Testfälle und bietet die technische Basis für Erweiterungen, die die potentiellen Fehlerursachen weiter einschränken können, wie zum Beispiel dynamische Untersuchungen der von einer Testmethode aufgerufenen Methoden. Durch die Kombination verschiedener Ansätze für die Analyse soll der Entwickler auf Basis dieser Arbeit möglichst zielstrebig zur Ursache seines fehlgeschlagenen Testfalls geführt werden.

A Grundlagen der Eclipse Plattform

Dieser Anhang soll keine grundlegende Einführung in Eclipse geben. Für diese sei zum Beispiel auf [Bach 2007], [Beck & Gamma 2004] oder [Beck & Gamma 2004 de] verwiesen. Statt dessen sollen hier lediglich Themen skizziert werden, die in den angegebenen Quellen nach der Meinung des Autors nicht oder nicht ausführlich genug dargestellt werden, zum Verständnis des entwickleten Plugins allerdings nötig sind. Im Kapitel „Das Plugin“ist bei Bedarf explizit auf diesen Abschnitt oder eine der Quellen verwiesen.

A.1 Classpath Repräsentation in Eclipse

Beim Start eines Java Programms kann über den Kommandozeilenparamter `-classpath` angegeben werden, in welchen Verzeichnissen und Bibliotheken (JAR-Dateien) nach Klassen gesucht werden soll, die zur Laufzeit des Programmes diesem zur Verfügung stehen.[26] In Eclipse werden die einzelnen Elemente, aus denen sich dieser Klassenpfad zusammensetzt, als Objektinstanzen dargestellt, denen gemeinsam ist, dass ihre zugrunde liegende Klasse das Interface `org.eclipse.jdt.core.IClasspathEntry` implementiert. In [JDT API] wird bei der Beschreibung des Interfaces `IClasspathEntry` dabei zwischen fünf Typen unterschieden:

- Quell-Ordnern in Projekten, die *.java* Dateien enthalten, *Source Classpath Entries* genannt..
- Ordner oder JAR-Bibliotheken, die *.class* Dateien enthalten, *Library Classpath Entries* genannt.
- Referenzen zu anderen Java-Projekten, *Required Project Classpath Entries* genannt.
- Referenzen auf ein Projekt oder eine Bibliothek, die mit einer Variable beginnen, *Variable Classpath Entries* genannt.
- Benannte Container, die andere IClasspathEntry-Typen enthalten können, *Classpath Container Entries* genannt.

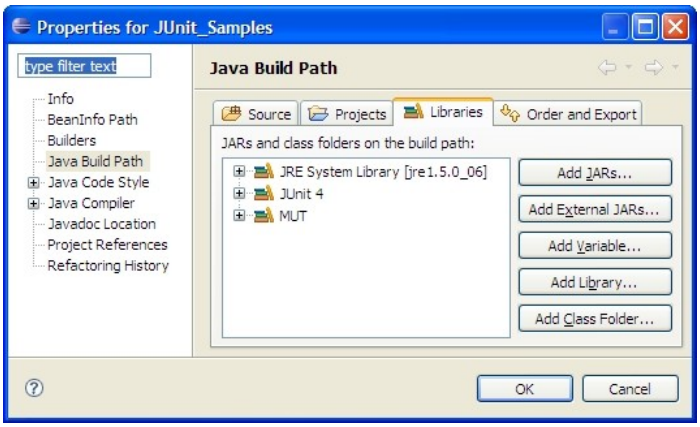

Abbildung 23: Bearbeitung des Erstellungspfades eines Java Projekts

26 Vergleiche [Ullenboom 2007], Kapitel 8.5.2

Abbildung 23 zeigt den Dialog für die Bearbeitung der Elemente des Erstellungspfades eines Java Projekts in Eclipse. Dabei fügen die folgenden Schaltflächen *Classpath Entries* vom jeweils angegebenen Typ hinzu:

- Reiter „Source“: *Source Classpath Entries*
- Reiter „Projects“: *Project Classpath Entries*
- „Add JARs“, „Add External JARs“ und „Add Class Folder“: *Library Classpath Entries*
- „Add Variable“: *Variable Classpath Entries*
- „Add Library“: *Container Classpath Entry*

Im folgenden wird das Wort Bibliothek nicht weiter für *Container Classpath Entries* verwendet, um eine Verwechslung mit dem Begriff JAR-Bibliothek zu vermeiden, der lediglich eine einzelne JAR Datei als Sammlung von Klassen meint.

