FERNUNIVERSITÄT IN HAGEN
FAKULTÄT FÜR MATHEMATIK UND INFORMATIK
LEHRGEBIET PROGRAMMIERSYSTEME
Prof. Dr. Friedrich Steimann

Abschlussarbeit im Studiengang
Master of Computer Science

Ein Refaktorisierungswerkzeug zum Ersetzen von kollaborierenden Objekten in Unit-Tests durch Mock-Objekte

vorgelegt von

Thomas Baumann
Drosselweg 4
83607 Holzkirchen
Matrikelnummer 7131003

betreut durch

Prof. Dr. Friedrich Steimann

27.03.2008

Inhaltsverzeichnis

1 Einleitung

In den heutigen agilen Softwareentwicklungs-Prozessen wird oftmals die testgetriebene Softwareentwicklung eingesetzt. Bei diesem Entwicklungsansatz werden Testfälle geschrieben, bevor der getestete Code überhaupt existiert. Neuer Code sollte nur dann geschrieben werden, wenn ein fehlerhafter Testfall vorliegt. In einem großen Software-Projekt wird man nicht sofort das Gesamtsystem testen, sondern man wird zunächst die Funktionalität der einzelnen Klassen sicherstellen. Dazu kann man die Klassen in sogenannten Unit-Tests separat testen. Für die Automatisierung von Unit-Tests gibt es spezielle Test-Frameworks. Im Java-Bereich ist das JUnit-Framework weit verbreitet, welches in die Entwicklungsumgebung Eclipse integriert ist. JUnit kann ganze Pakete von Unit-Tests ausführen und das Gesamtergebnis eines Testlaufs übersichtlich darstellen. Die Tests können jederzeit auf Knopfdruck wiederholt werden. So kann man sehr einfach bei jeder Codeänderung verifizieren, ob diese negative Auswirkungen auf bereits erfolgreich getestete Funktionen hat. Dies führt zu einer schnellen Aufdeckung von Fehlern und letztlich zu einer deutlich besseren Software-Qualität.

Eine getestete Klasse wird auch als „Class under Test“ (CUT) bezeichnet. Sie hat oftmals komplexe Schnittstellen zur Außenwelt, welche durch sogenannte „kollaborierende Objekte“ repräsentiert werden, zu denen eine Instanz der CUT in Beziehung steht. Die Klassen dieser Kollaborateure werden zu Beginn eines großen Software-Projekts i.d.R. noch gar nicht existieren bzw. nicht fehlerfrei zur Verfügung stehen. Des Weiteren sollte sich ein Unit-Test auf den Test der CUT konzentrieren und nicht den Code der Kollaborator-Klassen testen. Deshalb ist es naheliegend, die externen Schnittstellen einer CUT durch sogenannte Mock-Objekte zu simulieren. Diese Objekte bieten die gleiche Schnittstelle wie das Original-Objekt an und sind außerdem in der Lage, dessen Verhalten nachzubilden. In den letzten Jahren wurden im Java-Umfeld viele sogenannte „Mock-Frameworks“ entwickelt, welche Mock-Objekte für jede beliebige Klasse oder Schnittstelle erzeugen können. Außerdem ist es möglich, für jedes Mock-Objekt ein spezifisches Verhalten anzugeben. Allerdings haben alle Mock-Frameworks eine eigene, sehr spezifische Syntax, welche nicht standardisiert ist. In diese Syntax muss sich ein Entwickler erst einarbeiten, was für ihn zusätzlichen Aufwand neben der Erstellung der Testfälle bedeutet. Wenn er in seinen Testfällen einmal ein bestimmtes Mock-Framework eingesetzt hat, kann er nur mit großem Aufwand auf ein anderes Framework umsteigen.

Diese Arbeit untersucht, wie man Testfälle mit Mock-Objekten erstellen kann, ohne die Details der Mock-Frameworks zu kennen. Dazu wird ein neues Java-basiertes Refaktorisierungs-Werkzeug für die Eclipse-Platform vorgeschlagen, welches bestehende JUnit-Testfälle automatisch so umgestaltet, dass die Instanzen der CUT mit Mock-Objekten arbeiten. Das Refactoring muss die Kollaborateure der CUT-Instanzen in den Testfällen finden und diese durch Mock-Objekte ersetzen. Dies soll so flexibel gestaltet werden, dass der Einsatz beliebiger Mock-Frameworks möglich ist. Die Eclipse-Platform wurde nicht zuletzt deshalb gewählt, weil sie durch sogenannte Plugins beliebig erweitert werden kann und bereits einen Rahmen für Refactorings zur Verfügung stellt.

Kapitel 2 gibt eine Einführung in das Thema Mock-Objekte. Außerdem werden das JUnit-Testframework und mehrere Java-basierte Mock-Frameworks im Detail beschrieben. Weiterhin werden die externen Schnittstellen einer Klasse genauer beleuchtet. Kapitel 3 beschreibt die Aufgabenstellung und die Funktionalität des neuen Refaktorisierungs-Werkzeugs, während Kapitel 4 dessen Implementierung darstellt. In Kapitel 5 werden einige Tests beschrieben, die mit dem Refactoring durchgeführt wurden. Kapitel 6 bewertet das Werkzeug und beschreibt seine Einsatzmöglichkeiten. In Kapitel 7 wird schließlich ein Ausblick gegeben, wie das Refactoring verbessert und erweitert werden könnte.

2 Unit-Tests mit Mock-Objekten

2.1 Überblick

Ein objektorientiertes Programm besteht aus vielen miteinander in Beziehung stehenden Objekten. Als ersten Schritt des Programm-Tests führt man für jede einzelne Klasse Unit-Tests durch. Dabei hat eine Instanz der Class under Test (CUT) stets Schnittstellen zu anderen „kollaborierenden" Objekten in ihrer Umgebung (siehe [Freeman et al., 2004/II], „no object is an island..."). Diese Schnittstellen können sehr komplex werden, z.B. wenn man eine Anbindung an eine Datenbank oder eine Netzwerk-Verbindung zu einem Server hat. So kann es sehr aufwändig werden, vor den eigentlichen Tests die Umgebung zu initialisieren, insbesondere dann, wenn es sich um eine sehr umfangreiche Datenbank handelt oder wenn ein über ein Netzwerk angeschlossenes Gerät nicht verfügbar ist. Aus diesem Grund verwendet man für Unit-Tests oft statt der realen kollaborierenden Objekte sogenannte „Mock-Objekte", welche das Verhalten der realen Objekte imitieren.

[Thomas & Hunt, 2002] nennen sieben Gründe für die Verwendung von Mock-Objekten:

- Das reale Objekt zeigt ein nicht-deterministisches Verhalten.
- Man kann das reale Objekt nur schwer initialisieren.
- Das reale Objekt zeigt ein schwer triggerbares Verhalten (Ein Netzwerkfehler wäre z.B. nur für einen Test nur schwer hinzustellen).
- Das reale Objekt ist langsam.
- Das reale Objekt hat eine interaktive Benutzerschnittstelle.
- Das reale Objekt existiert noch nicht.
- Der Test braucht Informationen, wie das reale Objekt benützt wurde (z.B. ob eine bestimmte Methode aufgerufen wurde oder nicht).

Mock-Objekte registrieren jeden Methodenaufruf und ersetzen das Verhalten der Methoden durch einen „Stub", d.h. die Methoden haben keine Implementierung. Damit der Test trotzdem fehlerfrei abläuft, müssen die Mock-Objekte für jeden Methodenaufruf passende Rückgabewerte liefern.

Mock-Frameworks wie z.B. JMock, EasyMock und RMock können Mock-Objekte für einen gegebenen Typ (Interface oder Klasse) dynamisch erzeugen. Außerdem kann in einem Testfall spezifiziert werden, welche Methodenaufrufe für ein Mock-Objekt erwartet werden, und welchen Wert das Mock-Objekt bei einem bestimmten Methodenaufruf zurückgeben soll. Weiterhin kann überprüft werden, ob die Erwartungen bei der Ausführung des Tests wirklich erfüllt werden. Die genannten Frameworks sind in Kapitel 2.4 detailliert beschrieben.

2.2 Need-Driven Development

Werden reale kollaborierende Objekte durch Mock-Objekte ersetzt, muss ein Mock-Objekt nicht sämtliche Methoden der korrespondierenden Klasse simulieren, sondern nur die Methoden, welche das zu testende Objekt tatsächlich aufruft. Der Focus liegt hier auf der Interaktion zwischen den Objekten, wobei die Dienste entscheidend sind, welche ein Objekt benötigt, und nicht jene, die es zur Verfügung stellt. Letztendlich müssen die Interfaces identifiziert werden, welche den Rollen entsprechen, die die Objekte in einem objektorientierten Programm spielen.

[Freeman et al., 2004] stellen hierfür den Ansatz des „Need-Driven-Development“ vor: Wenn man eine zu testende Klasse implementieren will, stellt man zunächst fest, welche Dienste sie von ihrer Umgebung benötigt. Für diese Dienste definiert man eines oder mehrere Mock-Objekte, welche die entsprechenden kollaborierenden Objekte simulieren. Als ersten Schritt beschreibt man in einem Testfall, welche Methoden der zu testenden Klasse aufgerufen werden sollen, und welche Methodenaufrufe der Mock-Objekte (inkl. Rückgabewerten) man infolgedessen erwartet. Nun kann man den Code der Klasse schreiben, sie instanziieren und den Code der Klasse testen, ohne den Code für die Dienste zu implementieren. Ist der Code der Klasse getestet, kann man damit fortfahren, die bisher simulierten Dienste durch reale Objekte zu implementieren. Diese Objekte benötigen wieder Dienste, welche wiederum durch Mock-Objekte simuliert werden. Nach Implementierung und Test der neuen Klassen kann man fortfahren und wiederum die Mock-Objekte durch reale Objekte ersetzen usw. Diese Vorgehensweise kann man so lange fortführen, bis man eine Schicht erreicht, wo die benötigten Dienste durch bereits vorhandene Systembibliotheken bereitgestellt werden.

Das oben beschriebene Vorgehen ist ein testgetriebener Entwicklungsansatz. Zunächst wird ein Testfall geschrieben und erst anschließend der Code der CUT. Im Testfall wird dabei das Verhalten eines Mock-Objekts inklusive der erwarteten Methodenaufrufe spezifiziert. Daraus lässt sich ein Interface bestimmen, welches für das entsprechende kollaborierende Objekt im Code der CUT verwendet werden kann. [Freeman et al., 2004] sprechen hier von einem „Interface Discovery“ (siehe auch [Mackinnon et al., 2000]). Das komplette Interface ergibt sich nach Meinung des Autors allerdings nicht aus einem einzelnen Testfall, sondern aus der Summe aller Testfälle, denn jede getestete Funktion kann andere Methodenaufrufe eines kollaborierenden Objekts zur Folge haben.

Letztendlich erhält man beim „Need-Driven-Development“ ein Geflecht von Objekten, welche durch minimale Interfaces miteinander kommunizieren. Die Interfaces entsprechen den Rollen der verschiedenen Objekte. Sie haben auch den Vorteil dass der Code von ihrer tatsächlichen Realisierung entkoppelt wird. Dennoch kann bei steigender Komplexität auch bei dieser Vorgehensweise ein Refactoring des Code notwendig werden, wenn sich das Design des Programms als problematisch herausstellt.

2.3 Automatisierte Unit-Tests mit JUnit

Die oben erwähnten Mock-Frameworks unterstützen alle JUnit (siehe ([JUnit])). JUnit ist ein Java-Framework zum automatisierten Ablauf von Unit-Tests, welches zum Lieferumfang der Eclipse-Platform (siehe [Eclipse]) gehört. JUnit unterstützt u.a. folgende in [Link 2005], Kapitel 2.2, erwähnte Anforderungen an ein Framework zur Testautomatisierung:

- Testfälle müssen getrennt von der getesteten Klasse (CUT) definiert werden können. Sie sollten sich zweckmäßigerweise in einer eigenen Klasse befinden. Diese Klasse wird im Folgenden als Testklasse bezeichnet.
- Testfälle müssen voneinander unabhängig sein, d.h. das Testergebnis darf nicht von der Reihenfolge ihrer Abarbeitung abhängen.
- Testfälle können in beliebigen Gruppen zusammengefasst werden.

Bis zur JUnit-Version 3.8 musste die Testklasse von einer bestimmten Klasse abgeleitet sein. Außerdem mussten die Namen der Testfälle mit „test“ beginnen. Mit der Version 4 wurde JUnit auf Annotationen umgestellt. Testfälle können jetzt in einer beliebigen Klasse als normale Methoden deklariert werden, welche mit der Annotation `@Test` versehen sind. JUnit kann entweder einen einzelnen Testfall oder auch alle in einer Klasse vorhandenen Testfälle ausführen. Mit Hilfe der `@Suite`-Annotation können mehrere Klassen mit Testfällen zu einer Test-Suite zusammengefasst werden. Nach der Durchführung eines Tests kann das Ergebnis mit speziellen `assert`-Methoden überprüft werden, die JUnit zur Verfügung stellt.

Die Umgebung, unter denen die JUnit-Testfälle ablaufen, wird durch eine sogenannte TestRunner-Klasse bereitgestellt. JUnit 4 liefert lediglich einen textbasierten TestRunner mit. Mit Hilfe der `@RunWith`-Annotation ist es aber möglich, für eine Testklasse andere TestRunner zu verwenden. Die Eclipse-Platform stellt einen eigenen graphischen Test-Runner für JUnit zur Verfügung, welcher fest in die Entwicklungsumgebung eingebunden ist. Testfälle können hier einfach über den Menüpunkt „Run as -> JUnit Test“ im Kontextmenü gestartet werden. Das Ergebnis der Tests wird als grüner oder roter Balken angezeigt.

Ein Testfall läuft in einer festen Umgebung ab, welche als „Test-Fixture“ bezeichnet wird. Sie besteht aus einer Menge von Objekten, z.B. Instanzen der CUT. Aus Effizienzgründen sollten alle Testfälle einer Testklasse in der gleichen Umgebung laufen. In diesem Fall muss man die Test-Fixture nicht in jedem Testfall aufbauen, sondern kann den entsprechenden Code in eine eigene Methode packen. Diese Methode wird in JUnit mit der Annotation `@Before` versehen und automatisch vor jedem Testfall ausgeführt. Damit wird die Test-Fixture vor jedem Testfall neu initialisiert. Dies ist unbedingt notwendig, weil sonst jeder Testfall von den Zustands-Änderungen abhängig wäre, die seine Vorgänger in den Objekten der Test-Fixture verursacht hätten. Damit wäre die Unabhängigkeit der Testfälle nicht mehr gewährleistet, s.o.

Eine mit `@Before` annotierte Methode wird im Folgenden kurz als „Setup-Methode“ bezeichnet. In einer Setup-Methode erzeugte Objekte werden in Instanzvariablen gespeichert, damit sie in allen Testfällen sichtbar sind. JUnit erlaubt die Verwendung mehrerer Setup-Methoden in einer Testklasse. Für diese Methoden wird allerdings keine

bestimmte Aufrufreihenfolge garantiert. Nach den Tests kann es notwendig sein, die Fixture aufzuräumen und reservierte Ressourcen freizugeben. Dies kann in einer oder mehreren Methoden erfolgen, die mit `@After` annotiert sind, und von JUnit nach jedem Testfall ausgeführt werden.

Für rechenzeitintensive Initialisierungen, die für eine Testklasse nur einmal durchgeführt werden sollen, kann eine Klassenmethode definiert werden, die mit der Annotation `@BeforeClass` versehen ist. Nach Meinung des Autors ist es aber nicht sinnvoll, in einer solchen Klassenmethode Instanzen der CUT zu erzeugen (und diese in Klassenvariablen zu speichern), da alle Testfälle, welche diese Objekte benutzen, nicht mehr voneinander unabhängig wären, s.o. Eine Klassenmethode für Bereinigungen nach den Tests muss mit `@AfterClass` annotiert sein.

2.4 Mock-Frameworks

2.4.1 Ein Beispiel für eine CUT

Als Beispiel für eine CUT soll in den folgenden Kapiteln eine Klasse `Bank` verwendet werden, welche eine Reihe von Konten verwalten und diese auch (z.B. auf einem Kontoauszugsdrucker) ausgeben kann. Die Klasse `Bank` benötigt zwei kollaborierende Objekte in ihrer Umgebung: ein Datenbank-Objekt für das Speichern der Kontodaten, welches das Interface `IDatenbank` implementiert, sowie eine Instanz der Klasse `Ausgabe` für die Ausgabe der Kontodaten:

```
1:  public class Bank
2:  {
3:     private IDatenbank dieKonten;
4:     private Ausgabe outStream;
5:
6:     public Bank(IDatenbank kontenDb, Ausgabe stream)
7:     {  dieKonten = kontenDb;
8:        outStream = stream;
9:     }
10:
11:    public Konto kontoEröffnen(int nummer, int betrag) { ..... }
12:    public Konto leseKonto(int nummer) { ..... }
13:    public void kontoAusgabe(int nummer) { ..... }
14: }
```

Die Klassen der kollaborierenden Objekte sind folgende:

```
1:  public interface IDatenbank
2:  {
3:     public boolean addItem(int key, Object value);
4:     public boolean deleteItem(int key);
5:     public Object getItem(int key);
6:  }
```

```
1:  public class Datenbank implements IDatenbank
2:  {  ....
3:  }
```

```
1:  public class Ausgabe
2:  {  ....
3:     public Ausgabe(PrintStream stream) { ..... }
4:     public void print(String str) { ..... }
5:     public void println(String str) { ..... }
6:     public void printError(String str) { ..... }
7:  }
```

Die Bankkonten selbst werden durch die Klasse Konto repräsentiert:

```
1:  public class Konto
2:  {  ....
3:     public Konto(int nummer, int wert) { ..... }
4:     public void setKontostand(int wert) { ..... }
5:     public int getKontostand() { ..... }
6:     public int getKontonummer() { ..... }
7:     public String asString() { ..... }
8:     public boolean equals(Object anObject) { ..... }
9:  }
```

2.4.2 JMock

JMock ([JMock]) ist ein Open Source Framework, welches ein Java-API zur dynamischen Erzeugung von Mock-Objekten für Interfaces und Klassen bereitstellt. JMock unterstützt das JUnit-Testframework ([JUnit]), Version 3 und 4, kann aber auch mit anderen Testframeworks verwendet werden. Die in den folgenden Beispielen vorgestellte JMock-Version 2.2.0 (siehe auch [JMock-API 2.2.0]) setzt die Java-Version 5.0 oder höhere voraus.

Die Basis für die Erzeugung der Mock-Objekte bildet eine Instanz der Klasse `Mockery`, welche einen Context bildet, in dem die Umgebung des zu testenden Objekts simuliert wird. Die Klasse `Mockery` fungiert auch als Factory für die entsprechenden Mock-Objekte.

Möchte man JMock mit Junit 4 benützen, muss man die Klasse `JUnit4Mockery` instanziieren, welche von der Klasse `Mockery` abgeleitet ist. Außerdem muss mit Hilfe der RunWith-Annotation angegeben werden, dass als TestRunner die Klasse `JMock` verwendet werden soll:

```
1:  import org.jmock.Mockery;
2:  import org.jmock.integration.junit4.JMock;
3:  import org.jmock.integration.junit4.JUnit4Mockery;
4:
5:  @RunWith(JMock.class)
6:  public class BankTest
7:  { Mockery context = new JUnit4Mockery();
8:    .....
9:  }
```

Quelltext 2-1: JMock: Instanziierung der JUnit4Mockery

Der nächste Schritt ist die Initialisierung der Testumgebung. Für die Objekte, zu denen das zu testende Objekt (die CUT-Instanz) Abhängigkeiten hat, werden Mock-Objekte erzeugt. JMock erzeugt standardmäßig nur Mock-Objekte für Interfaces. Intern wird dazu das Java Reflection-API benützt. Mittels der statischen Methode `newProxyInstance()` der Klasse `Proxy` (Paket `java.lang.reflect`) wird ein Proxy-Objekt erzeugt, welches alle Methoden

des angegebenen Interfaces implementiert, und die Aufrufe für diese Methoden an einen JMock-internen Handler delegiert.

Ein Mock-Objekt muss sowohl dem Testfall als auch der CUT-Instanz bekannt sein. Außerdem muss es der CUT-Instanz statt dem realen kollaborierenden Objekt übergeben werden. Die verschiedenen Möglichkeiten zur Lösung dieses Problems sind in Kapitel 2.5 beschrieben. Eine davon wird in der in Kapitel 2.4.1 beschriebenen Klasse Bank gezeigt. Hier ist es möglich, die kollaborierenden Objekte als Konstruktor-Parameter von außen zu übergeben. Das folgende Beispiel zeigt die Übergabe eines Mock-Objekts für ein Interface beim Erzeugen der CUT-Instanz in der Setup-Methode:

```
1:  @RunWith(JMock.class)
2:  public class BankTest
3:  {
4:    private Mockery context = new JUnit4Mockery();
5:    private Bank dieBank;
6:    private IDatenbank mock_kontenDb;
7:
8:    @Before
9:    public void setUp() throws Exception
10:   {
11:      // Create Mock-Objects for Collaborators
12:      mock_kontenDb = context.mock(IDatenbank.class);
13:      // Create Testobject
14:      dieBank = new Bank(mock_kontenDb, new Ausgabe(System.out));
15:   }
16:   ......
```

Quelltext 2-2: JMock: Erzeugen eines Mock-Objekts für ein Interface

Mit Hilfe der JMock-Erweiterung `ClassImposteriser` können neben Mock-Objekten für Interfaces auch Mock-Objekte für Klassen erzeugt werden:

```
1:  import org.jmock.lib.legacy.ClassImposteriser;
2:  .....
3:  @RunWith(JMock.class)
4:  public class BankTest
5:  {
6:    private Mockery context = new JUnit4Mockery() {{
7:       setImposteriser(ClassImposteriser.INSTANCE); }};
8:    private IDatenbank mock_kontenDb;
9:    private Ausgabe mock_outputStream;
10:   .....
11:   @Before
12:   public void setUp() throws Exception
13:   {
14:     // Create Mock-Objects for Collaborators
15:     mock_kontenDb = context.mock(IDatenbank.class);
16:     mock_outputStream = context.mock(Ausgabe.class);
17:     // Create Testobject
18:     dieBank = new Bank(mock_kontenDb, mock_outputStream);
19:   }
20:   ......
```

Quelltext 2-3: JMock: Erzeugen von Mock-Objekten für Klassen und Interfaces

Intern erzeugt JMock Mock-Objekte für Klassen, indem mit Hilfe der „Code Generation Library“ ([cglib]) eine Proxy-Klasse erzeugt wird, welche eine Subklasse des zu ersetzenden

Typs ist. Die eigentliche Erzeugung des Mock-Objekts übernimmt die Google-Bibliothek „Objenesis“ (siehe [objenesis]), welche Klassen instanziieren kann, ohne dass deren Konstruktor aufgerufen wird. Dies hat den Vorteil, dass beim Instanziieren einer Mock-Klasse der Konstruktor der Originalklasse nicht benützt wird, und somit Abhängigkeiten zur Originalklasse vermieden werden.

Hat man einmal die Mock-Objekte erzeugt, kann man im jeweiligen Testfall mittels des JMock-API spezifizieren, welche Methodenaufrufe mit welchen Parametern man für diese Mock-Objekte während des Tests erwartet. Außerdem kann angegeben werden, wie ein Mock-Objekt auf diese Methodenaufrufe reagieren soll, d.h. welche Werte der jeweilige Methodenaufruf zurückliefern soll. Nach der Definition der Erwartungen wird der eigentliche Test durchgeführt. Während des Tests überprüft JMock automatisch alle spezifizierten Erwartungen. Wird eine Erwartung nicht erfüllt, meldet JMock einen Assertion-Fehler.

```
1:  @Test
2:  public void testcase()
3:  {
4:    // Expectations for Mock-Objects
5:    context.checking(new Expectations()
6:    {{ one(mock_kontenDb).addItem(4711, new Konto(4711,100));
7:         will(returnValue(true));
8:       one(mock_kontenDb).getItem(4711);
9:         will(returnValue(new Konto(4711,100)));
10:   }} );
11:   // Execute the Test; JMock will check the expectations
12:   Konto neuesKonto = dieBank.kontoEröffnen(4711, 100);
13:   assertNotNull(neuesKonto);
14:   dieBank.kontoAusgabe(4711);
15: }
```

Quelltext 2-4: JMock: Spezifikation der erwarteten Methodenaufrufe

In diesem Beispiel wird die Erwartung spezifiziert, dass für das Mock-Objekt `mock_kontenDb` die Methode `addItem()` zur Speicherung eines neuen Konto-Eintrags aufgerufen wird. Als Parameter werden die Kontonummer sowie ein entsprechendes `Konto`-Objekt übergeben. Die Methode soll den Wert `true` zurückliefern, d.h. die Speicherung des Kontos war erfolgreich. Außerdem soll für das gleiche Mock-Objekt die Methode `getItem()` mit der neuen Kontonummer aufgerufen werden, welche das eben erzeugte `Konto`-Objekt zurückliefert. Beide Methodenaufrufe sollen genau einmal erfolgen.

JMock bietet (durch spezifische Methoden der Klasse `Expectations`) folgende Möglichkeiten, die Anzahl der erwarteten Methodenaufrufe anzugeben:

`one(mockObject)`	Die Methode sollte genau einmal aufgerufen werden.
`exactly(n).of(mockObject)`	Die Methode sollte genau n Mal aufgerufen werden.
`atLeast(n).of(mockObject)`	Die Methode sollte mindestens n Mal aufgerufen werden.
`atMost(n).of(mockObject)`	Die Methode sollte höchstens n Mal aufgerufen werden.

`between(min, max).of( mockObject)`	Es werden zwischen min und max Methoden-aufrufe erwartet.
`allowing(mockObject), ignoring(mockObject)`	Die Anzahl der Methodenaufrufe kann beliebig sein. Es ist auch möglich, dass die Methode nicht aufgerufen wird.
`never(mockObject)`	Die Methode wird gar nicht aufgerufen.

Tabelle 2-1: JMock: Spezifikation der Anzahl der erwarteten Methodenaufrufe

Die erwarteten Parameter der Methoden können wie in Quelltext 2-4 direkt als Literale oder Ausdrücke spezifiziert werden. In diesem Fall werden die Parameter auf Gleichheit getestet. Die Klassen der Parameter müssen deshalb die Methode `equals()` implementiert haben. Möchte man flexiblere Bedingungen für die erwarteten Parameter eines Methodenaufrufs angeben, können sogenannte Matcher verwendet werden. Intern werden diese Matcher von Factory-Methoden der Klasse `Expectations` geliefert, welche dazu die Google „Hamcrest-Library“ (siehe [hamcrest]) verwenden. Jeder Matcher realisiert das gleichnamige Interface dieser Bibliothek.

Statt den Parameter direkt anzugeben, wird die Methode `with()` der Klasse `Expectations` aufgerufen, an welche ein Matcher übergeben wird. Der Ausdruck „`one(mock_outputStream).println(with(aNonNull(String.class)));`“ testet z.B., ob der Methode `println()` ein `String`-Objekt übergeben wird, welches nicht `null` ist. JMock unterstützt unter anderem folgende Matcher:

`equal(T` `value)`	Test auf Gleichheit.
`same``(T value)`	Test auf Identität.
`a``(Class<T> type)`, *`an`*`(Class<T> type)`	Test auf eine beliebige Instanz der angegebenen Klasse oder einer ihrer Subklassen.
`any``(Class<T> type)`	erlaubt ein beliebiges Objekt (liefert immer *true*).
`aNull``(Class<T> type)`	Test auf einen Parameter des gegebenen Typs, welcher *null* ist.
`aNonNull``(Class<T> type)`	Test auf einen Parameter des gegebenen Typs, welcher nicht *null* ist.

Tabelle 2-2: JMock-Matcher

Weiterhin kann angegeben werden, ob ein Matcher ein negatives Ergebnis zurückliefert, und es können die Ergebnisse mehrerer Matcher logisch verknüpft werden:

`not(Matcher m)`	Test auf negatives Ergebnis.
`anyOf(Matcher m1, Matcher m2, ...)`	Test ob ein Matcher ein positives Ergebnis liefert.
`allOf(Matcher m1, Matcher m2, ...)`	Test ob alle Matcher ein positives Ergebnis liefern.

Tabelle 2-3: Logische Verknüpfung von JMock-Matchern

Wie bereits in Quelltext 2-4 (Zeile 7) gezeigt, kann der Rückgabewert einer Methode mittels des Konstrukts „`will(returnValue(aValue))`“ spezifiziert werden (`will()` ist eine Methode der Klasse `Expectations`). Stattdessen kann man auch angeben, dass das Mock-Objekt beim Aufruf einer Methode eine Exception werfen soll:

```
one(mock_outputStream).println("A bad String"); will(throwException(
   new IllegalArgumentException("Don't call me with this!")));
```

In Quelltext 2-4 müssen die erwarteten Methoden nicht in der angegebenen Reihenfolge aufgerufen werden. Möchte man die Aufruf-Reihenfolge überprüfen, ist die Definition einer sog. Sequenz nötig:

```
1:  @Test
2:  public void testcase()
3:  { // Expectations for Mock-Objects
4:    final Sequence sequence = context.sequence("Mock-Sequence");
5:    context.checking(new Expectations()
6:    {{ one(mock_kontenDb).addItem(4711, new Konto(4711,100));
7:          will(returnValue(true)); inSequence(sequence);
8:       one(mock_kontenDb).getItem(4711);
9:          will(returnValue(new Konto(4711,100))); inSequence(sequence);
10:   }} );
11:   .....
12: }
```

Quelltext 2-5: JMock: Sequenz von Methodenaufrufen

Durch Definition einer „state machine“ kann man komplizierte Abhängigkeiten zwischen den verschiedenen Methodenaufrufen spezifizieren, z.B. dass eine Methode nur dann aufgerufen werden darf, wenn vorher eine bestimmte andere Methode aufgerufen wurde:

```
1:  @Test
2:  public void testcase()
3:  { // Expectations for Mock-Objects
4:    final States database = context.states("database").startsAs("empty");
5:    context.checking(new Expectations()
6:    {{ one(mock_kontenDb).addItem(4711, new Konto(4711,100));
7:          then(database.is("notEmpty")); will(returnValue(true));
8:       one(mock_kontenDb).getItem(4711);
9:          when(database.is("notEmpty"));
10:         will(returnValue(new Konto(4711,100)));
11:   }} );
12:   .....
13: }
```

Quelltext 2-6: JMock „state machine“

Schließlich kann JMock auf spezielle Benutzerbedürfnisse angepasst und entsprechend erweitert werden:

- Man kann eigene Matcher definieren, um spezielle Bedingungen für die Parameter der aufgerufenen Methoden zu prüfen. Dazu muss das Interface `org.hamcrest.Matcher` implementiert werden.
- Um ein spezielles Verhalten aufgerufener Methoden zu simulieren, kann man eigene „Actions“ definieren, indem man das Interface `org.jmock.api.Action` implementiert. Hier ist es auch möglich, die aktuellen Parameter einer Methode auszuwerten und daraus ein bestimmtes Verhalten abzuleiten.

Weitere Einzelheiten können der Beschreibung des JMock-API entnommen werden, siehe [JMock-API 2.2.0].

2.4.3 EasyMock

EasyMock (siehe [EasyMock]) ist wie JMock ein Java-API zur dynamischen Erzeugung von Mock-Objekten. EasyMock läuft ab Java Version 5.0 und kann mit einem beliebigen Testframework benützt werden. Die nachfolgenden Beispiele verwenden JUnit 4.

Für die meisten Tests mit EasyMock benötigt man lediglich die statischen Methoden der Klasse `EasyMock`. Mit der Methode `createMock()` kann ein Mock-Objekt für ein gegebenes Interface erzeugt werden:

```
1:  import static org.easymock.EasyMock.*;
2:  .....
3:  public class BankTest
4:  {
5:    private Bank dieBank;
6:    private IDatenbank mock_kontenDb;
7:    @Before
8:    public void setUp() throws Exception
9:    {
10:     // Create Mock-Objects
11:     mock_kontenDb = createMock(IDatenbank.class);
12:     // Create Testobject
13:     dieBank = new Bank(mock_kontenDb, new Ausgabe(System.out));
14:   }
15:   .....
```

Quelltext 2-7: EasyMock: Erzeugen eines Mock-Objekts für ein Interface

Zum Erzeugen eines Mock-Objekts für ein Interface benützt EasyMock intern wie JMock das Java Reflection-API (Methode `newProxyInstance()`, Klasse `Proxy`), siehe Kapitel 2.4.2.

Die „EasyMock Class Extension“ erlaubt es, Mock-Objekte auch für Klassen zu erzeugen. Außerdem kann man nur ein Subset der Methoden einer Klasse durch Mock-Methoden ersetzen. Alle anderen Methoden werden normal ausgeführt. Intern benützt EasyMock wie JMock zum Erzeugen von Mock-Objekten für Klassen die „Code Generation Library“ ([cglib]), siehe Kapitel 2.4.2. Die Library „Objenesis“ wird von EasyMock allerdings nicht benützt. Trotzdem wird beim Erzeugen eines Mock-Objekts der Konstruktor der Originalklasse nicht aufgerufen.

