LVA Testen von Softwaresystemen
WS 2004

Dr. Christoph Steindl

Peter Maurer, 0156185
Deepak Dhungana, 0057030
George Homorozeanu, 0256766

Inhaltsverzeichnis

Einführung

Was sind Mock Objekte?

Mock-Objekte sind Dummy- Objekte, die die Spezifikation des erwarteten Verhaltens beinhalten und die Überprüfung des tatsächlichen auftretenden Verhaltens durchführen können. Ein *Dummy-Objekt* ist ein Objekt, das ein anderes für die Dauer eines Tests ersetzt. Dabei implementiert es das gleiche Interface wie das »richtige« Objekt, ersetzt dabei jedoch komplexe Berechnungen durch konstante Rückgaben, wirft auf Befehl Exceptions, führt zusätzliche Parameterüberprüfungen durch oder tut andere Dinge, die nur in Tests benötigt werden. Mock Objekte simulieren nicht nur die jeweilige Ressource, sondern sie sind auch in der Lage, ihre korrekte Verwendung selber zu überprüfen. Sie bilden eines der mächtigsten Testmuster.

Warum Mock Objekte?

Wenn es ums Testen von einem System geht, hat man üblicherweise nicht alle Komponenten zur Verfügung und man kann nicht auf alle Komponenten Einfluss nehmen. Abhängigkeiten zwischen Klassen erschweren das Testen. In solchen Situationen spielen Mock Objekte eine wichtige Rolle. Komponenten wie das Netzwerk, eine Datenbank oder eine Servlet - Engine kann man nicht ohne weiteres kontrollieren. Die Abhängigkeit zwischen verschiedenen Komponenten wird mit speziellen Objekten überbrückt, sodass das Testen auch ohne die Objekte des Frameworks einen Sinn macht. Eine einzelne Klasse ist leicht zu testen. Im echten Projekt ist es oft anders. Wenn Klassen voneinander abhängen, kann man sie nur noch gemeinsam testen. Abhängigkeiten sind oft nur implizit. Soll eine Softwarekomponente getestet werden, die mit anderen Komponenten interagiert, müssen Attrappen für diese erstellt werden(Mock-Objekte). Diese simulieren die für den Testzweck zu ersetzenden Klassen in einer einfacheren Weise.

Vorteile der Verwendung von Mock Objekten

- früheres Testen von integrierten Objekten
- Abtrennung Komponente - Container in Testumgebung möglich
- Anwendungsdesign verändert sich(BIT vs. lose Kopplung)
- Einsparungen durch automatisierte Codegeneration von Mock-Klassen aus Schnittstellen - Funktionalität der Mock-Klassen sind Testkosten
- Kopplung zwischen Mocks vermeiden, funktionsreiche Mocks umstritten

Wann sollten Mock Objekte verwendet werden

- Wenn das echte Objekt nicht deterministische, nicht vorhersehbare Eigenschaften hat.
- Wenn es nicht leicht ist, reale Objekte zu installieren und zu konfigurieren
- Wenn das Verhalten des echten Objekts schwierig zu reproduzieren ist. Z.B. Netzwerkfehler
- Wenn das reale System Testfälle unnötig verlangsamt.

- Wenn manuelle Eingriffe notwendig sind, um das echte Objekt zu bedienen.
- Damit die Tests wiederholt werden können
- Wenn die echten Komponenten noch nicht verfügbar bzw. noch in der Entwicklung sind.

Diese Probleme kann man umgehen, wenn man mit Mock Objekten arbeitet. Ein großer Vorteil liegt daran, dass man bei der Programmierung gar nicht berücksichtigen braucht, wie das Programm eigentlich getestet wird und erzeugt keinen großen Overhead.

Ähnliche Begriffe

Ein *Stub* ist ein bislang nur rudimentär implementierter Teil einer Software, der später durch die richtige Implementierung ersetzt wird. Die Aufgabe eines Stub-Objekts ist die eines Platzhalters für geplante, aber noch nicht umgesetzte Funktionalität.
Ein *Dummy* dagegen kann die echte Implementierung für Testzwecke ersetzen. Ob das Echte oder ein Dummy-Objekt verwendet wird, entscheidet sich durch codeinterne oder externe Konfiguration.

Klassen der Mock-Objekte lassen sich in den Entwicklungsumgebungen oftmals automatisch erzeugen. Natürlich ist es auch möglich, Mock Objekte selbst zu schreiben. Aber es gibt viele verschiedene Tools, für verschiedene Sprachen, welche die automatische Erzeugung von Mock Objekten unterstützen. Die drei Schritte für das Einsetzen von Mock Objekten sind:

- Eine Schnittstellen Beschreibung für das Mock Objekt schreiben.
- Diese Schnittstellen implementieren.
- Das Mock Objekt zum Testen der Schnittstelle schreiben.