Um in ein Java-Projekt JAR-Bibliotheken einzubinden, die von einem Plugin mitgeliefert werden, eignen sich vor allem die beiden letzten *Classpath Entry* Varianten, da sie es ermöglichen, einem Projekt eine Referenz hinzuzufügen, die je nach Workspace, in dem das Projekt geöffnet ist (zum Beispiel auf unterschiedlichen Rechnern), unterschiedlich aufgelöst wird.

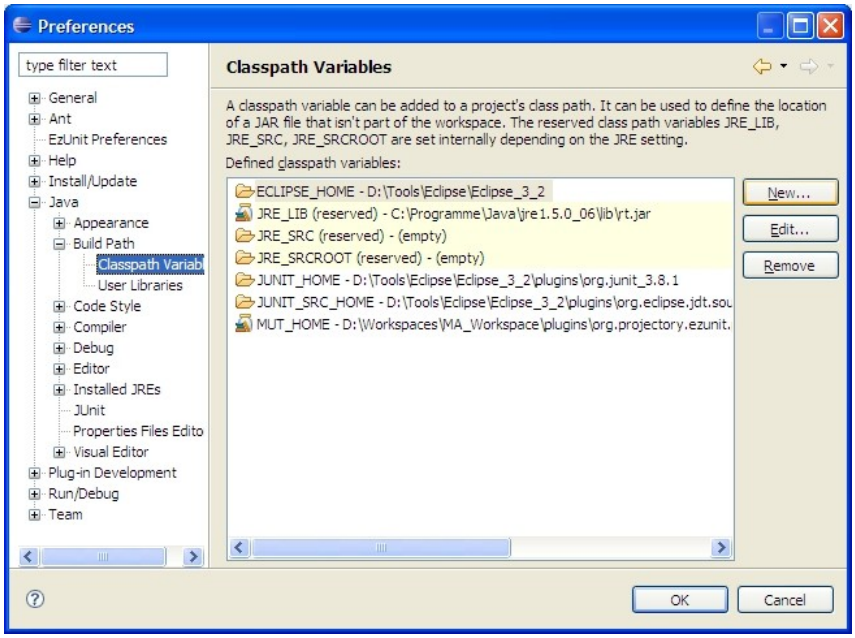

Abbildung 24: Erstellen einer Variable zum Hinzufügen eines Variable Classpath Entries

Die Variablen für einen *Variable Classpath Entry* können dabei entweder manuell angelegt werden (siehe Abbildung 24) oder über den Erweiterungspunkt `org.eclipse.jdt.core.classpathVariableInitializer` von einem Plugin erstellt werden. Im folgenden soll der Fokus vor allem auf der zweiten Variante liegen.

Listing 36 zeigt eine entsprechende Erweiterung, die eine Variable mit dem Namen „MUT_LIBRARY“ definiert. Die Initialisierung der Variable geschieht in der Klasse `MUTClasspathVariableInitializer`, die in den Zeilen 4 und 5 angegeben ist. Diese muss eine Kindklasse der Klasse `org.eclipse.jdt.core.ClasspathVariableInitializer` sein und deren abstrakte Methode `initialize(...)` implementieren.

```
1: <extension point="org.eclipse.jdt.core.
2:                   classpathVariableInitializer">
3:     <classpathVariableInitializer
4:        class="org.projectory.ezunit.buildPath.
5:               MUTClasspathVariableInitializer"
6:        variable="MUT_LIBRARY"/>
7: </extension>
```

Listing 36: Defintion des Classpath Variable Initilizers für die Variable MUT_LIBRARY