Ein Mock-Objekt befindet sich zunächst im Status "record". Zu Beginn des Testfalls können die erwarteten Methodenaufrufe spezifiziert werden. Im einfachsten Fall sieht die Syntax wie ein normaler Methodenaufruf aus. Möchte man aber für die Methoden auch Rückgabewerte angeben, muss die statische Methode `expect()` (Klasse `EasyMock`) benützt werden. Nach Spezifikation der erwarteten Methodenaufrufe muss das Mock-Objekt durch Aufruf der statischen Methode `replay()` in einen Zustand versetzt werden, in dem es wie ein Mock-Objekt reagiert. Anschließend führt man den eigentlichen Test durch. Wird während des Tests eine Methode eines Mock-Objekts aufgerufen, verifiziert EasyMock automatisch die Parameter der Methode und gibt gegebenenfalls den spezifizierten Rückgabewert zurück. Allerdings findet keine Überprüfung statt, ob eine Methode gar nicht aufgerufen wurde.

Möchte man auch dies verifizieren, muss man nach dem eigentlichen Test für das Mock-Objekt die statische Methode `verify()` aufrufen. Werden die Erwartungen nicht erfüllt, liefert EasyMock entsprechende Assertion-Fehler. Die folgende Quelltext 2-8 zeigt einen JUnit-Testfall mit EasyMock:

```
1:  @Test
2:  public void testcase()
3:  {
4:    // Expectations for Mock-Objects
5:    expect(mock_kontenDb.addItem(4711, new Konto(4711,100)))
6:       .andReturn(true);
7:    expect(mock_kontenDb.getItem(4711)).andReturn(new Konto(4711,100));
8:    expectLastCall().anyTimes();
9:    replay(mock_kontenDb);
10:   // Execute the Test
11:   Konto neuesKonto = dieBank.kontoEröffnen(4711, 100);
12:   assertNotNull(neuesKonto);
13:   dieBank.kontoAusgabe(4711);
14:   // Verify Mock Expectations
15:   verify(mock_kontenDb);
16: }
```

Quelltext 2-8: JUnit-Testfall mit EasyMock

Ähnlich wie JMock bietet auch EasyMock verschiedene Möglichkeiten, die Anzahl der erwarteten Methodenaufrufe zu spezifizieren, indem man nach der Angabe eines erwarteten Methodenaufrufs die statische Methode `expectLastCall()` aufruft. Direkt im Anschluss ruft man auf deren Ergebnis wiederum eine der folgenden Methoden auf:

`times(n)`	Die Methode sollte genau n mal aufgerufen werden.
`times(min, max)`	Es werden zwischen min und max Methodenaufrufe erwartet.
`atLeastOnce()`	Die Methode sollte mindestens ein Mal aufgerufen werden.
`anyTimes()`	Es wird eine beliebige Anzahl von Methodenaufrufen erwartet.

Tabelle 2-4: EasyMock: Spezifikation der Anzahl der erwarteten Methodenaufrufe

Ein Beispiel für `anyTimes()` findet sich in Quelltext 2-8, Zeile 8. Ohne diese Angaben wird genau ein Methodenaufruf erwartet.

Die erwarteten Parameter eines Methodenaufrufs können, wie z.B. in Quelltext 2-8, Zeile 5, direkt als Literale oder Ausdrücke angegeben werden. Wie schon bei JMock werden die Parameter mit den tatsächlichen aktuellen Parametern per `equals()` verglichen. Möchte man einen anderen Vergleich durchführen, kann man sog. „Argument Matcher“ benützen, welche als statische Methoden der Klasse `EasyMock` definiert sind. Der Ausdruck „`mock_outputStream.println(isA(String.class));`“ z.B. testet, ob das übergebene Argument eine Instanz der Klasse `String` ist. Unter anderem stellt EasyMock folgende Argument Matcher zur Verfügung:

`eq(T value)`	Test auf Gleichheit.
`aryEq(A array)`	Test eines Array's auf Gleichheit.
`same(T value)`	Test auf Identität.
`isA(Class<T> type)`	Test auf eine beliebige Instanz der angegebenen Klasse oder einer ihrer Subklassen.
`anyBoolean(),anyByte(), anyChar(),anyInt(),anyLong(), anyFloat(),anyDouble(), anyShort(),anyObject()`	Test auf einen beliebigen primitiven Wert oder ein beliebiges Objekt.
`isNull(), notNull()`	Test auf Parameterwert *null* oder *not null.*

Tabelle 2-5: EasyMock-Matcher

Außerdem werden spezielle Vergleiche für Strings (Test auf Substring, String-Anfang und String-Ende) und numerische bzw. `Comparable`-Typen (z.B. Tests auf einen bestimmten Wertebereich) unterstützt.

Ebenso wie bei JMock können auch bei EasyMock verschiedene Matcher logisch verknüpft werden:

`not(T matcher)`	Test auf negatives Ergebnis eines Matchers.
`or(T matcher1, T matcher2, ...)`	Test ob ein Matcher ein positives Ergebnis liefert.
`and(T matcher1, T matcher2, ...)`	Test ob alle Matcher ein positives Ergebnis liefern.

Tabelle 2-6: Logische Verknüpfung von EasyMock-Matchern

Ein eigener Argument-Matcher kann definiert werden, indem man das Interface `IArgumentMatcher` implementiert, und die entsprechende Klasse in einer ebenfalls neu definierten statischen Methode instanziiert. Näheres dazu siehe [EasyMock Dokumentation].

Wie in Quelltext 2-8 (Zeile 6) gezeigt, werden die Rückgabewerte der Methoden durch Aufruf der Methode „`andReturn(aValue)`" spezifiziert (die statische Methode `expect()` der Klasse `EasyMock` liefert ein Objekt, welches das Interface `IExpectationSetters` implementiert; dieses Interface definiert die Methode `andReturn()`). Statt der Rückgabe eines Wertes ist es auch möglich, mit `andThrow()` eine Exception zu werfen, z.B.:

```
expect(mock_kontenDb.getItem(999999999)).andThrow(
   new IllegalArgumentException("Illegal Number"));
```

Man kann das Verhalten einer Methode auch auf Benutzerbedürfnisse anpassen, indem man das Interface `IAnswer` implementiert. Wie JMock bietet auch EasyMock die Möglichkeit, auf die aktuellen Parameter einer aufgerufenen Methode zuzugreifen, z.B.:

```
1:  expect(mock_kontenDb.addItem(4711, new Konto(4711,100))).andAnswer(
2:     new IAnswer<Boolean>() {
3:        public Boolean answer()
4:        {  Object[] currArguments = getCurrentArguments();
5:           if (currArguments[0].equals(
6:                 ((Konto)currArguments[1]).getKontonummer()))
7:             return Boolean.TRUE;
8:           else return Boolean.FALSE;
9:        }
10:    }
11: );
```

Quelltext 2-9: EasyMock: Benutzerdefiniertes Verhalten einer Methode

Der in Quelltext 2-8 angegebene Testfall überprüft nicht, ob die Methoden der Mock-Objekte in der angegebenen Reihenfolge aufgerufen wurden. Möchte man nur für ein Objekt die Reihenfolge der Methodenaufrufe überprüfen, kann man zur Erzeugung des Mock-Objekts statt der Methode `createMock()` die Methode `createStrictMock()` verwenden. Bei mehreren Mock-Objekten braucht man aber eine sogenannte "Mock Control", in der die Checks für die einzelnen Mock-Objekte zusammengefasst werden:

```
1:  import org.easymock.IMocksControl;
2:  .....
3:  public class BankTest
4:  { .....
5:    private IMocksControl mockCtrl = createStrictControl();
6:    @Before
7:    public void setUp() throws Exception
8:    {
9:      // Create Mock-Objects
10:     mock_kontenDb = mockCtrl.createMock(IDatenbank.class);
11:     mock_outputStream = mockCtrl.createMock(Ausgabe.class);
12:     .....
13:   }
14:   @Test
15:   public void testcase() {
16:   {
17:     // Expectations for Mock-Objects
18:     expect(mock_kontenDb.addItem(4711, new Konto(4711,100)))
19:       .andReturn(true);
20:     expect(mock_kontenDb.getItem(4711)).andReturn(new Konto(4711,100));
21:     mock_outputStream.println(isA(String.class));
22:     mockCtrl.replay();
23:     // Execute the Test
24:     .....
25:     // Verify Mock Expectations
26:     mockCtrl.verify();
27:     mockCtrl.reset();
28:   }
```

Quelltext 2-10: EasyMock: Mock Control

2.4.4 RMock

RMock ([Rmock]) ist ein Open Source Mock-Framework, welches speziell für die Benützung mit JUnit ([JUnit]) entwickelt wurde. Bislang wird nur die JUnit Version 3.8.1 unterstützt, jedoch nicht die neueren Versionen 4.x, in denen die speziellen JUnit-Testklassen durch Annotationen ersetzt wurden.

Um in einem JUnit-Testfall Mock-Objekte verwenden zu können, muss er statt von der JUnit-Klasse `TestCase` von der RMock-spezifischen Klasse `RMockTestCase` abgeleitet werden. Mit der Methode `mock()` dieser Klasse können Mock-Objekte für Klassen oder Interfaces erzeugt werden:

```
1:  import com.agical.rmock.extension.junit.*;
2:  ......
3:  public class BankTest extends RMockTestCase
4:  {
5:    private Bank dieBank;
6:    private IDatenbank mock_kontenDb;
7:    private Ausgabe mock_outputStream;
8:    protected void setUp() throws Exception
9:    {  // Create Mock-Objects for Collaborators
10:      mock_kontenDb = (IDatenbank)mock(IDatenbank.class);
11:      mock_outputStream = (Ausgabe)mock(Ausgabe.class);
12:      // Create Testobject
13:      dieBank = new Bank(mock_kontenDb, mock_outputStream);
14:   }
15:   ......
```

Quelltext 2-11: RMock: Erzeugen von Mock-Objekten

Intern verwendet RMock zur Erzeugung von Mock-Objekten immer die „Code Generation Library“ ([cglib]), siehe Kapitel 2.4.2, ganz gleich, ob es sich um Mock-Objekte für Interfaces oder Klassen handelt. Allerdings ruft RMock im Gegensatz zu den anderen Mock-Frameworks beim Erzeugen eines Mock-Objekts für eine Klasse immer auch den Konstruktor der Originalklasse auf. Hat die Originalklasse keinen Default-Konstruktor, und erzeugt man ein Mock-Objekt mit der in Quelltext 2-11 angegeben Syntax, führt dies zu einer `CodeGenerationException` in der cglib! Dies ist nach Meinung des Autors eine große Einschränkung, da hier eine Abhängigkeit zur Original-Implementierung besteht. Man kann diese Exception vermeiden, in dem man RMock beim Aufruf der Methode `mock()` ein Array von Parameter-Objekten übergibt. RMock sucht dann automatisch den Konstruktor mit der Signatur heraus, die zu den Parameter-Typen passt:

```
mock_outputStream = (Ausgabe)mock(Ausgabe.class,
   new Object[]{System.out}, "Ausgabe");
```

Allerdings darf man auch hier keinen Fehler machen, denn sonst wirft RMock eine `NoSuch-ConstructorException`.

Zur Erzeugung eines Mock-Objekts für eine Klasse kann man statt der oben genannten Methode `mock()` auch die Methode `intercept()` verwenden. In diesem Fall leitet RMock standardmäßig alle Methodenaufrufe an die Originalklasse weiter, außer man spezifiziert für einen Methodenaufruf bestimmte Erwartungen (s.u.). Auf diese Weise ist es möglich, ein

Subset der Methoden einer Klasse durch Mock-Methoden zu ersetzen (ähnlich wie bei der „EasyMock Class Extension“, s.o.). Man kann auch mittels der Methode `modify()` (s.u.) die Parameter eines Methodenaufrufs modifizieren und RMock erst danach die Methode der Originalklasse aufrufen lassen. Ebenso kann der Rückgabewert der Original-Methode modifiziert werden. Für Interfaces gibt es neben `mock()` die Methode `fakeAndIntercept()`, mit der sich ein Mock-Objekt erzeugen lässt, das für alle Methodenaufrufe Default-Werte zurückgibt.

Ähnlich wie bei EasyMock ist jedes Mock-Objekt zunächst im State „recording“, d.h. man kann die für den jeweiligen Testfall erwarteten Methodenaufrufe spezifizieren. Anschließend werden alle Mock-Objekte durch Aufruf der Methode `startVerification()` in den State „verifying“ umgeschaltet, und der eigentliche Test kann durchgeführt werden. Für die Überprüfung des Testergebnisses bietet RMock eine Bibliothek von Assertions an (`AssertThat(...)`), welche umfangreichere Prüfungen ermöglichen als die JUnit 3.8.1 Assertions und diese ersetzen können. In JUnit 4.4 wurde inzwischen auch ein neuer „AssertThat“-Mechanismus eingeführt, der syntaktisch aber etwas anders aussieht.

Die erwarteten Methodenaufrufe kann man syntaktisch wie einen normalen Methodenaufruf auf das entsprechende Mock-Objekt angeben. Mit Hilfe der Methode `modify()` kann anschließend spezifiziert werden, wie viele Methodenaufrufe man erwartet und welchen Rückgabewert ein entsprechender Methodenaufruf liefern soll bzw. ob stattdessen eine Exception erwartet wird. Eine Exception kann statt mit einem try-catch-Block auch mit dem Ausdruck „`expectThatExceptionThrown(…)`“ gefangen werden:

```
1:  public void testcase()
2:  {
3:    // Expectations for Mock-Objects
4:    mock_kontenDb.addItem(4711, new Konto(4711,100));
5:       modify().returnValue(true);
6:    mock_kontenDb.getItem(0);
7:       modify().multiplicity(expect.once());
8:       modify().throwException(new IllegalArgumentException("Illegal" +
9:          " Account Number"));
10:   // Switch Mock-State
11:   startVerification();
12:   // Execute the Test
13:   Konto neuesKonto;
14:   neuesKonto = dieBank.kontoEröffnen(4711, 100);
15:   assertThat(neuesKonto, is.NOT_NULL);
16:   expectThatExceptionThrown(is.instanceOf(
17:           IllegalArgumentException.class));
18:   dieBank.kontoAusgabe(0);
19: }
```

Quelltext 2-12: JUnit-Testfall mit RMock

Es gibt folgende Möglichkeiten, die Anzahl der erwarteten Methodenaufrufe zu spezifizieren:

`once()`	Die Methode sollte genau einmal aufgerufen werden (Standardeinstellung; kann auch weggelassen werden).
`exactly(n)`	Die Methode sollte genau n mal aufgerufen werden.
`atLeastOnce()`	Die Methode sollte mindestens einmal aufgerufen werden.

`atLeast(n)`	Die Methode sollte mindestens n Mal aufgerufen werden.
`atMostOnce()`	Die Methode sollte höchstens einmal aufgerufen werden.
`atMost(n)`	Die Methode sollte höchstens n Mal aufgerufen werden.
`from(m)[.to(n)]`	Es werden zwischen m und n Methodenaufrufe erwartet. Die Angabe eines Maximalwertes ist optional.

Tabelle 2-7: RMock: Spezifikation der Anzahl der erwarteten Methodenaufrufe

Um die erwarteten Parameterwerte eines Methodenaufrufs flexibler angeben zu können, definiert RMock in der Klasse `AbstractStrategyTestCase` (welche eine Superklasse von `RMockTestCase` ist) eine Instanzvariable `is`, die auf ein Objekt zeigt, welches das Interface `ConstraintFactory` (Paket `com.agical.rmock.core.match.constraint`) implementiert. Folgende Constraints sind möglich:

`eq(value)`	Test ob ein gegebener Wert gleich *value* ist. *value* kann ein primitiver Typ short, int, long, char, byte float oder double sein.
`eq(object)`	Test auf Gleichheit zweier Objekte (via equals()).
`lt(value)`	Test ob ein gegebener Wert kleiner als *value* ist (mögliche Typen von *value* s.o.).
`lt(object)`	Test ob ein gegebenes Objekt kleiner als *object* ist (beide Objekte müssen das Interface *Comparable* implementiert haben).
`gt(value)`	Test ob ein gegebener Wert größer als *value* ist (mögliche Typen von *value* s.o.).
`gt(object)`	Test ob ein gegebenes Objekt größer als *object* ist (beide Objekte müssen das Interface *Comparable* implementiert haben).
`le(value)`	Test ob ein gegebener Wert kleiner oder gleich *value* ist (mögliche Typen von *value* s.o.).
`le(object)`	Test ob ein gegebenes Objekt kleiner oder gleich *object* ist (beide Objekte müssen das Interface *Comparable* implementiert haben).
`ge(value)`	Test ob ein gegebener Wert größer oder gleich *value* ist (mögliche Typen von *value* s.o.).
`ge(object)`	Test ob ein gegebenes Objekt größer oder gleich *object* ist (beide Objekte müssen das Interface *Comparable* implementiert haben).
`not(expression)`	Vergleich mit dem invertierten Ausdruck *expression*.
`same(object)`	Test auf Identität zweier Objekte.
`instanceOf(class)`	Test auf eine beliebige Instanz der angegebenen Klasse oder einer ihrer Subklassen.
`containing(string)`	Test, ob *string* enthalten ist.
`startingWith(string)`	Test, ob ein gegebener String mit *string* anfängt.

`endingWith(string)`	Test, ob ein gegebener String mit *string* endet.
`NULL, NOT_NULL`	Test auf *null* oder ungleich *null*.
`TRUE, FALSE`	Test auf *true* oder *false*.
`ANYTHING`	Test auf einen beliebigen Wert.
`AS_RECORDED`	Test auf den Parameterwert, der bereits im Methodenaufruf angegeben wurde.

Tabelle 2-8: RMock-Constraints

Möchte man mit Hilfe dieser Constraints flexible Bedingungen für die möglichen Parameter eines Methodenaufrufs angeben, geschieht dies wieder mit der Methode `modify()`, welche gewissermaßen die bereits im Methodenaufruf angegebenen Werte modifiziert, z.B.:

```
mock_outputStream.println("Ein String");
   modify().args(is.instanceOf(String.class));
mock_kontenDb.addItem(99999, new Konto(4711,100));
   modify().args(is.ANYTHING, is.AS_RECORDED);
```

Die Constraints können auch in `assertThat`-Statements verwendet werden, siehe Quelltext 2-12, Zeile 15.

Ähnlich wie bei den anderen Mock-Frameworks kann man auch bei RMock sicherstellen, dass die erwarteten Methodenaufrufe in der angegebenen Reihenfolge ausgeführt werden. Dies geschieht mit Hilfe einer sog. "ordered section", welche mit Hilfe der Instanzvariable `s` der Klasse `RMockTestCase` erzeugt werden kann. Daneben kann man auch noch eine sog. "defaults section" spezifizieren, wo man i.d.R. Methodenaufrufe gruppiert, die beliebig oft mit beliebigen Parametern auftreten können. Die "defaults section" wird üblicherweise in der Setup-Methode definiert. Die dort angegebenen Methodenaufrufe werden von RMock überprüft, wenn keine der in den Testfällen spezifizierten Erwartungen auf einen im Test aufgetretenen Methodenaufruf passt:

```
1:  public class BankTest extends RMockTestCase
2:  { private Bank dieBank;
3:    private IDatenbank mock_kontenDb;
4:    private Ausgabe mock_outputStream;
5:    protected void setUp() throws Exception
6:    {
7:      // Create Mock-Objects for Collaborators
8:      mock_kontenDb = (IDatenbank)mock(IDatenbank.class);
9:      mock_outputStream = (Ausgabe)mock(Ausgabe.class);
10:     // RMock defaults-section
11:     appendToSection("defaults");
12:     {  mock_outputStream.println("Irgendeine Kontoausgabe");
13:           modify().args(is.containing("Kontonummer:"));
14:           modify().multiplicity(expect.from(0));
15:     }
16:     endSection();
17:     // Create Testobject
18:     dieBank = new Bank(mock_kontenDb, mock_outputStream);
19:   }
```

```
20:   public void testcase()
21:   {
22:     // Expectations for Mock-Objects
23:     beginSection(s.ordered("mySection"));
24:     {  mock_kontenDb.addItem(4711, new Konto(4711,100));
25:           modify().returnValue(true);
26:        mock_kontenDb.getItem(4711);
27:           modify().returnValue(new Konto(4711,100));
28:     }
29:     endSection();
30:     ......
31:   }
```

Quelltext 2-13: RMock: Sequenzen erwarteter Methodenaufrufe

2.5 Übergabe von kollaborierenden Objekten an eine CUT-Instanz

Wenn man in einem Testfall das Verhalten eines Mock-Objekts spezifizieren will, muss er dieses Objekt kennen. Außerdem muss der CUT-Instanz dieses Mock-Objekt statt dem realen kollaborierenden Objekt übergeben werden. Dies kann man dadurch erreichen, dass der Testfall in die Erzeugung des kollaborierenden Objekts involviert ist. Laut [Freeman et al., 2004] gibt es dafür folgende Möglichkeiten:

a) Man übergibt der CUT-Instanz das kollaborierende Objekt als Konstruktor-Parameter. Martin Fowler bezeichnet dies in seinem Artikel über „Dependency Injection“ als „Constructor Injection“, siehe [Fowler, 2004].

   Ein Beispiel zeigt die Definition der Klasse `Bank` in Kapitel 2.4.1. Hier werden die kollaborierenden Objekte für die Konten-Datenbank und die Konten-Ausgabe als Konstruktor-Parameter übergeben. Natürlich ist es sinnvoll, wenn wie bei der Konten-Datenbank als Typ des Konstruktor-Parameters ein Interface (hier `IDatenbank`) verwendet wird, denn dies entkoppelt die konkrete Implementierung des Kollaborateurs vom Code der CUT. Die Beschreibung der Mock-Frameworks in Kapitel 2.4. enthält mehrere Beispiele für die Übergabe von Mock-Objekten beim Instanziieren der CUT.

b) Der CUT-Instanz wird das kollaborierende Objekt als Parameter einer Methode übergeben. Soll eine CUT-Instanz konsequent von ihren externen Abhängigkeiten entkoppelt werden, sollte ihr das kollaborierende Objekt nur einmal übergeben und anschließend in einer Instanzvariable gespeichert werden. Für die Übergabe verwendet man sinnvollerweise eine Setter-Methode. Martin Fowler bezeichnet dies in ([Fowler, 2004]) als „Setter Injection“.

   Die Klasse `Bank` könnte beispielsweise folgendermaßen definiert sein:

```
1:  public class Bank
2:  {  .....
3:     public Bank() { }
4:     public void setKontoDb(IDatenbank kontenDb)
5:     {  dieKonten = kontenDb;
6:     }
7:     public void setOutputStream(Ausgabe stream)
8:     {  outStream = stream;
9:     }
10:    .....
11: }
```

Quelltext 2-14: Setter-Methoden in der Klasse Bank

Die Übergabe der Mock-Objekte an eine CUT-Instanz würde z.B. so aussehen:

```
1:  @Before
2:  public void setUp() throws Exception
3:  {  .....
4:     // Create Testobject
5:     dieBank = new Bank();
6:     dieBank.setKontoDb(mock_kontenDb);
7:     dieBank.setOutputStream(mock_outputStream);
8:  }
```

Quelltext 2-15: Übergabe von Mock-Objekten durch „Setter-Injection“

c) Man übergibt der CUT-Instanz das kollaborierende Objekt nicht direkt, sondern ein Fabrik-Objekt, welches das kollaborierende Objekt erzeugt. Das Fabrik-Objekt kann wieder als Konstruktor-Parameter oder Parameter einer Methode übergeben werden. Das folgende Beispiel zeigt die Übergabe als Konstruktor-Parameter:

```
1:  public class Bank
2:  {
3:     private IDatenbank dieKonten;
4:     .....
5:     public Bank(IDatenbankFactory dbFactory, ...)
6:     {
7:        dieKonten = dbFactory.createDatenbank();
8:        .....
9:     }
10:    .....
11: }
```

Quelltext 2-16: Übergabe eines Fabrik-Objekts im Konstruktor der Klasse Bank

Folgt man dem „Abstrakte Fabrik“-Pattern (siehe [Gamma et al., 2004], Seite 107), wird ein Interface als abstrakte Fabrik definiert. Eine weitere Klasse bildet die konkrete Fabrik und implementiert dieses Interface:

```
1:  public interface IDatenbankFactory
2:  {  public IDatenbank createDatenbank();
3:  }
```

```
1:  public class DatenbankFactory implements IDatenbankFactory
2:  {
3:     public IDatenbank createDatenbank()
3:     {  return new Datenbank();
4:     }
5:  }
```

Quelltext 2-17: Abstrakte und konkrete Fabrik

Die Übergabe eines Mock-Objekts an eine CUT-Instanz kann man z.B. so realisieren, dass in der Klasse, in der die Testfälle definiert sind, eine innere Klasse deklariert wird, die das oben gezeigte Interface `IDatenbankFactory` implementiert und ein Mock-Objekt zurückgibt, welches das Interface `IDatenbank` simuliert. Das folgende Beispiel verwendet das JMock-Framework:

```
1:  public class Testclass
2:  {  .....
3:     private IDatenbank mock_kontenDb;
4:     private class MockDbFactory implements IDatenbankFactory
5:     {
6:        public IDatenbank createDatenbank()
7:        {  mock_kontenDb = context.mock(IDatenbank.class);
8:           return mock_kontenDb;
9:        }
10:    }
11:    @Before
12:    public void setUp() throws Exception
13:    {
14:       // Create Testobject
15:       dieBank = new Bank( new MockDbFactory(),... );
16:    }
17:    ......
18: }
```

Quelltext 2-18: Übergabe eines Mock-Objekts mittels einer Factory

Alternativ kann man die Fabrik selbst als Mock-Objekt definieren und dafür ein Verhalten spezifizieren, welches als Rückgabewert der Methode `createDatenbank()` ein Mock-Objekt zurückgibt:

```
1:  public class Testclass
2:  {  .....
3:     private IDatenbankFactory mock_dbFactory;
4:     private IDatenbank mock_kontenDb;
5:     @Before
6:     public void setUp() throws Exception
7:     {
8:        // Create Mock-Objects for Collaborators
9:        mock_kontenDb = context.mock(IDatenbank.class);
10:       // Mock-Object Factory
11:       mock_dbFactory = context.mock(IDatenbankFactory.class);
12:       context.checking(new Expectations()
13:       {{  one(mock_dbFactory).createDatenbank();
14:              will(returnValue(mock_kontenDb));
15:       }} );
16:       // Create Testobject
17:       dieBank = new Bank( mock_dbFactory, ... );
18:    }
19:    ......
20: }
```

Quelltext 2-19: Fabrik als Mock-Objekt

Diese Vorgehensweise hat den Vorteil, dass man das Verhalten der aufgerufenen Fabrik-Methode abhängig von deren Argumenten spezifizieren kann (falls es anders als in Quelltext 2-19 gezeigt welche gibt). Man könnte z.B. in einigen Fällen Mock-Objekte, in anderen Fällen jedoch reale kollaborierende Objekte zurückgeben.

d) Statt der oben genannten aufwändigen Fabrik-Lösung kann die CUT auch Fabrik-Methoden für die Erzeugung der Kollaborator-Objekte definieren, welche sie selbst aufruft. In einem Testfall kann man dann statt der CUT eine abgeleitete Klasse instanziieren, welche diese Fabrik-Methoden überschreibt. Ein Nachteil dieser Vorgehensweise ist allerdings, dass der Testfall abhängig von der Realisierung der CUT wird, denn er muss in der abgeleiteten Klasse das Verhalten der überschriebenen Fabrik-Methode nachbilden. Diese könnte beispielsweise abhängig von übergebenen Argumenten völlig unterschiedliche Objekte erzeugen.

Das folgende Beispiel zeigt das Überschreiben einer Fabrik-Methode der CUT, um Mock-Objekte zu erzeugen. Es wird das JMock-Framework verwendet.

```
1:  public class Bank
2:  {
3:     private IDatenbank dieKonten;
4:     .....
5:     public Bank()
6:     {
7:        dieKonten = createDatenbank();
8:        .....
9:     }
10:    public IDatenbank createDatenbank()
11:    {
12:       return new Datenbank();
13:    }
14: }
```

```
1:  public class Testclass
2:  {  .....
3:     private Bank dieBank;
4:     private IDatenbank mock_kontenDb;
5:     private class BankWithMock extends Bank
6:     {
7:        @Override
8:        public IDatenbank createDatenbank()
9:        {  mock_kontenDb = context.mock(IDatenbank.class);
10:          return mock_kontenDb;
11:       }
12:    }
13:    @Before
14:    public void setUp() throws Exception
15:    {
16:       // Create Testobject
17:       dieBank = new BankWithMock();
18:    }
19:    ......
20: }
```

Quelltext 2-20: Überschreiben einer Fabrikmethode der CUT in der Testklasse

e) [Fowler, 2004] beschreibt als weitere Möglichkeit das „Service Locator“-Pattern. Der „Service Locator“ ist ein Objekt, welches einer Applikation alle benötigten Services zur Verfügung stellt. Jedes kollaborierende Objekt kann man als Service auffassen. Der „Service-Locator“ lässt sich z.B. durch Anwendung des „Singleton“-Patterns (siehe [Gamma et al., 2004], Seite 157) als Klasse realisieren, von der es nur eine Instanz gibt. Dieses „Singleton“-Objekt hält die kollaborierenden Objekte (Services) in Instanz-

variablen oder in einer Hash-Tabelle. Die kollaborierenden Objekte werden dem „Service Locator“ z.B. als Konstruktor-Parameter oder als Parameter einer Methode von außen übergeben. Die CUT-Instanz kann sich die kollaborierenden Objekte über entsprechende Methoden der „Service Locator“-Klasse holen. Das folgende Beispiel zeigt die Anwendung des „Service Locator“-Patterns in einem Test, welcher das JMock-Framework verwendet:

```
1:  public class ServiceLocator
2:  {
3:     private static ServiceLocator instance = null;
4:     IDatenbank datenbank = null;
5:     .....
6:     private ServiceLocator() { }
7:     public static ServiceLocator getInstance()
8:     {
9:        if (instance == null) instance = new ServiceLocator();
10:       return instance;
11:    }
12:    public static void loadDatenbank(IDatenbank datenbank)
13:    {
14:       getInstance().datenbank = datenbank;
15:    }
16:    public static IDatenbank getDatenbank()
17:    {
18:       return getInstance().datenbank;
19:    }
20:    .....
21: }
```

```
1:  public class Bank
2:  {
3:     private IDatenbank dieKonten;
4:     .....
5:     public Bank()
6:     {
7:        dieKonten = ServiceLocator.getDatenbank();
8:        .....
9:     }
10: }
```

```
1:  public class Testclass
2:  {  .....
3:     private IDatenbank mock_kontenDb;
4:     @Before
5:     public void setUp() throws Exception
6:     {
7:        // Create Mock-Objects for Collaborators
8:        mock_kontenDb = context.mock(IDatenbank.class);
9:        // Load Service Locator
10:       ServiceLocator.loadDatenbank(mock_kontenDb);
11:       .....
12:       // Create Testobject
13:       dieBank = new Bank();
14:    }
15:    ......
16: }
```

Quelltext 2-21: Übergabe eines Mock-Objekts durch das „Service Locator“-Pattern

3 Ein Werkzeug zum Ersetzen kollaborierender Objekte

3.1 Aufgabenstellung

Wie schon in Kapitel 2.1 erwähnt, kann es sehr schwierig sein, einen Testfall mit realen kollaborierenden Objekten auszuführen. Die oben beschriebenen Mock-Frameworks bieten die Möglichkeit, dynamisch Mock-Objekte zu erzeugen, welche die kollaborierenden Objekte ersetzen können. Folgt man dem „Need Driven Development"-Ansatz aus Kapitel 2.2, muss man schon vor dem Schreiben des CUT-Code in einem Testfall das Verhalten der Mock-Objekte beschreiben. Dies setzt allerdings voraus, dass man eines der oben beschriebenen Mock-Frameworks kennt und dessen Syntax beherrscht. Da man die Mock-Objekte manuell implementieren muss, ist diese Vorgehensweise außerdem relativ fehleranfällig.

Eine Alternative wäre, dass man den Testfall zunächst ohne Mock-Objekte schreibt. Man erzeugt eine Test-Fixture mit Instanzen der CUT und übergibt ihnen reale kollaborierende Objekte (siehe Kapitel 2.5). Anschließend spezifiziert man den Test, welcher auf die CUT-Instanzen eine Reihe von Methoden aufruft, sowie die erwarteten Testresultate. Nach der Erstellung des Testfalls schreibt man den Code der CUT. Schließlich startet man ein Refaktorisierungswerkzeug, welches den Testfall automatisiert so umgestaltet, dass er statt der realen kollaborierenden Objekte Mock-Objekte verwendet.

Diese Arbeit stellt ein neues „Create Mock Objects (CMO)"-Refactoring vor, welches als Plugin für Eclipse realisiert ist. Es soll JUnit4-Testfälle unabhängig von komplexen externen Schnittstellen der CUT machen. Dazu muss es alle kollaborierenden Objekte identifizieren, die den verwendeten CUT-Instanzen von außen übergeben werden, siehe Kapitel 2.5. Die JUnit4-Testfälle sind in einer Testklasse definiert, siehe Kapitel 2.3. Findet das CMO-Refactoring keine kollaborierenden Objekte, soll es dem Anwender einen Hinweis anzeigen, dass als Vorbedingung ein weiteres Refactoring notwendig ist, um die CUT von ihren externen Abhängigkeiten zu entkoppeln. Infrage kommt hier z.B. ein „Inject Dependency"-Refactoring, siehe Kapitel 6.1. Gibt es kollaborierende Objekte, kann der Anwender auswählen, welche davon durch Mock-Objekte ersetzt werden sollen. Außerdem kann er das Mock-Framework selektieren, welches für die Erzeugung der Mock-Objekte verwendet werden soll. Das CMO-Refactoring modifiziert den Java-Classpath des Testprojekts entsprechend, um dieses Mock-Framework in der Testklasse verwenden zu können.