Wir müssen jedoch darauf achten, dass unsere Mock-Objekte nicht zu komplex werden. Anzeichen für zu große Komplexität sind:

- Sie duplizieren Programmlogik aus den »richtigen« Klassen.
- Sie rufen ihrerseits andere Mock- oder Dummy-Objekte auf.
- Wir haben das Bedürfnis, Testfälle für die Mock-Objekte selbst zu schreiben.

In diesen Fällen hilft es, einen Schritt zurückzutreten und uns zu fragen, ob wir die Mocks durch Aufteilung nicht vereinfachen können, ob wir sie vielleicht gar nicht benötigen oder ob unser Mock-Problem nicht eigentlich unsere Aufmerksamkeit auf ein Designproblem lenken möchte.

Easy Mock, Mock Creator und Mock Maker sind Generatoren, die die Erstellung von Mock-Objekten weitgehend vereinfachen und automatisieren.

Einsatz von Mock Objekten

Aussehen eines einzelnen Unit Tests:

1. Erzeuge die nötigen Mock-Objekte.
2. Setze, wenn nötig, den internen Zustand dieser Mock-Objekte.
3. Setze die erwarteten Ausgaben in den Mock-Objekten.
4. Rufe den zu testenden Code mit den Mock-Objekten als Parameter auf.
5. Überprüfe, wenn angebracht, Zustandsänderungen in den zu testenden Objekten durch direkte Tests.
6. Verifiziere die Konsistenz der Mock-Objekte.

Ansätze, ein Dummy-Objekt zu bauen:

1. Indem wir es als Unterklasse von der richtigen Implementierung ableiten.
2. Indem sowohl die richtige als auch die Mock-Klasse das gleiche Interface implementieren.

Während die erste Variante die einfachere ist, da wir kein eigenes Interface implementieren müssen, birgt sie gewisse Gefahren. So passiert es relativ schnell, dass man bei einer Änderung der Signatur der richtigen Klasse vergisst, die Mock-Klasse anzupassen.
Die zweite Variante dagegen erzeugt zusätzlichen Programmieraufwand, da sie zunächst einmal die Extraktion des Interfaces und auch die Implementierung aller Methoden in der Mock- Klasse erfordert. Änderungen der Signatur ziehen dementsprechend auch Änderungen an (mindestens) drei unterschiedlichen Stellen nach sich: dem Interface selbst, der richtigen Implementierung und aller Mock-Klassen. Dennoch bevorzugen wir meist diese Variante, da das Interface zusätzlich eine dokumentierende Funktion ausübt und die zu betrachtende Komplexität spürbar verringert. Zudem kann der Aufwand zur Synchronisation zwischen Interface und Implementierung durch eine entsprechend ausgestattete Entwicklungsumgebung minimiert werden.

Konkretes Vorgehen

- Mock Objekt initialisieren
  - Zustand setzen
  - Verhalten parametrisieren
- Mock Objekt an getesteten Code übergeben
- Zustand des Mock Objekts verifizieren

z.B. Netzwerkkommunikation

- Client versendet Strings per HTTP
  - Netzwerkkommunikation über Schnittstelle
  - Mock-Implementierung für Tests
  - weitere nützliche Implementierungen

Beispiel

für generierte Mockobjekte für das Interface Foo:

```
public interface Foo {
  String doSomething( String something );
  double squareRoot( double input );
}
```

Hier die generierte Klasse:

```
public class MockFoo extends MockObject implements Foo {
  // Methoden aus dem Interface Foo
  public String doSomething( java.lang.String something )  {...}
  public double squareRoot( double input )  {...}

  // setup MockObject: Setzen der Erwartungen,
  // ggf. mit Rückgabewert oder zu werfender Exception.
  public void expectDoSomething( String something, String returnValue )  {...}
  public void expectDoSomething( String something, Throwable throwable )  {...}
  public void expectSquareRoot( double input, double returnValue ) {...}
  public void expectSquareRoot( double input, Throwable throwable )  {...}

  // setup MockObject: Angegebene Methoden bei Aufruf nicht beachten.
  // ggf. mit Rückgabewert oder zu werfender Exception.
  public void setDoSomethingDummy( Throwable throwable )  {...}
  public void setDoSomethingDummy( String returnValue )  {...}
  public void setSquareRootDummy( Throwable throwable )  {...}
  public void setSquareRootDummy( double returnValue )  {...}
}
```

Wichtige geerbte Methoden:

```
public class MockObject {
  public void startBlock()  {...}
  public void endBlock()  {...}
  public void verify()  {...}
  public Boolean checkDummy( Boolean flag, boolean value )  {...}
  [...]
}
```

Einfache Verwendung

Der Mock Creator erstellt MockObjects die das gewählte Interface implementieren. Im Test werden die Aufrufe die auf den Mockobjekten erwartet werden in folgender Form angegeben. Wir erwarten in diesem Fall, dass auf einem Objekt Foo die Methode doSomething mit dem Parameter "nobody expects" aufgerufen wird. Als Folge dieses Aufrufs gibt das MockObject das Ergebnis "the spanish inquisition" zurück. Wird hingegen squareRoot(...) oder doSomething mit einem anderen Parameter aufgerufen, so wirft MockFoo eine Exception.

```
// setup des MockObjects
MockFoo mockFoo = new MockFoo();
mockFoo.expectDoSomething( "nobody expects", "the spanish inquisition" );
// Testcode aus der Testklasse
assertEquals( "the spanish inquisition", mockFoo.doSomething( "nobodyExpects" ) );
// Verifizierung
mockFoo.verify();
```