Listing 37 zeigt eine Implementierung dieser abstrakten Methode, die die Variable „MUT_LIBRARY“ an die JAR-Bibliothek `MUT.jar` im Verzeichnis `lib` des EzUnit-Plug-ins bindet.

```
1:  public void initialize(String variable) {
2:   Bundle bundle = EzUnitPlugin.getDefault().getBundle();
3:   if (bundle != null) {
4:    try {
5:         URL local = FileLocator.
6:            toFileURL(bundle.getEntry("/lib/MUT.jar"));
7:     String fullPath = new
8:            File(local.getPath()).getAbsolutePath();
9:         IPath path = Path.fromOSString(fullPath);
10:        JavaCore.setClasspathVariable(variable, path, null);
11:   } catch (IOException e) {
12:     ...
13:   } catch (JavaModelException e) {
14:         ...
15:   }
16:  }
17: }
```

Listing 37: Beispielhafte Implementierung der Methode initalize(....) eines Classpath Variable Initilizers für die Variable MUT_LIBRARY

Analog zu den *Variable Classpath Entries* können auch *Container Classpath Entries* manuell angelegt werden. Der entsprechende Punkt in den Eclipse-Voreinstellungsdialog nennt analog zur Konfiguration des Erstellungspfades eines Eclipse-Projekts *User Libraries*. Abbildung 25 zeigt eine manuell angelegte Bibliothek „MUT_USER_LIBRARY“, die lediglich eine JAR-Bibliothek enthält.

Für die automatische Auflösung eines *Classpath Container Entries* kommt die Erweiterung `org.eclipse.jdt.core.classpathContainerInitializer` zum Einsatz. Im Unterschied zu Variablen, die direkt nach der Definition einer `classpathVariableInitializer` Erweiterung im Dialog für das Hinzufügen einer Variable zum Erstellungspfad zur Verfügung stehen, ist dazu für *Classpath Container Entries* noch eine Erweiterung des

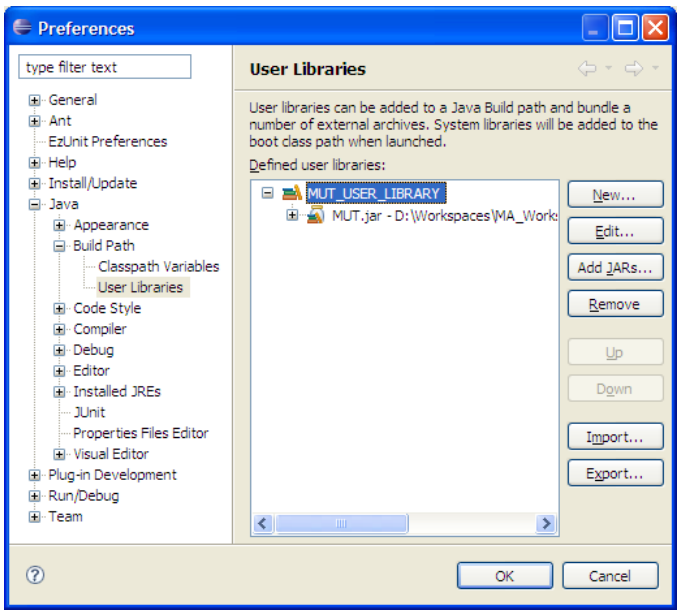

Abbildung 25: Anlegen einer MUT User Library

Punktes `org.eclipse.jdt.ui.classpathContainerPage` hinzuzufügen. Diese definiert einen Namen, unter dem der Container angezeigt wird und stellt eine Wizard-Seite zur Verfügung in der weitere Einstellungen zu dem *Container* beim Hinzufügen zu einem Projekt abgefragt werden können.

Für ein Beispiel zur Anwendung dieser beiden Erweiterungspunkte sei an dieser Stelle auf das Kapitel 4.3, „Der MUT Classpath Container“, verwiesen.