Anschließend soll das CMO-Refactoring den Code der Testklasse modifizieren und Mock-Objekte erzeugen, die das von der CUT benötigte Protokoll zur Verfügung stellen. Diese Mock-Objekte sollen dem jeweiligen Testobjekt statt der entsprechenden kollaborierenden Objekte übergeben werden. Die Modifikationen im Testprojekt und im Code der Testklasse, welche vom verwendeten Mock-Framework abhängig sind, sollen über einen Eclipse Extension-Point realisiert werden. Dadurch kann das Mock-Framework flexibel ausgetauscht werden. Das Verhalten der Mock-Objekte muss im Rahmen dieser Arbeit nicht spezifiziert werden. Es soll aber in den betroffenen Testfällen ein TODO-Stub eingefügt werden, um den Anwender darauf aufmerksam zu machen, dass dieses Verhalten angegeben werden muss, damit die modifizierten Testfälle korrekt ablaufen.

3.2 Voreinstellungen des CMO-Plugins

Bevor der Anwender das CMO-Refactoring verwenden kann, muss er das Framework auswählen, welches für die Erzeugung der Mock-Objekte benützt werden soll. Dies geschieht über eine Eclipse-Voreinstellungsseite:

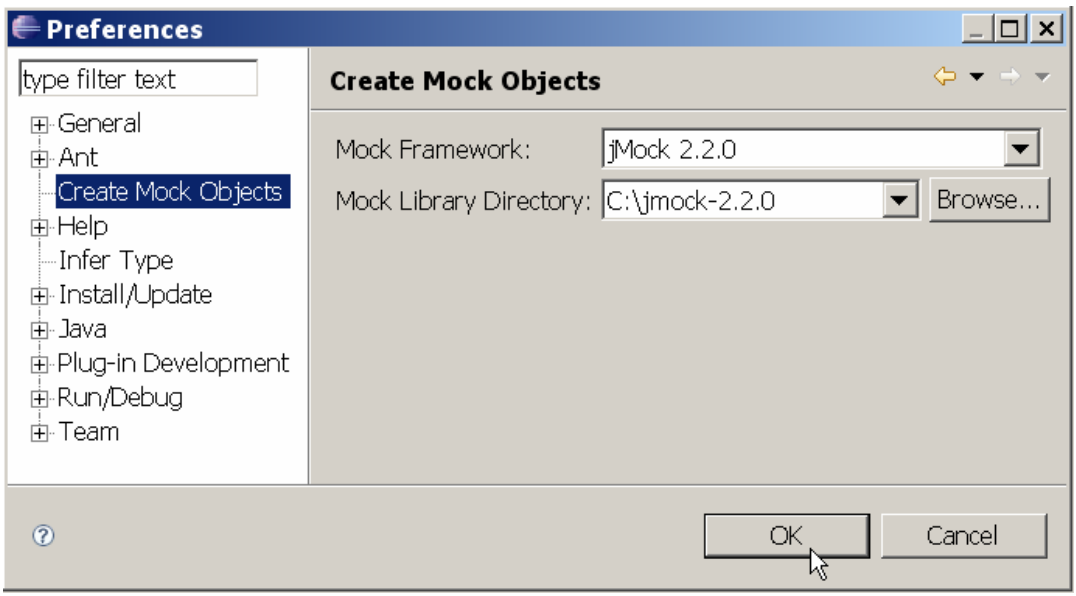

Abbildung 3-1: Voreinstellungen des CMO-Plugins

Die in Kapitel 4 beschriebene Implementierung des CMO-Plugins unterstützt das JMock-Framework. Die Bibliotheken des Mock-Frameworks müssen in ein Datei-Verzeichnis kopiert werden, welches auf der Voreinstellungsseite anzugeben ist.

3.3 Annotationen

Das CMO-Refactoring stellt zwei Annotationen zur Verfügung:

- Eine Testklasse kann mit einer „Class under Test“-Annotation (`@CUT`) versehen werden, welche als Parameter den voll qualifizierten Namen der CUT enthält, die in den Testfällen getestet wird.
- Das Refactoring versieht die Testklasse automatisch mit der Marker-Annotation `@DoMockCollaborators`. Diese Annotation wird im Code der umgestalteten Testfälle abgefragt (siehe Kapitel 3.7.4). Ist sie vorhanden, werden die Testfälle mit Mock-Objekten statt mit realen kollaborierenden Objekten ausgeführt.

Um die Annotationen im Projekt der Testklasse benützen zu können, stellt das CMO-Refactoring einen Classpath Container zur Verfügung (siehe Kapitel 4.10.1). Dieser wird dem Classpath des Java-Projekts durch das Refactoring automatisch hinzugefügt. Er kann dem Projekt aber auch manuell über die Seite „Java Build Path“ der Projekt-Eigenschaften hinzugefügt werden, falls man die Testklasse schon vor dem Refactoring mit der `@CUT`-Annotation versehen möchte. Dazu muss die Schaltfläche „Add Library“ im Ordner „Libraries“ betätigt werden:

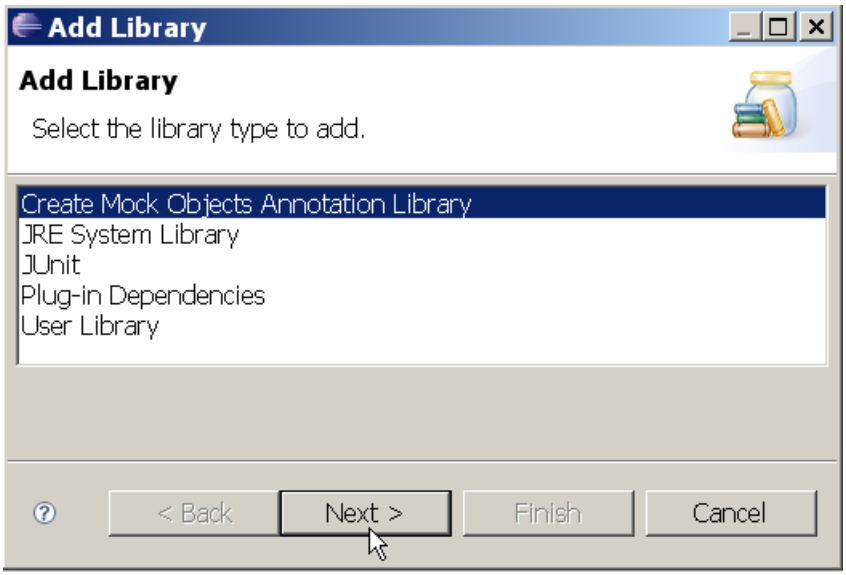

Abbildung 3-2: Hinzufügen der Create Mock Objects Annotation Library

3.4 Start des CMO-Refactorings

Das CMO-Refactoring soll kollaborierende Objekte in JUnit4-Testfällen finden, welche den CUT-Instanzen von außen übergeben werden, und diese durch Mock-Objekte ersetzen. Die Übergabe der Kollaborateure kann nicht nur im Testfall selbst, sondern auch in einer Setup-Methode stattfinden. Diese Methode baut eine Test-Fixture auf, welche von allen Testfällen benützt wird, siehe Kapitel 2.3. Ersetzt man dort die Kollaborateure der erzeugten CUT-Instanzen durch Mock-Objekte, wirkt sich diese Änderung auf alle Testfälle aus. Deshalb ist es sinnvoll, das CMO-Refactoring entweder auf einen einzelnen Testfall oder auf alle Testfälle inklusive der Setup-Methoden[1] anzuwenden. Eine mit `@BeforeClass` annotierte Klassenmethode wird nicht berücksichtigt, da hier keine CUT-Instanzen erzeugt werden sollten, siehe Kapitel 2.3. Das CMO-Refactoring untersucht auch keine Methoden der Testklasse, welche nicht annotiert sind, denn bei diesen Methoden ist nicht klar, ob es sich um Testcode oder zu testenden Code handelt.[2]

Bevor ein Anwender das CMO-Refactoring starten kann, muss er die Testklasse sowie die Class under Test (CUT) auswählen. Dies kann er tun, indem er im Code der Testklasse einen Typen selektiert, welcher der CUT entspricht. Wählt er anschließend im Kontext-Menü den Punkt „Replace Collaborators with Mock-Objects“ aus, wird das CMO-Refactoring für alle Testfälle in der Testklasse inkl. der Setup-Methoden gestartet:

[1] JUnit erlaubt mehrere mit @Before annotierte Methoden, siehe Kapitel 2.3.
[2] Obwohl Testfälle vom Code der CUT getrennt werden sollten, ist es nicht auszuschließen, dass beides vermischt wird.

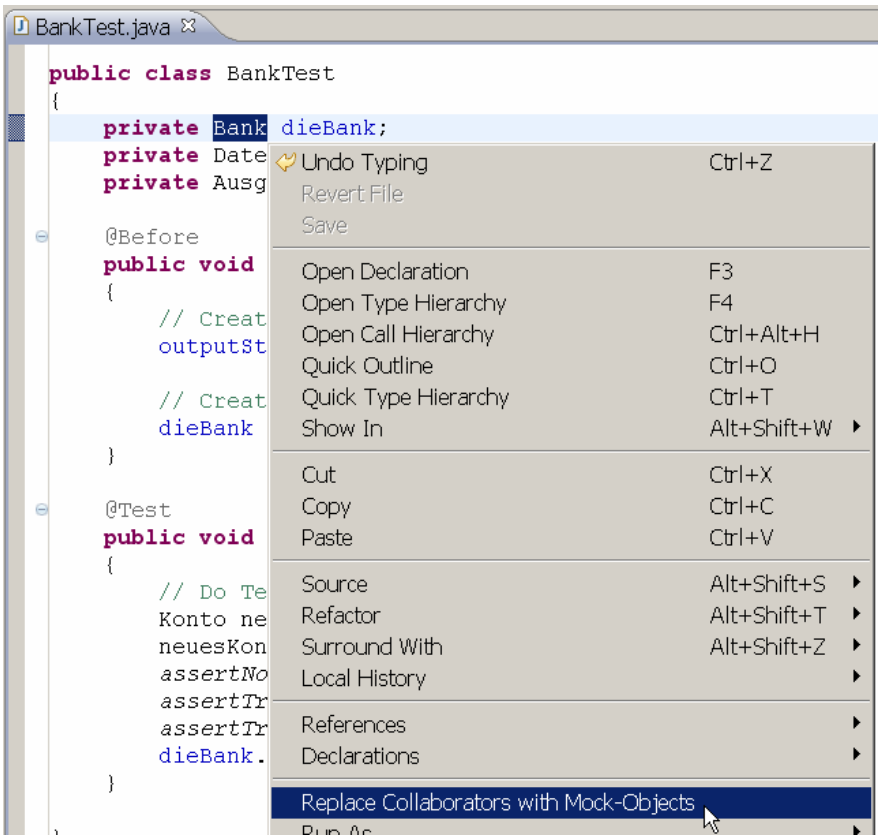

Abbildung 3-3: Selektion des CUT-Typen im Code der Testklasse

Alternativ kann der Anwender im Package Explorer einen Testfall oder die Testklasse selbst selektieren und auch hier den oben genannten Menüpunkt auswählen. Im ersten Fall wird das CMO-Refactoring nur auf den selektierten Testfall angewendet, im zweiten Fall auf alle Testfälle inkl. der Setup-Methoden. Da die CUT hier nicht ausgewählt wird, versucht das CMO-Refactoring, sie aus dem Namen der Testklasse zu ermitteln. Dieser besteht oft aus dem Namen der CUT mit dem Zusatz „Test“, z.B. „BankTest“. Alternativ kann die Testklasse mit einer neuen „Class under Test“-Annotation (`@CUT`) versehen werden (siehe Kapitel 3.3). Diese Annotation enthält als Parameter den voll qualifizierten Namen der CUT:

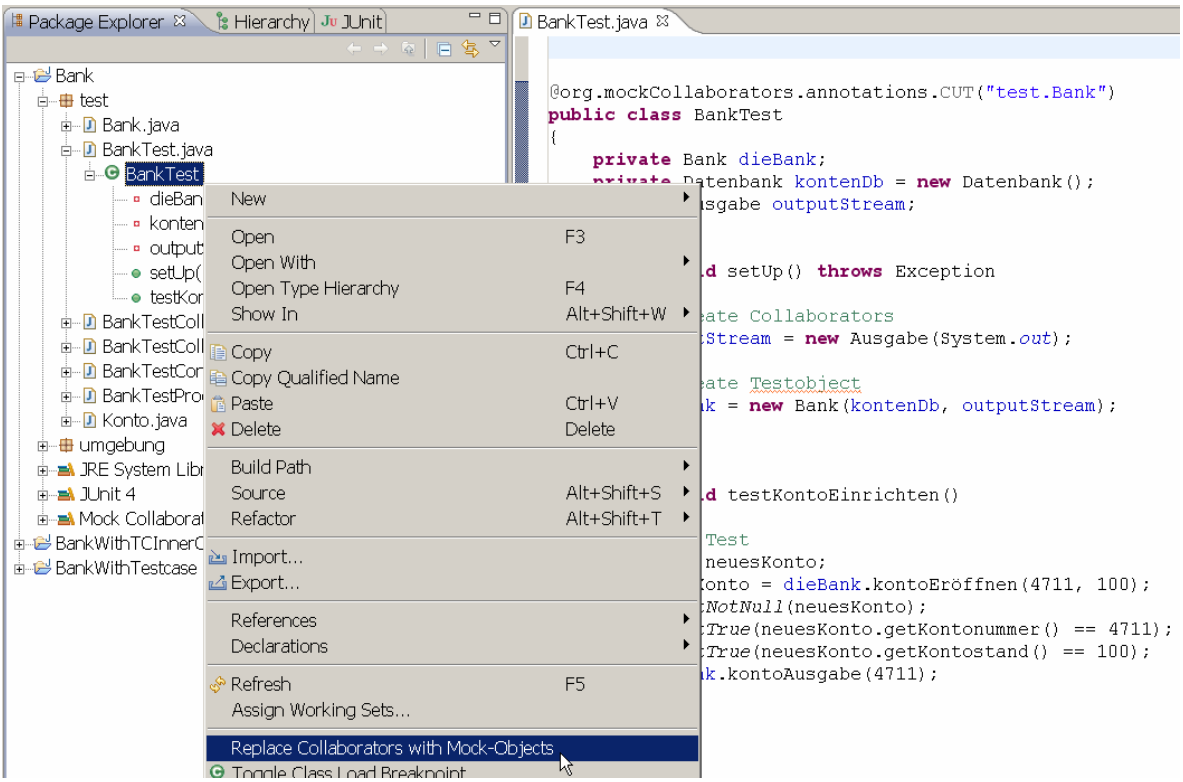

Abbildung 3-4: Selektion der Testklasse

Wurde das CMO-Refactoring durch Auswahl des Menüpunkts „Replace Collaborators with Mock-Objects“ gestartet, erscheint folgende Startseite:

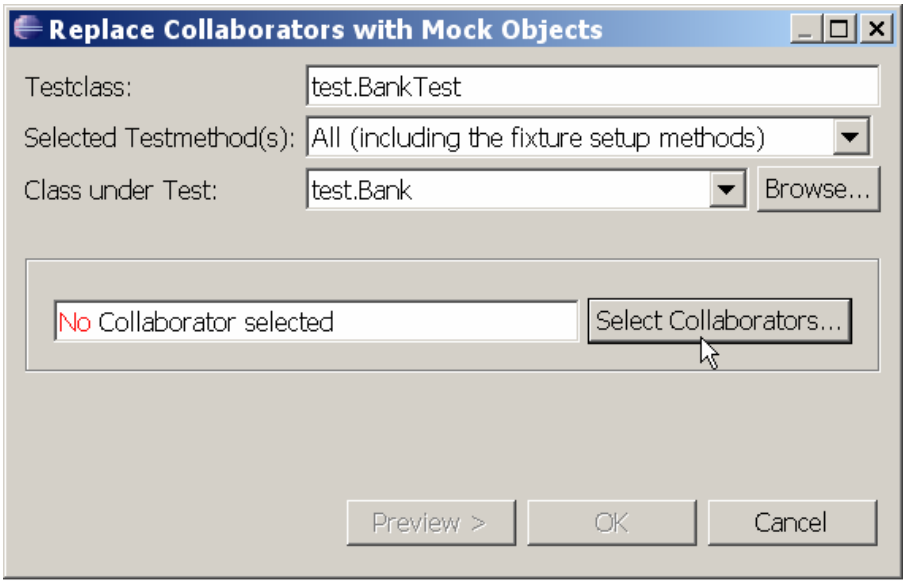

Abbildung 3-5: Startseite des CMO-Refactorings

Auf dieser Seite kann der Anwender noch einmal die Einstellungen für das Refactoring verändern. Drückt er die „Browse“-Schaltfläche zur Auswahl der CUT, wird ein Dialog angezeigt, welcher alle Klassen anbietet, die im Projekt der Testklasse deklariert sind:

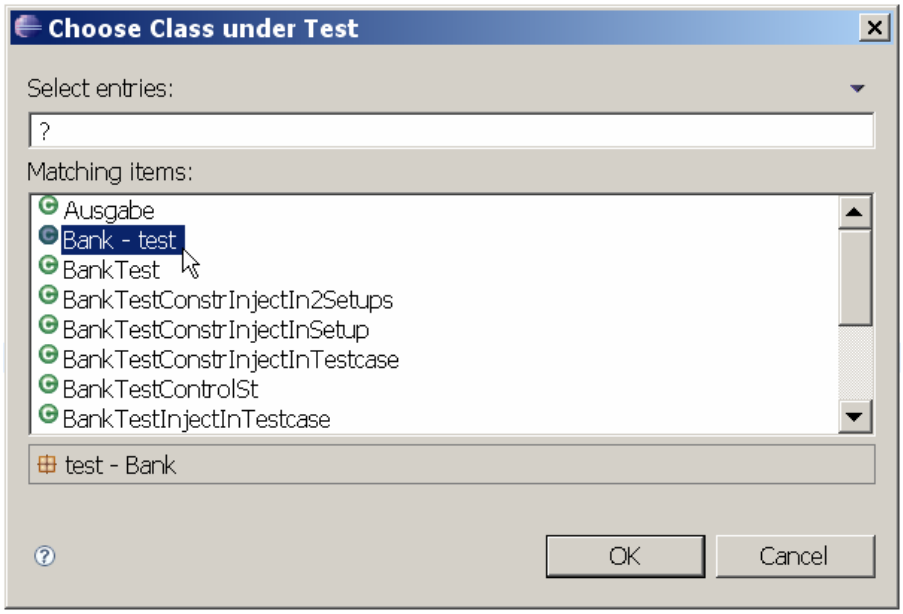

Abbildung 3-6: Dialog zur Auswahl der CUT

Außerdem kann man über ein Pulldown-Menü einen einzelnen Testfall in der Testklasse oder alle Testfälle inkl. der Setup-Methoden auswählen:

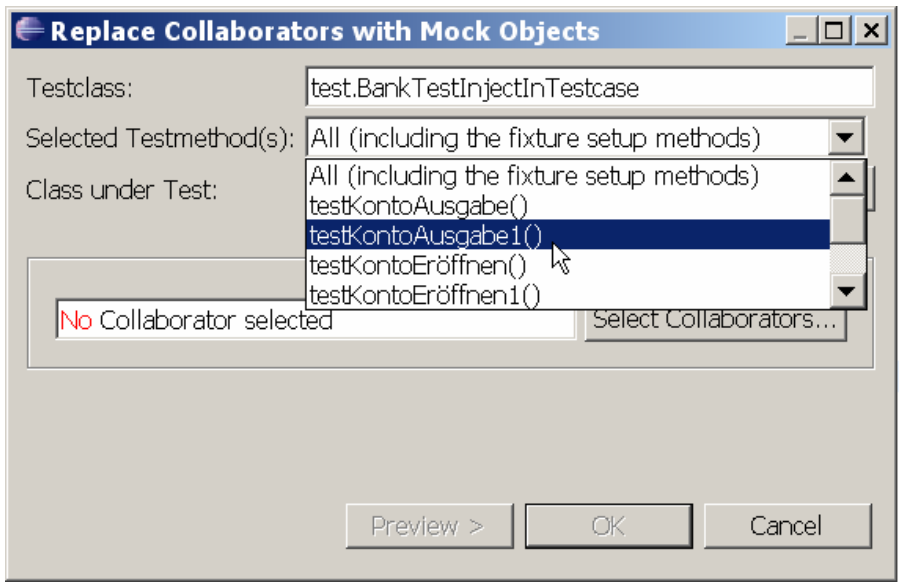

Abbildung 3-7: Pulldown-Menü zur Auswahl des Testfalls

Nach jeder Änderung der Einstellungen sucht das CMO-Refactoring automatisch nach Kollaborateuren der CUT in den selektierten Methoden, siehe Kapitel 3.5. Anschließend kann der Anwender über die Schaltfläche „Select Collaborators“ diejenigen auswählen, welche durch Mock-Objekte ersetzt werden sollen, siehe Kapitel 3.6.

3.5 Analyse der Testfälle auf Kollaborateure der CUT

Für Tests mit Mock-Objekten muss das Design der CUT so ausgelegt sein, dass man ihren Instanzen die kollaborierenden Objekte von außen übergeben kann. Kapitel 2.5 beschreibt dazu verschiedene Möglichkeiten. Das CMO-Refactoring unterstützt die Übergabe der kollaborierenden Objekte als Konstruktor-Parameter („Constructor-Injection“) oder als Parameter eines Methodenaufrufs. Die in der Testklasse selektierten Testfälle bzw. Setup-Methoden werden mit Hilfe einer statischen Codeanalyse auf folgende Ausdrücke untersucht:

- Ausdrücke, in denen die CUT instanziiert wird. Diese Ausdrücke werden in der Java Language Specification als „Class Instance Creation Expressions“ bezeichnet, siehe [JLS3], Kapitel 15.9. Sie enthalten das Schlüsselwort „new“ sowie einen Aufruf eines Konstruktors der zu instanziierenden Klasse, in diesem Fall die CUT.
- Ausdrücke, in denen eine Methode der CUT aufgerufen wird. Sie heißen „Method Invocation Expressions“, siehe [JLS3], Kapitel 15.12. Das CMO-Refactoring beschränkt sich hier auf die in Kapitel 2.5 beschriebene „Setter Injection“, und wertet nur Methodenaufrufe mit genau einem Parameter aus. Hätte eine Methode mehrere Parameter, wäre es wahrscheinlich, dass es sich nicht bei allen um kollaborierende Objekte handelt. In diesem Fall wären die externen Schnittstellen der CUT nicht klar von ihrer Funktionalität getrennt.

Für einen Konstruktor- bzw. Setter-Parameter müssen folgende Bedingungen gelten, damit er vom CMO-Refactoring als Kollaborateur erkannt wird:

- Der aktuelle Parameter ist entweder ein Variablenname oder ein Ausdruck „`this`.variablenname“, in dem mittels der Variable `this` auf eine Instanzvariable zugegriffen wird.
- Der Typ des korrespondierenden formalen Konstruktor- bzw. Setter-Parameters ist keiner der folgenden Typen:
  - der CUT-Typ selbst
  - `Object`, `Class`, `String` oder Subklassen von `Throwable`
  - primitive Typen `byte`, `short`, `int`, `long`, `boolean`, `float`, `double`, `char`
  - Arrays

Die in Kapitel 2.4 beschriebenen Mock-Frameworks erlauben die Erzeugung von Mock-Objekten nur für Klassen und Interfaces, nicht für primitive Typen und Arrays. Für Instanzen der Metaklasse `Class` oder der Klasse `String` können keine Mock-Objekte erzeugt werden. Das RMock-Framework liefert beim Versuch, ein Mock-Objekt für die Klasse `Object` zu erzeugen, eine `NullPointerException`. Außerdem ist es nach Meinung des Autors nicht sinnvoll, Mock-Objekte für Fehlermeldungen oder Exceptions zu erzeugen. Ist der Typ eines formalen Konstruktor- bzw. Setter-Parameters keiner der genannten Typen, handelt es sich bei diesem Parameter um eine potentielle externe Abhängigkeit der CUT.

Zusätzlich muss der aktuelle Parameter aber eine Variable sein, weil das CMO-Refactoring derzeit nur solche Argumente durch Mock-Objekte austauschen kann, siehe Kapitel 3.7.1.

Findet das CMO-Refactoring nur Konstruktor- bzw. Setter-Argumente, welche diese Bedingung nicht erfüllen, zeigt es dem Anwender einen entsprechenden Hinweis an. Enthält einer der oben genannten Ausdrücke mindestens einen Parameter, der alle Bedingungen erfüllt und dadurch als kollaborierendes Objekt erkannt wurde, wird dieser Ausdruck im Folgenden als „Kollaborator-Ausdruck“ bezeichnet.

Findet das CMO-Refactoring in allen Testfällen inkl. der Setup-Methoden keine potentiellen externen Abhängigkeiten der CUT, zeigt es eine Fehlermeldung an und weist den Anwender darauf hin, dass die CUT z.B. mit einem „Inject Dependency“-Refactoring umgestaltet werden sollte (siehe Kapitel 7.1). Wurde nur ein einzelner Testfall selektiert, empfiehlt das CMO-Refactoring dem Anwender, die Suche nach kollaborierenden Objekten auf alle Testfälle inkl. der Setup-Methoden auszudehnen.

3.6 Auswahl der zu ersetzenden kollaborierenden Objekte

Hat das CMO-Refactoring Ausdrücke mit kollaborierenden Objekten gefunden, kann der Anwender auswählen, welche der darin enthaltenen kollaborierenden Objekte durch Mock-Objekte ersetzt werden sollen. Dazu drückt er die Schaltfläche „Select Collaborators“ auf der Startseite des Refactorings (siehe Abbildung 3-5). Daraufhin werden ihm auf einer Reihe Eingabeseiten die Anweisungen angezeigt, in denen sich die Kollaborator-Ausdrücke befinden. Aus Gründen der Übersichtlichkeit werden für die Anzeige immer die Anweisungen verwendet, welche sich direkt im Rumpf der jeweiligen Testmethode (Testfall oder Setup-Methode) befinden, selbst wenn es sich dabei um Kontrollstrukturen oder Blöcke handelt, die ihrerseits wieder eine Menge von Anweisungen enthalten, in denen sich letztendlich die Kollaborator-Ausdrücke befinden. Die kollaborierenden Objekte werden auf der Eingabeseite farblich hervorgehoben und können über Checkboxen selektiert werden:

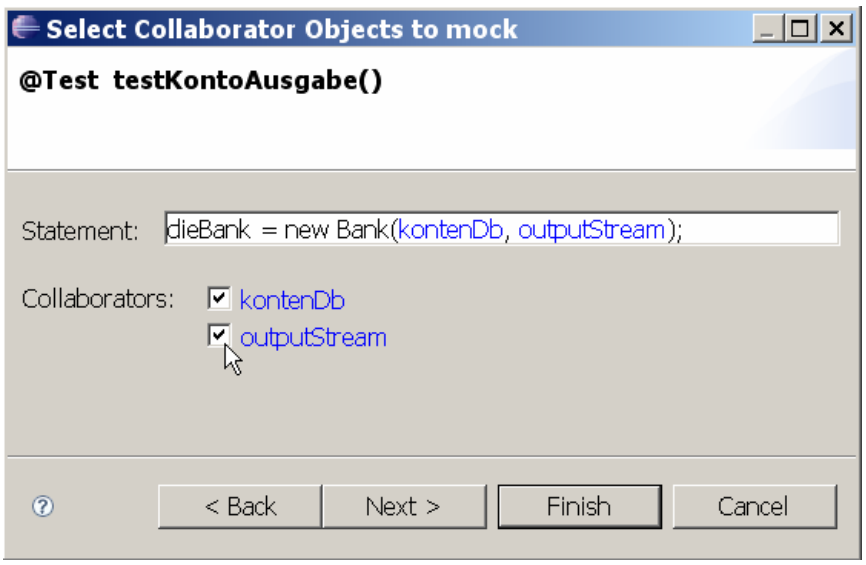

Abbildung 3-8: Selektion der kollaborierenden Objekte

Nach der Auswahl gelangt der Anwender durch Drücken der Finish-Schaltfläche zur Startseite des CMO-Refactorings zurück (siehe Abbildung 3-5). Dort kann er das Refactoring über die OK-Schaltfläche starten.

3.7 Refactoring der Testklasse

3.7.1 Überblick

Ein Refactoring, welches einen Testfall so umgestalten soll, dass er mit Mock-Objekten statt den realen kollaborierenden Objekten läuft, muss folgende Aufgaben erfüllen:

- Durch eine statische Code-Analyse muss festgestellt werden, wo die kollaborierenden Objekte an die CUT-Instanz übergeben werden. Das CMO-Refactoring unterstützt die Übergabe als Konstruktor-Parameter oder Parameter einer Setter-Methode, siehe Kapitel 3.5.
- Für die kollaborierenden Objekte müssen Mock-Objekte erzeugt werden. Außerdem muss das Verhalten der Mock-Objekte spezifiziert werden, welches vom jeweiligen Testfall abhängt.
- Die Mock-Objekte müssen der CUT-Instanz statt der kollaborierenden Objekte übergeben werden.

Durch diese Modifikationen wird die CUT mit Mock-Objekten statt realen kollaborierenden Objekten arbeiten. Das Verhalten der Mock-Objekte kann i.d.R. direkt nach ihrer Erzeugung spezifiziert werden. Dies gilt allerdings nicht für Mock-Objekte in einer Setup-Methode, denn JUnit ruft eine mit `@Before` annotierte Methode vor jedem Testfall auf, siehe Kapitel 2.3. Für die Angabe des Mock-Verhaltens müsste die Setup-Methode stets wissen, welcher Testfall als nächstes gestartet wird.[3] Das Verhalten der Mock-Objekte muss deshalb immer in den Testfällen angegeben werden.[4] Das CMO-Refactoring fügt anstelle des Verhaltens lediglich TODO-Stub's in die Testfälle ein, aber die Umgestaltung des Code sollte dennoch so ausgelegt sein, dass eine Spezifikation des Mock-Verhaltens an diesen Stellen möglich ist.

Für die Übergabe der Mock-Objekte an die CUT-Instanz gibt es verschiedene Möglichkeiten. Wird ein kollaborierendes Objekt durch eine Variable bezeichnet, könnte man die Anweisung suchen, in der das kollaborierende Objekt der Variablen zugewiesen wird. Diese Anweisung könnte man ändern und der Variablen ein neu erzeugtes Mock-Objekt zuweisen. Somit müsste man den Konstruktor- bzw. Methodenaufruf nicht antasten. Die genannte Zuweisung ist aber nicht immer leicht zu finden. Sie könnte z.B. in verschachtelten Kontrollstrukturen enthalten sein, in denen u.U. nicht jeder Zweig tatsächlich durchlaufen wird. Außerdem braucht man die Variable, um das erwartete Verhalten des Mock-Objekts zu spezifizieren. Dies ist unproblematisch, solange man dies an der gleichen Stelle tun kann, wo auch das Mock-Objekt erzeugt wird. Wird ein kollaborierendes Objekt der CUT-Instanz aber in einer Setup-Methode übergeben, muss dessen Verhalten in den Testfällen spezifiziert werden, s.o. Dazu muss die Variable in allen Testfällen sichtbar sein. Dies ist bei einer Instanzvariablen der Fall, bei einer lokalen Variablen jedoch nicht. Selbst bei einer Instanzvariablen hätte man immer noch das Problem, dass ihr nach der Speicherung des

[3] Selbst wenn sie das wüßte, ist es nicht praktikabel, dass die Setup-Methode Testfall-spezifischen Code enthält.

[4] Eine Ausnahme ist die Spezifikation eines Default-Verhaltens, welches für alle Testfälle gleich ist; vgl. die "defaults section" des RMock-Frameworks in Kapitel 2.4.4 und Quelltext 2-13.

Mock-Objekts im weiteren Verlauf der Setup-Methode noch ein anderes Objekt zugewiesen werden könnte. Damit wäre das Mock-Objekt verloren.

Damit dies nicht passieren kann, deklariert das CMO-Refactoring für jedes Mock-Objekt eine eigene Instanzvariable in der Testklasse, im Folgenden als „Mock-Variable“ bezeichnet. Vor der Übergabe des kollaborierenden Objekts an die CUT-Instanz wird das Mock-Objekt erzeugt und der Mock-Variablen zugewiesen. Anschließend tauscht das CMO-Refactoring im Konstruktor- bzw. Methodenaufruf den Namen der ursprünglichen Variable gegen den Namen der Mock-Variable aus. Diese Lösung hat auch den Vorteil, dass für das Verhalten des Mock-Objekts nur die Methoden angegeben werden müssen, welche die CUT-Instanz aufruft. Methoden, die der Testfall selbst auf das reale kollaborierende Objekt aufruft (z.B. um es zu initialisieren), sind außen vor, denn dieses Objekt wird weiterhin durch die ursprüngliche Variable bezeichnet.