Erweiterte Features

Durch die Verwendung von Mock Objekten in Tests sowie die einfache Verfügbarkeit von MockImplementationen ist es sehr einfach möglich, Tests zu schreiben, die eine Änderung des zu testenden Codes massiv erschweren - schließlich schreiben die Mockobjekte jeden einzelnen Aufruf genau vor. Hier gibt es mehrere Abhilfen... wir raten dazu, alle zu verwenden.

- MockObjects vorsichtig einsetzen
- Dummyaufrufe parametrisieren
- Blöcke von Code mit beliebiger Reihenfolge
- MockObjects vorsichtig einsetzen

Wahrscheinlich muss jeder einmal die Schwierigkeiten durchlaufen haben, Mock-getesteten Code umzuschreiben (refaktorisieren) um dieses Problem auch selbst zu sehen.

Dummyaufrufe parametrisieren

Die vom MockCreator generierten Klassen ermöglichen es, einige Aufrufe als "unerheblich für den Testverlauf" zu kennzeichnen - ein guter Kandidat hierfür sind beispielsweise einfache getXY-Methoden. Dadurch wird der tatsächliche Aufruf dieser Methoden nicht überwacht:

```
// setup für das MockObject
MockCustomer mockCustomer = new MockCustomer();
mockCustomer.setDummyGetName( "Otto Normalverbraucher" );
mockCustomer.expectGetBalance( new Money( 10 ) );

// Erzeugen der zu testenden Klasse und Test
```

```
ThisClassIsTested testee = new ThisClassIsTested( mockCustomer );
testee.thisMethodIsTested();

// Testcode aus der zu testenden Klasse
public class ThisClassIsTested {
  private Customer _customer;
  public ThisClassIsTested( Customer customer ) { _customer = customer; }
  public void thisMethodIsTested() {
    doSomething( _customer.getName() );
    doSomethingElse( _customer.getName() );
    Money balance = _customer.getBalance();
    System.out.println( "Balance for Customer " + _customer.getName()
      + ": " + balance + " Euro" );
  }
  [...]
}

// Verifizierung
mockCustomer.verify();
```

Das Standardverhalten der generierten MockObjects ist, dass die Reihenfolge der Methodenaufrufe beachtet wird. Es ist jedoch möglich, dieses Standardverhalten zu ändern - und zwar auch für einzelne "Methodenblöcke":

```
// setup für das MockObject
MockFoo mockFoo = new MockFoo();
mockFoo.expectDoSomething( "nobody expects", "the spanish inquisition" );
mockFoo.startBlock();
mockFoo.expectSquareRoot( 4, 2 );
mockFoo.expectSquareRoot( 9, 3 );
mockFoo.expectSquareRoot( 16, 4 );
mockFoo.endBlock();
// Testcode aus der Testklasse
// Testcode aus der zu testenden Klasse
public class ThisClassIsTested {
  private Foo _foo;
  public ThisClassIsTested( Foo foo ) { _foo = foo; };
  public void thisMethodIsTested() {
_foo.doSomething( "nobody expects" ) );
System.out.println( "sqrt(16) is " + _foo.squareRoot( 16 ) );
System.out.println( "sqrt(4) is "  + _foo.squareRoot( 4 ) );
System.out.println( "sqrt(9) is "  + _foo.squareRoot( 9 ) );
```

```
  }
  [...]
}
// Verifizierung
mockFoo.verify();
```

Generierte Objekte in Kürze

Jeder Aufruf einer expect-Methode (hier expectGet) speichert ein Objekt-Array mit Methodenname und Parametern in einer ExpectationList -> mockobjects und den Rückgabewert (hier "Eintrag 1") in eine Liste von Rückgabewerten.

Jeder Methodenaufruf vom zu testenden Objekt auf dem Mockobjekt vergleicht den Aufruf mit dem zuvor im Test als erwartet angegebenen Aufruf.

Um auch Fehlerfälle testen zu können generiert der MockCreator zu jeder Methode im Interface zwei expect-Methoden in der folgenden Form (hier für die Methode "methodCall"):

```
public void expectMethodName
               ( [alle Parameter]
                 [, Rückgabewert sofern
                         nicht void]
               );
public void expectMethodName
               ( [alle Parameter]
                 [, Throwable der bei Aufruf
                         geworfen wird]
               );
```

Mit Hilfe der zweiten Version ist es möglich, auch einen Fehlerfall im Test zu simulieren. Die Mock-Implementation gibt in diesem Fall bei Aufruf keinen Rückgabewert zurück, sondern löst die übergebene Exception aus. Hierbei ist die Schnittstelle zu beachten: Sofern nicht im Interface eine Exception deklariert ist, können hier nur RuntimeExceptions und Errors angegeben werden, da sonst während des Tests eine ClassCastException auftritt. Wenn das Interface Exceptions deklariert, können natürlich auch die deklarierten Exceptions als Werte für den Parameter Throwable verwendet werden.