A.2 Typ-Hierachie

Mit dem *Hierarchy View* ist es in Eclipse möglich, den Baum der Kind- und Eltern-Klassen einer ausgewählten Klasse anzuzeigen. Abbildung 26 zeigt als Beispiel die Hierarchie der Eltern-Klassen der Klasse `NormalAnnotation`.

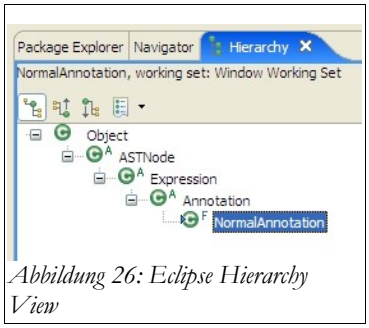

Abbildung 26: Eclipse Hierarchy View

Diese Hierarchie ist mit Hilfe der Methode `org.eclipse.jdt.core.IType.newTypeHierarchy(IProgressMonitor)` auch über die Eclipse API verfügbar. Das von dieser Methode zurückgegebene Objekt vom Typ `org.eclipse.jdt.core.ITypeHierarchy` bietet Methoden für die direkte Abfrage aller Eltern- und Kind-Klassen. Ein Bei-

spiel für die Verwendung findet sich in dem im Rahmen dieser Arbeit entwickelten Plugin in der Methode `org.projectory.ezunit.internal.methodwrapper.CalledMethodExtractorASTVisitor.getOverwritingMethods(IMethod)`.

Die Berechnung der Typ-Hierarchie zu einer Klasse ist allerdings recht aufwendig. Ein Aufruf der Methode `newTypeHierarchy` kann durchaus einige Sekunden dauern. Über den als Parameter erwarteten `IProgressMonitor` ist es möglich, dem Benutzer sowohl den Fortschritt zu visualisieren, als auch ihm die Möglichkeit zu geben, den Vorgang abzubrechen.

A.3 Hinzufügen von Menüpunkten zu einem Kontextmenü

Über den Erweiterungspunkt `org.eclipse.ui.popupMenus` können in Eclipse Kontextmenüs um zusätzliche Einträge erweitert werden. [Beck & Gamma 2004] beschreibt diesen Erweiterungspunkt auf den Seiten 124-128, [Beck & Gamma 2004 de] auf den Seiten 138-141.

Dieser Erweiterungspunkt unterscheidet dabei zwischen zwei verschiedenen Arten von bereitgestellten Menüpunkten: *Object Contributions* und *Viewer Contributions*. Während eine *Viewer Contribution* an einen Eclipse-View oder -Editor gebunden ist und in dessen Kontextmenüs unabhängig von der momentanen Selektion erscheint, steht eine *Object Contribution* in jedem Kontextmenü von jedem Editor und jedem View zur Verfügung, wenn ein Objekt von einem bestimmten Typ, zum Beispiel eine Java-Methode ausgewählt ist.

Listing 38 zeigt die Definition einer *Viewer* und einer *Object Contribution*. Beide definieren in den Zeilen 6 bis 11 beziehungsweise 16 bis 21 eine Action, die dem Menüpunkt entspricht. Das Attribut `id` identifiziert die Action dabei eindeutig. Das Attribut `name` verweist auf die Datei `plugin.properties`, und steht für die Bezeichnung unter der die Action im Menü erscheint. Das Attribut `menubarPath` gibt eine Position im Kontextmenü an. Die Position `additions` sollte von jedem Kontextmenü vorgesehen sein. Andere Werte sind von der Definition des Kontextmenüs abhängig.