Der Austausch des Konstruktor- bzw. Methoden-Parameters durch eine Mock-Variable würde auch dann funktionieren, wenn das kollaborierende Objekt als Ergebnis eines Ausdrucks übergeben wird, z.B.: „`dieBank = new Bank(new Datenbank(),...);`“. Bevor der Konstruktor der Klasse Bank aufgerufen wird, wird hier zunächst der Ausdruck „`new Datenbank()`“ ausgewertet. Dieser Ausdruck kann aber Seiteneffekte haben, ohne die der Testfall nicht mehr laufen würde. Mit dem bloßen Austauschen des Ausdrucks ist es also nicht getan, sondern das Refactoring wäre hier deutlich aufwändiger. Aus diesem Grund konzentriert sich das CMO-Refactoring auf kollaborierende Objekte, die durch Variablen bezeichnet werden, siehe auch Kapitel 3.5.

3.7.2 Dynamische Aspekte

Werden in der Testklasse verschiedene CUT-Instanzen erzeugt, kann man die gleiche Variable mehrmals zur Übergabe eines kollaborierenden Objekts verwenden, z.B.:

```
1:  // Create Testobjects
2:  kontenDb = ServiceLocator.getDatenbank();
3:  dieBank  = new Bank(kontenDb, ...);
4:  kontenDb = ServiceLocator.getDatenbank();
5:  dieBank2 = new Bank(kontenDb, ...);
```

Quelltext 3-1: Mehrmalige Verwendung einer Variablen als Konstruktor-Parameter

Eine Variable stellt einen Objektbezeichner dar. Ihr können während der Programmausführung verschiedene Objekte zugewiesen werden. Auf der rechten Seite einer Zuweisung steht ein beliebiger Ausdruck. Dies kann z.B. die Instanziierung einer Klasse sein. Der Ausdruck kann aber auch - wie in Zeile 2 des obigen Beispiels - einen oder mehrere Methodenaufrufe enthalten. Ist ein Methodenaufruf dynamisch gebunden, wird erst zur Laufzeit ermittelt, welche Methode tatsächlich ausgeführt wird. In diesem Fall kann durch eine statische Codeanalyse nicht mit Sicherheit bestimmt werden, welches Objekt der Ausdruck zurückliefert.

Selbst wenn in mehreren Zuweisungen der gleiche Ausdruck verwendet wird, kann man nicht davon ausgehen, dass er immer das gleiche Objekt zurückliefert. Der Getter in den Zeilen 2 und 4 des obigen Beispiels wird dies wahrscheinlich tun. Wird stattdessen aber eine

Factory-Methode aufgerufen, z.B. „`dbFactory.createDatenbank()`“, dann liefert der Ausdruck möglicherweise stets verschiedene Objekte. Um sicher zu gehen, müsste man den Code der Methoden analysieren, die in dem Ausdruck aufgerufen werden, sowie die Methoden, die dort wiederum aufgerufen werden usw. Sind dynamisch gebundene Methoden im Spiel, kommt man aber auch hier mit einer statischen Codeanalyse nicht weiter.

Gewissheit über die Identität der Objekte, die durch eine Variable bezeichnet oder durch einen Ausdruck geliefert werden, liefert letztendlich nur eine Laufzeitanalyse der Testfälle z.B. mit Hilfe entsprechender Java-Profiler. Diese erscheint aber im Rahmen eines Refactorings nicht sinnvoll, denn dieses wird nicht während der Ausführung eines Programms bzw. Testfalls durchgeführt. Vielmehr wird ein Refactoring beim Editieren einer Java-Quelldatei gestartet, um die Code-Struktur des Programms zu verbessern. Das CMO-Refactoring konzentriert sich deshalb auf eine statische Codeanalyse. Es erzeugt für jedes an die CUT übergebene (und vom Anwender selektierte) kollaborierende Objekt ein eigenes Mock-Objekt nebst eigener Mock-Variable, selbst wenn wie in Quelltext 3-1 möglicherweise das gleiche kollaborierende Objekt an mehrere CUT-Instanzen übergeben wird. Dies erscheint aus Sicht der Mock-Frameworks nicht problematisch, denn für jedes Mock-Objekt kann ein eigenes Verhalten angegeben werden.

3.7.3 Projekteinstellungen und globale Codeanpassungen

Als erster Schritt des Refactorings muss die Testklasse so angepasst werden, dass die Testfälle das eingestellte Mock-Framework benützen können. Dazu werden zum Classpath des Java-Projektes, in dem die Testklasse deklariert ist, die Bibliotheken (`.jar`-Dateien) des Mock-Frameworks hinzugefügt. Außerdem werden die notwendigen import-Deklarationen in die Java-Quelldatei der Testklasse eingefügt.

Die in Kapitel 4 beschriebene Implementierung des CMO-Plugins unterstützt das JMock-Framework. Für dieses Framework muss die Klasse Klasse `JUnit4Mockery` instanziiert werden, siehe Kapitel 2.4.2. Da der formale Typ eines Konstruktor- oder Methoden-Parameters sowohl eine Klasse als auch ein Interface sein kann, wird die JMock-Erweiterung `ClassImposteriser` verwendet und folgende Instanzvariable in der Testklasse deklariert:

```
private Mockery context = new JUnit4Mockery() {{
   setImposteriser(ClassImposteriser.INSTANCE); }};
```

Die Testklasse selbst wird mit der Annotation `@CUT` (siehe Kapitel 3.3) versehen, welche den Namen der CUT enthält. Ist diese Annotation bereits vorhanden, wird der Name der CUT aktualisiert. Um die Testfälle nach der Umgestaltung mit Mock-Objekten oder realen kollaborierenden Objekten ausführen zu können, wird die Testklasse mit der Marker-Annotation `@DoMockCollaborators` (siehe ebenfalls Kapitel 3.3) versehen. Diese Annotation wird im umgestalteten Code abgefragt, s.u.

3.7.4 Umgestaltung der Kollaborator-Ausdrücke

In einem Kollaborator-Ausdruck wird die CUT instanziiert oder eine Setter-Methode der CUT aufgerufen. Außerdem hat das CMO-Refactoring in diesem Ausdruck Kollaborateure gefunden, siehe Kapitel 3.5. Jedes vom Anwender selektierte kollaborierende Objekt muss durch ein Mock-Objekt ersetzt werden. Ein kollaborierendes Objekt wird entweder durch einen Variablennamen oder durch einen Ausdruck „`this`.variablenname“ bezeichnet, siehe die Bedingungen in Kapitel 3.5. Diese Variable wird im Folgenden als „Kollaborator-Variable“ bezeichnet. Wie bereits in den Kapiteln 3.7.1 und 3.7.2 beschrieben, wird für jedes zu ersetzende kollaborierende Objekt ein eigenes Mock-Objekt erzeugt. Dies gilt auch dann, wenn die gleiche Kollaborator-Variable mehrfach als Parameter in Konstruktor- oder Methodenaufrufen verwendet wird. Jedes Mock-Objekt wird einer neuen Instanzvariablen, genannt Mock-Variable, zugewiesen, die in der Testklasse deklariert wird.

Als Typ eines Mock-Objekts und Typ der zugehörigen Mock-Variable wird der Typ des formalen Konstruktor- oder Methoden-Parameters verwendet, für den die Kollaborator-Variable als aktueller Parameter übergeben wurde. Es wäre auch möglich, den Typen der Kollaborator-Variable selbst zu verwenden, denn dieser ist entweder identisch mit dem formalen Parametertypen oder er ist ein Subtyp desselben. Im letzteren Fall würde man das Protokoll des Mock-Objekts aber unnötig erweitern, denn im CUT-Konstruktor bzw. in der Methode der CUT können nur die Methoden des Mock-Objekts aufgerufen werden, die im Typ des formalen Pameters oder seinen Supertypen deklariert sind.[5] Handelt es sich bei dem formalen Parametertyp um einen generischen Typ, muss als Typ des Mock-Objects der sogenannte „raw type“ verwendet werden, d.h. der generische Typ ohne Typ-Parameter. Ein parametrisierter Typ wird von den Mock-Frameworks nicht akzeptiert. Existiert für den Typ des Mock-Objekts noch keine import-Deklaration in der Java-Quelldatei der Testklasse, wird sie eingefügt.

Der Name einer Mock-Variablen folgt folgendem Schema:

`mock_«Name Kollaborator-Variable»_[«Index»_]«Testfallname»`

Wird die gleiche Kollaborator-Variable in einem Testfall mehrmals zur Übergabe eines kollaborierenden Objekts verwendet, unterscheiden sich die entsprechenden Mock-Variablen durch einen Index. Bei der ersten Mock-Variablen wird der Index nicht angegeben. Ab der zweiten Mock-Variablen gleichen Namens beginnt er mit 1 und wird bei jeder weiteren gleichnamigen Mock-Variablen um 1 inkrementiert.

Zu Beginn eines Testfalls oder einer Setup-Methode wird für jedes dort selektierte kollaborierende Objekt ein Mock-Objekt erzeugt und der entsprechenden Mock-Variablen zugewiesen. Anschließend fügt das CMO-Refactoring einen TODO-Stub für die Verhaltens-Spezifikation der Mock-Objekte ein, falls die Mock-Objekte in einem Testfall erzeugt wurden. Geschah dies aber in einer Setup-Methode, kann das Mock-Verhalten dort nicht angegeben werden, siehe Kapitel 3.7.1. Stattdessen wird in diesem Fall ein TODO-Stub am Beginn aller Testfälle

[5] Mit einem Type-Cast könnten auch Methoden einer anderen Klasse aufgerufen werden; benützt die CUT allerdings dieses Konstrukt, wäre ihre externe Schnittstelle nicht sauber definiert.

eingefügt, denn die in der Setup-Methode erzeugte Test-Fixture wird von jedem Testfall benützt.

Nach der Erzeugung der Mock-Objekte müssen die Argumente der Kollaborator-Ausdrücke, welche den selektierten kollaborierenden Objekten entsprechen, durch die Namen der korrespondierenden Mock-Variablen ersetzt werden. Der Rumpf des Testfalls `testKonto-Ausgabe()` könnte z.B. folgende Anweisung enthalten:

```
1:  dieBank = new Bank(kontenDb, outputStream);
```

Quelltext 3-2: Zuweisung einer neu erzeugten CUT-Instanz

Hier werden einer neu erzeugten CUT-Instanz die beiden kollaborierenden Objekte `kontenDb` und `outputStream` übergeben. Sollen beide Objekte durch Mock-Objekte ersetzt werden, könnte ein mögliches Refactoring inklusive der Instanziierung der Mock-Objekte und des TODO-Stubs folgendermaßen aussehen:

```
1:  // Mock-objects for collaborator simulation
2:  mock_kontenDb_testKontoAusgabe = context.mock(IDatenbank.class);
3:  mock_outputStream_testKontoAusgabe = context.mock(Ausgabe.class);
4:  // TODO Specify the behavior of the mock-objects
5:  .....
6:  // Collaborators have been replaced by mock-objects!
7:  dieBank = new Bank(mock_kontenDb_testKontoAusgabe,
8:                      mock_outputStream_testKontoAusgabe);
```

Quelltext 3-3: Ersetzen kollaborierender Objekte durch Mock-Objekte

Mit dieser Änderung würde die erzeugte CUT-Instanz nur noch mit Mock-Objekten arbeiten. Dies ist aber nicht immer im Sinne des Anwenders, denn dieser möchte den Testfall in manchen Fällen auch mit den realen kollaborierenden Objekten laufen lassen. Um dies steuern zu können, wird in der Testklasse eine neue `boolean`-Klassenvariable `collaboratorsAreMocked` deklariert, welche das Vorhandensein der `@DoMockCollaborators`-Annotation überprüft:

```
static final boolean collaboratorsAreMocked =
      BankTest.class.isAnnotationPresent(DoMockCollaborators.class);
```

Ein Kollaborator-Ausdruck kann in einem anderen Ausdruck, z.B. einer Zuweisung (s.o.), enthalten sein. Dieser Ausdruck ist wiederum in einer Anweisung enthalten. Das CMO-Refactoring sucht diese Anweisung und erzeugt eine neue `If`-Abfrage, in der die obige Klassenvariable abgefragt wird. Die Anweisung wird dupliziert und mit Mock-Variablen in den `then`-Zweig sowie in ihrem ursprünglichen Zustand in den `else`-Zweig eingefügt. Für das Beispiel aus Quelltext 3-2 sieht das tatsächliche Refactoring inklusive der Deklaration der Mock-Variablen folgendermaßen aus:

```
1:  public class BankTest
2:  {
3:    .....
4:    static final boolean collaboratorsAreMocked =
5:        BankTest.class.isAnnotationPresent(DoMockCollaborators.class);
6:    private IDatenbank mock_kontenDb_testKontoAusgabe;
7:    private Ausgabe mock_outputStream_testKontoAusgabe;
8:    .....
```

```
9:    @Test
10:   public void testKontoAusgabe() throws Throwable
11:   {
12:     // Mock-objects for collaborator simulation
13:     mock_kontenDb_testKontoAusgabe = context.mock(IDatenbank.class);
14:     mock_outputStream_testKontoAusgabe = context.mock(Ausgabe.class);
15:     // TODO Specify the behavior of the mock-objects
16:     .....
17:     // Inject mock-objects or real collaborators into CUT-instance
18:     if (collaboratorsAreMocked)
19:         dieBank = new Bank(mock_kontenDb_testKontoAusgabe,
20:                            mock_outputStream_testKontoAusgabe);
21:     else
22:         dieBank = new Bank(kontenDb, outputStream);
23:     .....
24:   }
25    .....
26: }
```

Quelltext 3-4: Refactoring einer Anweisung im Rumpf einer Testmethode

Werden die kollaborierenden Objekte der CUT als Methoden-Parameter übergeben, sieht das Refactoring ähnlich aus. Der Kollaborator-Ausdruck ist in diesem Fall ein Setter-Aufruf. Werden in mehreren Setter-Aufrufen kollaborierende Objekte ersetzt, und befinden sich diese Aufrufe in direkt aufeinanderfolgenden Anweisungen, erzeugt das CMO-Refactoring nur eine „`if(collaboratorsAreMocked)`“-Abfrage. Z.B. könnten sich im Rumpf des Testfalls `testKontoAusgabe()` folgende Anweisungen befinden:

```
1:  dieBank = new Bank();
2:  dieBank.setKontoDb(kontenDb);
3:  dieBank.setOutputStream(outputStream);
```

Quelltext 3-5: Übergabe der Kollaborateure durch Setter-Aufrufe

Selektiert der Anwender die kollaborierenden Objekte `kontenDb` und `outputStream`, sieht das Refactoring-Ergebnis folgendermaßen aus:

```
1:  // Mock-objects for collaborator simulation
2:  mock_kontenDb_testKontoAusgabe = context.mock(IDatenbank.class);
3:  mock_outputStream_testKontoAusgabe = context.mock(Ausgabe.class);
4:  // TODO Specify the behavior of the mock-objects
5:  .....
6:  dieBank = new Bank();
7:  // Inject mock-objects or real collaborators into CUT-instance
8:  if (collaboratorsAreMocked)
9:  {
10:    dieBank.setKontoDb(mock_kontenDb_testKontoAusgabe);
11:    dieBank.setOutputStream(mock_outputStream_testKontoAusgabe);
12: }
13: else
14: {
15:    dieBank.setKontoDb(kontenDb);
16:    dieBank.setOutputStream(outputStream);
17: }
```

Quelltext 3-6: Refactoring zweier Setter-Aufrufe

Die folgenden Java-Anweisungen können einen oder mehrere Kollaborator-Ausdrücke enthalten:

- Eine Anweisung, die einen Ausdruck enthält (z.B. die Zuweisung in Quelltext 3-2)
- Eine Return- oder Throw-Anweisung (enthält genau einen Ausdruck)
- Eine Assert-Anweisung (enthält einen oder zwei Ausdrücke)
- Ein Konstruktor- oder Superkonstruktor-Aufruf, der Ausdrücke für die Argumente des Konstruktors enthält.
- Eine Anweisung für eine Variablen-Deklaration, welche aus einem Typ und mehreren Variablen-Deklarations-Fragmenten besteht. Die Fragmente enthalten jeweils einen Variablen-Namen und optional einen Ausdruck als Initialisierer.
- Eine Anweisung für eine `If`-, `Do`-, `While`-, `For`-, `Switch`-, oder `Synchronized`-Kontrollstruktur. Diese Anweisungen enthalten einen oder mehrere Ausdrücke, z.B. die `If`-Bedingung oder die Durchlaufbedingung der Schleife. Außerdem enthalten sie eine oder mehrere Anweisungen, die einmal, mehrmals oder abhängig von einer Bedingung ausgeführt werden sollen. Dabei werden mehrere Anweisungen meist in einem Block gruppiert, der auch eine Anweisung ist.

Alle Anweisungen mit Ausnahme der Variablen-Deklarationen und der Kontrollstrukturen können, wie in Quelltext 3-4 gezeigt, dupliziert und mit bzw. ohne Mock-Objekten in die Zweige der „`if(collaboratorsAreMocked)`“-Abfrage eingefügt werden.

Eine Eigenschaft der Kontrollstrukturen ist, dass sie sowohl Ausdrücke als auch Anweisungen enthalten. Beide Sprachkonstrukte können wiederum Kollaborator-Ausdrücke enthalten, z.B.:

```
1:  if ((dieBank = new Bank(kontenDb, new Ausgabe(System.out))) != null)
2:  {
3:     dieBank2 = new Bank(new Datenbank(), outputStream);
4:  }
```

Quelltext 3-7: If-Anweisung mit mehreren Kollaborator-Ausdrücken

Hat der Anwender die kollaborierenden Objekte `kontenDb` und `outputStream` selektiert, dupliziert das CMO-Refactoring lediglich die umgebende `If`-Anweisung, da sie auch die Anweisung mit dem zweiten Kollaborator-Ausdruck enthält. Das Ergebnis inklusive der Instanziierung der Mock-Objekte würde folgendermaßen aussehen, wenn sich die `If`-Anweisung im Rumpf der Methode `testKontoAusgabe()` befindet:

```
1:  // Mock-objects for collaborator simulation
2:  mock_kontenDb_testKontoAusgabe = context.mock(IDatenbank.class);
3:  mock_outputStream_testKontoAusgabe = context.mock(Ausgabe.class);
4:  // TODO Specify the behavior of the mock-objects
5:  .....
6:  // Inject mock-objects or real collaborators into CUT-instance
7:  if (collaboratorsAreMocked)
8:  {
9      if ((dieBank = new Bank(mock_kontenDb_testKontoAusgabe,
10:        new Ausgabe(System.out))) != null)
11:    {
12:       dieBank2 = new Bank(new Datenbank(),
13:          mock_outputStream_testKontoAusgabe);
14:    }
15: }
16: else
17: {
18:    if ((dieBank = new Bank(kontenDb, new Ausgabe(System.out))) != null)
19:    {
20:       dieBank2 = new Bank(new Datenbank(), outputStream);
21:    }
22: }
```

Quelltext 3-8: Refactoring einer Kontrollanweisung mit mehreren Kollaborator-Ausdrücken

Ein anderes Problem ist das Einfügen einer Variablen-Deklaration in die neue „`if(collaboratorsAreMocked)`“-Abfrage. Sie müsste in einem Block deklariert werden, der den `then`- bzw. `else`-Zweig der `If`-Abfrage bildet. In diesem Fall wären die deklarierten Variablen aber außerhalb des Blocks nicht sichtbar. Deshalb zerlegt das CMO-Refactoring eine Variablendeklaration:

- Zunächst wird eine neue Variablendeklaration erzeugt. Sie enthält lediglich die Variablen-Namen aller Fragmente ohne Initialisierer, sowie die Variablen-Namen aller Fragmente, in deren Initialisierern sich Kollaborator-Ausdrücke mit selektierten kollaborierenden Objekten befinden.
- Anschließend werden neue Anweisungen für die Fragmente mit Initialisierer erzeugt. Dabei muss darauf geachtet werden, dass alle Variablen im umgestalteten Code genau in der gleichen Reihenfolge initialisiert werden wie im ursprünglichen Code, denn die Initialisierungen können voneinander abhängen! Es kann z.B. zunächst einer Variablen ein Objekt zugewiesen werden. Anschließend kann diese Variable einer zweiten Variablen zugewiesen werden, so dass beide Variablen auf das gleiche Objekt zeigen. Die zweite Zuweisung würde aber nicht funktionieren, wenn die erste Variable noch nicht initialisiert wurde.
- Für Fragmente, deren Initialisierer keine Kollaborator-Ausdrücke mit selektierten kollaborierenden Objekten enthält, wird eine neue Variablendeklaration erzeugt. Dabei werden gleichartige aufeinanderfolgende Fragmente in einer Deklaration zusammengefasst.

- Für Fragemente, deren Initialisierer Kollaborator-Ausdrücke mit selektierten kollaborierenden Objekten enthält, wird eine neue Anweisung erzeugt, in welcher der Variablen ihr Initialisierer zugewiesen wird. Diese Anweisung wird dupliziert und mit Mock-Objekten bzw. realen kollaborierenden Objekten in beide Zweige einer „`if(collaboratorsAreMocked)`“-Abfrage eingefügt. Ähnlich wie oben werden bei gleichartigen aufeinanderfolgenden Fragmenten die entsprechenden Anweisungen in die gleiche `If`-Abfrage eingefügt.

Im Rumpf der Methode `testKontoAusgabe()` befindet sich z.B. folgende Variablendeklaration:

```
1:  Bank dieBank, dieBank2 = new Bank(kontenDb, new Ausgabe(System.out)),
2:       dieBank3 = new Bank(new Datenbank(), outputStream),
3:       dieBank4 = dieBank2;
```

Quelltext 3-9: Variablendeklaration mit mehreren Kollaborator-Ausdrücken

Selektiert der Anwender die kollaborierenden Objekte `kontenDb` und `outputStream` in den Initialisierern der Variablen `dieBank2` und `dieBank3`, sieht das Refactoring inkl. der Erzeugung der Mock-Objekte folgendermaßen aus:

```
1:  // Mock-objects for collaborator simulation
2:  mock_kontenDb_testKontoAusgabe = context.mock(IDatenbank.class);
3:  mock_outputStream_testKontoAusgabe = context.mock(Ausgabe.class);
4:  // TODO Specify the behavior of the mock-objects
5:  .....
6:  Bank dieBank, dieBank2, dieBank3;
7:  // Inject mock-objects or real collaborators into CUT-instance
8:  if (collaboratorsAreMocked)
9:  {
10:    dieBank2 = new Bank(mock_kontenDb_testKontoAusgabe,
11:                        new Ausgabe(System.out));
12:    dieBank3 = new Bank(new Datenbank(),
13:                        mock_outputStream_testKontoAusgabe);
14: }
15: else
16: {
17:    dieBank2 = new Bank(kontenDb, new Ausgabe(System.out));
18:    dieBank3 = new Bank(new Datenbank(), outputStream);
19: }
20: Bank dieBank4 = dieBank2;
```

Quelltext 3-10: Refactoring einer Variablendeklaration

4 Die Realisierung des „Create Mock Objects“-Plugins

4.1 Pakete und Architektur

Eclipse ([Eclipse]) ist eine universelle Open-Source-Platform für die Entwicklung beliebiger Software-Applikationen. Seit der Version 3.0 basiert Eclipse auf den Standards der Open Services Gateway Initiative ([OSGi]). Der Eclipse-Kern selbst besteht aus einem OSGi-Framework namens „Equinox“ ([Equinox]), welches einen OSGi-Server realisiert. Die eigentliche Funktionalität wird durch OSGi-Bundles, auch Eclipse-Plugins genannt, bereitgestellt, welche von diesem Kern beim Start von Eclipse geladen werden. Auch die weithin bekannte Java-Entwicklungsumgebung besteht aus mehreren dieser Plugins. Durch diese flexible Erweiterbarkeit entwickelt sich Eclipse zunehmend zum Industriestandard. Beispielsweise basieren sowohl das SAP Netweaver Developer Studio ([NetWeaver]) zur Entwicklung von Java EE-Unternehmensanwendungen als auch die Wind River Workbench ([Wind River]) zur Entwicklung von embedded Software inzwischen auf Eclipse.

Das „Create Mock Objects (CMO)“-Refactoring ist ebenfalls ein Eclipse-Plugin. Die Grundlagen zur Erstellung von Eclipse-Plugins sind in [Clayberg&Rubel 2006], [Gamma&Beck 2004] und [Daum 2006] ausführlich beschrieben und sollen deshalb hier nicht weiter erläutert werden. Die Konfiguration des Plugins, wie z.B. die implementierten Eclipse-Erweiterungspunkte, Abhängigkeiten zu anderen Eclipse-Plugins oder Build-Einstellungen sind in der Datei `plugin.xml` zu finden. Für die Refactoring-Anteile, welche spezifisch für ein bestimmtes Mock-Framework sind, definiert das CMO-Plugin den Erweiterungspunkt `mockRewriter`. Diese Arbeit stellt zwei Eclipse-Plugins „JMock 2.2.0 Extension for CMO“ und „JMock 2.4.0 Extension for CMO“ zur Verfügung, welche den Erweiterungspunkt `mockRewriter` für die Versionen 2.2.0 und 2.4.0 des JMock-Frameworks implementieren:

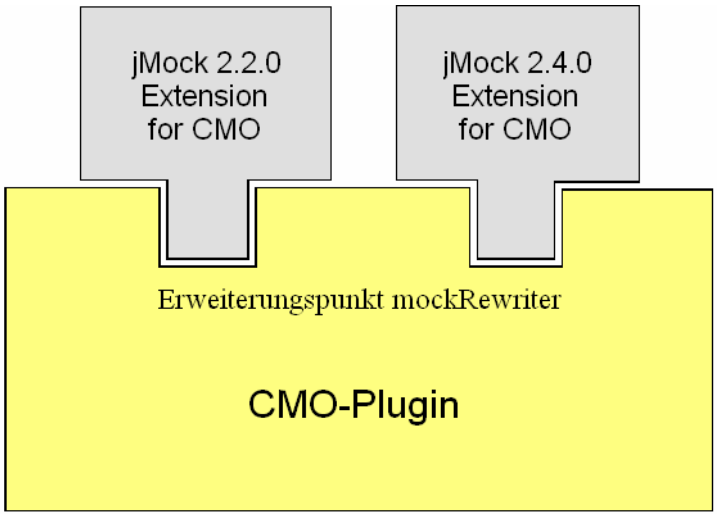

Abbildung 4-1: CMO-Plugin und Extension-Plugins für JMock

Die beiden Extension-Plugins bestehen lediglich aus der Plugin-Klasse selbst, sowie einer weiteren Klasse, die das Interface `IMockRewriter` implementiert und die Mock-Framework-spezifischen Codeanpassungen in der Testklasse durchführt:

Plugin org.JMock220ExtForCMO	
Klasse	Paket
JMock220Extension	org.JMock220ExtForCMO
JMock220Rewriter	org.JMock220ExtForCMO.rewrite

Plugin org.JMock240ExtForCMO	
Klasse	Paket
JMock240Extension	org.JMock240ExtForCMO
JMock240Rewriter	org.JMock240ExtForCMO.rewrite

Tabelle 4-1: Klassen der Extension-Plugins

Der Erweiterungspunkt `mockRewriter` sowie das Interface `IMockRewriter` sind in Kapitel 4.3 detailliert beschrieben. Nähere Informationen zu den Klassen `JMock220Rewriter` und `JMock240Rewriter` findet sich im Kapitel 4.10.

Das CMO-Plugin besteht aus folgenden Paketen:

org.createMockObjects	Dieses Paket enthält die zentralen Klassen für das CMO-Plugin und -Refactoring.
org.createMockObjects.actions	Hier befinden sich die Action-Klassen für den Start des CMO-Refactorings über Kontext-Menüs im Editor oder Package Explorer.
org.createMockObjects.annotations	Enthält die Klassen für die Annotations @CUT und @DoMockCollaborators.
org.createMockObjects.classpath	Hier befinden sich die Klassen für den Class-path Container, der die CMO Annotations enthält, sowie die Klassen, welche den Classpath eines Projektes im Rahmen des Refactorings verändern.
org.createMockObjects.collsearch	Enthält die Klassen für die Suche nach möglichen kollaborierenden Objekten im Code der Testfälle und Setup-Methoden.
org.createMockObjects.mockFrameworkIf	Enthält Klassen, welche die Schnittstelle zu Mock Framework-spezifischen Refactoring-Anteilen bilden.
org.createMockObjects.preferences	Enhält Klassen für die Voreinstellungen des CMO-Plugins.
org.createMockObjects.rewrite	Hier befinden sich Klassen, welche die Code-Änderungen in der Testklasse und den selektierten Testmethoden durchführen.
org.createMockObjects.ui	Dieses Paket enthält Klassen für die graphische Benutzeroberfläche zum Start des Plugins und zur Auswahl der zu ersetzenden Kollaborateure.
org.createMockObjects.unittest	Enhält Hilfsklassen zur Verwaltung der Testklasse und der CUT.

`org.createMockObjects.visitors`	Dieses Paket enthält Visitor-Klassen zum Durchsuchen des AST der Testklasse.

Tabelle 4-2: Pakete des CMO-Plugins

Das folgende UML-Diagramm zeigt die wichtigsten Objekte des CMO-Refactorings:

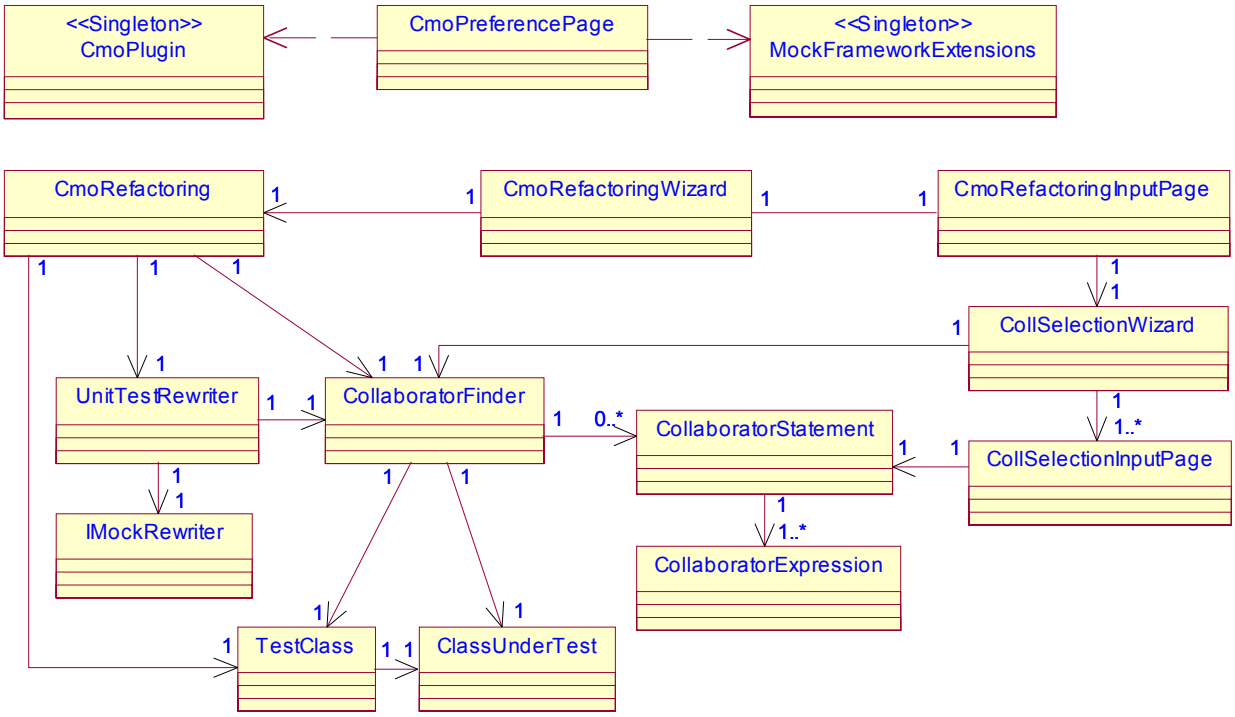

Abbildung 4-2: Die wichtigsten Objekte des CMO-Refactorings

Das CMO-Plugin wird durch die Klasse `CmoPlugin` realisiert. Von dieser Klasse gibt es nur eine Instanz, die das Eclipse-Framework beim Laden des Plugins erzeugt. Die Erkennung der spezifischen Anpassungen für verschiedene Mock-Frameworks übernimmt eine Singleton-Instanz der Klasse `MockFrameworkExtensions`. Ein `CmoPreferencePage`-Objekt baut eine graphische Seite auf, über die eine bestimmte Mock-Framework-Erweiterung für das Refactoring ausgewählt werden kann. Diese Selektion wird persistent in den Voreinstellungen des CMO-Plugins gespeichert. Ein `CmoRefactoringWizard`-Objekt erzeugt eine Instanz der Klasse `CmoRefactoring`, welche das CMO-Refactoring selbst realisiert. Die graphische Startseite des Refactorings wird durch ein `CmoRefactoring-InputPage`-Objekt aufgebaut. Ein `CollSelectionWizard`-Objekt ist für die graphische Selektion der zu ersetzenden kollaborierenden Objekte zuständig, wobei die einzelnen Eingabeseiten durch `CollSelectionInputPage`-Objekte realisiert werden. Eine Instanz der Klasse `CollaboratorFinder` übernimmt die Suche nach Ausdrücken, in denen einer CUT-Instanz kollaborierende Objekte übergeben werden. Jeder dieser Ausdrücke wird in einem `CollaboratorExpression`-Objekt gespeichert. Außerdem werden die Anweisungen im Rumpf der jeweiligen Testmethode, welche diese Kollaborator-Ausdrücke enthalten, in `CollaboratorStatement`-Objekten abgelegt. Die Testklasse und die CUT werden durch ein `TestClass`- bzw. `ClassUnderTest`-Objekt repräsentiert. Eine Instanz der Klasse `UnitTestRewriter` berechnet die Änderungen im Code der Testklasse. Dabei

werden die Mock-Framework-spezifischen Anpassungen durch ein Objekt durchgeführt, welches das Interface `IMockRewriter` implementiert. Dieses Objekt wird von einem der Extension-Plugins (s.o.) zur Verfügung gestellt.