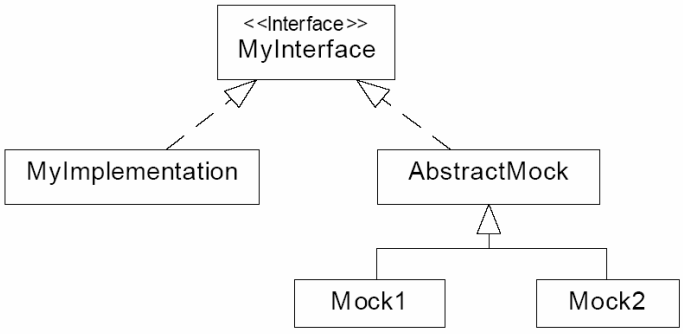
Wenn die Anzahl von Methodenaufrufen auf einem Objekt unerheblich ist, dient ein Aufruf der setDummyXY-Methoden dem Festlegen der entsprechenden Rückgabewerte bei Aufruf der Methode.

```
public void setDummyMethodCall( [Rückgabewert sofern nicht void] );
public void setDummyMethodCall( [Throwable, der bei Aufruf geworfen wird] );
```

Blöcke von Code, deren Aufrufreihenfolge unerheblich ist, lassen sich mit startBlock und endBlock festlegen.

```
public void startBlock();
public void endBlock();
```

Dieses UML-Diagramm soll den Vollausbau eines kleinen Musters zur Einführung von Mock-Objekten verdeutlichen. Die Idee dahinter ist, dass die Klasse AbstractMock für alle im Interface deklarierten Methoden eine NotImplementedException wirft. Konkrete Mock-Klassen leiten von ihr ab und überschreiben nur die interessanten Methoden. Gemeinsamkeiten der konkreten Mock Objekte lassen sich zudem nach oben in AbstractMock verschieben, um auch in den Tests Codeduplikation zu vermeiden.



Tricks zur Klassenbegrenzung

Durch die Einführung abstrakter Klassen diverser Unterklassen hat man auf Dauer mit einer sich stetig vermehrenden Zahl von Mock-Klassen zu kämpfen, von denen die meisten nur ein einziges Mal Verwendung fänden. Diese übermäßige Vermehrung von Klassen lässt sich in Java durch folgende Tricks vermeiden:

- Wird eine bestimmte Mock-Implementierung nur für einen einzelnen Test gebraucht, so erzeugt man sie als *anonyme Klasse* direkt in der Testmethode. Dieser Trick ist vor allem geeignet, um Methoden zu simulieren, die im Test feste Werte zurückgeben. Komplexere Validierungsfunktionen sind in anonymen Klassen nur durch leichte bis mittelschwere Verrenkungen zu erreichen, da Java diesen leichtgewichtigen Klassen einige Einschränkungen auferlegt.
- Wird eine bestimmte Mock-Implementierung nur innerhalb einer Testklasse benötigt, dann legt man sie als *innere Klasse* der Testklasse an. Es ist nichts Anderes als die Reduzierung der Sichtbarkeit der Mock-Klasse; Ob man eine Mock-Klasse im konkreten Fall als innere oder »normale« Klasse implementiert, hängt nicht zuletzt von der Unterstützung dieses Java-Features durch die verwendete Entwicklungsumgebung ab.
- Zu guter Letzt soll auch die Möglichkeit nicht unerwähnt bleiben, die Testklasse selbst als Mock-Objekt zu nutzen, indem man sie das entsprechende Interface selbst implementieren lässt. Dies ist eine leicht abgewandelte Form des Ansatzes mit der inneren Klasse, jedoch ohne die Möglichkeit, von einer bestehenden abstrakten Klasse zu erben.

Mock Objekte unter .NET

Testgetriebene Entwicklung und das damit zusammenhängende Unit-Testen gehören zu den wichtigsten Themen in der Entwicklerszene. Nicht zuletzt die unter Microsoft .Net verfügbaren Test-Frameworks wie das auf JUnit aus der Java-Welt basierende NUnit (www.nunit.org) oder csUnit (www.csunit.org) haben durch ihren professionellen Charakter dazu beigetragen.

Auch in der .Net-Welt kann man handgestrickte oder dynamische Darstellung von Mock-Objekten durchführen.