Das Attribut `class` gibt den voll qualifizierten Namen einer Klasse an, die das Verhalten der Action beinhaltet. Je nach Typ der *Contribution* muss diese Klasse eins der folgenden Interfaces implementieren:

```
1: <extension
2:  point="org.eclipse.ui.popupMenus">
3:   <objectContribution
4:       id="org.projectory.ezunit.iMethodContribution"
5:       objectClass="org.eclipse.jdt.core.IMethod">
6:     <action
7:         class="org.projectory.ezunit.internal.
8:                contribution.MUTDialogAction"
9:         id="org.projectory.ezunit.mutDialog"
10:        label="%editMUTs"
11:        menubarPath="additions"/>
12:  </objectContribution>
13:  <viewerContribution
14:      id="org.projectory.ezunit.plugin.viewerContrib"
15:      targetID="#CompilationUnitEditorContext">
16:    <action
17:        class="org.projectory.ezunit.internal.
18:               contribution.MUTDialogAction"
19:        id="org.projectory.ezunit.viewerMutDialog"
20:        label="%editMUTs"
21:        menubarPath="additions"/>
22:  </viewerContribution>
23:</extension>
```

Listing 38: Definition von zwei Menüpunkten, die zu Kontextmenüs hinzugefügt werden

- `org.eclipse.ui.IObjectActionDelegate` falls es sich um eine *Object Contribution* handelt
- `org.eclipse.ui.IViewActionDelegate` falls es sich um eine *Viewer Contribution* zu einem *View* handelt
- `org.eclipse.ui.IEditorActionDelegate` falls es sich um eine *Viewer Contribution* zu einem *Editor* handelt

Der wesentliche Unterschied zwischen den beiden Arten liegt in den Attributen `objectClass` (Zeile 5) und `targetID` (Zeile 15). Während für eine *Object Contribution* auf diese Art und Weise in Form eines voll qualifizierten Klassennamens angegeben wird, für welche Art von Objekten sie angezeigt werden soll, wird für eine *Viewer Contribution* angegeben in welchen Kontextmenüs sie angezeigt werden sollen. Dazu wird die ID angegeben, unter der das Kontextmenü registriert wurde. Diese entspricht in der Regel des definierende Plugins, es gibt aber, wie im Beispiel, Sonderfälle. Für die Details dazu sei aber auf [Beck & Gamma 2004], Seite 126 verwiesen.

A.4 Methodensignaturen

[Gosling et. al 1996] und [Gosling et. al 2000] beschreiben den Begriff der Signatur einer Methode übereinstimmend jeweils im Kapitel 8.4.2 recht einfach. Sie besteht in Java aus dem Namen einer Methode und der Anzahl und den Typen ihrer Parameter. In der zur

Java-Version 5.0 gehörenden dritten Ausgabe des Buches ([Gosling et. al 2005]), ist der Kern der Definition zwar identisch, aber durch die Einführung der *Generics* formal etwas komplexer.

Die interne Darstellung dieser Typen wird in [Lindholm & Yellin 1999] im Kapitel 4.3.2 beschrieben. Dort wird zwischen drei Typen unterschieden, die jeweils unterschiedlich dargestellt werden.

- *BaseType*: primitiver Datentyp, dargestellt durch ein einzelnes Zeichen. Die Zuordnung kann folgender Tabelle entnommen werden:

BaseType Character	***Datentyp***
B	byte
C	char
D	double
F	float
I	integer
J	long
S	short
Z	boolean

- *ObjectType*: Klassen- oder Interface-Typ, dargestellt in der Form

  `L<classname>;`

  <classname> ist dabei ein voll qualifizierter Klassen- oder Interface-Name. Statt der üblichen Notation mit einem „.“ als Trennzeichen wird in der internen Repräsentation ein „/“ eingesetzt.

- *ArrayType*: Feld-Typ, dargestellt in der Form

  `[ComponentType`

  ComponentType ist dabei entweder ein *BaseType*, ein *ObjectType* oder wieder ein *ArrayType*

Um diese Darstellung zu verdeutlichen enthält folgende Tabelle ein paar Beispiele:

Java Quellcode Notation	***Interne Notation***
int	I
int[ ]	[I
double[ ][ ][ ]	[[[D
java.lang.Object	Ljava/lang/Object;
java.lang.Object[ ]	[Ljava/lang/Object;

Eine ähnlich Form der Parametertyp Darstellung verwendet die Eclipse API für das Auffinden von Methoden in einer Klasse. Im Interface `org.eclipse.jdt.core.IType` ist dazu die Methode `getMethod(String, String[])` deklariert. Sie erwartet als Parameter den Namen der Methode und die Typen in einer Notation, die der ober erläuterten

sehr ähnlich ist. Diese Notation inklusive der feinen Unterschiede die Eclipse an dieser Stelle macht ist in der API-Dokumentation der Klasse `org.eclipse.jdt.core.Signature` beschrieben.

