Seminar Simulation und Bildanalyse mit Java

Testen objektorientieter Software mit JUnit

Helge Scheffler

1 – Motivation
2 – Stubs
3 – Mock-Objekte
4 – Vorteile und Probleme von Stubs und Mock-Objekten
5 – Vererbung

1 Motivation

- Probleme beim Testen von Klassen
  - Klassen sind meistens von vielen anderen Klassen abhängig
  - Klassen sind von externen Quellen abhängig
- das erfordert aufwändige Tests oder Tests von Gruppen von Klassen
- der Testaufwand steigt überproportional
- Tests werden langsam, löchrig und komplex
- Lösung: benutze Stubs oder Mock-Objekte, um das zu testende Objekt von Abhängigkeiten zu entkoppeln

2 Stubs

Beispiel:

Wie teste ich ein Währungsrechner, US Dollar in Euro, der sich den Wechselkurs von einem Server holt?

```
public class EuroCalculator  {

   public double valueInEuro(double amount, String fromCurrency)  {
      double euroValue;
      String euro = "EUR";
      ExchangeRateProvider rate = new ExchangeRateProvider();
      euroValue = amount * rate.getRateFromTo(fromCurrency, euro);
      return euroValue;
   }
}

public class ExchangeRateProvider  {
   public double getRateFromTo(String fromCurrency, String to)  {
       double retrievedRate = ... // Netzzugriff auf Server
       return retrievedRate;
   }
}
```

```
public interface IRateProvider  {
   public double getRateFromTo(String fromCurrency, String to);
}

public class EuroCalculator  {
   private IRateProvider rateProvider;

   public EuroCalculator(IRateProvider rateProvider)  {
      this.rateProvider = rateProvider;
   }
   [...]
}
```

```
public class StubProvider implements IRateProvider  {
   private double stubRate;
   public StubProvider(double stubRate)  {
       this.stubRate = stubRate;
   }
   public double getRateFromTo(String from, String to)  {
       return stubRate;
   }
}
```

```
public class EuroCalculatorTest extends TestCase  {
   public EuroCalculatorTest(String name)   {
       super(name);
   }
   public void testUSD2EUR()  {
       double result =
       new EuroCalculator(new StubProvider(1.1934)).valueInEuro(1.0, "USD");

       assertEquals(1.1934, result, 0.00001);
   }
}
```

- Stubs ersetzen echte Implementierung für den Test und liefern geeignete Werte
- sie ersetzen die Klassen, die wir nicht mittesten wollen / können
- Beispiele:
  - Server:
    - lange Antworten
    - Überlastung
  - Dateizugriff
  - Datenbankzugriff

- Wannn werden Stubs eingesetzt?
  - Zugriff auf externe sich ändernde Resourcen
  - Komponenten mit Seiteneffekten (Wiederholbarkeit des Tests gefährdet)
  - Objekte in Fehler-/Ausnahmezustand (Server reagiert nicht)
  - langsame Objekte (Datenbanken)
  - Klassen mit vielen Abhängigkeiten, die Methoden aus mehreren anderen Klassen aufrufen
  - Komponenten deren Code und Methodenaufrufe ständig geändert werden
  - noch nicht vorhandene Klassen

- wenn der Test fehlschlägt, wissen wir wo: im zu testenden Objekt, nicht Extern
- nebenbei wird auch eine Designverbesserung erreicht