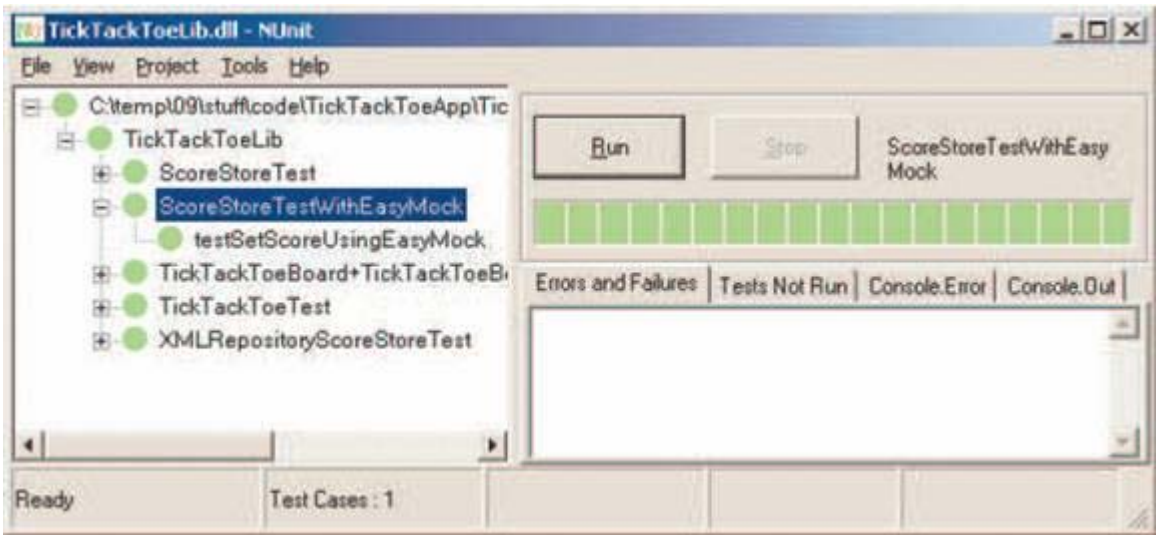
Aber, auch wenn das Erstellen von Mock-Objekten einfach ist und Erfolge schnell sichtbar werden, ergeben sich trotzdem kleinere Nachteile aus dem Einsatz manuell erstellter Mock-Objekte:

- Da sich die Mock-Objekte in separaten Klassen befinden, muss deren Code gelesen werden, um einen Unit-Test zu verstehen.
- Es kann mühsam sein, Mock Objekte zu erstellen. Unter Umständen schleichen sich Fehler ein.
- Wird eine Methode zu einem Interface hinzugefügt, so ist das Mock-Objekt ebenfalls um diese Methode zu ergänzen.
- Wird der Name einer Methode verändert, ist dies in allen Mock-Objekten zu berücksichtigen.
- Wird eine Methode aus einem Interface gelöscht, so ist diese in den Mock-Objekten ebenfalls zu löschen.

An dieser Stelle schafft das Open-Source-Produkt EasyMock Abhilfe. EasyMock ist ein in der Java-Welt bewährtes und einfach zu bedienendes Werkzeug für die dynamische Erzeugung von Mock-Objekten. Das Prinzip der Java-Lösung wurde mit dem Open-Source-Werkzeug EasyMock.NET auf die .NET-Welt übertragen.

EasyMock.NET

Das Tool EasyMock.NET verfolgt einen dynamischen und hochflexiblen Ansatz. Ein so genanntes Mock-Control erstellt dabei zur Testausführung ein Mock-Objekt. Nach dem Erzeugen befindet sich dieses Objekt im Setup-Modus. In diesem Modus werden die Erwartungen spezifiziert. Nach dem Wechsel in den aktiven Modus verhält sich das Mock Objekt wie zuvor programmtechnisch spezifiziert. Wird das Mock-Objekt in unerlaubter Art und Weise verwendet, wird umgehend ein Fehler signalisiert.
Anschließend wird die Funktion des echten Objekts ausgeführt und der Einsatz des Mock-Objekts über die Mock-Control verifiziert.