Während sich die Darstellung der *ArrayTypes* und der *BaseTypes* nicht unterscheidet, kommt in der Eclipse-Darstellung der *ObjectTypes* der gewohnte „.“ als Trennzeichen zum Einsatz. Man schreibt also `Ljava.lang.Object;`. Darüber hinaus sieht Eclipse noch eine Darstellung für nicht-aufgelöste Typen vor. Nicht-aufgelöst bedeutet dabei, dass statt dem voll qualifizierten Namen, der sich durch die Auflösung über die importierten Typen einer Klasse ergibt, der lokal verwendete Name eines Typs zum Einsatz kommt. Diese Typen werden in der Form `Q<Name>;` dargestellt. Die nicht-aufgelöste Variante kommt zum Einsatz, wenn eine Klasse im Quellcode vorliegt. Für binär vorliegende Klassen, kommt die aufgelöste Variante zum Einsatz.

Klarer wird auch dies an einem Beispiel. Ausgangspunkt für dieses Beispiel ist das Objekt `demoKlasseType` vom Typ `org.eclipse.jdt.core.IType`, das die in Listing 39 dargestellte Klasse DemoKlasse repräsentiert. Um eine Referenz auf ein `org.eclipse.jdt.core.IMethod` Objekt zu bekommen, das die Methode `setPoint(Point)` repräsentiert muss die Methode `getMethod(String, String[])` mit unterschiedlichen Parameters aufgerufen werden:

- `demoKlasseType.getMethod("setPoint", new String[] {"QPoint;"})`
  wenn die Klasse DemoKlasse im Quellcode vorliegt.
- `demoKlasseType.getMethod("setPoint", new String[] {"Ljava.awt.Point;"})`
  wenn die Klasse nicht im Quellcode vorliegt, also zum Beispiel nur die .class Datei in einer JAR-Bibliothek vorhanden ist.

```
1: import java.awt.Point;
2: public class DemoKlasse {
3:       private Point point = null;
4:       public Point getPoint() {
5:              return point;
6:       }
7:       public void setPoint(Point point) {
8:              this.point = point;
9:       }
10:}
```

Listing 39: Beispielklasse für die Erläuterung der Eclipse-internen Typdarstellung

Da es sich bei der Methode `getMethod(String, String[])` um eine so genannte *handle-only* Methode handelt, wird in jedem Fall ein IMethod-Objekt zurück gegeben. Die durch das Objekt dargestellte Methode muss allerdings nicht zwingend existieren. Die Existenz lässt sich mit der Methode `exists()` des IMethod-Objekts überprüfen.

Dass die erste Anfrage ein IMethod-Objekt zurück gibt, das nicht existiert wenn die Klasse nur in binärer Form vorliegt, ist nicht weiter überraschend. Der Typ `Point` ist nicht zwingend eindeutig auflösbar. Es wäre ja denkbar, dass neben der Methode `setPoint(java.awt.Point)` auch noch eine Methode `setPoint(org.eclipse.swt.graphics.Point)` existiert.