4.2 Implementierung der Annotationen

Das CMO-Refactoring stellt die beiden Annotationen `@CUT` und `@DoMockCollaborators` zur Verfügung, siehe Kapitel 3.3. Sie sind im Paket `org.createMockObjects.annotations` in den gleichnamigen Klassen implementiert. Beide Annotation-Klassen sind mit der Annotation „`@Retention(RetentionPolicy.`*`RUNTIME`*`)`“ versehen, so dass sie vom Java-Compiler in die Class-Datei der Klasse eingefügt werden, in der sie verwendet werden. Dadurch kann man sie zur Laufzeit über das Java Reflection-API lesen. In Kapitel 3.7.4 wird z.B. gezeigt, wie das Vorhandensein der `@DoMockCollaborators`-Annotation durch Aufruf der Methode `isAnnotationPresent()` der Klasse `Class` geprüft und das Ergebnis der Klassenvariablen `collaboratorsAreMocked` zugewiesen wird.

Die Class-Dateien der Annotationen werden beim Übersetzen des CMO-Plugins in dessen Verzeichnis `lib` in der Bibliothek `org_createMockObjects_annotations.jar` gespeichert. Die Bibliothek kann einem Projekt in Form eines Classpath Containers hinzugefügt werden. Dieser „Create Mock Objects Annotation-Container“ wird durch die Klasse `CmoAnnotationContainer` im Paket `org.createMockObjects.classpath` realisiert, siehe Kapitel 4.10.1. Er wird im Folgenden kurz als „Annotation-Container“ bezeichnet. Die Klasse `CmoAnnotationContainer` implementiert das Interface `org.eclipse.jdt.core.IClasspathContainer`.

Der Annotation-Container kann einem Projekt über die Seite „Java Build Path“ in den Projekt-Eigenschaften hinzugefügt werden.[6] Anschließend betätigt man im Ordner „Libraries“ die Schaltfläche „Add Library“. Für die entsprechende graphische Eingabeseite implementiert das CMO-Plugin den Eclipse-Erweiterungspunkt `org.eclipse.jdt.ui.classpathContainerPage`, über den sich eine Wizard-Seite zum Erzeugen eines Classpath Containers einrichten lässt. Dazu wird die Identifikation `org.createMockObjects.annotationContainer` des Annotation-Containers angegeben, sowie die Klasse `CmoAnnotationContainerPage`, welche das Interface `org.eclipse.jdt.ui.wizards.IClasspathContainerPage` implementiert.

Die Klasse `CmoAnnotationContainerPage` befindet sich im Paket `org.createMockObjects.classpath` und ist eine Subklasse der Klasse `org.eclipse.jface.wizard.WizardPage`. Sie überschreibt die Methode `createControl()` ihrer Superklasse, um mit Hilfe von SWT-Widgets eine graphische Eingabeseite für die Erzeugung des Annotation-Containers aufzubauen. Eine detaillierte Beschreibung des Eclipse Standard Widget Toolkits (SWT) findet sich in [Daum 2006]. Deshalb soll hier nicht näher darauf eingegangen werden. Da der Anwender auf der Eingabeseite keine Parameter eingeben muss, enthält sie lediglich den Namen der Bibliothek, die hinzugefügt werden soll:

[6] Dies ist auch programmatisch durch das Refactoring möglich, siehe Kapitel 4.10.1.

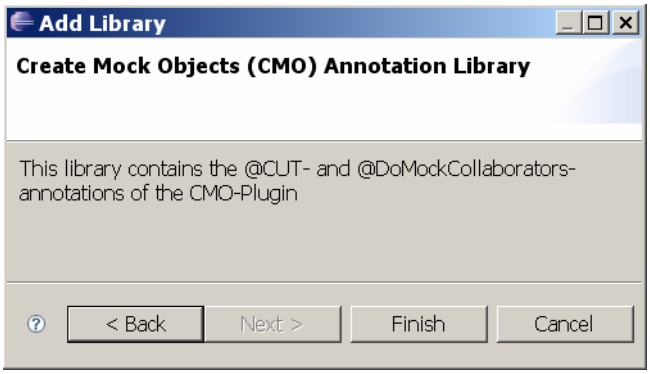

Abbildung 4-3: Wizard-Seite zum Hinzufügen der CMO Annotation Library

Die eigentliche Erzeugung des Annotation-Containers wird in der Methode `getSelection()` angestoßen. Sie ist im Interface `IClasspathContainerPage` (s.o.) definiert und wird vom Eclipse-Framework aufgerufen, nachdem die Finish-Schaltfläche der Eingabeseite gedrückt wurde. Diese Methode erzeugt durch Aufruf der Klassenmethode `newContainerEntry()` der Klasse `org.eclipse.jdt.core.JavaCore` einen neuen Classpath Container Entry. Als Pfad des Containers wird die Identifikation `org.createMockObjects.annotationContainer` (s.o.) des Annotation-Containers mitgegeben, welche in der Klasse `CmoAnnotationContainer` als Konstante deklariert ist:

```
1:  public IClasspathEntry getSelection()
2:  {   classPathEntry = JavaCore.newContainerEntry( new Path(
3:          CmoAnnotationContainer.CMO_ANNOTATION_CONTAINER_ID));
4:      return classPathEntry;
5:  }
```

Quelltext 4-1: Methode getSelection() der Klasse CmoAnnotationContainerPage

Eine detaillierte Beschreibung der verschiedenen Eclipse Classpath-Entries findet sich in [Meyer 2007], Anhang A.1. Sie sollen deshalb hier nicht näher beschrieben werden. Nach der Erzeugung des Classpath Container Entry's initialisiert das Eclipse-Framework den Classpath Container. Dazu sucht es nach einer Implementierung des Erweiterungspunktes `org.eclipse.jdt.core.classpathContainerInitializer` für die angegebene Container-Identifikation. Das CMO-Plugin implementiert diesen Erweiterungspunkt und definiert dafür im Paket `org.createMockObjects.classpath` die Klasse `CmoAnnotationContainerInit`. Sie ist eine Subklasse der Klasse `org.eclipse.jdt.core.ClasspathContainerInitializer` und überschreibt deren abstrakte Methode `initialize()` durch eine konkrete Implementierung.

Dabei wird zunächst der absolute Pfad ermittelt, unter dem die Bibliothek `org_createMockObjects_annotations.jar` in der Verzeichnisstruktur des CMO-Plugins zu finden ist. Der entsprechende Code ist in der Methode `getAnnotationLibPath()` realisiert und folgt im Wesentlichen der Vorgehensweise, die in [Meyer 2007], Anhang A.1, Listing 37, beschrieben ist. Anschließend erzeugt die Methode `initialize()` eine neue Instanz der Klasse `CmoAnnotationContainer` und übergibt ihr den Bibliothekspfad. Schließlich wird der neue Annotation-Container unter der schon oben erwähnten Container-Identifikation `org.createMockObjects.annotationContainer` an das übergebene Java-Projekt

gebunden. Dazu wird die Klassenmethode `setClasspathContainer()` der Klasse `org.eclipse.jdt.core.JavaCore` benützt:

```
1:  public void initialize(IPath path, IJavaProject project)
2:        throws CoreException
3:  {
4:     .....
5:     // Create CMO Annotation Container
6:     cmoAnnotationContainer = new CmoAnnotationContainer(path);
7:     // Get path of Annotation Library
8:     IPath annotationPath = getAnnotationLibPath();
9:     if (annotationPath != null)
10:      cmoAnnotationContainer.addLibrary(annotationPath);
11:    // Setup Classpath Entries in Container
12:    cmoAnnotationContainer.createClasspathEntries();
13:    // Set new Classpath Container for the specified project
14:    JavaCore.setClasspathContainer( path,
15:            new IJavaProject[] { project },
16:            new IClasspathContainer[] { cmoAnnotationContainer },
17:            null );
18: }
```

Quelltext 4-2: Initialisierung des CMO Annotation-Containers

4.3 Der Erweiterungspunkt mockRewriter

Einige Anteile des Refactorings, wie z.B. die Erzeugung der Mock-Objekte, sind vom verwendeten Mock-Framework abhängig. Deshalb definiert das CMO-Plugin für jede Erweiterung, die diese Anteile für ein spezifisches Mock-Framework zu Verfügung stellt, den Erweiterungspunkt `mockRewriter`. Sein Schema ist in der Datei `mockRewriter.exsd` zu finden. Jede Implementierung dieses Erweiterungspunkts muss ein Element `mockRewriter` spezifizieren, welches zwei obligatorische Attribute hat:

- den Namen des verwendeten Mock-Frameworks
- den vollqualifizierten Namen einer Klasse, welche das Interface `IMockRewriter` implementiert.

Das erwähnte Interface `IMockRewriter` befindet sich im Paket `org.createMock-Objects.mockFrameworkIf`. Es enthält folgende Methoden:

`void rewriteGlobalDefinitions(`
`  ASTRewrite astRewrite,`
`  ImportRewrite importRewrite,`
`  TypeDeclaration node);`
	Fügt die globalen Definitionen (z.B. Instanzvariablen und Imports) für das Mock-Framework in die übergebene Klassen-Deklaration ein.

`Statement buildMockObjectCreationStatement(`
`  AST ast,`
`  String varName,`
`  Type varType);`
	Generiert eine neue Anweisung, welche ein Mock-Objekt des übergebenen Typs erzeugt und einer Variablen des übergebenen Namens zuweist. Die Methode gibt die neu erzeugte Anweisung zurück.

`boolean isValidMockLibDirectory(` `  String dirPath);`	Überprüft, ob das übergebene Verzeichnis alle jar-Bibliotheken des Mock-Frameworks enthält.

`List<IPath> getMockLibraryPaths(` `  String dirPath);`	Liefert die Pfade aller jar-Bibliotheken des Mock-Frameworks als Liste von `IPath`-Objekten.

Tabelle 4-3: Interface IMockRewriter

Die Klassen `JMock220Rewriter` und `JMock240Rewriter` der JMock 2.2.0/2.4.0 Extension-Plugins (siehe Kapitel 4.1) implementieren das Interface `IMockRewriter` für die Versionen 2.2.0 und 2.4.0 des JMock-Frameworks, siehe Kapitel 4.10.

Beim Laden des CMO-Plugins muss dieses feststellen, welche Implementierungen für spezifische Mock-Frameworks (z.B. JMock, EasyMock, RMock oder andere) verfügbar sind. Dies übernimmt die Klasse `MockFrameworkExtensions` im Paket `org.createMock-Objects.mockFrameworkIf`, welche entsprechend des „Singleton“-Patterns (siehe [Gamma et al., 2004], Seite 157) realisiert ist und nur eine Instanz hat. Im Konstruktor der Klasse `CmoPlugin` wird die Methode `resolveExtensions()` dieser Singleton-Instanz aufgerufen, welche über die Extension-Registry der Eclipse-Platform alle Implementierungen des Erweiterungspunkts `mockRewriter` abfragt:

```
1:  public class MockFrameworkExtensions
2:  { .....
3:    private static final String EXTENSION_POINT_MOCK_REWRITER =
4:       "org.createMockObjects.mockRewriter";
5:    private static final String ELEMENT_MOCK_REWRITER = "mockRewriter";
6:    private static final String ATTRIBUTE_FRAMEWORK_NAME = "frameworkName";
7:    private static final String ATTRIBUTE_CLASS = "class";
8:    .....
9:    private ArrayList<String> frameworkNames = ...;
10:   private HashMap<String, IConfigurationElement> mockConfigElements = ...;
11:   private HashMap<String, IMockRewriter> mockRewriter = ...;
12    public void resolveExtensions()
13:   { // Get all extensions contributed to the extension point mockRewriter
14:     IExtensionRegistry registry = Platform.getExtensionRegistry();
15:     IExtensionPoint point = registry.getExtensionPoint(
16:         EXTENSION_POINT_MOCK_REWRITER);
17:     .....
18:     IExtension[] extensions = point.getExtensions();
19:     // Search for the element mockRewriter in all extensions
20:     for (IExtension ext : extensions)
21:     { IConfigurationElement[] configElements =
22:          ext.getConfigurationElements();
23:       for (IConfigurationElement ce : configElements)
24:       { if (ce.getName().equals(ELEMENT_MOCK_REWRITER))
25:         { // Get attribute frameworkName
26:           String frameworkName= ce.getAttribute(ATTRIBUTE_FRAMEWORK_NAME);
27:           // Save only the framework name and the config element
28:           frameworkNames.add(frameworkName);
29:           mockConfigElements.put(frameworkName, ce);
30:         }
31:       }
32:     }
33:   }
34:   .....
```

Quelltext 4-3: Abfrage der Eclipse Extension-Registry

Bei dieser Abfrage werden aus Performance-Gründen lediglich die Namen der unterstützten Mock-Frameworks sowie die zugehörigen Objekte des Typs `org.eclipse.core.run-`

time.IConfigurationElement gespeichert. Implementiert ein Plugin einen Erweiterungspunkt, wird in der Datei „plugin.xml“ ein entsprechender „extension“-Eintrag angelegt, welcher durch ein IConfigurationElement-Objekt repräsentiert wird. Ist eines der Attribute dieses „extension“-Eintrags eine Klasse, kann diese mittels der Methode createExecutableExtension() instanziiert werden. In der Methode getMockRewriter() des MockFrameworkExtensions-Objekts wird die Rewriter-Klasse erst instanziiert, wenn sie tatsächlich gebraucht wird (lazy initialization):

```
1:  public IMockRewriter getMockRewriter(String frameworkName)
2:  {   IMockRewriter rewriter = null;
3:      if (mockRewriter.containsKey(frameworkName))
4:        return mockRewriter.get(frameworkName);
5:      else
6:      {  if (mockConfigElements.containsKey(frameworkName))
7:         {  try
8:            {  Object o = mockConfigElements.get(frameworkName).
9:                    createExecutableExtension(ATTRIBUTE_CLASS);
10:              if (o instanceof IMockRewriter)
11:              {  rewriter = (IMockRewriter)o;
12:                 mockRewriter.put(frameworkName, rewriter);
13:              }
14:           } catch (CoreException e) { ..... }
15:        }
16:     }
17:     return rewriter;
18: }
```

Quelltext 4-4: Instanziierung des Mock Rewriters

4.4 Die Voreinstellungs-Seite des Plugins

Um das CMO-Refactoring zu starten, muss der Anwender zunächst das zu verwendende Mock-Framework aus den verfügbaren Implementierungen des oben genannten Erweiterungspunktes mockRewriter (s.o.) auswählen. Außerdem muss er das Verzeichnis angeben, in dem sich die jar-Bibliotheken des Mock-Frameworks befinden, siehe Abbildung 3-1 in Kapitel 3.2.

Die Voreinstellungs-Seite wird mit Hilfe des Eclipse-Erweiterungspunktes org.eclipse.ui.preferencePages realisiert. Dieser Erweiterungspunkt erfordert die obligatorische Angabe einer Klasse, welche das Interface org.eclipse.ui.IworkbenchPreferencePage implementiert. Die Klasse CmoPreferencePage (Paket org.createMockObjects.preferences) übernimmt dies für das CMO-Plugin. Die graphische Eingabe wird durch SWT-Widgets realisiert (siehe [Daum 2006]). Auf der Voreinstellungs-Seite werden die Namen aller Frameworks angezeigt, für die eine Implementierung des Erweiterungspunktes mockRewriter gefunden wurde. Diese Namen werden durch die Methode getFrameworkNames() des in Kapitel 4.3 beschriebenen MockFrameworkExtensions-Objekts geliefert.

Das Bibliotheksverzeichnis kann über einen Browse-Button ausgewählt werden. Beim Drücken dieses Buttons ruft die SelectionAdapter-Instanz des entsprechenden Button-Widgets die Methode mockDirButtonSelectionHandler() der Klasse CmoPreferencePage auf, welche das Ereignis verarbeitet. In dieser Methode wird ein Dialog

geöffnet, mit dem man durch das Dateisystem navigieren und ein Verzeichnis selektieren kann. Dieser Dialog ist bereits als SWT-Widget in der Klasse `org.eclipse.swt.widgets.DirectoryDialog` vordefiniert. Nach der Auswahl eines Verzeichnisses überprüft die Klasse `CmoPreferencePage`, ob in diesem Verzeichnis alle `jar`-Bibliotheken des ausgewählten Mock-Frameworks vorhanden sind. Dies geschieht über das in Kapitel 4.3 beschriebene `MockFrameworkExtensions`-Objekt. Die Methode `mockDirectoryIsValid()` dieses Objekts sucht die dem Mock-Framework-Namen entsprechende Implementierung des Interfaces `IMockRewriter`. Auf diese Implementierung wird anschließend die Methode `isValidMockLibDirectory()` (siehe Tabelle 4-3) aufgerufen.

Sind die Eingaben in Ordnung, wird in der Methode `checkStatusAndUpdatePage()` der Klasse `CmoPreferencePage` die Voreinstellungs-Seite gültig gesetzt, und damit der OK-Button freigeschaltet. Drückt der Anwender diesen Button, wird vom Eclipse-Framework die Methode `performOk()` derselben Klasse aufgerufen, welche die Einstellungen im Preference-Store des CMO-Plugins speichert.

Dieser Preference-Store ist ein persistentes Objekt, welches für ein Plugin genau einmal existiert. Es kann über das Interface `org.eclipse.jface.preference.IPreferenceStore` angesprochen werden. Der Preference-Store wird durch die Methode `preferenceStore()` der Klasse `CmoPlugin` geliefert, welche ihrerseits die Methode `getPreferenceStore()` auf die statische Instanz des Plugins aufruft. Er besteht aus einer Liste von Name/Wert-Paaren. Für das Lesen und Speichern des konkreten Mock-Framework-Bezeichners und des Mock-Framework-Verzeichnisses sind Klassenmethoden der Klasse `CmoPreferences` zuständig. Beide Werte werden als Strings gespeichert. In der Klasse `CmoPreferences` sind Konstanten für die Namen definiert, unter denen beide Werte im Preference-Store abgelegt werden.

Die Standardwerte für die beiden oben genannten Voreinstellungen werden durch Implementierung des Erweiterungspunktes `org.eclipse.core.runtime.preferences` gesetzt. Dabei wird die Klasse `CmoPreferencesInitializer` angegeben, welche das Interface `org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer` implementiert. Initialisiert Eclipse den Preference-Store des CMO-Plugins, wird diese Klasse instanziiert und die Methode `initializeDefaultPreferences()` aufgerufen. Sie setzt als Voreinstellung für den Mock-Framework-Namen den entsprechenden Namen der ersten gefundenen `mockRewriter`-Implementierung und als Voreinstellung für das Mock-Framework-Directory einen leeren String:

```
1:  public class CmoPreferencesInitializer
2:          extends AbstractPreferenceInitializer
3:  { @Override
4:    public void initializeDefaultPreferences()
5:    { // Initialize default preferences
6:      CmoPreferences.setDefaultMockFrameworkName(
7:          MockFrameworkExtensions.getInstance().getFirstFrameworkName());
8:      CmoPreferences.setDefaultMockFrameworkDirectory("");
9:      .....
10:   }
11: }
```

Quelltext 4-5: Initialisierung der CMO-Voreinstellungen

4.5 Repräsentation der Testklasse und der CUT

Das CMO-Refactoring gestaltet JUnit4-Testfälle um. Die Testklasse, in der diese Testfälle deklariert sind, sowie die CUT werden durch Instanzen der Klassen `TestClass` und `ClassUnderTest` repräsentiert. Beide Klassen befinden sich im Paket `org.createMock-Objects.unittest`. Das `TestClass`-Objekt verwendet folgende Hilfsobjekte:

- eine Instanz der Klasse `TestAST`, welche die Java-Quelldatei repräsentiert, in der die Testklasse deklariert ist.
- eine Instanz der Klasse `TestMethods`, welche die Testfälle und die Setup-Methoden verwaltet, die in der Testklasse enthalten sind.

Die Java Development Tools (JDT) der Eclipse-Platform verwalten den Java-Quellcode intern in einer Baumstruktur, die als „Abstract Syntax Tree“ (AST) bezeichnet wird. Der Eclipse-Artikel [Kuhn&Thomann 2006] enthält eine Übersicht über den Aufbau des AST. Im Konstruktor der Klasse `TestAST` wird deren Methode `createAstForCompilation-Unit()` aufgerufen. Hier wird mit Hilfe eines Objekts des Typs `org.eclipse.jdt.core.dom.ASTParser` die Java-Quelldatei der Testklasse analysiert und dafür ein AST des Typs `org.eclipse.jdt.core.dom.CompilationUnit` erzeugt. Die Klasse `TestAST` enthält zahlreiche Hilfsmethoden, um diesen Baum auszuwerten bzw. ihn zu durchsuchen. Z.B. sucht die Methode `resolveTypeDeclarationNode()` den AST-Knoten in diesem Baum, welcher der Deklaration eines bestimmten Typen entspricht. Sie wird von der Klasse `TestClass` benötigt, um den AST-Knoten mit der Typdeklaration der Testklasse zu finden.

Um die Testklasse nach JUnit4-Testfällen und Setup-Methoden zu durchsuchen, kann die Methode `resolveTestCasesAndFixtureSetupMethods()` des `TestClass`-Objekts aufgerufen werden. Die eigentliche Funktionalität wird hier an das `TestAST`-Objekt delegiert. In dessen Methode `searchTestCasesAndFixtureSetupMethods()` wird die Testklasse mit Hilfe eines `TestMethodSearchVisitor`-Objekts nach Methoden durchsucht, die mit den Annotationen `@Before` oder `@Test` versehen sind. Dabei wird das „Visitor“-Pattern angewandt, welches in [Gamma et al., 2004], Seite 301, beschrieben ist. Eclipse stellt für das Durchsuchen eines AST die abstrakte Klasse `org.eclipse.jdt.core.dom.ASTVisitor` zur Verfügung. Alle Visitor-Klassen des CMO-Plugins sind von dieser Klasse abgeleitet und überschreiben die notwendigen `visit()`-Methoden. Die gefundenen Testfälle und Setup-Methoden werden in einer neuen Instanz der Klasse `TestMethods` gespeichert. Dieses Objekt merkt sich außerdem, welche dieser Methoden vom Anwender (über die Startseite des CMO-Refactorings) für ein mögliches Refactoring selektiert wurden. Für die Änderung der Selektion gibt es in der Klasse `TestMethods` entsprechende Methoden, z.B. `selectAllTestMethods()` oder `selectTestMethod()`. Die Klasse `TestClass` stellt gleichnamige Wrapper-Methoden zur Verfügung, welche ihre Aufrufe lediglich an das `TestMethods`-Objekt weiterleiten. So braucht ein anderes Objekt nur das `TestClass`-Objekt zu kennen, wenn es auf die Testfälle oder Setup-Methoden zugreifen will.[7]

[7] Dadurch wird auch das Gesetz von Demeter eingehalten, welches besagt, dass Objekte nur mit denjenigen Objekten in ihrer Umgebung kommunizieren sollten, zu denen sie eine direkte Beziehung haben.

Die `setCut()`-Methoden der Klasse `TestClass` erzeugen eine neue Instanz der Klasse `ClassUnderTest`, um den Typ der CUT zu speichern. Der jeweiligen Methode wird entweder dieser Typ oder der vollqualifizierte CUT-Name als Parameter übergeben. Im letzteren Fall wird ein Typ im Projekt der Testklasse gesucht, dessen Name dem übergebenen String entspricht. Wenn beim Start des CMO-Refactorings (s.u.) die CUT nicht bekannt ist, wird mit Hilfe der Methode `setDefaultCut()` versucht, die CUT aus dem Namen der Testklasse zu ermitteln. Oft besteht der Name der Testklasse aus dem Namen der CUT und dem Anhang „Test“. Wird die Klasse „Bank“ getestet, könnte die Testklasse z.B. „BankTest“ heißen. Die Methode `setDefaultCut()` überprüft deshalb mit Hilfe der Klasse `java.util.regex.Matcher`, ob der Name der Testklasse mit dem regulären Ausdruck "(.+)(Test)" übereinstimmt. Ist dies der Fall, sucht sie mit Hilfe der Eclipse Java Search Engine (siehe Anhang A.2) eine Klasse im Java-Projekt der Testklasse, deren Name dem ersten Teil dieses Ausdrucks entspricht.

4.6 Start des Refactorings über Eclipse-Actions

Der Anwender kann das CMO-Refactoring starten, indem er im Code der Testklasse einen Typen bzw. im Package Explorer die Testklasse selbst oder eine Methode der Testklasse selektiert (siehe Kapitel 3.4). In beiden Fällen wählt er im entsprechenden Kontext-Menü den Punkt „Replace Collaborators with Mock-Objects“ aus.

Für die Erweiterung eines Kontext-Menüs stellt Eclipse den Erweiterungspunkt `org.eclipse.ui.popupMenus` zur Verfügung. Handelt es sich um das Menü eines selektierten Objekts, z.B. einer Klasse, wird diesem Erweiterungspunkt ein `objectContribution`-Element hinzugefügt. Ist das Menü unabhängig von der konkreten Selektion, z.B. bei der Selektion im Text einer editierten Java-Quelldatei, benützt man ein `viewerContribution`-Element. Jede dieser Kontributionen enthält eine Menge von `action`-Elementen, welche den neuen Menüeinträgen entsprechen. Die folgende Abbildung zeigt die Kontributionen in der Datei `plugin.xml` des CMO-Plugins:

```
1:  <extension
2:        point="org.eclipse.ui.popupMenus">
3:     <objectContribution
4:           .....
5:           objectClass="org.eclipse.jdt.core.IType">
6:        <action
7:              .....
8:              label="Replace Collaborators with Mock-Objects"
9:              class="org.createMockObjects.actions.CmoITypeAction"
10:             enablesFor="1"/>
11:    </objectContribution>
12:    <viewerContribution
13:          .....
14:          targetID="#CompilationUnitEditorContext">
15:       <action
16:             .....
17:             label="Replace Collaborators with Mock-Objects"
18:             class="org.createMockObjects.actions.CmoEditorAction"
19:             menubarPath="additions"/>
20:    </viewerContribution>
```

```
21:     <objectContribution
22:           .....
23:           objectClass="org.eclipse.jdt.core.IMethod">
24:        <action
25:              .....
26:              label="Replace Collaborators with Mock-Objects"
27:              class="org.createMockObjects.actions.CmoIMethodAction"
28:              enablesFor="1"/>
29:     </objectContribution>
30: </extension>
```

Quelltext 4-6: Kontributionen zum Erweiterungspunkt org.eclipse.ui.popupMenus

Bei den Objekt-Kontributionen muss im Attribut `objectClass` der Typ des Objekts angegeben werden, in dessen Kontextmenü der neue Menüpunkt erscheinen soll. Hier ist das entweder eine Klasse oder eine Methode. Die entsprechenden Interfaces im Eclipse Java-Modell (siehe [Eclipse 3.3 Java Model]) sind `org.eclipse.jdt.core.IType` und `org.eclipse.jdt.core.IMethod`. Jede Objekt-Kontribution enthält eine Action, in deren Attribut `label` der Text spezifiziert wird, der im Kontext-Menü des selektierten Objekts angezeigt werden soll. Anschließend wird im Attribut `class` eine Klasse angegeben, welche das Interface `org.eclipse.ui.IObjectActionDelegate` implementiert. Schließlich enthält das Attribut `enablesFor` die Information, dass die Action nur für die Selektion eines einzelnen Objekts ausgeführt werden soll.

Im Falle der Viewer-Kontribution wird im Attribut `targetID` angegeben, dass sich die Kontribution auf den Editor einer Java-Quelldatei bezieht. Die Action enthält im Attribut `label` den gleichen Text wie die Actions oben, sowie im Attribut `class` eine Klasse, die das Interface `org.eclipse.ui.IEditorActionDelegate` implementiert. Im Attribut `menubarPath` muss hier zusätzlich die Stelle im Kontext-Menü angegeben werden, an der der neue Menüpunkt erscheinen soll. Hier ist das die „standard additions group“.

Die Klasse `CmoAction` ist die Basisklasse für alle Actions des CMO-Plugins. Sie enthält im wesentlichen die Methode `runCmoRefactoring()`, welche von den abgeleiteten Klassen aufgerufen wird, um das CMO-Refactoring zu starten. Diese Methode ist in Kapitel 4.7 beschrieben, siehe Quelltext 4-7.

Die Actions der Objekt-Kontributionen werden durch Instanzen der Klassen `CmoIType-Action` und `CmoIMethodAction` realisiert. Wird im Package Explorer eine Klasse bzw. eine Methode selektiert, ruft das Eclipse-Framework die Methode `selectionChanged()` dieser Objekte auf. Diese Methode speichert lediglich das übergebene Selektions-Objekt in einer Instanzvariablen. Dieses Objekt ist eine Implementierung des Interfaces `org.eclipse.jface.viewers.IStructuredSelection`. Wird der neue Menüpunkt „Replace Collaborators with Mock-Objects“ im Kontextmenü des selektierten Objekts ausgewählt, ruft das Eclipse-Framework die Methode `run()` der Action-Klasse auf.

Die `run()`-Methode sieht in der Klasse `CmoITypeAction` so aus, dass zunächst das Java-Element bestimmt wird, welches dem selektierten Typen entspricht. Anschließend wird eine Instanz der Klasse `TestClass` erzeugt, welche die Testklasse innerhalb des CMO-Refactorings repräsentiert, siehe Kapitel 4.5. Durch Aufruf der Methode `resolveTestCasesAndFixtureSetupMethods()` dieses Objekts können alle Methoden

der selektierten Klasse bestimmt werden, welche mit `@Test` oder `@Begin` annotiert sind. Es muss mindestens ein Testfall vorhanden sein. Durch Aufruf der Methode `selectAllTest-Methods()` des `TestClass`-Objekts werden alle Testfälle und Setup-Methoden für ein mögliches Refactoring selektiert. Falls die Testklasse mit der `@CUT`-Annotation versehen ist, kann die CUT durch Aufruf der Methode `resolveCutAnnotation()` bestimmt werden. Falls nicht, wird mittels der Methode `setDefaultCut()` des `TestClass`-Objekts (siehe Kapitel 4.5) versucht, aus dem Namen der Testklasse eine Voreinstellung für die CUT zu ermitteln.[8] Schließlich wird das CMO-Refactoring durch Aufruf der Methode `runCmoRefactoring()` der Basisklasse `CmoAction` (s.o.) gestartet. Dabei wird dem Refactoring das neu erzeugte `TestClass`-Objekt mitgegeben, siehe Quelltext 4-7 in Kapitel 4.7.

Die Methode `run()` der Klasse `CmoIMethodAction` sieht ähnlich aus. Das selektierte Objekt ist hier allerdings eine Methode, und deshalb muss die Klasse bestimmt werden, in der diese Methode deklariert ist. Anschließend werden wieder die Testfälle und die Setup-Methoden dieser Klasse bestimmt, aber es wird nur die selektierte Methode für ein mögliches Refactoring markiert. Ist diese Methode kein Testfall, wird eine Fehlermeldung ausgegeben. Schließlich wird wie oben eine mögliche `@CUT`-Annotation ausgewertet und das Refactoring gestartet.

Die Klasse `CmoEditorAction` realisiert die Action der Viewer-Kontribution. Wird eine Java-Quelldatei editiert, ruft das Eclipse-Framework die Methode `setActiveEditor()` auf, und übergibt so der `CmoEditorAction`-Instanz den gerade aktiven Editor. Wird ein Teil des Textes der Java-Quelldatei selektiert und im Kontextmenü des Editors der neue Menüpunkt „Replace Collaborators with Mock-Objects“ angeklickt, ruft das Eclipse-Framework die Methode `run()` auf. Diese Methode ermittelt das Java-Element, welches im Java-Quelltext selektiert wurde. Es muss entweder der Name einer Klasse sein, welche der CUT entspricht, oder der Name eines CUT-Konstruktors. Anschließend wird die Klasse bestimmt, in der sich der selektierte Text befindet. Diese Klasse wird als Testklasse interpretiert und, wie bei den oben beschriebenen Actions, auf vorhandene Testfälle überprüft. Details zur Bestimmung der CUT und der Testklasse finden sich in Anhang A.1. Letztendlich wird wiederum eine Instanz der Klasse `TestClass` erzeugt, welcher die ermittelte CUT durch Aufruf der Methode `setCut()` übergeben wird. Außerdem werden alle Testmethoden für ein mögliches Refactoring ausgewählt. Schließlich wird das CMO-Refactoring mittels der Methode `runCmoRefactoring()` der Basisklasse `CmoAction` gestartet.