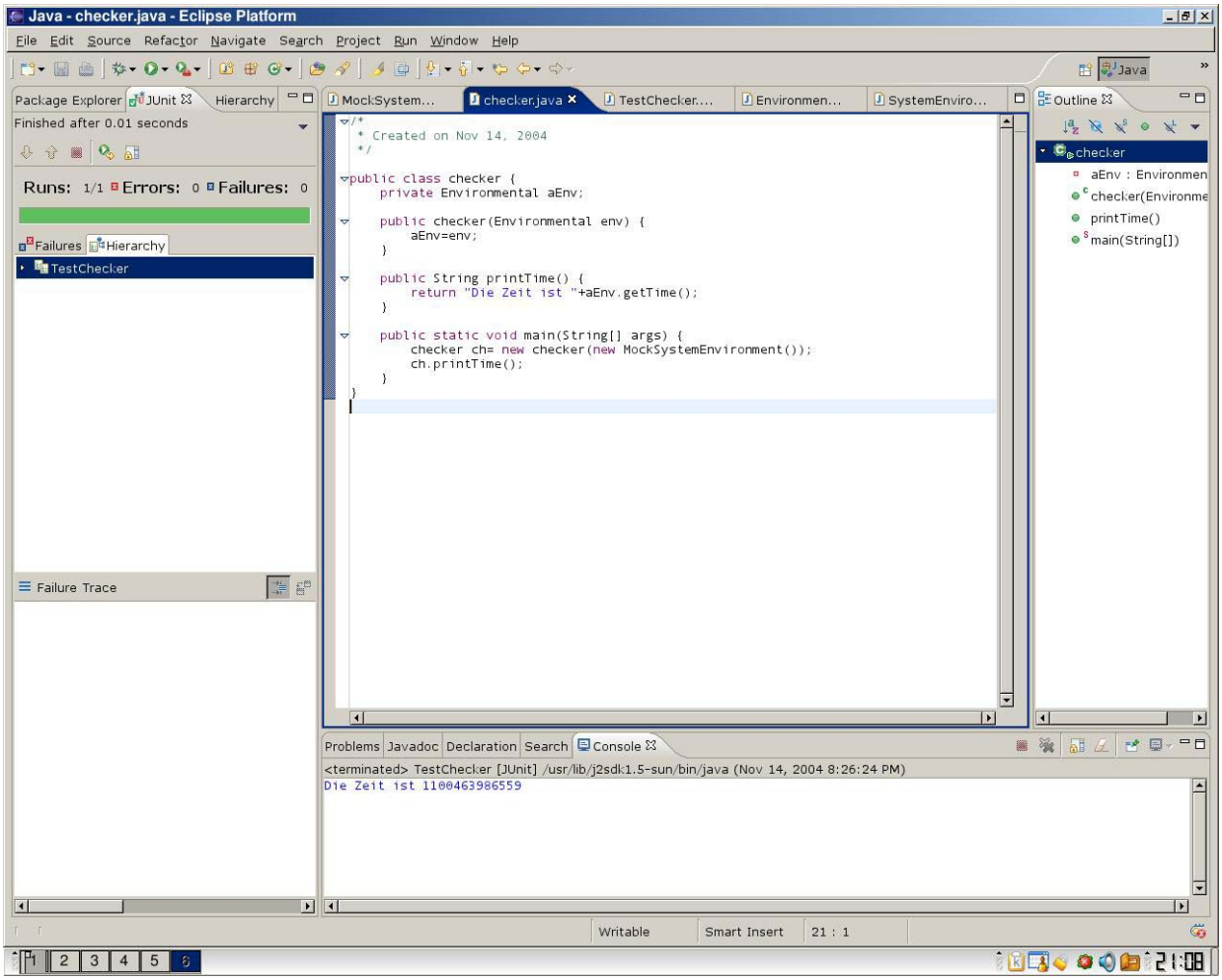
Mock Objekte in Java

Anhand eines einfachen Beispiels wollen wir aufzeigen wie man mit Eclipse und JUnit in Java einfach Mock - Objekte zum Testen verwenden kann. Wir implementieren dazu eine Klasse System Environment, die das dahinterliegende System widerspiegeln soll. Mit Hilfe dieser Klasse kann man auf die aktuelle Zeit zugreifen. Man sollte sich allerdings vorstellen, dass in der Realität für wesentlich komplexere Schnittstellen und Klassen Mock Objekte erzeugt werden.

```
/*
 * Für das Interface Environmental werden wir unsere Mock - Implementierung erzeugen.
 */
public class SystemEnvironment implements Environmental{
        public long getTime() {
        /*
         * Hier könnte eine wesentlich kompliziertere Implementierung liegen.
         * Wichtig anzumerken ist dabei aber auch, dass wir auf die aktuelle Systemzeit
         * normalerweise keinen Einfluss haben.
         */
                return System.currentTimeMillis();
        }
}
/*
 * Das dazugehörige Interface sieht folgendermaßen aus. Es könnten hier auch mehrere
 * Methoden stehen.
 */
public interface Environmental {
        public long getTime();
```

```
}

/*
 * Nun wollen wir ein Mock Objekt dazu erstellen. Also ein Objekt das dieselbe Schnittstelle
 * implementiert, das aber den Vorteil hat, dass wir die Reaktionen für unsere Testfälle
 * steuern können. Zu diesem Zweck haben wir die Methode setTime hinzugefügt. In diesem
 * einfachen Objekt verzichten wir darauf, auch noch mitschreiben zu lassen welche
 * Methoden aufgerufen werden.
 */
public class MockSystemEnvironment implements Environmental{
        private long current_time=0;

        public long getTime() {
                return current_time;
        }

        /*
         * Mit Hilfe dieser Methode können wir jede erdenkliche Zeit simulieren.
         */
        public void setTime(long aTime) {
                current_time=aTime;
        }
}
/*
 * Werfen wir nun einen Blick auf das Objekt, das auf die Implementierung zugreift.
 * Der Code kann unmodifiziert bleiben. Hier ist kein Hinweis darauf zu finden, dass
 * wir die implementierte Klasse SystemEnvironment, die auf dem Interface
 * Environmental basiert, mit Mock Objekten testen.
 */
public class checker {
        private Environmental aEnv;
        public checker(Environmental env) {
                aEnv=env;
        }
        public String printTime() {
                return "Die Zeit ist "+aEnv.getTime();
        }
        public static void main(String[] args) {
                checker ch= new checker(new SystemEnvironment());
                ch.printTime();
        }
}
```

```
/*
 * Der größte Vorteil ergibt sich erst wenn wir zum eigentlichen Testen schreiten.
 * Deshalb werfen wir nun einen Blick auf das Testen mittels JUnit - TestCases.
 * Um verschiedene Zeiten zu Testen muss man nun nicht mehr selbst die Systemzeit
 * händisch umstellen, sondern kann sich getrost auf JUnit verlassen.
 * Die Zeit wird also ohne Zeitverlust als geändert vorgetäuscht.
 */
public class TestChecker extends TestCase {
        public TestChecker() {}

        public void testSomeTimes() {
                MockSystemEnvironment mse= new MockSystemEnvironment();
                long someTime =System.currentTimeMillis();

        // einige Beispieltestfall
                mse.setTime(someTime);
                checker ch= new checker(mse);
                assertEquals("Teste die Ausgabe von printTime",
                        ch.printTime(), "Die Zeit ist "+someTime);

                someTime=1;
                checker ch= new checker(mse);
                assertEquals("Teste die Ausgabe von printTime",
                        ch.printTime(), "Die Zeit ist 1");
        // weitere Testfälle
        }
}
```