Dass die zweite Anfrage allerdings bei einer im Quellcode vorliegenden Klasse nicht die Referenz auf die tatsächlich existierende Methode zurück gibt ist erstaunlich. Über die Methode `resolveType(String)`, die im Interface `IType` definiert ist, kann ein lokal verwendeter Name in einen voll qualifizierten Namen umgewandelt werden. Würde über diese Methode jede Typbezeichnung eines jeden Parameters jeder Methode in einen voll qualifizierten Typnamen konvertiert, könnte mit der zweiten Anfrage (mit dem übergebenen voll qualifizierten Typnamen) eine entsprechende Methode auch in einer Datei gefunden werden, die im Quellcode vorliegt. In der Implementierung in Eclipse 3.2 findet diese Umsetzung allerdings nicht statt. Betrachtet man die Methode `equals(Object)` der Klasse `org.eclipse.jdt.internal.core.SourceMethod` (die Implementierung des Interfaces IMethod für Klassen, die im Quellcode vorliegen) stellt man fest, dass dort die Parametertypen einfach über die Methode `java.lang.String.equals(Object)` miteinander verglichen werden. Weiß man, dass der zweite Aufruf intern ein SourceMethod Objekt erzeugt das „Ljava.lang.String;“ als Parametertyp enthält und weiß man, dass die Methode `exists()` probiert aus der Liste der SourceMethod-Objekte im Objekt `demoKlasseType` das Objekt zu finden das mit dem erzeugten Objekt identisch ist, wird klar warum auch die zweite Anfrage auf eine Klasse die im Quellcode vorliegt keine Referenz auf die existierende Methode zurück gibt.

Um die passenden Darstellung der Parametertypen für den Aufruf der Methode `getMethod(String, String[])` zu erhalten, bietet das Interface `org.eclipse.jdt.core.IMethod` die Methode `getParameterTypes()` an. Sie liefert ein String-Array mit nicht-aufgelösten Typnamen zurück, wenn die Methode im Quellcode vorliegt und ein String-Array mit aufgelösten Typnamen, wenn die Methode nur binär vorliegt.

Der Umgang mit *Generics* wurde in diesem Kapitel nicht betrachtet. Da im Rahmen dieser Arbeit keine Darstellungen von Typen im Sinne der Klasse `org.eclipse.jdt.core.Signature` „von Hand“ erzeugt werden (sondern nur über die Methode `getParameterTypes()` angefragt werden) ist ein Verständnis dieser Details auch nicht nötig. Für diese sei an dieser Stelle auf die API-Dokumentation zu der Klasse `Signature` verwiesen.

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

[Bach 2007]: Markus Bach (2007): Design und Implementierung eines Eclipse-Plug-Ins zur Anzeige von möglichen Typgeneralisierungen im Quelltext

[Beck & Andres 2005]: Kent Beck, Cynthia Andres (2005): Extreme Programming Explained (Second Edition), Addison-Wesley, Boston

[Beck & Gamma 2004]: Kent Beck, Erich Gamma (2004): Contributing to Eclipse, Addison-Wesley, Boston

[Beck & Gamma 2004 de]: Kent Beck, Erich Gamma (2004): Eclipse erweitern, Addison-Wesley, München

[Beck 2005]: Kent Beck (2005): Test-Driven Development by Example, Addison-Wesley, Boston

[Bouillon et. al 2007]: Philipp Bouillon, Jens Krinke, Nils Meyer, Friedrich Steimann (2007): EzUnit: A Framework for Associating Failed Unit Testswith Potential Programming Errors. In: , Seiten , ,

[Cornett 2007]: Steve Cornett (2007): Code Coverage Analysis. Online unter http://www.bullseye.com/coverage.html

[Daum 2005]: Dr. Berthold Daum (2005): Java-Entwicklung mit Eclipse 3.1, dpunkt.verlag, Heidelberg

[Dmitriev 2006]: Mikhail Dmitriev (2006): Testar. Online unter http://google-testar.sourceforge.net/

[ECL2001]: Eclipse.org (2001): Eclipse.org Consortium Forms to Deliver New Era Application Development Tools. Online unter http://www.eclipse.org/org/pr.html

[ECL2004]: The Eclipse Foundation (2004): Eclipse 3.0 Delivers Universal Platform forApplication Construction. Online unter http://www.eclipse.org/org/press-release/jun212004r30pr.html

[Gamma et. al 1997]: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1997): Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, USA

[Gauthier 2003]: Laurent Gauthier (2003): Building and delivering a table editor with

SWT/JFace. Online unter http://www.eclipse.org/articles/Article-Table-viewer/table_viewer.html