4.7 Einbindung in das Eclipse Refactoring-Framework

Ein Refactoring ist ein Werkzeug, welches den Code eines Programms in einer bestimmten Weise umgestaltet, ohne dass die Funktion des Programms verändert wird. Eclipse stellt für Refactorings mit dem Plugin `org.eclipse.ltk.core.refactoring` ein eigenes Framework zur Verfügung. Die dazugehörige graphische Schnittstelle ist im Plugin `org.eclipse.ltk.ui.refactoring` realisiert. Der Eclipse-Artikel [Widmer 2006]

[8] Oft besteht der Name der Testklasse aus dem CUT-Namen mit dem Zusatz „Test“, z.B. „BankTest“, siehe Kapitel 3.4.

beschreibt die beiden Frameworks anhand eines Beispiel-Refactorings. Die Hauptklassen des CMO-Refactorings sind folgende:

- Die Klasse `CmoRefactoring` im Paket `org.createMockObjects` steht für das CMO-Refactoring selbst. Sie ist von der abstrakten Klasse `org.eclipse.ltk.core.refactoring.Refactoring` abgeleitet, welche die Superklasse aller Eclipse-Refactorings ist.
- Die Klasse `CmoRefactoringWizard` repräsentiert das graphische User-Interface des Refactorings. Sie ist von der abstrakten Klasse `org.eclipse.ltk.ui.refactoring.RefactoringWizard` abgeleitet.
- Die graphische Startseite des Refactorings selbst wird durch die Klasse `CmoRefactoringInputPage` realisiert, welche von der abstrakten Klasse `org.eclipse.ltk.ui.refactoring.UserInputWizardPage` abgeleitet ist.

Die Methode `runCmoRefactoring()` der Klasse `CmoAction` (siehe Kapitel 4.6) beinhaltet den Start des CMO-Refactorings. Hier wird zunächst die Klasse `CmoRefactoring` instanziiert und dieses Objekt einer neuen `CmoRefactoringWizard`-Instanz mitgegeben. Das `CmoRefactoringWizard`-Objekt wird wiederum einer neuen Instanz der Klasse `org.eclipse.ltk.ui.refactoring.RefactoringWizardOpenOperation` übergeben. Über den Aufruf der Methode `run()` dieses Objekts wird das Refactoring schließlich gestartet:

```
1:  protected void runCmoRefactoring(TestClass theTestClass)
2:  {   try
3:      {   CmoRefactoring refactoring= new CmoRefactoring(theTestClass);
4:          CmoRefactoringWizard refactoringWizard =
5:              new CmoRefactoringWizard(refactoring);
6:          RefactoringWizardOpenOperation openOperation =
7:              new RefactoringWizardOpenOperation(refactoringWizard);
8:          openOperation.run(getActiveShell(),
9:              "Replace Collaborators with Mock Objects");
10:     }
11:     .....
12: }
```

Quelltext 4-7: Start des CMO-Refactorings in der Klasse CmoAction

Ein neues Eclipse-Refactoring muss grundsätzlich folgende Methoden der Klasse `org.eclipse.ltk.core.refactoring.Refactoring` überschreiben:

- Die Methode `checkInitialConditions()` wird vom Eclipse-Framework beim Start des Refactorings aufgerufen. Hier werden die Vorbedingungen des Refactorings geprüft. Anschließend wird über das oben erwähnte `CmoRefactoringWizard`-Objekt die graphische Eingabeseite des Refactorings aufgebaut.
- Wird auf dieser Eingabeseite die OK- oder Preview-Schaltfläche gedrückt, ruft das Eclipse-Framework die Methode `checkFinalConditions()` auf, welche noch einmal abschließend prüft, ob alle Bedingungen für das Refactoring erfüllt sind.
- Ist dies der Fall, ruft das Eclipse-Framework die Methode `createChange()` auf, welche die Codeänderungen in Form eines Arrays von Objekten des Typs `org.`

`eclipse.ltk.core.refactoring.Change` zur Verfügung stellt. Wurde die Preview-Schaltfläche gedrückt, werden die Änderungen in einem Vorschaufenster angezeigt. Ansonsten werden sie direkt in den Code eingearbeitet.

Die Methode `checkInitialConditions()` sieht in der Klasse `CmoRefactoring` so aus, dass zunächst überprüft wird, ob in den Plugin-Voreinstellungen ein gültiges Verzeichnis eingestellt wurde, in dem die Bibliotheken des ausgewählten Mock-Frameworks zu finden sind. Die voreingestellten Werte werden dabei über Klassenmethoden der Klasse `CmoPreferences` (siehe Kapitel 4.4) ermittelt. Anschließend werden sie mittels der Methode `mockDirectoryIsValid()` des `MockFrameworkExtensions`-Objekts überprüft.[9] Weiterhin prüft die Methode `checkInitialConditions()`, ob die Java-Quelldatei, in der die Testklasse deklariert ist, Compiler-Fehler aufweist. Außerdem wird sichergestellt, dass das CMO-Refactoring noch nicht auf diese Java-Quelldatei angewendet wurde. Tritt bei einer der genannten Prüfungen ein Fehler auf, wird er als „Fatal Error" in einem Objekt des Typs `org.eclipse.ltk.core.refactoring.RefactoringStatus` an das Eclipse-Framework zurückgegeben. Dies führt zu einem Abbruch des Refactorings.

Nach Beendigung der Methode `checkInitialConditions()` baut das Eclipse-Framework die graphische Eingabeseite des CMO-Refactorings (s.u.) auf. Da auf dieser Seite alle Benutzereingaben überprüft werden, sind bei ihrer Beendigung via „OK"- oder „Preview"-Schaltfläche alle Bedingungen für das Refactoring erfüllt. Deshalb können in der Methode `checkFinalConditions()` der Klasse `CmoRefactoring` die notwendigen Codeänderungen für die Testklasse (inkl. Änderungen im Classpath des Java-Projekts) berechnet werden. Dies übernimmt ein Objekt vom Typ `UnitTestRewriter`.[10] Darauf wird die Methode `rewriteTestClass()` aufgerufen, welche eine Statusinformation zurückgibt, ob die Codeänderungen erfolgreich erzeugt werden konnten. War dies nicht der Fall, wird dem Eclipse-Framework ein „Fatal Error" gemeldet und das Refactoring wird abgebrochen.

Die Methode `createChange()` der Klasse `CmoRefactoring` muss die berechneten Codeänderungen schließlich nur noch durch Aufruf der Methode `getTestClassChanges()` von der `UnitTestRewriter`-Instanz abholen. Sie werden dem Eclipse-Framework als ein Array von `Change`-Objekten zurückgegeben.

Für die graphische Benutzerschnittstelle muss ein Refactoring die abstrakte Klasse `org.eclipse.ltk.ui.refactoring.RefactoringWizard` (s.o.) erweitern. Diese Klasse sorgt dafür, dass ein graphischer Rahmen aufgebaut wird, mit dessen Hilfe man durch verschiedene Seiten navigieren kann. So wird z.B. beim Anklicken einer „Preview"-Schaltfläche eine Vorschauseite mit den Codeänderungen des Refactorings angezeigt. Drückt man die „OK"-Schaltfläche, werden diese Änderungen tatsächlich in den Code eingearbeitet. Über die Methode `addUserInputPages()` kann man eigene graphische Seiten in diesen Rahmen einfügen. Die Klasse `CmoRefactoringWizard` erzeugt in dieser Methode ein `CmoRefactoringInputPage`-Objekt. Wenn das Eclipse-Framework die Methode `createControl()` dieses Objekts aufruft, wird mit Hilfe von SWT-Widgets (siehe [Daum 2006]) die in Abbildung 3-5 gezeigte Startseite des CMO-Refactorings aufgebaut.

[9] Vgl. hierzu die Verzeichnisprüfung in der Klasse `CmoPreferences` in Kapitel 4.4.

[10] Die Klasse `UnitTestRewriter` führt das Refactoring der Testklasse durch. Sie ist in Kapitel 4.10 beschrieben.

Hier werden die Namen der Testklasse und der CUT angezeigt, sowie die Testfälle bzw. Setup-Methoden, welche in der Testklasse für ein Refactoring selektiert wurden. Diese Einstellungen können durch den Anwender verändert werden.

Die Selektion der Testfälle wird in der Klasse `CmoRefactoringInputPage` über ein `Combo`-Widget realisiert. Hier wird dem Anwender eine Liste aller Testfälle angezeigt, die in der Testklasse gefunden wurden. Er kann entweder einen einzelnen Testfall oder alle Testfälle inklusive der Setup-Methoden auswählen. Die Methode `testMethodCombo-SelectionHandler()` wertet die entsprechende Selektion aus und speichert sie in der Instanz der Klasse `TestClass`, welche die Testklasse innerhalb des CMO-Refactoring repräsentiert, siehe Kapitel 4.5.

Um eine andere CUT auszuwählen, kann der Anwender auf der Refactoring-Startseite eine „Browse“-Schaltfläche betätigen. Daraufhin wird in der Methode `cutButtonSelection-Handler()` der Klasse `CmoRefactoringInputPage` ein Dialog aufgebaut, in dem er eine der Klassen auswählen kann, die in den Quelldateien des Testklassen-Projekts deklariert sind, siehe Abbildung 3-6. Dazu wird die statische Methode `createTypeDialog()` der Klasse `org.eclipse.jdt.ui.JavaUI` benützt, siehe Anhang A.3.

Nach jeder Änderung der Refactoring-Einstellungen sucht das `CmoRefactoringInput-Page`-Objekt erneut nach kollaborierenden Objekten, welche einer Instanz der selektierten CUT in den ausgewählten Testfällen übergeben werden. Dazu wird die Methode `searchCollaborators()` der Klasse `CmoRefactoring` aufgerufen. Hier wird ein neues `CollaboratorFinder`-Objekt erzeugt, welches die Suche nach den Kollaborateuren übernimmt, siehe Kapitel 4.8. War die Suche erfolgreich, zeigt die Methode `checkStatusAndUpdatePage()` der Klasse `CmoRefactoringInputPage` auf der Refactoring-Startseite eine „Select Collaborators“-Schaltfläche an. Betätigt der Anwender diese Schaltfläche, wird eine Folge von Dialogen aufgebaut, über die er die kollaborierenden Objekte selektieren kann, welche durch Mock-Objekte ersetzt werden sollen. Details zu diesen Dialogen finden sich in Kapitel 4.9. Die Anzahl der ausgewählten kollaborierenden Objekte wird nach Beendigung der Dialoge auf der Refactoring-Startseite angezeigt. Wurde mindestens ein Objekt selektiert, konfiguriert die Methode `checkStatusAndUpdate-Page()` die Refactoring-Startseite so, dass die Schaltflächen „Preview“ und „OK“ betätigt werden können, um eine Vorschau der Codeänderungen anzuzeigen bzw. das Refactoring direkt durchzuführen.

Findet das `CollaboratorFinder`-Objekt keine kollaborierenden Objekte, liefert die oben erwähnte Methode `searchCollaborators()` einen Fehlerstatus zurück. Dabei wird differenziert, ob überhaupt keine Kollaborateure gefunden wurden, oder nur solche, die keine Variablen sind, siehe Kapitel 3.5. Der Fehlerstatus wird in der Methode `checkStatusAndUpdatePage()` der Klasse `CmoRefactoringInputPage` überprüft und dem Anwender angezeigt.

4.8 Suche nach kollaborierenden Objekten

Wie bereits im obigen Kapitel erwähnt, übernimmt eine Instanz der Klasse `CollaboratorFinder` (Paket `org.createMockObjects.collsearch`) die Suche nach kollaborierenden Objekten. Die Methode `searchCollaborators()` dieser Klasse durchsucht alle vom Anwender ausgewählten Testfälle bzw. Setup-Methoden nach Ausdrücken, in denen kollaborierende Objekte einer CUT-Instanz übergeben werden. Wie in Kapitel 3.5 beschrieben, werden dabei „Class Instance Creation Expressions“ und „Method Invocation Expressions“ berücksichtigt. Für die Suche nach diesen Ausdrücken wird das „Visitor“-Pattern angewandt, welches in [Gamma et al., 2004], Seite 301, detailliert beschrieben ist. Die entsprechenden Besucher-Objekte werden durch Instanzen der Klassen `ClassInstanceCreationSearchVisitor` und `MethodInvocationSearchVisitor` aus dem Paket `org.createMockObjects.visitors` realisiert. Sie werden dem AST-Knoten der zu durchsuchenden Methode übergeben und liefern die AST-Knoten der gefundenen Ausdrücke zurück.

War die Suche erfolgreich, werden die Argumente der gefundenen Ausdrücke untersucht. Ein aktueller Parameter wird als kollaborierendes Objekt erkannt, wenn die in Kapitel 3.5 beschriebenen Bedingungen erfüllt sind. Zunächst wird mittels der Methode `checkFormalParameter()` der Typ des korrespondierenden formalen Konstruktor- bzw. Methoden-Parameters überprüft:

```
1:  private boolean checkFormalParameter(ITypeBinding typeBinding)
2:  {  // There will be no type for a BaseTypeBinding or an ArrayBinding!
3:     IType type = (IType)typeBinding.getJavaElement();
4:     // The formal parametertype must be a Class or an Interface
5:     if ( (typeBinding.isClass() || typeBinding.isInterface()) &&
6:          (type != null) )
7:     {  // Check if the type of the formal parameter is not
8:        // Object, Class, String or the CUT itself
9:        String qualifiedName = typeBinding.getQualifiedName();
10:       if ( (qualifiedName.equals("java.lang.Object"))  ||
11:            (qualifiedName.contains("java.lang.Class")) ||
12:            (qualifiedName.equals("java.lang.String"))  ||
13:            (qualifiedName.equals(cut.getFullyQualifiedName())) )
14:           return false;
15:       // Check if the type of the formal parameter is
16:       // not a subtype of Throwable
17:       try
18:       {  ITypeHierarchy typeHierarchy = type.newSupertypeHierarchy(null);
19:          for (IType t : typeHierarchy.getAllClasses())
20:          {  if (t.getFullyQualifiedName().equals("java.lang.Throwable"))
21:                return false;
22:          }
23:          return true;
24:       }
25:       catch (JavaModelException e) { e.printStackTrace(); return false; }
26:    }
27:    else return false;
28: }
```

Quelltext 4-8: Prüfung eines formalen Parametertypen in der Klasse CollaboratorFinder

Ist die Prüfung des formalen Parameter-Typen erfolgreich, wird der aktuelle Parameter selbst untersucht. Er muss entweder ein Variablenname oder ein Ausdruck „`this`.variablenname“

sein, mit dem auf eine Instanzvariable zugegriffen wird. Dies wird mittels der Methode `isVariable()` überprüft:

```
1:  boolean isVariable(ASTNode node)
2:  {  IBinding nameBinding = null;
3:     if ((node instanceof SimpleName) || (node instanceof FieldAccess))
4:     {
5:        if (node instanceof SimpleName)
6:        {  nameBinding = ((SimpleName)node).resolveBinding();
7:        }
8:        else if (node instanceof FieldAccess)
9:        {  if (((FieldAccess)node).getExpression() instanceof ThisExpression)
10:          {  SimpleName varName = ((FieldAccess)node).getName();
11:             nameBinding = varName.resolveBinding();
12:          }
13:       }
14:       if ((nameBinding!=null) && (nameBinding instanceof IVariableBinding))
15:       {
16:          return true;
17:       }
18:    }
19:    return false;
20: }
```

Quelltext 4-9: Prüfung eines aktuellen Parameters in der Klasse CollaboratorFinder

Erfüllt mindestens ein Argument eines gefundenen Ausdrucks die oben genannten Bedingungen, wird der AST-Knoten dieses „Kollaborator"-Ausdrucks in einer neuen Instanz der Klasse `CollaboratorExpression` gespeichert. Außerdem enthält dieses Objekt die AST-Knoten und die entsprechenden formalen Parametertypen aller Argumente, die als kollaborierende Objekte erkannt wurden.

Für die Auswahl der gefundenen kollaborierenden Objekte durch den Anwender ist es aus Gründen der Übersichtlichkeit besser, ihm nicht den Kollaborator-Ausdruck allein anzuzeigen, sondern die komplette Anweisung im Rumpf der jeweiligen Testmethode, die den Kollaborator-Ausdruck enthält. Diese Anweisung wird mittels der Methode `resolveMethodStatement()` ermittelt:

```
1:  private Statement resolveMethodStatement(Expression expression,
2:     MethodDeclaration methodNode)
3:  {  ASTNode node = expression;
4:     Block methodBody = methodNode.getBody();
5:     List statementList = methodBody.statements();
6:     // Search for a parent of the expression node,
7:     //which is contained in the statement list
8:     while (node != null)
9:     {  if ((node instanceof Statement) && (statementList.contains(node)))
10:          return (Statement)node;
11:       node = node.getParent();
12:    }
13:    return(null);
14: }
```

Quelltext 4-10: Bestimmung der „Parent"-Anweisung eines Kollaborator-Ausdrucks

Für jede derartige Anweisung wird eine Instanz der Klasse `CollaboratorStatement` erzeugt. Dieses Objekt enthält die Anweisung selbst, sowie eine Liste mit `CollaboratorExpression`-Objekten für alle Kollaborator-Ausdrücke, die diese Anweisung enthält.

Als Ergebnis der Kollaborator-Suche legt das `CollaboratorFinder`-Objekt eine Hashtabelle an. Sie enthält für jede Testmethode (Testfall oder Setup-Methode), in der kollaborierende Objekte gefunden wurden, eine Liste von `CollaboratorStatement`-Objekten.[11] Die Schlüssel der Hashtabelle sind die AST-Knoten der Testmethoden. Die Methode `getMethods()` der Klasse `CollaboratorFinder` liefert alle Schlüsselwerte der Tabelle, d.h. alle Testfälle bzw. Setup-Methoden, in denen kollaborierende Objekte gefunden wurden. Über die Methode `getCollaboratorStatements()` kann die `CollaboratorStatement`-Liste für eine bestimmte Testmethode abgefragt werden.

4.9 Auswahl zu ersetzender kollaborierender Objekte

Wurden für wenigstens einen selektierten Testfall kollaborierende Objekte gefunden, kann der Anwender auf der Startseite des CMO-Refactorings eine „Select Collaborators"-Schaltfläche betätigen, siehe Kapitel 4.7. Daraufhin wird eine Folge von Dialogen aufgebaut, über die er die zu ersetzenden kollaborierenden Objekte auswählen kann. Eclipse bietet für solche Dialoge sog. „Wizards" an. Sie sind in [Daum 2006], Seite 271, genauer beschrieben. Ein Wizard bietet einen graphischen Rahmen an, der den Anwender durch mehrere Dialogseiten leitet. Über „Next" bzw. „Back"-Schaltflächen kann zur nächsten bzw. vorgerigen Seite gewechselt werden. Die Eingabe wird über eine „Finish"-Schaltfläche beendet oder über eine „Cancel"-Schaltfläche abgebrochen. Ein konkreter Wizard muss auf Basis der abstrakten Klasse `org.eclipse.jface.wizard.Wizard` implementiert werden. Die einzelnen graphischen Seiten werden durch eine Klasse realisiert, die von der abstrakten Klasse `org.eclipse.jface.wizard.WizardPage` abgeleitet ist. Die entsprechenden konkreten Klassen für die Kollaborator-Auswahl heißen `CollSelectionWizard` und `CollSelectionInputPage`. In der Methode `collButtonSelectionHandler()` der Klasse `CmoRefactoringInputPage` wird ein `CollSelectionWizard`-Objekt erzeugt und einer Instanz der Klasse `org.eclipse.jface.wizard.WizardDialog` übergeben. Über die Methode `open()` dieses Objekts wird der Wizard schließlich gestartet:

```
1:  private void collButtonSelectionHandler()
2:  {  // Build Wizard Dialog for Collaborator Selection
3:     CollSelectionWizard wizard = new CollSelectionWizard(
4:         refactoring.getCollaboratorFinder());
5:     WizardDialog dialog = new WizardDialog(getShell(), wizard);
6:     // Open dialog
7:     if (dialog.open() == Window.OK)
8:     .....
9:  }
```

Quelltext 4-11: Start des Wizards für die Kollaborator-Selektion

Bei seiner Erzeugung wird dem `CollSelectionWizard`-Objekt das `CollaboratorFinder`-Objekt übergeben, welches vom CMO-Refactoring für die Suche nach den kollaborierenden Objekten benützt wurde. Für jede gefundene Anweisung mit kollaborierenden Objekten, d.h. für jede Instanz der Klasse `CollaboratorStatement` (siehe Kapitel 4.8) wird ein eigenes `CollSelectionInputPage`-Objekt erzeugt, welches über SWT-Widgets

[11] Vgl. dazu auch Abbildung 4-2, die eine UML-Darstellung der wichtigsten Objekte des CMO-Refactorings zeigt.

eine graphische Dialogseite aufbaut. Hier wird neben der Anweisung selbst auch der Name des Testfalls angezeigt, welcher die Anweisung enthält. Jedes kollaborierende Objekt in der Anweisung wird farblich hervorgehoben. Außerdem wird für jeden Kollaborateur ein eigenes `Button`-Widget erzeugt, welches eine Checkbox realisiert. Darüber kann der Anwender auswählen, ob das kollaborierende Objekt durch ein Mock-Objekt ersetzt werden soll oder nicht.

Das `CollaboratorStatement`-Objekt enthält eine Liste von `CollaboratorExpression`-Objekten, welche den Ausdrücken entsprechen, in denen kollaborierende Objekte gefunden wurden, siehe Kapitel 4.8. Diese Objekte enthalten Hashtabellen, in denen die Selektionsinformation für jeden Kollaborateur als `boolean`-Wert abgelegt ist. Diese Hashtabellen werden durch das `CollSelectionInputPage`-Objekt aber nicht sofort aktualisiert, denn der Anwender könnte den Selektions-Dialog durch Drücken der „Cancel"-Schaltfläche abbrechen. Stattdessen sind in der Klasse `CollSelectionInputPage` Schattentabellen definiert, in denen die Selektionen zwischengespeichert werden. Beendet der Anwender die Eingabe über die „Finish"-Schaltfläche, wird die Methode `performFinish()` des `CollSelectionWizard`-Objekts aufgerufen. Diese ruft wiederum für jede Eingabeseite die Methode `updateCollaboratorSelections()` der Klasse `CollSelectionInputPage` auf. Hier werden die Selektionswerte aus den Schattentabellen in die `CollaboratorExpression`-Objekte kopiert. In der Methode `performFinish()` wird außerdem die Gesamtzahl der selektierten kollaborierenden Objekte ermittelt, welche dem Anwender als Rückmeldung auf der Refactoring-Startseite angezeigt wird. Startet der Anwender anschließend das Refactoring, können über die Methode `getSelectedCollExpressions()` des `CollaboratorFinder`-Objekts für einen Testfall alle `CollaboratorExpression`-Objekte ermittelt werden, die selektierte kollaborierende Objekte enthalten, d.h. alle Ausdrücke in diesem Testfall, in denen es Objekte gibt, die durch Mock-Objekte ersetzt werden müssen.

4.10 Modifikation der Testklasse

Die notwendigen Änderungen in der Testklasse werden von einer Instanz der Klasse `UnitTestRewriter` (Paket `org.createMockObjects.rewrite`) berechnet. Dabei muss sowohl der Code der entsprechenden Java-Quelldatei als auch der Java-Classpath des zugehörigen Projekts angepasst werden. Letztendlich müssen diese Modifikationen dem CMO-Refactoring als Objekte übergeben werden, die von der abstrakten Klasse `org.eclipse.ltk.core.refactoring.Change` abgeleitet sind.[12] Für die Code-Änderungen wird ein Objekt des Typs `org.eclipse.ltk.core.refactoring.TextFileChange` erzeugt. Die Änderungen im Classpath des Java-Projektes werden von einem Objekt des Typs `CmoClassPathChange` durchgeführt, siehe Kapitel 4.10.1. Die Methode `rewriteTestClass()` der Klasse `UnitTestRewriter` koordiniert die Berechnung der Änderungen und die Erzeugung der oben erwähnten `Change`-Objekte.[13] Sie speichert diese in Instanz-

[12] Ein Eclipse-Refactoring muss dem Eclipse-Framework die Codeänderungen in der Methode `createChange()` als ein Array von `Change`-Objekten bereitstellen, siehe Kapitel 4.7.

[13] Sie wird von der Methode `checkFinalConditions()` der Klasse `CmoRefactoring` aufgerufen, siehe Kapitel 4.7.

variablen und gibt eine Fehlerinformation zurück. Die `Change`-Objekte können anschließend über die Methode `getTestClassChanges()` gelesen werden.[14]

Die Mock-Framework-spezifischen Änderungen werden von einem `IMockRewriter`-Objekt durchgeführt. Konkret handelt es sich um die Instanz einer Klasse aus einem Plugin, welches den Erweiterungspunkt `mockRewriter` implementiert, siehe Kapitel 4.3. Die beiden Extension-Plugins für JMock 2.2.0 und JMock 2.4.0 (siehe Kapitel 4.1) stellen die Klassen `JMock220Rewriter` und `JMock240Rewriter` als `IMockRewriter`-Implementierungen zur Verfügung. Im Konstruktor der Klasse `UnitTestRewriter` wird das Objekt erzeugt, welches dem vom Anwender ausgewählten Mock-Framework entspricht. Dazu wird die Methode `getMockRewriter()` des `MockFrameworkExtensions`-Singletons aufgerufen, siehe Quelltext 4-4.

4.10.1 Anpassungen im Classpath des Java-Projektes

Die CMO-Annotations werden dem Classpath des Java-Projektes als Classpath Container Entry hinzugefügt, siehe Kapitel 4.2. Die Klasse `CmoAnnotationContainer` im Paket `org.createMockObjects.classpath` realisiert diesen Annotation-Container. Sie ist von der abstrakten Klasse `LibraryContainer` abgeleitet, welche eine Menge von Bibliothekspfaden in Form von Objekten des Typs `org.eclipse.core.runtime.IPath` enthält. Die Methode `createClasspathEntries()` erzeugt aus diesen Pfaden Library Classpath Entries. Im Falle des Annotation-Containers wird nur ein einziger Library Classpath Entry benötigt, siehe Kapitel 4.2. Wenn der Annotation-Container initialisiert wird (siehe Quelltext 4-2), holt sich das Eclipse-Framework die Library Classpath Entries über die Methode `getClasspathEntries()`.

Die Bibliotheken des eingestellten Mock-Frameworks werden nicht in einem Classpath Container zusammengefasst, denn sie hängen von den jeweiligen Extension-Plugins ab, welche für spezifische Mock-Frameworks installiert sind. Ändern sich diese Plugins, könnte ein schon hinzugefügter Classpath Container für ein Projekt nach dem Neustart von Eclipse völlig anders initialisiert werden, siehe Anhang A.4. Dies würde u.U. nicht mehr zu den Codeänderungen passen, die das CMO-Refactoring in den Testfällen des Projektes durchgeführt hatte. Deshalb werden die Mock-Framework-Bibliotheken dem Classpath des Projektes als einzelne Library Classpath Entries hinzugefügt. Dazu wird eine Instanz der Klasse `MockFrameworkLibraries` verwendet, welche ebenfalls von der oben erwähnten abstrakten Klasse `LibraryContainer` abgeleitet ist. Bei der Erzeugung des `MockFrameworkLibraries`-Objekts werden die Pfade der Mock-Bibliotheken des ausgewählten Frameworks ermittelt. Eine detaillierte Beschreibung befindet sich in Anhang A.4.

Die Methode `createClassPathChange()` der Klasse `UnitTestRewriter` ändert den Classpath des Java-Projektes nur dann, wenn er den Annotation-Container bzw. die Mock-Framework-Biblitheken noch nicht enthält. Dies kann über entsprechende Methoden der Klassen `CmoAnnotationContainer` und `MockFrameworkLibraries` abgefragt werden. Falls eine Anpassung notwendig ist, wird ein Objekt des Typs `CmoClassPathChange`

[14] Die Methode `createChange()` der Klasse `CmoRefactoring` holt die `Change`-Objekte ab, siehe Kapitel 4.7.

erzeugt, welches den Classpath in der Methode `perform()` ändert. Dabei wird auch der Fall berücksichtigt, dass der Classpath schon einen Teil der Bibliotheken enthält. Details der programmatischen Realisierung sind in Anhang A.4 beschrieben. Für die Rücknahme der Änderungen erzeugt das `CmoClassPathChange`-Objekt ein `CmoClassPathChangeUndo`-Objekt. Wählt der Anwender im Eclipse Edit-Menü den Punkt „Undo CMO Refactoring“ aus, wird die Methode `perform()` dieses Objekts ausgeführt.

4.10.2 Änderungen im Java-Quellcode

In den Eclipse Java Development Tools (JDT) wird ein Java-Programm intern als „Abstract Syntax Tree“ (AST) repräsentiert. Codeänderungen, die sich auf einen AST beziehen, können mit Hilfe der Klasse `ASTRewrite` im Paket `org.eclipse.jdt.core.dom.rewrite` aufgezeichnet werden, ohne dass der ursprüngliche AST verändert wird.[15] Sind in der entsprechenden Java-Quelldatei auch Änderungen der import-Deklarationen notwendig, können diese mit Hilfe der Klasse `ImportRewrite` erfasst werden. Wenn die Änderungen komplett sind, können sie mit Hilfe der oben genannten Klassen in eine textuelle Form konvertiert werden. Die Methode `createTestClassChange()` der Klasse `UnitTestRewriter` erzeugt die Codeänderungen in der Java-Quelldatei der Testklasse.[16] Anschließend übernimmt die Methode `createTextChange()` die Konvertierung in ein `TextFileChange`-Objekt.

Als ersten Schritt fügt die Methode `createTestClassChange()` die Definitionen in den Quelltext ein, welche für den Einsatz des ausgewählten Mock-Frameworks benötigt werden. Dazu wird die Methode `rewriteGlobalDefinitions()` des `IMockRewriter`-Objekts aufgerufen. Deren Realisierung sieht z.B. in der Klasse `JMock220Rewriter` so aus, dass die für das JMock-Framework notwendigen import-Deklarationen in die Java-Quelldatei der Testklasse eingefügt werden. Außerdem wird die Testklasse mit einer Annotation „`@RunWith(JMock.class)`“ versehen. Schließlich wird in der Testklasse eine neue Instanzvariable `context` erzeugt, der ein `JUnit4Mockery`-Objekt zugewiesen wird. Da auch Mock-Objekte für Klassen erzeugt werden sollen, wird hierbei die JMock-Erweiterung `ClassImposteriser` verwendet.[17]

Im Folgenden wird die Testklasse mit der `@CUT`-Annotation versehen. Falls sie schon vorhanden ist, wird der CUT-Name aktualisiert. Die `@DoMockCollaborators`-Annotation wird der Testklasse ebenfalls hinzugefügt. Zusätzlich fügt die Methode `insertDoMockCollaboratorsAnnotation()` der Klasse `UnitTestRewriter` eine neue Klassenvariable `collaboratorsAreMocked` vom Typ `boolean` in die Testklasse ein, welche das Vorhandensein dieser Annotation prüft, siehe Kapitel 3.7.4.