Weiterte Anwendungsgebiete:

- Möchte man die genaue *Reihenfolge von empfangenen Nachrichten* verifizieren, dann bietet sich auch hier die Umwandlung der Nachrichten im Mock-Objekt zu Strings an. Unter Umständen erfordert dieses Vorgehen jedoch häufige Anpassungen im Zuge von Methoden - Umbenennungen.
- Die Überprüfung einer Nachrichten- oder Event-Sequenz über mehrere Clients (und damit Mock-Objekte) hinweg, lässt sich bewerkstelligen, indem man allen betroffenen Mock-Objekten einen *Nachrichtenregistrator* mitgibt. Dieser Registrator spielt die Rolle des eigentlichen Mocks und vergleicht die erwartete mit der tatsächlichen Nachrichtenfolge.

Das wirklich Bemerkenswerte an Mock-Objekten ist, dass sie zu guten objekt-orientierten Entwürfen zwingen, die auf lose gekoppelten über Schnittstellen interagierenden Objekten beruhen. Diese Objekte sagen einander, was zu tun ist, anstatt sich gegenseitig Löcher in den Bauch zu Fragen, um dann Dinge zu tun, die das gefragte Objekt besser selbst gemacht hätte.

Ausgewählte Tools

Tool: EasyMock
Sprache: Java, .NET
Lizenz: MIT license
Homepage: http://www.easymock.org/
EasyMock stellt Mock-Objekte für Schnittstellen in den JUnit Tests zur Verfügung, die „on the fly“ durch Java Proxy Mechanismen erzeugt werden. Wegen EasyMocks einzigartiger Art der Aufnahme von Erwartungen beeinflussen die meisten „Refactoring“s die Mock Objekte nicht. Darum ist EasyMock ein perfektes Werkzeug für Test-Driven Development. EasyMock ist ein OpenSource Projekt das unter der MIT Lizenz zur Verfügung steht.

Tool: JMock
Sprache: Java
Lizenz: jMock.org
Homepage: http://www.jmock.org
Mock Objekte sind leicht zu Erstellen und zu Ändern. Schnell und ohne Unterbrechung der eigentlichen Implementierungsphase kann man flexible Constraints für Interaktion zwischen Objekten definieren. Es ist auch sehr leicht erweiterbar.

Tool: MockCreator
Sprache: Java
Lizenz:
Homepage: http://mockcreator.sourceforge.net/
dieses Tools bietet ein Eclipse - Plugin http://mockcreator.sourceforge.net/eclipse/

Tool: MockMaker
Sprache: MIT license
Lizenz: GPL/LGPL
Homepage: http://www.mockmaker.org
bietet Plugins für Eclipse, VisualAge und JBuilder

Tool: VBMock
Sprache: Visual Basic
Lizenz: GPL/LGPL
Homepage: http://vbmock.sourceforge.net

Pro und Contra

Die Verwendung von Dummy- und Mock-Objekten ist sowohl in der Gemeinde der Softwaretester als auch in der XP-Welt nicht unumstritten. Tragen wir die Argumente beider Seiten noch mal zusammen.