3 Mock-Objekte

Beispiel: Programm eines Videoverleihs, das unter anderem einen Beleg mit dem Filmnamen druckt

```
public class Customer ...
   public void printStatementDetail(IPrinter printer)
      throws OutOfPaperException {
      for(Iterator iterator = rentals.iterator(); iterator.hasNext(); )  {
         Rental rental = (Rental) iterator.next();
         printer.print(rental.getMovieTitle());
      }
         printer.cutPaper();
   }
}

public interface IPrinter  {
    public void print(String line) throws OutOfPaperException;
    public void cutPaper() throws OutOfPaperException;
}
```

```
import com.mockobjects.*;
public class MockPrinter extends MockObject implements IPrinter  {
    private ExpectationList printerOutput
      = new ExpectationList("printer output");
    private ExpectationCounter cutPaperCalls
      = new ExpectationCounter("cutPaper() calls");

    void addExpectedOutput(String output)  {
        printerOutput.addExpected(output);
    }
    void addExpectedCutPaperCalls(int numberOfCalls)  {
        cutPaperCalls.setExpected(numberOfCalls);
    }

    void verify()  {
        Verifier.verify(this);
    }
```

```
    public void print(String output) throws OutOfPaperException  {
        printerOutput.addActual(output);
    }
    public void cutPaper() throws OutOfPaperException  {
        cutPaperCalls.inc();
    }
}
```

```
public class CustomerTest ...
    MockPrinter mockPrinter;

    protected void setUp()...
        mockPrinter = new MockPrinter();
    }
    public void testStatementDetailForRentalLines()
    throws Exception  {
        mockPrinter.addExpetedOutput("Buffalo 66");
        mockPrinter.setExpectedCutPaperCalls(1);

        customer.rentMovie(buffalo66);
        customer.printStatementDetail(mockPrinter);

        mockPrinter.verify();
    }
}
```

- Mock-Objekte sind Stubs mit eingebetteter Testfunktionalität
  - Verifizieren von korrektem Verhalten
  - Mock wird von aussen kontrolliert
  - Einstellung der gewünschten Reaktionen
  - Mocks sollen dumm sein und so einfach, dass sie nicht getestet werden müssen
  - man kann kleine, fokussierte Tests schreiben, um präzise Fehlermeldungen zu erhalten

- Testen von Innen
  - kontrollieren, ob die gewünschten Methodenaufrufe erfolgen
  - kontrollieren, ob die Methodenaufrufe der Testobjekte mit den richtigen Werten geschehen
  - Fehler werden besser / früher erkennbar

- Muster:
  - erzeuge Mocks
  - setze internen Zustand der Mocks
  - setze Erwartungen
  - Aufrufen des zu testenden Codes mit Mock als Parameter
  - Verifizieren des Verhaltens

- korrektes Verhalten in Ausnahmesituationen testen
  - zu testendes Objekt muss eine Exception auffangen
  - Mock implementieren, das eine Exception wirft
    - zum Beispiel die Datenbank reißt ab

- Mockimplementierung
  - Mock Objekte werden beispielsweise über ein Interface von dem Testobjekt verwendet
  - Mocks aus Bibliotheken
    * www.mockobjects.com
    * man kann Expectation Classes verwenden
    * diese Klassen trennen das Mock Objekt inhaltlich in die Bereiche
      · Stub Teil
      · erwartetes Verhalten
      · tatsächliches Verhalten
      · Verifizierung

- EasyMocks (www.easymock.org)
  - das Mock Objekt muss nicht mehr selbst implementiert werden
  - das Verhalten des Mock Objektes wird in der Testklasse angegeben
  - das MockControl Objekt übernimmt das Verifizieren
  - man kann bestimmen wie streng das Verhalten des Testobjektes geprüft werden soll
    * richtige Reihenfolge oder beliebige Reihenfolge
    * exakte oder beliebige Anzahl von Aufrufen einer Funktion
    * unerwartete Aufrufe sind möglich