Für jedes selektierte kollaborierende Objekt wird ein Mock-Objekt erzeugt, welches einer neuen Instanzvariablen in der Testklasse, genannt Mock-Variable, zugewiesen wird, siehe Kapitel 3.7.4. Für diese Aufgaben verwendet die Methode `createTestClassChange()` ein Hilfsobjekt vom Typ `MockObjectRewriter`. Diesem werden die AST-Knoten aller Testfälle

[15] Der AST und das ASTRewrite-API sind im Eclipse-Artikel [Kuhn&Thomann 2006] beschrieben.
[16] Sie wird in der Methode `rewriteTestClass()` (s.o.) aufgerufen.
[17] vgl. dazu die Beschreibung des JMock-Frameworks in Kapitel 2.4.2, Quelltext 2-3.

oder Setup-Methoden übergeben, in denen der Anwender kollaborierende Objekte für das Refactoring selektiert hat. Außerdem werden die Kollaborator-Ausdrücke mitgegeben, welche die selektierten Kollaborateure enthalten. Beide Informationen können über das in Kapitel 4.8 beschriebene `CollaboratorFinder`-Objekt abgefragt werden. Anschließend wird die Methode `insertMockObjectCreations()` des `MockObjectRewriter`-Objekts aufgerufen. Sie fügt zunächst die Deklarationen der Mock-Variablen in den Rumpf der Testklasse ein. Dazu ruft sie (über die Methode `createMockObjects()`) für jedes selektierte kollaborierende Objekt die Methode `createMockVariable()` auf. Der Typ der Mock-Variablen ist der Typ des entsprechenden formalen Konstruktor- bzw. Methoden-Parameters (siehe Kapitel 3.7.4):

```
1:  private String createMockVariable(Type mockVarType,
2:          IVariableBinding collVarBinding, String methodName)
3:  {  // Create Name of Mock Variable
4:     String mockVarName = createMockVarName(
5:         collVarBinding.getName(), methodName);
6:     // Create new field declaration for Mock variable
7:     VariableDeclarationFragment varFrag =
8:         ast.newVariableDeclarationFragment();
9:     varFrag.setName(ast.newSimpleName(mockVarName));
10:    FieldDeclaration mockField = createFieldDeclarationWithVarDeclFrag(
11:        mockVarType, varFrag, ModifierKeyword.PRIVATE_KEYWORD);
12:    // Add a new field to the type declaration of the testclass
13:    TypeDeclaration typeDeclNode = unitTestRewriter.getTestClass().
14:        getTypeDeclarationNode();
15:    ListRewrite lrw = astRewrite.getListRewrite(typeDeclNode,
16:        TypeDeclaration.BODY_DECLARATIONS_PROPERTY);
17:    unitTestRewriter.insertBeforeOldElements(lrw, mockField);
18:    return mockVarName;
19: }
```

Quelltext 4-12: Einfügen einer Deklaration für eine Mock-Variable in die Testklasse

Die Namen der Mock-Variablen werden in einer Hashtabelle der `MockObjectRewriter`-Instanz gespeichert, welche als Schlüssel den AST-Knoten des Arguments enthält, das in einem Kollaborator-Ausdruck durch die jeweilige Mock-Variable ersetzt werden soll. Anschließend generiert die Methode `createMockObjects()` für jedes Mock-Objekt eine Anweisung, in der es erzeugt und der korrespondierenden Mock-Variablen zugewiesen wird. Dazu verwendet sie die Methode `buildMockObjectCreationStatement()` des `IMockRewriter`-Objekts. Ihr werden der Name und der Typ der Mock-Variablen übergeben. In der Klasse `JMock220Rewriter` sieht diese Methode folgendermaßen aus:

```
1:  public Statement buildMockObjectCreationStatement(
2:          AST ast, String varName, Type varType)
3:  {
4:      // Invoke the method "mock" on the JMock context
5:      // to create a new mock object
6:      MethodInvocation mockMethInv = ast.newMethodInvocation();
7:      mockMethInv.setName(ast.newSimpleName("mock"));
8:      mockMethInv.setExpression(ast.newSimpleName("context"));
9:      // The method argument is the literal of the mock variable type
10:     TypeLiteral mockTypeLit = ast.newTypeLiteral();
11:     mockTypeLit.setType(varType);
12:     ((List<Expression>)mockMethInv.arguments()).add(mockTypeLit);
13:     // Create a new expression statement, which assigns
14:     // the mock object to the mock variable
15:     ExpressionStatement mockExprStatement =
16:         createExpressionStatementWithAssignment(
17:             ast.newSimpleName(varName), mockMethInv);
18:     return mockExprStatement;
19: }
```

Quelltext 4-13: Erzeugung eines neuen Mock-Objekts in der Klasse JMock220Rewriter

Die Methode `buildMockObjectCreationStatement()` der Klasse `JMock240Rewriter` sieht ähnlich aus. Jedoch wird hier der JMock-Methode `mock()` als zweites Argument der Name der Mock-Variable übergeben. Dieser wird von JMock als Name des Mock-Objekts benützt. Leider ist JMock 2.4.0 ohne diesen Namen nicht in der Lage, zwei Mock-Objekte desselben Typs zu erzeugen. Es wirft bei der Erzeugung des zweiten Mock-Objekts eine `IllegalArgumentException`, weil es standardmäßig den Typ des Mock-Objekts als dessen Name verwendet und zwei gleiche Namen nicht verwalten kann. Es wäre nach Ansicht des Autors sinnvoller, wenn JMock die Mock-Objekte anhand ihrer Objekt-Identifizierer unterscheiden würde.

Die Methode `insertMockObjectCreations()` des `MockObjectRewriter`-Objekts fügt schließlich die Anweisungen für die Erzeugung der Mock-Objekte am Anfang derjenigen Methode (Testfall oder Setup-Methode) ein, in der ein kollaborierendes Objekt durch das jeweilige Mock-Objekt ersetzt werden soll. Falls es sich bei der Methode um einen Testfall handelt, fügt sie nach diesen Anweisungen einen TODO-Stub für die Spezifikation des Verhaltens der Mock-Objekte ein. Falls Mock-Objekte in einer Setup-Methode erzeugt wurden, fügt sie anschließend einen TODO-Stub in allen Testfällen ein, in denen noch keiner vorhanden ist.

Nach der Erzeugung der Mock-Objekte muss das `UnitTestRewriter`-Objekt diejenigen Argumente der Kollaborator-Ausdrücke, welche den vom Anwender selektierten kollaborierenden Objekten entsprechen, durch die Namen der korrespondierenden Mock-Variablen ersetzen. Außerdem muss in den Testfall bzw. die Setup-Methode eine neue `If`-Abfrage eingefügt werden, die dafür sorgt, dass der Testfall mit oder ohne Mock-Objekten ablaufen kann. Diese Aufgaben sind abhängig von der Anweisung, in der sich der jeweilige Kollaborator-Ausdruck befindet, siehe Kapitel 3.7.4. Diese Anweisung muss sich nicht direkt im Rumpf der Testmethode befinden, sondern kann z.B. auch in einer verschachtelten Kontrollstruktur enthalten sein. Sie ist deshalb nicht zu verwechseln mit der Anweisung, welche das `CollaboratorFinder`-Objekt für die Anzeige und Auswahl der

kollaborierenden Objekte erzeugt! Die entsprechenden `CollaboratorStatement`-Objekte werden von der Klasse `UnitTestRewriter` nicht benutzt. Stattdessen ermittelt die Methode `createTestClassChange()` über das `CollaboratorFinder`-Objekt lediglich die Kollaborator-Ausdrücke, welche selektierte Kollaborateure enthalten. Diese Ausdrücke werden durch Instanzen der Klasse `CollaboratorExpression` repräsentiert. Die Methode `resolveContainingStatement()` bestimmt anschließend für jeden dieser Kollaborator-Ausdrücke den nächsten Elternknoten, welcher eine Anweisung ist.

Für jede so gefundene Anweisung erzeugt die Methode `createStatementRewriters()` der Klasse `UnitTestRewriter` ein eigenes Objekt, welches die oben beschriebenen Codeänderungen durchführt. Diesem Objekt werden alle in der Anweisung enthaltenen Kollaborator-Ausdrücke übergeben. Für die meisten Anweisungen hat das Objekt den Typ `StatementRewriter`. Für eine Variablen-Deklaration wird stattdessen eine Instanz der Klasse `VarDeclStatementRewriter` verwendet, welche eine Spezialisierung der Klasse `StatementRewriter` ist. Befindet sich eine Anweisung mit einem Kollaborator-Ausdruck wiederum in einer Anweisung, welche auch einen Kollaborator-Ausdruck enthält, wird nur für die umgebende Anweisung ein `StatementRewriter`-Objekt erzeugt, welches alle darin befindlichen Kollaborator-Ausdrücke enthält. Schließlich ruft die Methode `createTestClassChange()` auf jedes `StatementRewriter`-Objekt die Methode `rewriteStatement()` auf.

Diese Methode sieht in der Klasse `StatementRewriter` so aus, dass zunächst über die Methode `copySubtree()` der Klasse `org.eclipse.jdt.core.dom.ASTNode` zwei Kopien der Original-Anweisung erzeugt werden. In einer der beiden Kopien werden die selektierten Kollaborateure durch die Namen der entsprechenden Mock-Variablen ersetzt. Dazu muss erst einmal für jeden Kollaborator-Ausdruck aus der Original-Anweisung das entsprechende Pendant innerhalb der kopierten Anweisung gefunden werden. Dies ermöglicht die Methode `getCorrespondingExpression()` der Klasse `Rewriter`, welche eine abstrakte Basisklasse aller Rewriter-Klassen ist, und Hilfsmethoden für die Auswertung und Manipulation des AST enthält. Die obige Methode wendet das „Visitor“-Pattern[18] an und übergibt dem AST-Knoten der kopierten Anweisung eine Instanz der Klasse `ExpressionSearchVisitor`. Dieses Objekt vergleicht in der Methode `preVisit()` jeden besuchten Knoten, der eine Instanz der Klasse `org.eclipse.jdt.core.dom.Expression` ist, mit dem jeweiligen originären Kollaborator-Ausdruck.[19] Sind beide Ausdrücke strukturell äquivalent, und befinden sie sich an der gleichen Stelle im Quellcode, war die Suche erfolgreich:

[18] siehe [Gamma et al., 2004], Seite 301

[19] Eine Methode `visit()` für einen abstrakten Typ `Expression` gibt es in der Klasse `ASTVisitor` nicht.

```
1:  public void preVisit(ASTNode node)
2:  {  // Expression already found?
3:     if (result == null)
4:     {  // Check for an expression
5:        if (node instanceof Expression)
6:        {  // Check if the node is equivalent to the given expression
7:           // and if it is located at the same position
8:           if ( (node.subtreeMatch(new ASTMatcher(), expressionToSearch)) &&
9:                (node.getStartPosition() ==
10:                    expressionToSearch.getStartPosition()) )
11:          {
12:             result = (Expression)node;
13:          }
14:       }
15:    }
16: }
```

Quelltext 4-14: Suche nach dem Pendant des Kollaborator-Ausdrucks

In den gefundenen Ausdrücken werden anschließend die Argumente ersetzt, welche den vom Anwender selektierten Argumenten der originären Kollaborator-Ausdrücke entsprechen. Dies übernimmt die Methode `rewriteCollaboratorExpression()` der Klasse `StatementRewriter`, welche wiederum die Methode `replaceArgumentByVariable-Name()` benützt:

```
1:  protected void rewriteCollaboratorExpression(
2:     CollaboratorExpression collExpr, Expression expressionToMock)
3:  {  // Process all selected collaborators
4:     for (Expression collaborator : collExpr.getSelectedCollaborators())
5:     {  // Get name of mock variable
6:        String mockVarName =
7:            mockObjectRewriter.getMockVariable(collaborator);
8:        if (mockVarName != null)
9:        {  // Replace corresponding argument in cloned expression
10:          // by mock variable name
11:          replaceArgumentByVariableName(expressionToMock,
12:              collExpr.getCollaboratorIndex(collaborator),
13:              mockVarName);
14:       }
15:    }
16: }
17: protected void replaceArgumentByVariableName(
18:     Expression expression, int argIndex, String varName)
19: {  List arguments = null;
20:    if (expression instanceof ClassInstanceCreation)
21:    {  arguments = ((ClassInstanceCreation)expression).arguments();
22:    }
23:    else if (expression instanceof MethodInvocation)
24:    {  arguments = ((MethodInvocation)expression).arguments();
25:    }
26:    if (arguments != null)
27:    {  // Get the argument at the specified index and replace it
28:       Expression argument = (Expression)arguments.get(argIndex);
29:       astRewrite.replace(argument, ast.newSimpleName(varName), null);
30:    }
31: }
```

Quelltext 4-15: Ersetzen der selektierten Kollaborateure durch Mock-Objekte

Schließlich erzeugt die Methode `rewriteStatement()` über die Methode `createMock-IfStatement()` eine neue `If`-Abfrage, welche die Klassenvariable `collaboratorsAre-`

Mocked abfragt und in ihrem then- bzw. else-Zweig die beiden oben erwähnten Kopien der Original-Anweisung enthält. Diese If-Abfrage und ein entsprechender Kommentar werden mittels der Methode insertNewStatements() statt der Original-Anweisung in den Quellcode der entsprechenden Testmethode eingefügt.

Die Methode rewriteStatement() der Klasse VarDeclStatementRewriter ist etwas komplizierter, da die Variablendeklaration zerlegt werden muss. Details dazu beschreibt Kapitel 3.7.4. Zunächst werden die Kollaborator-Ausdrücke mittels der Methode sortCollExpressions() den Variablen-Deklarations-Fragmenten zugeordnet, in denen sie enthalten sind. Die Methode createNewVarDeclStatement() erzeugt anschließend eine neue Variablendeklaration. Diese Anweisung beinhaltet die Variablen-Namen aller Fragmente ohne Initialisierer, sowie die Variablen-Namen aller Fragemente, deren Initialisierer Kollaborator-Ausdrücke enthält. Mit Hilfe der Methode sortVarDeclFragments() werden als nächstes die Variablen-Deklarations-Fragmente mit Initialisierer in Gruppen zusammengefaßt, die Fragmente mit bzw. ohne Kollaborator-Ausdrücke enthalten. Dabei bleibt die Reihenfolge der Fragmente unverändert. Für die Fragment-Gruppen werden anschließend die entsprechenden Codeänderungen erzeugt.

Für eine Gruppe von Fragmenten mit Kollaborator-Ausdrücken übernimmt dies die Methode rewriteFragmentsWithCollaborators(). Sie erzeugt für jedes Fragment zwei identische neue Anweisungen, in denen der jeweiligen Variablen ihr Initialisierer zugewiesen wird. In einer der beiden Anweisungen werden alle selektierten Argumente der enthaltenen Kollaborator-Ausdrücke durch die Namen der entsprechenden Mock-Variablen ersetzt. Dazu werden die oben beschriebenen Methoden getCorrespondingExpression() und rewriteCollaboratorExpression() der Klasse StatementRewriter benützt. Anschließend werden beide Anweisungen in den then- bzw. else-Zweig einer neu erzeugten „if(collaboratorsAreMocked)“-Abfrage eingefügt. Für eine Gruppe von Fragmenten ohne Kollaborator-Ausdrücke erzeugt die Methode rewriteFragmentsWOCollaborators() lediglich eine neue Variablendeklarations-Anweisung, welche diese Fragmente enthält. Schließlich ersetzt die Methode rewriteStatement() die Original-Anweisung im Quellcode der jeweiligen Testmethode (Testfall oder Setup-Methode) durch die neu erzeugten Anweisungen. Dies geschieht durch Aufruf der Methode insertNewStatements() der Klasse StatementRewriter.

5 Test des CMO-Refactorings

Das CMO-Refactoring wurde mit der Eclipse Europa-Release, Version 3.3.1.1, und dem Java Runtime Environment 1.5.0_10 unter Microsoft Windows XP getestet. Da die oben genannte Eclipse-Version mit JUnit 4.3.1 ausgeliefert wird, wurde mit der JMock-Version 2.2.0 getestet. Zur Verifikation des JMock 2.4.0 Extension-Plugins wurden die Testfälle eines der Testprojekte mit JMock 2.4.0 und JUnit 4.4 wiederholt, s.u.

Auf der beiliegenden CD sind folgende Testprojekte enthalten:

- Das Projekt `Bank` enthält eine Reihe von Testfällen für die in Kapitel 2.4.1 beschriebene Klasse `Bank`.
- Das Projekt `BankWithTestcase` testet wieder die Klasse `Bank`, allerdings befindet sich der Testcode in der Klasse `Bank` selbst.
- Im Projekt `BankWithTCInnerClass` befindet sich der Testcode in einer Testklasse, welche innerhalb der Klasse `Bank` definiert ist.
- Im Projekt `BankWithGenerics` wird ebenfalls die Klasse `Bank` getestet. Die Kollaborator-Klasse `Datenbank` und das zugehörige Interface `IDatenbank` sind hier jedoch als generische Klassen definiert.
- Das Projekt `BankJUnit4.4` enthält die gleichen Testfälle wie das Projekt `Bank`, ist jedoch für den Test mit der JMock-Version 2.4.0 vorgesehen, welche die Version 4.4 des JUnit-Frameworks erfordert. Deshalb befindet sich im Classpath des Projektes das Archiv `junit4.4.jar`. Dieses Archiv wurde in einem neuen Unterverzeichnis `\junit4.4` abgelegt.

Die Testprojekte enthalten folgende Testklassen:

Projekte `Bank` und `BankJUnit4.4`	
Testklasse	Kommentar
`BankTest`	testet die Kollaborator-Übergabe als Konstruktor-Parameter in der Setup-Methode.
`BankTestConstrInjectInSetup`	testet die Kollaborator-Übergabe als Konstruktor-Parameter an zwei CUT-Instanzen in der Setup-Methode.
`BankTestConstrInjectIn2Setups`	derselbe Test wie oben, jedoch ist die Kollaborator-Übergabe auf zwei Setup-Methoden verteilt.
`BankTestConstrInjectInTestcase`	testet die Kollaborator-Übergabe als Konstruktor-Parameter in den Testfällen.
`BankTestControlSt`	testet die Kollaborator-Übergabe in verschiedenen Kontrollstrukturen.

`BankTestInjectInTestcase`	enthält eine Reihe von Testfällen, in denen Kollaborateure als Konstruktor- oder Setter-Parameter an CUT-Instanzen übergeben werden, z.T. findet die Übergabe innerhalb einer Variablendeklaration statt.
`BankTestNoCollVariable`	testet die Fehlermeldung des CMO-Refactorings, dass es nur Variablen als Kollaborateure unterstützt; hier werden die Kollaborateure ausschließlich als Ausdrücke an die CUT-Instanzen übergeben.
`BankTestProcCall`	testet die Kollaborator-Übergabe als Konstruktor-Parameter, wobei die Konstruktoraufrufe der CUT als Parameter in einem Methodenaufruf enthalten sind.
`BankTestSetterInjectInSetup`	testet die Kollaborator-Übergabe als Setter-Parameter in der Setup-Methode.
`BankTestSetterInjectInTestcase`	testet die Kollaborator-Übergabe als Setter-Parameter in den Testfällen.

Projekt `BankWithTestcase`	
Testklasse	Kommentar
`Bank`	diese Klasse ist die CUT und die Testklasse in einem; sie enthält als Testcode einen JUnit4-Testfall und eine Setup-Methode für den Aufbau der Test-Fixture.

Projekt `BankWithTCInnerClass`	
Testklasse	Kommentar
`Bank$BankTest`	innerhalb der CUT `Bank` ist eine innere Klasse `BankTest` deklariert, welche einen JUnit4-Testfall und eine Setup-Methode enthält.

Projekt `BankWithGenerics`	
Testklasse	Kommentar
`BankTest`	enthält einen Testfall für die Klasse `Bank`, wobei die Kollaborator-Klasse `Datenbank` als generische Klasse mit zwei Typparametern deklariert ist.

6 Diskussion

6.1 Bewertung des CMO-Refactorings

Das CMO-Refactoring ist ein Werkzeug, welches den Entwickler bei der Erstellung von Testfällen mit Mock-Objekten unterstützt. Dazu analysiert es die Testklasse und sucht kollaborierende Objekte, welche den Instanzen der CUT in den Testfällen bzw. beim Aufbau der Test-Fixture von außen übergeben werden. Hierfür ist ein entsprechendes Design der CUT unbedingte Voraussetzung. Das CMO-Refactoring unterstützt die in Kapitel 2.5 beschriebenen Entwurfsmuster Constructor-Injection und Setter-Injection. Dabei analysiert es aber nicht die CUT, sondern ermittelt deren Konstruktor- bzw. Setter-Aufrufe in der Testklasse. Ist es nicht möglich, einer CUT-Instanz die externen Kollaborateure („Dependencies“) von außen vorzugeben, muss die CUT durch ein anderes Refactoring-Werkzeug umgestaltet werden. Dies könnte z.B. ein „Inject Dependency“-Refactoring sein, welches derzeit im Rahmen des intoJ-Projektes entwickelt wird, siehe [intoJ]. Natürlich ist die Dependency Injection nicht die einzige Möglichkeit für eine Übergabe externer Abhängigkeiten. Martin Fowler beschreibt z.B. in seinem Artikel [Fowler, 2004] auch das „Service Locator“-Pattern. Hier werden einer CUT-Instanz die Kollaborateure nicht von außen vorgegeben, sondern sie holt sie sich selbst als „Services“ von einem ServiceLocator-Objekt ab, indem sie darauf entsprechende Methoden aufruft. Dieses Pattern unterstützt das CMO-Refactoring derzeit allerdings nicht. Dies gilt ebenso für die in Kapitel 2.5 beschriebenen Fabrik-Objekte und Fabrik-Methoden, mit denen sich ein Testfall ebenfalls in die Erzeugung der kollaborierenden Objekte einschalten kann.

Das CMO-Refactoring präsentiert dem Anwender die gefundenen kollaborierenden Objekte auf einer Reihe von graphischen Dialogseiten. Er kann diejenigen Objekte selektieren, welche durch Mock-Objekte ersetzt werden sollen. Die Mock-Objekte werden mit Hilfe eines voreingestellten Mock-Frameworks erzeugt. Die Anpassung für dieses Mock-Framework befindet sich in einem eigenen Eclipse-Plugin und ist beliebig austauschbar. Das CMO-Refactoring gestaltet den Code der Testklasse so um, dass den betroffenen CUT-Instanzen entweder die neuen Mock-Objekte oder die ursprünglichen Kollaborateure übergeben werden. Dies kann der Anwender über eine neue Marker-Annotation steuern, und so entscheiden, ob er die entsprechenden Testfälle in einem realen oder simulierten Umfeld laufen lassen will.

Der Austausch der kollaborierenden Objekte unterliegt einigen Einschränkungen. So kann das CMO-Refactoring derzeit nur Variablen durch Mock-Objekte ersetzen, siehe Kapitel 3.7.1. In Zukunft wäre es sicherlich wünschenswert, dass auch beliebige Ausdrücke ersetzt werden können. Ein weiteres Problem ist die tatsächliche Identität der übergebenen kollaborierenden Objekte, welche ohne eine Laufzeitanalyse der Testfälle nicht zweifelsfrei bestimmt werden kann, siehe Kapitel 3.7.2. Hier geht das CMO-Refactoring einen Kompromiß ein und ersetzt jedes kollaborierende Objekt durch ein eigenes Mock-Objekt. Dies ist sicherlich ein nicht immer wünschenswerter Weg, denn wenn ein Entwickler einen Testfall so schreibt, dass mehreren CUT-Instanzen dasselbe kollaborierende Objekt übergeben wird, dann möchte er dieses Objekt oftmals auch durch dasselbe Mock-Objekt

ersetzen. Eine mögliche Abhilfe wäre ein neues Profiling-Werkzeug, welches die Identitäten der Kollaborateure bestimmt, siehe Kapitel 7.1.

Des Weiteren findet das CMO-Refactoring kollaborierende Objekte nur in Instanziierungen oder Setter-Aufrufen der CUT. Ein so übergebenes Objekt kann durch eine Variable referenziert und im Testfall auch anderweitig verwendet werden. So kann der Testfall z.B. Methoden des kollaborierenden Objekts aufrufen, um es zu initialisieren. Diese Methoden-Aufrufe sind für ein entsprechendes Mock-Objekt nicht relevant. Dieses muss anders initialisiert werden, nämlich durch die Spezifikation seines Verhaltens, das es in einem Testfall an den Tag legen soll. Wenn das kollaborierende Objekt aber in einer JUnit `assert`-Methode zur Verifikation des Testergebnisses verwendet wird, kann dies zu Problemen führen. Z.B. könnte ein kollaborierendes Objekt, welches einer CUT-Instanz übergeben wurde, mittels eines Getters wieder gelesen werden. Anschließend kann man es über den Aufruf einer `assert`-Methode mit dem ursprünglichen Kollaborateur vergleichen, z.B.:

```
1:  // Create Collaborators
2:  kontenDb = new Datenbank();
3:  .....
4:  // Create Testobject
5:  dieBank = new Bank();
6:  dieBank.setKontoDb(kontenDb);
7:  .....
8:  // Do Test
9:  .....
10: assertEquals(dieBank.getKontoDb(), kontenDb);
```

Quelltext 6-1: Assert-Anweisung mit einem kollaborierenden Objekt

Wird der `assertEquals`-Aufruf in Zeile 10 ausgeführt, ruft das JUnit-Framework auf das erste Argument die Methode `equals()` auf und übergibt ihr das zweite Argument als Parameter. Wenn man der CUT-Instanz in Zeile 6 statt dem Objekt `kontenDb` ein Mock-Objekt übergibt, wird der Getter-Aufruf in Zeile 10 wieder dieses Mock-Objekt liefern. Man müsste also im Verhalten des Mock-Objekts für einen Aufruf der Methode `equals()` den Rückgabewert `true` angeben, damit der `assertEquals`-Aufruf erfolgreich ist. Wenn man allerdings in Zeile 10 statt der Methode `assertEquals()` die Methode `assertSame()` verwenden würde, hilft eine Spezifikation des Mock-Verhaltens nicht weiter, denn mit `assertSame()` wird die Identität der übergebenen Argumente überprüft. In diesem Fall müsste man also sowohl in Zeile 6 als auch in Zeile 10 das Objekt `kontenDb` durch dasselbe Mock-Objekt ersetzen, z.B. indem man das entsprechende Argument in den Methodenaufrufen durch den Namen einer Variable ersetzt, die das Mock-Objekt enthält. Dies sollte man aber nur dann tun, wenn man sicher nachweisen kann, dass die Variable `kontenDb` in beiden Zeilen tatsächlich dasselbe Objekt enthält! Dafür ist eine Laufzeit-analyse des Testfalls erforderlich, s.o.

Ein durch das CMO-Refactoring umgestalteter JUnit-Testfall wird derzeit nicht fehlerfrei ablaufen, da statt des Verhaltens der Mock-Objekte lediglich ein TODO-Stub eingefügt wird. Lässt man den Testfall laufen, meldet z.B. das JMock-Framework einen `java.lang.AssertionError` mit dem Kommentar „no expectations specified“. Wenn das CMO-Refactoring von Entwicklern als hilfreiches Werkzeug akzeptiert werden soll, wäre eine

automatische Generierung des Mock-Verhaltens der wichtigste Schritt. Dies könnte ein zusätzliches Profiling-Werzeug ermöglichen, welches das Verhalten der Kollaborateure durch eine Laufzeitanalyse der Testfälle ermittelt, siehe Kapitel 7.1.

6.2 Ein Szenario für den Einsatz des CMO-Refactorings

Zu Beginn eines größeren Software-Projekts wird man die zunächst die Anforderungen an das System beschreiben. Sind diese Anforderungen hinreichend festgehalten, kann man sich den Aufbau des Systems überlegen. In einer Funktions-Spezifikation können die wichtigsten Klassen festgelegt und die Interaktionen zwischen ihren Instanzen in einer Reihe von Meldungsflüssen beschrieben werden. Verfolgt man einen testgetriebenen Entwicklungsansatz, wird man noch vor der Implementierung der Software Testfälle schreiben, welche die verschiedenen Funktionen des Systems abdecken.

In großen Projekten ist es üblich, die Software-Entwicklung international zu verteilen, beispielsweise zwischen Hochlohn- und Niedriglohn-Standorten. Ein Entwickler kann für die Realisierung einiger Klassen zuständig sein. Deren Schnittstellen werden aber häufig von ganz anderen Entwicklern realisiert, die sich nicht einmal am gleichen Standort aufhalten müssen. Der Entwickler befindet sich also in der Situation, dass er seine Klassen zunächst ohne ihre kollaborierenden Objekte testen muss.

Dazu kann der Entwickler zunächst Unit-Tests schreiben, beispielsweise mit Unterstützung von JUnit 4. Diese Unit-Tests können so aufgebaut werden, als wollte man mit den realen kollaborierenden Objekten testen. Das Design der CUT muss so ausgelegt sein, dass ihr die kollaborierenden Objekte von außen übergeben werden können. Nach der Implementierung der CUT kann man die Unit-Tests mit Hilfe des CMO-Refactorings so umgestalten, dass sie mit Mock-Objekten ablaufen. Mit der derzeitigen Version des CMO-Refactorings sind die Testfälle allerdings noch nicht lauffähig, da das Verhalten der Mock-Objekte nicht automatisch generiert wird (siehe obiges Kapitel). Man müsste das Mock-Verhalten deshalb manuell in den Testfällen implementieren. Eine Grundlage dafür bilden z.B. die Meldungsflüsse in der oben erwähnten Funktions-Spezifikation.

Setzt man einen lauffähigen Testfall mit Mock-Objekten voraus, kann ein Entwickler seine CUT schon testen, obwohl die realen kollaborierenden Objekte noch nicht verfügbar sind. Sind die realen kollaborierenden Objekte in einer späteren Projektphase schließlich verfügbar, kann er in seinen Testfällen die „@DoMockCollaborators“-Annotation auskommentieren und die Tests mit den realen Objekten wiederholen. Hier wird sich i.d.R. herausstellen, dass es Unterschiede zwischen der oben erwähnten funktionalen Beschreibung und dem tatsächlichen Verhalten der kollaborierenden Objekte gibt. Infolgedessen ist eine Abstimmung zwischen den beteiligten Entwicklern notwendig und der Code der CUT bzw. die Testfälle sind entsprechend anzupassen.

Ist das System hinreichend stabil, könnte man mittels eines Profiling-Tools die Interaktion zwischen den CUT-Instanzen und den kollaborierenden Objekten aufzeichnen und daraus das Verhalten der Mock-Objekte generieren, siehe Kapitel 7.1. Dieses Verhalten könnte als Referenz für Regressionstests benützt werden. Läuft ein Testfall nach einer Änderung im

Code der CUT mit den realen kollaborierenden Objekten nicht mehr korrekt ab, kann man ihn noch einmal mit den entsprechenden Mock-Objekten testen. Wenn er dann fehlerfrei läuft, ist die Ursache des Fehlers eine Änderung im Verhalten der kollaborierenden Objekte.

6.3 Vergleich mit verwandten Arbeiten

6.3.1 InferType

InferType ([Kegel 2007]) ist ein Refaktorisierungswerkzeug für Eclipse, welches für ein selektiertes Deklarationselement eines Java-Programms einen „maximal verallgemeinerten Typ“ berechnet, d.h. einen Typ, der nur die Methoden enthält, welche das Programm für das Deklarationselement benötigt. InferType redeklariert das Deklarationselement anschließend mit diesem Typ und passt das Java-Programm so an, dass alle Zuweisungen typkorrekt bleiben.

InferType kann z.B. auf den Parameter eines CUT-Konstruktors angewendet werden, der für die Übergabe eines kollaborierenden Objekts verwendet wird. In diesem Fall berechnet InferType das minimal notwendige Protokoll für dieses Objekt. Der Typ des Kollaborateurs wird durch ein neues Interface ersetzt, welches dieses Protokoll enthält. In gewisser Weise geht InferType damit den umgekehrten Weg wie der in Kapitel 2.2 beschriebene „Need-Driven Development“-Ansatz. Hier ergibt sich das minimal notwendige Interface eines kollaborierenden Objekts während der Programmentwicklung, da sein Verhalten zunächst durch ein Mock-Objekt simuliert wird. InferType dagegen bestimmt dieses Interface aus dem fertig entwickelten Code der CUT.

Es ist sinnvoll, den Code der CUT von der Realisierung ihrer externen Schnittstellen zu entkoppeln. Deshalb sollten für die Übergabe der kollaborierenden Objekte generell Interface-Typen verwendet werden. Dadurch wird auch das Protokoll der durch das CMO-Refactoring erzeugten Mock-Objekte minimal, denn die Typen dieser Mock-Objekte entsprechen den Typen der formalen Konstruktor- bzw. Setter-Parameter, welche für die Übergabe der Kollaborateure verwendet werden. Folgt die CUT diesem Ansatz nicht, sollte sie vor dem Einsatz des CMO-Refactorings mit InferType entsprechend umgestaltet werden.

6.3.2 AgitarOne

[AgitarOne] ist ein kommerzieller Testfall-Generator, welcher als Eclipse-Plugin realisiert ist. AgitarOne analysiert eine gegebene Java-Klasse und generiert automatisch JUnit-Testfälle, welche nach den Angaben der Firma Agitar Software bis zu 80% des analysierten Code abdecken. Für die Simulation komplexer Schnittstellen, wie z.B. Datenbankverbindungen, und für das Erzielen bestimmter Testresultate, z.B. Exceptions, verwenden einige der generierten Testfälle Mock-Objekte, welche über ein proprietäres "Mockingbird"-Framework erzeugt werden. Außerdem stellt AgitarOne ein sog. „Management Dashboard“ zur Verfügung, mit dem Testmetriken, wie z.B. die Testabdeckung, die Anzahl der Fehler oder die Anzahl der getesteten Klassen, übersichtlich dargestellt werden können. AgitarOne ist

unter dem Namen „JUnit Factory“ auch als Freeware verfügbar, die allerdings nur einen Teil der Funktionalität enthält.

Die durch AgitarOne erzeugten Testfälle können für einen Entwickler nützlich sein, wenn er Regressionstests seiner Software durchführen will. Man kann dadurch besser abschätzen, wie sich Änderungen im Code einer getesteten Klasse auswirken. Die generierten Tests können jedoch keine selbst geschriebenen Testfälle ersetzen. Ein objektorientiertes Programm besteht aus einem Geflecht von Objekten, welche einen Zustand haben und miteinander in Beziehung stehen. Läuft das Programm ab, interagieren die Objekte, indem sie Methoden ihrer Schnittstellenpartner aufrufen. Dadurch ändert sich wiederum ihr Zustand. Wenn man z.B. in einem Unit-Test die Interaktionen zwischen einer CUT-Instanz und ihren Kollaborateuren testen will, muss man diese Objekte zunächst initialisieren, damit sie als Voraussetzung für den Test einen klar definierten Zustand haben. Dazu gehört auch die Übergabe der Kollaborateure an die CUT-Instanz. Anschließend ruft man mehrere Methoden des CUT-Objekts in einer bestimmten Reihenfolge auf, so dass zwischen dem CUT-Objekt und den kollaborierenden Objekten ein erwarteter Meldungsfluss entsteht, welcher z.B. in einer Funktions-Spezifikation vorgegeben ist. Diesen Meldungsfluss kann aber ein Testfallgenerator wie AgitarOne nicht kennen, und so wird er oftmals Testfälle produzieren, die von diesen Vorgaben stark abweichen und unrealistische Szenarien testen. Deshalb muss ein Entwickler Unit-Tests stets auch manuell erstellen. Möchte er diese Testfälle mit Mock-Objekten laufen lassen, kann er sie mit Hilfe des CMO-Refactorings entsprechend umgestalten.