Vorteile

- Wir können mit einer feineren Granularität und größeren Genauigkeit testen. Dies zeigt sich u.a. darin, dass wir eine *Test-Failure* immer auf einen Fehler im Testobjekt oder den Test selbst zurückführen können und nicht in Schichten des Programms wühlen müssen, die augenblicklich nicht von Interesse sind.
- Testeigene Mock-Objekte stellen die Wiederholbarkeit unserer Tests sicher. Echte Serverobjekte ändern durch einen Test möglicherweise ihren Zustand und müssten nach dem Test wieder zurückgesetzt werden. Dies ist im besten Fall zusätzlicher Aufwand, unter Umständen sogar völlig unmöglich.
- Die Verwendung echter Serverobjekte im Test stellt eine Art von *Mikrointegrationstest* dar. *Integrationstests* überlappen jedoch stark mit den *Funktionstests* und sind nur in besonderen Fällen Teil der *Entwicklertests*. Die Erfahrung zeigt auch, dass Tests, die Objekte vieler Schichten integrieren, auf Dauer zu langsam für ein schnelles und ständiges *Feedback* werden.
- Herkömmliche Tests sind auf das Überprüfen von Rückgabewerten und nach außen sichtbaren Zustandsänderungen des Testobjekts angewiesen. Mock-Objekte erlauben es uns, zu überprüfen, ob der Zugriff des Testobjekts auf seine Helfer- und Serverobjekte richtig ist. Wir testen sozusagen von innen.
- Mock-Objekte dienen als Behälter, in welchem wir sich wiederholende Testfunktionalität sammeln können. Sie erleichtern das *Refactoring* von Testcode und stellen ein Muster dar, das die Kommunikation des Codes verbessert.

Nachteile

Alle genannten Punkte haben entweder mit der Erhöhung der Unabhängigkeit (von Tests und Code) oder der Verbesserung der Kommunikation zu tun. All diesen Vorteilen stehen auch Nachteile gegenüber:

- Mock-Klassen können Fehler enthalten. Dieses Problem wird jedoch dadurch relativiert, dass die Wahrscheinlichkeit, dass sich Fehler in der Dummy-Klasse und in der Testklasse gegenseitig aufheben, sehr klein ist. Fehler der Dummy-Klasse werden daher meist sofort entdeckt.
- Mock-basiertes Testen findet keine Fehler, die sich aus dem Zusammenspiel mehrerer Komponenten ergeben. Diese Fehlerkategorie decken wir einfacher durch Funktionstests ab. Werden solche Fehler dennoch zu einem häufigen Problem, sollte man über zusätzliche lokale Integrationstests an den kritischen Stellen des Systems nachdenken, die idealerweise aber nur zwei aneinander grenzende Schichten integrieren.

- Änderungen am Interface der echten Implementierung erfordern Änderungen am Dummy-Objekt. Dieser zusätzliche Aufwand macht erfahrungsgemäß jedoch nur einen kleinen Teil des Gesamtaufwands zur Aktualisierung aller Tests aus. Häufig hilft auch die IDE beim Finden der zu ändernden bzw. zu ergänzenden Signaturen.
- Das Testen mit Attrappen muss von den Entwicklern erlernt werden. Mit der Zeit wächst jedoch nicht nur die Erfahrung, sondern auch die Bibliothek an wiederverwendbaren Dummy- und Mock- Objekten.
- »Testen von Innen« erfordert, dass man weiß, was in der Klasse geschieht bzw. geschehen soll. Ändert sich die Implementierung, z.B. weil man bessere Wege gefunden hat, mit einem Serverobjekt zu arbeiten, muss häufig auch der Testcode (inklusive Mock- Objekten) geändert werden, obwohl das Testobjekt nach außen ein unverändertes Verhalten zeigt. Mock-Objekte werden daher bevorzugt für das Testen relativ stabiler Implementierungen verwendet.

Literaturverzeichnis:

(1) Mackinnon, T., Freeman, S., Craig, P. Endo-Testing: Unit Testing with Mock Object. *Proceedings XP2000*.

(2) Matthew A. Brown, Eli Tapolcsanyi, Mock Object Patterns, 5/19/2003, PDF

(3) Andy Hunt and Dave Thomas, Pragmatic Unit Testing, Sept. 2003, PDF

(4) Dave Thomas and Andy Hunt, Mock Objects, IEEE Software, May/June 2002

(5) Shaun Smith, Jennitta Andrea, Gerard Meszaros, Test-driven development with Mock J2EE, JMS, and JNDI, PDF

(6) Venkat Subramaniam, Test Driven Development – Part II: Mock Objects, http://www.agiledeveloper.com/download.aspx, PDF

(7) Asim Jalis, Lance Kind, Automatically Generating System Mock Objects, PDF

(8) Stefan Schwarzer, Unit testing with mock code, EuroPython 2004, PDF

(9) Joe Walnes, Tim Mackinnon, Mock Objects: Driving object design top-down, PDF

(10) Tim Mackinnon (Connextra), Steve Freeman (BBST), Philip Craig (Independent), **Endo-Testing: Unit Testing with Mock Objects**, XP eXamined, Addison-Wesley, PDF

(11) http://www.mockobjects.com/HowMockObjectsHappened.html

(12) http://www-106.ibm.com/developerworks/library/j-mocktest.html

(13) http://msdn.microsoft.com/msdnmag/issues/04/10/NMock/default.aspx

(14) http://javaboutique.internet.com/tutorials/mock_objects