- Testablauf bei EasyMocks
  - das erwartete Verhalten wird definiert
  - die Mock Funktionalitäten werden aktiviert
  - Prüfung, ob erwartetes Verhalten eingetreten ist

```
public class Option{
   private IValue computevalue;
   private IPrice computeprice;
   ...

   public Option(String name, double K, double S, double r, double v,
                 double T, IValue computevalue, IPrice computeprice)
   {
       this.computevalue = computevalue;
       this.computeprice = computeprice;
        ...
   }
     ...

   public double get_value()  {
       this.value = computevalue.get_value(this);
       return this.value;
```

```
    }

    public double get_price()  {
        double value = get_value();
        this.price = computeprice.getPrice(value);
        return this.price;
    }
}
```

```
import junit.framework.*;
import org.easymock.MockControl;

public class Option1Test extends TestCase {

   private Option testOption;
   private MockControl controlValue, controlPrice;
   private IValue mockValue;
   private IPrice mockPrice;

   protected void setUp()  {
       controlValue = MockControl.createControl(IValue.class);
       mockValue = (IValue) control.getMock();
       controlPrice = MockControl.createControl(IPrice.class);
       mockPrice = (IPrice) control.getMock();
       testOption = new Option("test", 40, 42, 0.1, 0.2, 0.5, mockValue,
 mockPrice);
```

```
    }
    public void testGetPrice()  {
        mockValue.get_value(testOption);
        controlValue.setReturnValue(5.0);

        mockPrice.getPrice(5.0);
        controlPrice.setReturnValue(5.5);

        controlValue.replay();
        controlPrice.replay();

        assertEquals(testOption.get_price(), 5.5, 0.001);

        controlPrice.verify();
        controlValue.verify();
     }
}
```

4 Vorteile und Probleme von Stubs und Mock-Objekten

- Vorteile von Stubs
  - Fehler sind auf Testobjekt oder Test zurückzuführen
  - Wiederholbarkeit des Tests
  - Verhalten des Testobjekts an den Rändern des Wertebereichs
  - Testen von Exceptions

- zusätzliche Vorteile von Mock Objekten
  - Testen von Innen
    - nicht bloß Überprüfen der Rückgabewerte
    - sondern ob das Testobjekt auf andere Objekte richtig zugreift

- Probleme
  - Änderungen des Interfaces erfordern Änderungen im Stub / Mock-Objekt
  - man findet keine Fehler, die aus Zusammenspiel mehrerer Komponenten entstehen
    * folglich braucht man Interaktionstests

- Falsche Verwendung von Stubs und Mock-Objekten
  - Mocks sind so komplex, dass sie getestet werden müssten
  - als Faustregel (laut Frank Westphal) gilt:
    * sie rufen weitere Stubs / Mock-Objekte auf
    * man braucht mehr als drei Mock-Objekte für ein Testobjekt

5 Vererbung

- Unterklassen erben auch unerwünschte Eigenschaften der Oberklasse
- dadurch wird es schwieriger, sie zu testen
- Ersetzungsprinzip [Liskov]: Objekt des Untertyps muss Objekt des Obertyps stets ersetzen können

```
public class Rectangle  {
    private int x;
    private int y;
    public Rectangle(int x, int y)  {...}
    public int getX()  {...}
    public int getY()  {...}
    public int stretchX(int factor) {
        x = x * factor;
        return x;
    }
}

public class Square extends Rectangle  {
    public Square(int x)  {
        super(x, x);
    }
}
```

- Teile der Testsuite der Oberklasse können zum Testen der Unterklasse verwendet werden, falls das Ersetzungsprinzip erfüllt ist
- Vermutung bei Tests:
  - unveränderte Methoden müssen nicht mehr getestet werden
  - überschriebene Methoden können mit Tests der Oberklasse adäquat getestet werden
- beide Vermutungen sind meistens nicht richtig

Literatur

[1] J. Link: Unit Tests mit Java, dpunkt.verlag, 2002

[2] V. Massol, T. Husted: JUnit in Action, Manning Verlag, 2003

[3] F. Westphal: Testgetriebene Entwicklung mit JUnit und FIT.
http://www.frankwestphal.de/TestgetriebeneEntwiclungmitJUnitundFIT.html

[4] MockObjects Homepage: http://www.mockobjects.com

[5] EasyMocks Homepage: http://www.easymock.org

[6] B. Liskov: Data Abstraction and Hierarchy, SIGPLAN Notices. 23(5), 1988