6.3.3 Mock Central

MockCentral ([MockCentral]) ist ein Mock-Framework, welches die Java Enterprise Edition (siehe [JEE]) unterstützt. Das besondere an diesem Framework ist, dass es einen graphischen Editor für die Definition von Mock-Objekten besitzt. Jedem Mock-Objekt wird dabei ein eigener Name zugewiesen. Außerdem kann der Typ des Mock-Objekts angegeben werden. Weiterhin ist eine Spezifikation der für das Mock-Objekt erwarteten Methodenaufrufe nebst ihren Parametern möglich. Der Rückgabewert einer Methode kann auch angegeben werden. Die folgende Abbildung zeigt ein Beispiel, welches mit MockCentral mitgeliefert wird:

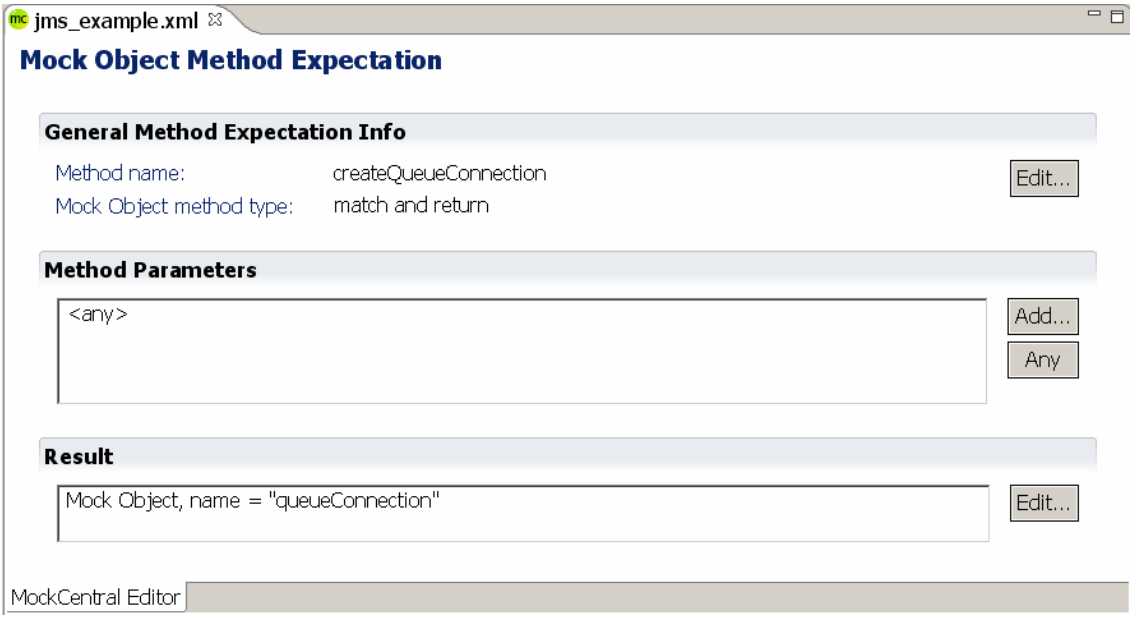

Abbildung 6-1: MockCentral Editor

Der MockCentral-Editor erlaubt es, Mock-Objekte in Gruppen zusammenzufassen, welche als „Fixtures“ bezeichnet werden. Alle eingegebenen Daten werden in einer XML-Datei abgespeichert. Will der Anwender die definierten Mock-Objekte in einem Testfall verwenden, muss er dort die Klasse `MockCentralServer` instanziieren und diesem Objekt den Pfad der XML-Datei übergeben. Anschließend kann er über Methoden des `MockCentral-Server`-Objekts die benötigten Mock-Objekte erzeugen, welche das in der XML-Datei spezifizierte Verhalten an den Tag legen. Allerdings muss der Anwender, ähnlich wie bei den in Kapitel 2.4 beschriebenen Mock-Frameworks, selbst entscheiden, welche Kollaborateure der CUT-Instanzen durch die erzeugten Mock-Objekte ersetzt werden sollen. Hier bietet MockCentral keine Unterstützung. Für die Spezifikation des Mock-Verhaltens hingegen ist der graphische Editor sicherlich eine große Hilfe, weil der Anwender diesen Code nicht manuell in den Testfall einfügen muss.

7 Schlußbetrachtungen

7.1 Mögliche Erweiterungen des CMO-Refactorings

Die wichtigste Erweiterung des CMO-Refactorings wäre sicherlich die automatische Generierung des Verhaltens der erzeugten Mock-Objekte. Dazu muss eine Laufzeitanalyse aller Testfälle durchgeführt werden, in denen CUT-Instanzen verwendet werden, deren Kollaborateure durch Mock-Objekte ersetzt werden sollen. Man könnte sich vorstellen, dass dies ein neues Profiling-Werkzeug übernimmt, welches während der Testausführung im Hintergrund mitläuft. Für die dynamische Programmanalyse könnte z.B. das in [Meyer 2007], Seite 21, erwähnte Java Virtual Machine Tool Interface (JVM TI) benützt werden. Für die Angabe des Verhaltens eines Mock-Objekts werden folgende Informationen benötigt:

- Welche Methoden des zu ersetzenden kollaborierenden Objekts werden in einem Testfall in welcher Reihenfolge aufgerufen?
- Welche Objekte werden diesen Methoden als Parameter übergeben?
- Welches Objekt gibt die jeweilige Methode zurück?

Die Fragen nach den übergebenen bzw. zurückgegebenen Objekten sind nicht so trivial wie es auf den ersten Blick aussieht. Jedes der Parameter-Objekte und auch das zurückgegebene Objekt haben einen Zustand, welcher durch die Belegung ihrer Instanzvariablen bestimmt ist. Jede Instanzvariable kann entweder einen primitiven Wert oder ein Objekt enthalten, das wiederum einen Zustand hat. Genau genommen geht es hier also um ein ganzes Geflecht von Objekten, welches einer Methode übergeben bzw. von ihr zurückgeliefert wird. Bezüglich der Parameterobjekte könnte man sich das Leben leichter machen und die Typen der formalen Methoden-Parameter bestimmen. Das Mock-Verhalten spezifiziert man dann so allgemein, dass die Methode mit irgendeinem Objekt des entsprechenden Typs aufgerufen werden soll, siehe z.B. die JMock-Matcher in Tabelle 2-2. Das zurückgegebene Objektgeflecht muss man jedoch im Verhalten des Mock-Objekts nachbilden, denn es wird i.d.R. vom Aufrufer verwendet. Alternativ kann die Methode aber auch ein weiteres Mock-Objekt zurückgeben, welches wiederum das erwartete Verhalten des Rückgabe-Objekts nachbildet.

Das oben erwähnte Profiling-Werkzeug könnte die ermittelten Daten in einer Datei abspeichern, welche anschließend vom CMO-Refactoring gelesen wird, um die notwendigen Anweisungen für das Verhalten der Mock-Objekte zu erzeugen. Da diese Anweisungen spezifisch für das jeweils verwendete Mock-Framework sind, muss das Interface `IMockRewriter` entsprechend erweitert werden. Derzeit unterstützt das CMO-Refactoring lediglich das JMock-Framework. Für andere Mock-Frameworks, wie z.B. EasyMock oder RMock, können eigene Extension-Plugins entwickelt werden. Leider haben die in Kapitel 2.4 beschriebenen Mock-Frameworks eine sehr unterschiedliche Syntax, insbesondere wenn die Reihenfolge der erwarteten Methodenaufrufe spezifiziert werden soll. Auch benötigen z.B. EasyMock und RMock spezielle Anweisungen, um das Verhalten der erzeugten Mock-Objekte zu verifizieren, während JMock dies automatisch tut. Diese Unterschiede muss man

in Betracht ziehen und das Interface `IMockRewriter` so erweitern, dass es möglichst generisch ist.

Das Profiling-Werkzeug könnte neben den Methodenaufrufen auch die Identität der kollaborierenden Objekte bestimmen. Dadurch könnte man feststellen, welche Kollaborateure einer CUT-Instanz tatsächlich übergeben werden. Wenn dasselbe kollaborierende Objekt mehreren CUT-Instanzen übergeben wird, könnte es das CMO-Refactoring immer durch dasselbe Mock-Objekt ersetzen. Außerdem könnte das CMO-Refactoring so feststellen, ob ein ersetzter Kollaborateur in einem `assert`-Aufruf verwendet wird und diesen gegebenenfalls auch hier durch das korrespondierende Mock-Objekt ersetzen.

Es wäre auch sinnvoll, das CMO-Refactoring hinsichtlich der Erkennung der Kollaborateure zu erweitern. So könnte es die Übergabe von kollaborierenden Objekten mittels Fabrik-Methoden, Fabrik-Objekten oder des „Service Locator“-Patterns unterstützen, siehe Kapitel 2.5. Außerdem könnte man das CMO-Refactoring so ausbauen, dass es neben Variablen auch Ausdrücke durch Mock-Objekte ersetzen kann. Dabei müssten die Seiteneffekte eines Ausdrucks berücksichtigt werden, siehe Kapitel 3.7.1. Wenn das CMO-Refactoring keine kollaborierenden Objekte findet, könnte die CUT durch ein „Inject Dependency“-Refactoring umgestaltet werden, siehe Kapitel 6.1. Der Aufruf dieses Refactorings könnte in das CMO-Refactoring integriert werden. Man könnte das „Inject Dependency“-Refactoring auf Wunsch des Anwenders starten und nach Beendigung desselben zum CMO-Refactoring zurückkehren, welches dann voraussichtlich kollaborierende Objekte finden würde.

7.2 Fazit

Die vorliegende Arbeit unterstützt den Software-Entwickler hinsichtlich der Erstellung von Testfällen mit Mock-Objekten. Sie geht dabei einen anderen Weg als das in [Freeman et al., 2004] beschriebene Need-Driven Development. Es lohnt sich aber, diesen Weg weiterzuverfolgen, denn ein Entwickler sollte sich auf seine primäre Aufgabe, nämlich die Entwicklung von Software konzentrieren. Er sollte sich nicht mit der Syntax von Mock-Frameworks beschäftigen müssen, die letztendlich nur der Entwicklungsunterstützung dienen.

Deshalb sollte man das CMO-Refactoring weiter zu einem in Eclipse integrierten „Mock Support-Toolset“ ausbauen, das dem Entwickler hinsichtlich aller Aspekte unterstützt, die bei der Entwicklung mit Mock-Objekten anfallen. Dies sind vor allem die Erzeugung der Mock-Objekte sowie die komplette Definition und Verifikation ihres Verhaltens. Das Ziel muss sein, dass ein Testfall sowohl mit den realen kollaborierenden Objekten als auch mit den entsprechenden Mock-Objekten fehlerfrei abläuft.

A Eclipse API's für die Plugin-Entwicklung

A.1 Selektion eines Java-Elementes im Quelltext

Will man das Kontextmenü eines Java-Editors erweitern, muss man für den Eclipse-Erweiterungspunkt `org.eclipse.ui.popupMenus` ein `viewerContribution`-Element definieren, siehe Kapitel 4.6. Diese Kontribution enthält für jeden neuen Menü-Eintrag ein sogenanntes `action`-Element. In dieser Action wird der Text des Menüeintrags definiert, sowie eine Klasse, welche das Interface `org.eclipse.ui.IEditorActionDelegate` implementiert. Im Falle des CMO-Plugins ist dies die Klasse `MockEditorAction`. Wird eine Java-Quelldatei editiert, ruft das Eclipse-Framework die Methode `setActiveEditor()` des Interfaces `IEditorActionDelegate` auf, um den aktiven Editor zu übergeben. Wird anschließend ein Textfragment selektiert und im Kontextmenü des Java-Editors der in der obigen Action definierte neue Menüeintrag ausgewählt, ruft das Eclipse-Framework die Methode `run()` auf. Hier muss der selektierte Text verarbeitet werden. Dies soll am Beispiel der Klasse `CmoEditorAction` gezeigt werden.

Der selektierte Text kann über den aktiven Editor ermittelt werden. Das entsprechende Objekt ist eine Implementierung des Interfaces `org.eclipse.jface.text.ITextSelection`. Weiterhin kann über die Eclipse-interne Klasse `org.eclipse.jdt.internal.ui.JavaPlugin` die editierte Java-Quelldatei bestimmt werden. Sie wird im Eclipse Java-Modell (siehe [Eclipse 3.3 Java Model]) durch das Interface `org.eclipse.jdt.core.ICompilationUnit` repräsentiert:

```
1:  public class CmoEditorAction extends CmoAction
2:          implements IEditorActionDelegate
3:  {
4:     private ISelection selection;
5:     private ITextEditor activeEditor = null;
6:     .....
7:     public void setActiveEditor(IAction action,IEditorPart targetEditor)
8:     {
9:        activeEditor = (ITextEditor)targetEditor;
10:    }
11:    public void run(IAction action)
12:    {
13:       ITextSelection textSelection;
14:       ICompilationUnit editedCompUnit;
15:       .....
16:       // Get text selection
17:       selection= activeEditor.getSelectionProvider().getSelection();
18:       textSelection = (ITextSelection)selection;
19:       // Get the edited Compilation Unit
20:       IWorkingCopyManager manager =
21:          JavaPlugin.getDefault().getWorkingCopyManager();
22:       editedCompUnit =
23:          manager.getWorkingCopy(activeEditor.getEditorInput());
24:       .....
25:    }
26: }
```

Quelltext A-1: Bestimmung der Text-Selektion und der editierten Java-Quelldatei

Die Methode `getSelectedType()` der Klasse `CmoEditorAction` zeigt, wie man anschließend die Java-Elemente ermitteln kann, die dem selektierten Text entsprechen. Dies geschieht über die Methode `codeSelect()` des `ICompilationUnit`-Objekts, welcher der Offset und die Länge der Textselektion im Java-Quelltext übergeben werden. Anschließend wird die Art des ersten selektierten Elements überprüft, denn die Klasse `MockEditor-Action` erwartet, dass im Java-Quelltext ein Typ selektiert wird, welcher der CUT entspricht. Ist das selektierte Element eine Klasse, wird diese zurückgegeben. Ist es ein Konstruktor, wird der Typ zurückgegeben, in dem dieser Konstruktor deklariert ist:

```
1:  private IType getSelectedType(ICompilationUnit compUnit,
2:                    ITextSelection textSelection)
3:  {  IJavaElement selectedElement = null;
4:     IJavaElement[] javaElements;
5:     try
6:     {  // Get selected java element
7:        javaElements = compUnit.codeSelect(
8:           textSelection.getOffset(), textSelection.getLength());
9:        if (javaElements.length == 0) return(null);
10:       selectedElement = javaElements[0];
11:       if (selectedElement.isReadOnly()) return(null);
12:       // Check type of selected java element
13:       if (selectedElement instanceof IType)
14:       {  // The CUT must be a class
15:          IType selectedType = (IType)selectedElement;
16:          if (selectedType.isClass()) return(selectedType);
17:          else return null;
18:       }
19:       else if (selectedElement instanceof IMethod)
20:       {  IMethod selectedMethod = (IMethod)selectedElement;
21:          // Check if the CUT constructor has been selected
22:          if (selectedMethod.isConstructor())
23:             return selectedMethod.getDeclaringType();
24:          else return null;
25:       }
26:       else return null;
27:    }
28:    catch (JavaModelException e) { ..... }
29: }
```

Quelltext A-2: Bestimmung des selektierten Java-Elements

Als letzter Schritt muss die Klasse bestimmt werden, in der sich der selektierte Text befindet, denn dies ist die Testklasse, auf die das CMO-Refactoring angewendet werden soll. Hierbei ist zu beachten, dass auch eine innere Klasse Testfälle enthalten kann, d.h. Methoden, die mit `@Test` annotiert sind. In der Methode `getTestClass()` der Klasse `CmoEditorAction` wird deshalb nach der innersten Klasse gesucht, welche die Textselektion enthält. Dazu werden alle Typen überprüft, die in der editierten Java-Quelldatei (Parameter `compUnit`) deklariert sind:

```
1:  private TestClass getTestClass(ICompilationUnit compUnit,
2:                       ITextSelection textSelection)
3:  {  .....
4:     try
5:     {  // The testclass may not only be the primary type
6:        // with the same name as the compilation unit, but
7:        // also an inner class declared in it
8:        IType[] typesInCompUnit = compUnit.getAllTypes();
9:        // The last type in the compilation unit is checked first,
10:       // so the innermost class is found.
11:       for (int i = typesInCompUnit.length-1; i >= 0; i--)
12:       {  if (typesInCompUnit[i].isClass())
13:          {  if ( typeContainsTextSelection(
14:                      typesInCompUnit[i], textSelection) )
15:             {  .....
16:             }
17:          }
18:       }
19:    }
20:    catch (JavaModelException e) { ..... }
21:    .....
22: }
23: private boolean typeContainsTextSelection(IType type,
24:                      ITextSelection textSelection)
25: {  boolean result = false;
26:    int selectionBegin = textSelection.getOffset();
27:    int selectionEnd = selectionBegin + textSelection.getLength() - 1;
28:    try
29:    {  ISourceRange typeRange = type.getSourceRange();
30:       int typeBegin = typeRange.getOffset();
31:       int typeEnd = typeBegin + typeRange.getLength() - 1;
32:       if ((selectionBegin >= typeBegin) && (selectionEnd <= typeEnd))
33:       {  result = true;
34:       }
35:    }
36:    catch (JavaModelException e) { ..... }
37:    return result;
38: }
```

Quelltext A-3: Bestimmung der Klasse, die den selektierten Text enthält

A.2 Suche von Java-Elementen mit der Search Engine

Eclipse bietet mit der Klasse `org.eclipse.jdt.core.search.SearchEngine` eine komfortable Suchmaschine, um in Java-Projekten nach bestimmten Java-Elementen zu suchen. Vor einer Suche wird zunächst mittels der Klassenmethode `createJavaSearch-Scope()` ein bestimmter Bereich definiert, in dem gesucht werden soll. Anschließend erzeugt man durch Aufruf der Klassenmethode `createPattern()` der Klasse `SearchPattern` ein Suchmuster. Hier kann die Art der gesuchten Java-Elemente angegeben werden und man kann spezifizieren, ob man die Elemente direkt sucht oder Verweise auf dieselben. Für das Einsammeln der Suchergebnisse muss ein Objekt erzeugt werden, dessen Typ von der Klasse `SearchRequestor` abgeleitet ist und die Methode `acceptSearchMatch()` überschreibt. Schließlich kann die Suche durch Aufruf der Methode `search()` eines `SearchEngine`-Objekts gestartet werden.

Ein Beispiel zeigt die Methode `searchAllDeclaredClasses()` der Klasse `TestClass`. Hier werden alle Klassen gesucht, die in den Java-Quelldateien des Projekts deklariert sind, in dem sich auch die Testklasse befindet:

```
1:  private List<IType> searchAllDeclaredClasses()
2:  {  final List<IType> typeList = new ArrayList<IType>();
3:     // Search sources files of java project
4:     IJavaSearchScope searchScope = SearchEngine.createJavaSearchScope(
5:         new IJavaElement[] { getProject() }, IJavaSearchScope.SOURCES);
6:     // Search for declarations of classes
7:     SearchPattern pattern = SearchPattern.createPattern(
8:          "*",
9:          IJavaSearchConstants.CLASS,
10:         IJavaSearchConstants.DECLARATIONS,
11:         SearchPattern.R_PATTERN_MATCH);
12:    SearchRequestor requestor = new SearchRequestor()
13:    {  // Collect the search matches
14:       public void acceptSearchMatch(SearchMatch match)
15:       {  if (match.getAccuracy() == SearchMatch.A_ACCURATE)
16:          {  Object element = match.getElement();
17:             if (element instanceof IType)
18:             {   typeList.add((IType)element);
19:             }
20:          }
21:       }
22:    };
23:    // Search
24:    SearchEngine searchEngine = new SearchEngine();
25:    try
26:    {  searchEngine.search(
27:           pattern,
28:           new SearchParticipant[] {
29:              SearchEngine.getDefaultSearchParticipant() },
30:           searchScope,
31:           requestor,
32:           null );
33:    }
34:    catch (CoreException e) { e.printStackTrace(); }
35:    return typeList;
36: }
```

Quelltext A-4: Suche von Klassen mit der Search Engine

A.3 Aufbau eines Dialogs zur Auswahl von Java-Elementen

Soll auf einer graphischen Eingabeseite ein Dialog aufgebaut werden, welcher die Selektion eines Java-Elementes aus einer bestimmten Auswahl erlaubt, kann dazu die statische Methode `createTypeDialog()` der Klasse `org.eclipse.jdt.ui.JavaUI` verwendet werden. Sie erwartet als Parameter einen Bereich, in dem sich die auszuwählenden Java-Elemente befinden sollen, sowie den Typ dieser Elemente. Ein Beispiel zeigt die Methode `cutButtonSelectionHandler()` der Klasse `CmoRefactoringInputPage`. Hier soll eine Klasse aus dem Projekt der Testklasse selektiert werden. Dabei werden nur Klassen berücksichtigt, die in den Java-Quelldateien des Projekts deklariert sind. Der entsprechende

Bereich für die Suche wird mit der schon oben erwähnten Klassenmethode `createJavaSearchScope()` der Klasse `SearchEngine` erzeugt:

```
1:  private void cutButtonSelectionHandler()
2:  {  IType selectedType;
3:     IJavaProject javaProject = testClass.getProject();
4:     // Create a type selection dialog for the scope of the
5:     // TestClass project (source files only)
6:     IJavaSearchScope searchScope = SearchEngine.createJavaSearchScope(
7:          new IJavaElement[] { javaProject}, IJavaSearchScope.SOURCES);
8:     try
9:     {  SelectionStatusDialog dialog =
10:         (SelectionStatusDialog) JavaUI.createTypeDialog(
11:             getShell(), getContainer(), searchScope,
12:             IJavaElementSearchConstants.CONSIDER_CLASSES,
13:             false, "?");
14:       dialog.setTitle("Choose Class under Test");
15:       if (dialog.open() == Window.OK)
16:       {  selectedType = (IType)dialog.getFirstResult();
17:          .....
18:       }
19:    }
20:    catch (JavaModelException e) { e.printStackTrace(); }
21: }
```

Quelltext A-5: Aufbau eines „Type Selection“-Dialogs

Den in dieser Methode neu erzeugten graphischen Dialog zeigt Abbildung 3-6 in Kapitel 3.4.

A.4 Programmatische Anpassung des Classpath eines Java-Projektes

Soll ein Classpath Container einem Java-Projekt programmgesteuert hinzugefügt werden, muss zunächst über die Klasse `org.eclipse.jdt.core.JavaCore` ein neuer Classpath Container Entry erzeugt werden. Anschließend besorgt man sich über die gleiche Klasse den Initialisierer des Classpath Containers und ruft dessen Methode `initialize()` auf. Dadurch wird der eigentliche Container erzeugt und an das übergebene Java-Projekt gebunden. Schließlich fügt man den neu erzeugten Classpath Container Entry dem Classpath des Projektes hinzu. Ein Beispiel zeigt die Methode `perform()` der Klasse `CmoClassPathChange`, welche den CMO Annotation-Container einem Projekt hinzufügt:

```
1:  public Change perform(IProgressMonitor pm) throws CoreException
2:  {
3:     IClasspathEntry cmoAnnotationContainerEntry = null;
4:     .....
5:     // Get entries of Project classpath
6:     List<IClasspathEntry> projectClassPath = new ArrayList<IClasspathEntry>(
7:         Arrays.asList(project.getRawClasspath()));
8:     if (addCmoAnnotationContainer)
9:     {  // Create new Classpath Container Entry
10:       // for the CMO Annotation Container
11:       cmoAnnotationContainerEntry = JavaCore.newContainerEntry(
12:           new Path(CmoAnnotationContainer.CMO_ANNOTATION_CONTAINER_ID));
13:       // Initialize CMO Annotation Container
14:       ClasspathContainerInitializer containerInit =
15:           JavaCore.getClasspathContainerInitializer(
16:               CmoAnnotationContainer.CMO_ANNOTATION_CONTAINER_ID);
17:       containerInit.initialize(new Path(
18:           CmoAnnotationContainer.CMO_ANNOTATION_CONTAINER_ID), project);
```

```
19:       .....
20:    }
21:    .....
22:    // Add Container to Java Project
23:    if (cmoAnnotationContainerEntry != null)
24:        projectClassPath.add(cmoAnnotationContainerEntry);
25:    .....
26:    // Set new project classpath
27:    project.setRawClasspath(projectClassPath.toArray(
28:        new IClasspathEntry[projectClassPath.size()]), null);
29:    .....
30: }
```

Quelltext A-6: Programmatisches Hinzufügen des CMO Annotation-Containers

Classpath Container bieten eine elegante Möglichkeit, mehrere Classpath Entries zusammenzufassen. Sie haben allerdings einen Nachteil: Wenn Eclipse neu gestartet wird, wird für jedes Projekt im Workspace der Classpath neu aufgelöst. Dabei wird für jeden Classpath Container Entry der entsprechende Initializer aufgerufen, der das eigentliche Container-Objekt neu erzeugt und an das Projekt bindet! Offensichtlich werden diese Objekte nicht persistent gespeichert. Dies bedeutet, dass man nicht sicher sagen kann, ob ein dem Projekt hinzugefügter Classpath Container nach dem Neustart von Eclipse noch denselben Inhalt hat, denn dieser hängt vom Code des Plugins ab, welches den entsprechenden Container-Initialisierer zur Verfügung stellt. Im schlimmsten Fall kann der Container-Inhalt von den Voreinstellungen des Plugins und auch vom Vorhandensein bzw. Nicht-Vorhandensein anderer Plugins abhängig sein.

Das CMO-Plugin fügt aus diesem Grund die Mock-Framework-Bibliotheken dem Classpath des Projektes als einzelne Library Classpath Entries hinzu. Dazu wird ein Hilfsobjekt vom Typ `MockFrameworkLibraries` benützt. Bei dessen Erzeugung werden die Pfade der Mock-Bibliotheken des ausgewählten Mock-Frameworks über das entsprechende Extension-Plugin bestimmt:[20]

```
1:  public MockFrameworkLibraries()
2:  {
3:     // Get the paths of the Mock Libraries
4:     String mockFrameworkName = CmoPreferences.getMockFrameworkName();
5:     IMockRewriter mockRewriter = MockFrameworkExtensions.getInstance().
6:         getMockRewriter(mockFrameworkName);
7:     if (mockRewriter != null)
8:     {  List<IPath> mockLibraryPaths = mockRewriter.getMockLibraryPaths(
9:            CmoPreferences.getMockFrameworkDirectory());
10:       if (mockLibraryPaths != null) addLibraries(mockLibraryPaths);
11:    }
12:    // Build Library Classpath Entries
13:    createClasspathEntries();
14: }
```

Quelltext A-7: Abfrage der Pfade der Mock-Framework-Bibliotheken

Die in Zeile 13 aufgerufene Methode `createClasspathEntries()` der Basisklasse `LibraryContainer` erzeugt für jeden Pfad via `JavaCore` einen Library Classpath Entry:

[20] Details zum MockFrameworkExtensions-Singleton und zum Interface IMockRewriter beschreibt Kapitel 4.3.

```
1:  public void createClasspathEntries()
2:  {  ArrayList<IClasspathEntry> entries = new ArrayList<IClasspathEntry>();
3:     for (IPath p : containedLibraries)
4:     {  entries.add(JavaCore.newLibraryEntry(p, null, null));
5:     }
6:     classPathEntries = entries.toArray(new IClasspathEntry[entries.size()]);
7:  }
```

Quelltext A-8: Erzeugung der Library Classpath Entries in der Klasse LibraryContainer

Die Methode `perform()` der Klasse `CmoClassPathChange` fügt die so erzeugten Library Classpath Entries dem Classpath des Java-Projekts in der gleichen Weise hinzu wie den Classpath Container Entry in Quelltext A-6, Zeile 24.

Tabellenverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[AgitarOne]

AgitarOne JUnit Generator. Online unter http://www.agitar.com/solutions/products/automated_junit_generation.html. Freeware-Version unter http://www.junitfactory.com

[Beck, Gamma]

Kent Beck, Erich Gamma: *Test Infected: Pogrammers Love Writing Tests*. Online unter http://members.pingnet.ch/gamma/junit.htm

[cglib]

Code Generation Library. Online unter http://cglib.sourceforge.net/

[Clayberg&Rubel 2006]

Eric Clayberg, Dan Rubel (2006): *Eclipse: Building Commercial-Quality Plug-ins*, Addison-Wesley, Boston

[Daum 2006]

Dr. Berthold Daum (2006): *Java-Entwicklung mit Eclipse 3.2*, dpunkt.verlag, Heidelberg

[EasyMock]

EasyMock Framework. Online unter http://www.easymock.org/

[EasyMock Dokumentation]

EasyMock Dokumentation. Online unter http://www.easymock.org/EasyMock2_3_Documentation.html

[Eclipse]

Eclipse Platform. Online unter http://www.eclipse.org

[Eclipse 3.3 Java Model]

Eclipse 3.3 Programmers Guide: Java Model. Online unter http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_int_model.htm

[Equinox]

Equinox OSGi-Framework. Online unter http://www.eclipse.org/equinox/

[Fowler, 2004]

Martin Fowler: *Inversion of Control Containers and the Dependency Injection pattern*. Online unter http://www.martinfowler.com/articles/injection.html

[Freeman et al., 2004]

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes (OOPSLA 2004): *Mock Roles, not Objects*. Online unter http://www.jmock.org/oopsla2004.pdf

[Freeman et al., 2004/II]

Steve Freeman, Tim Mackinnon, Nat Pryce, Joe Walnes (OOPSLA 2004): *JMock: Supporting Responsibility-Based Design with Mock Objects*

[Gamma&Beck 2004]

Erich Gamma, Kent Beck (2004): *Contributing to Eclipse: Principles, Patterns and Plug-Ins*, Addison-Wesley, Boston

[Gamma et al., 2004]

Erich Gamma, Richard Helm, Raph Johnson, John Vlissides (2004): *Entwurfsmuster*, Addison-Wesley, München

[hamcrest]

Google Hamcrest-Library. Online unter http://code.google.com/p/hamcrest/

[intoJ]

intoJ-Projekt. Online unter http://www.intoj.org

[JEE]

Java Platform, Enterprise Edition (Java EE). Online unter http://java.sun.com/javaee/

[JLS3]

The Java Language Specification, Third Edition. Online unter http://java.sun.com/docs/books/jls/

[JMock]

JMock Framework. Online unter http://www.jmock.org

[JMock-API 2.2.0]

JMock-API, Version 2.2.0. Online unter http://www.jmock.org/javadoc/2.2.0/

[JUnit]

Junit Testframework. Online unter http://www.junit.org/home

[Kegel 2007]

Hannes Kegel (2007): *Constraint-basierte Typinferenz für Java 5*. Diplomarbeit an der FernUniversität Hagen

[Kuhn&Thomann 2006]

Thomas Kuhn, Olivier Thomann (2006): *Abstract Syntax Tree*. Online unter http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html

[Link 2005]

Johannes Link (2005): *Softwaretests mit JUnit*, dpunkt.verlag, Heidelberg

[Mackinnon et al., 2000]

Tim Mackinnon, Steve Freeman, Philip Craig (XP2000 conference): *Endo-Testing: Unit Testing with Mock Objects*

[Meyer 2007]

Nils Meyer (2007): *Ein Eclipse-Framework zur Markierung von logischen Fehlern im Quellcode*. Masterarbeit an der FernUniversität Hagen

[MockCentral]

MockCentral Framework. Online unter http://mockcentral.org/

[NetWeaver]

SAP Netweaver Developer Studio. Online unter http://www.sap.com/germany/plattform/netweaver/components/developerstudio/index.epx

[objenesis]

Google Objenesis-Library Online unter http://objenesis.googlecode.com/svn/docs/index.html

[OSGi]

Open Services Gateway Initiative. Online unter http://www.osgi.org/

[RMock]

RMock Framework. Online unter http://rmock.sourceforge.net/

[Thomas & Hunt, 2002]

Dave Thomas, Andy Hunt: Mock Objects. IEEE Software 2002

[Widmer 2006]

Tobias Widmer: *Unleashing the power of refactoring*. Eclipse Magazine, July 4, 2006. Online unter http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html

[Wind River]

Wind River Workbench. Online unter http://www.windriver.com/products/product-overviews/Workbench-overview.pdf

Inhalt der beiliegenden CD

VERZEICHNIS	INHALT
Doc	Diese Ausarbeitung als PDF-Datei
Src	Die Quelldateien des CMO-Plugins und der JMock 2.2.0/JMock 2.4.0 Extension-Plugins
JavaDoc	Die Dokumentation der Plugins als HTML-Seiten
Tests	Die Quelldateien der Testprojekte
Install	Eine gepackte Update-Site mit den Plugins und ein Archiv mit den Testprojekten
Frameworks	Das JMock-Framework (Version 2.2.0 und 2.4.0), das EasyMock-Framework (Version 2.3) incl. der EasyMock Class Extension (Version 2.2.2), das RMock-Framework (Version 2.0.0) und die Version 4.4 des JUnit-Frameworks

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.

Holzkirchen, den 27. März 2008

Thomas Baumann