[Gosling et. al 1996]: James Gosling, Bill Joy, Guy L. Steele (1996): The Java(TM) Language Specification, Addison-Wesley, Boston, Online unter http://java.sun.com/docs/books/jls/first_edition/html/index.html

[Gosling et. al 2000]: James Gosling, Bill Joy, Guy Steele, Gilad Bracha (2000): Java(TM) Language Specification (2nd Edition), Prentice Hall, New Jersey, Online unter http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[Gosling et. al 2005]: James Gosling, Bill Joy, Guy Steele, Gilad Bracha (2005): ava(TM) Language Specification, The (3rd Edition), Prentice Hall, New Jersey, Online unter http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

[Hunt & Thomas 2004]: Andrew Hunt, David Thomas (2004): Unit-Tests mit JUnit, Carl Hanser Verlag, München

[JDT API]: IBM Corp. and others (2006): Eclipse JDT API Specification. Online unter http://help.eclipse.org/help32/topic/org.eclipse.jdt.doc.isv/reference/api/overview-summary.html

[JUnit API]: (2007): JUnit API. Online unter http://junit.sourceforge.net/javadoc_40/index.html

[junit.org 2007]: Object Mentor, Incorporated (2007): JUnit, Testing Resource for Extreme Programming. Online unter http://junit.org

[JVMTI 2005]: Sun Microsystems, Inc. (2005): JVM Tool Interface 1.0.38. Online unter http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[Kehn et. al 2002]: Dan Kehn, Scott Fairbrother, Cam-Thu Le (2002): How to Internationalize your Eclipse Plug-In. Online unter http://www.eclipse.org/articles/Article-Internationalization/how2I18n.html

[Kuhn & Thomann 2006]: Thomas Kuhn, Olivier Thomann (2006): Abstract Syntax Tree. Online unter http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

[Lindholm & Yellin 1999]: Tim Lindholm, Frank Yellin (1999): The Java(TM) Virtual Machine Specification (2nd Edition), Prentice Hall, New Jersey, Online unter http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

[Nowak 2005]: Johannes Nowak (2005): Fortgeschrittene Programmierung mit Java 5, dpunkt.verlag, Heidelberg

[Saff & Ernst 2003]: David Saff, Michael D. Ernst (2003): Reducing wasted development time via continous testing. In: Fourteenth International Symposium on Software Reliability Engineering, Seiten 281-292, , Denver, CO

[Saff & Ernst 2004]: David Saff, Michael D. Ernst (2004): An experimental evaluation of continuous testing during development. In: Proceedings of the 2004 International Symposium on Software Testing and Analysis, Seiten 76-85, , Boston, MA, USA

[Saff 2006]: David Saff (2006): Continuous Testing. Online unter http://pag.csail.mit.edu/continuoustesting/

[Steimann et al. 2005]: F. Steimann, D. Keller, B. Aziz Safi (2005): Moderne Programmiertechniken und -methoden, ,

[Steimann et. al 2005]: F. Steimann, D. Keller, B. Aziz Safi (2005): Moderne Programmiertechniken und -methoden, Unit-Testen, Seiten 81-101, Fernuniversität in Hagen

[Steimann et. al 2007]: Friedrich Steimann, Philipp Bouillon, Nils Meyer (2007): EzUnit: A Framework for Associating Failed Unit Testswith Potential Programming Errors. In: , Seiten , ,

[TPTP 2007]: Eclipse TPTP Project Management Committee (): Eclipse Test and PerformanceTools Platform [TPTP] Project. Online unter http://www.eclipse.org/tptp/home/project_info/general/tptp_datasheet.pdf

[Ullenboom 2007]: Christian Ullenboom (2007): Java ist auch eine Insel, Galileo Press, Bonn, Online unter http://www.galileocomputing.de/openbook/javainsel6/

[Wahler 2006]: Vera Wahler et al. (2006): moreUnit. Online unter http://www.moreunit.org

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nils Meyer

Pfaffenhofen, den 4. April 2007