JUnit

1 Was ist JUnit? (... und wieso brauchen wir es?)

Vermutlich kennt jeder das Problem, ein Programm schreiben zu müssen und sich dabei im ‚Blindflug'zu befinden. Wir wissen was wir wollen, aber nicht unbedingt, wie weit unsere Bemühungen gediehen sind.
Es ist in diesen Situationen wünschenswert zu wissen, ob die Methoden einer Klasse, das Richtige tun, anstatt warten zu müssen, bis das Programm lauffähig ist, um dann suchen zu müssen, welche Methode(n) dafür verantwortlich sind, daß das Program *nicht* richtig funktioniert.

An diesem Punkt setzt JUnit an: Junit ist ein Java-package, das dazu dient, die Methoden von Klassen einfach zu testen.
Statt warten zu müssen, bis man das Programm ausführen kann, testet man mit JUnit, ob die Methoden das Richtige tun. Wenn ein Test nicht funktioniert, weiß man welche Methode überarbeitet werden muß. Diese Tests können *einfach* wiederholt werden, was einen großen Vorteil birgt, wenn Methoden überarbeitet wurden. Statt zu schauen, ob alle Methoden mit dieser Überarbeitung zurechtkommen (von Hand?) startet man ein Testprogramm. Ein weiterer Vorteil ist, daß die Tests nicht den Sourcecode verschandeln, da sie in eigenen Klassen untergebracht werden können.

2 Und wie benutze ich JUnit?

2.1 Vorbereitung

1. junit3.7.zip entpacken
2. Die Variable CLASSPATH muß gesetzt werden. z.B.:

   ```
   setenv CLASSPATH $CLASSPATH:~/swp/junit3.7/junit.jar:.
   ```

3. Starten einer TestRunner Klasse z.B.:

   ```
   java junit.swingui.TestRunner
   ```

4. Angabe der gewünschten Testklasse.
   (Man kann auch beim Aufruf von TestRunner die Testklasse als Argument angeben.)

2.2 Testklassen schreiben

Um eine Testklasse zu schreiben, dürfte es wohl am einfachsten sein, sich an einem Beispiel zu orientieren.
Im folgenden wird vorausgesetzt, daß es Klassen *Feld* und *Ritter* gibt, die eine equals Methode implementieren. Anhand der zugehörigen Testklassen werden die wichtigsten Dinge erläutert:

```
public class FeldTest extends TestCase{

   public FeldTest(String s){super(s);}

   public static Test suite(){
      TestSuite suite=new TestSuite();
      suite.addTest(new FeldTest("testEquals"));
      return suite;
   }

   public void testEquals(){
      Feld f = new Feld(3);
      Feld g = new Feld(3);
      Feld h = new Feld(2);
      Assert.assertTrue(!f.equals(null));
      Assert.assertEquals(f,f);
      Assert.assertTrue(f.equals(g));
      Assert.assertTrue(!f.equals(h));
   }
}
```

Um auch Methoden testen zu können, auf die nur innerhalb eines Packages zugegriffen werden kann, ist es sinnvoll, wenn sich die Testklasse und die zu testende Klasse im gleichen Package befinden.
Soll mehr als eine Methode getestet werden - und werden dafür die gleichen Objekte mehrfach verwendet - so ist es sinnvoll diese Objekte für jeden Test neu zu erzeugen. Dafür gibt es die Methoden setUp() und TearDown().

```
public class  RitterTest extends TestCase{

   private Ritter a;
   private Ritter b;
   private Ritter c;

   public RitterTest(String s){super(s);}

   protected void setUp(){
```

```
      a = new Ritter(2);
      b = new Ritter(3);
      c = new Ritter(2);
   }

   public static Test suite(){
      // return new TestSuite(KnightTest.class);
      TestSuite suite = new TestSuite();
      suite.addTest (new RitterTest("testEquals"));
      suite.addTest (new RitterTest("testIt"));
      return suite;
   }

   public void testEquals(){
      Assert.assertTrue(!a.equals(null));
      Assert.assertEquals(a,a);
      Assert.assertEquals(a,c);
      Assert.assertTrue(!a.equals(b));
   }

   public void testIt(){
      Assert.assertTrue(a.it());
      Assert.assertTrue(c.it());
   }

   public void tearDown(){
      // hier werden offene Files geschlossen,
      // Threads beendet,...
      // sofern sie in setUp() initialisiert wurden
   }

}
```

Die auskommentierte Zeile der Methode suite() kann alternativ benutzt werden. Dann werden alle testXXX Methoden durchgeführt.
Jedesmal wenn ein Test der Klasse durchgeführt wird, wird die Methode setUp() von JUnit ausgeführt, um sicherzustellen, da sich die Objekte, mit denen gearbeitet wird (‚Fixture ‘), in einem genau definierten Zustand befinden.

```
public class AllTests extends TestCase{

   public AllTests(String s){super(s);}
```

```
   public static Test suite(){
      TestSuite suite= new TestSuite();
      suite.addTest(RitterTest.suite());
      suite.addTest(FeldTest.suite());
      return suite;
   }
}
```

2.3 Testen in der Realität

2.3.1 Wann sollte man testen?

Optimal ist es, wenn von Anfang an Tests geschrieben werden. Einmal täglich sollten alle Tests *erfolgreich* durchlaufen. Auch ist es praktisch einen Test zusammen mit der zu testenden Methode zu schreiben, weil man dann am besten weiß, was die Methode leisten soll (und was nicht).

2.3.2 Was sollte man testen?

Hier gilt: nach Möglichkeit alles, was überhaupt einen Fehler produzieren kann. Natürlich ist es nicht sinnvoll, Tests zu schreiben, die in jedem Fall erfolgreich laufen. Vielmehr sollten Tests die Möglichkeit bieten, zu prüfen, ob etwas (unerwarteter weise) bereits läuft, oder sich zu vergewissern, daß alles wie erwartet läuft.

2.3.3 Wenn Tests nicht (mehr) laufen

Wer irgendeinen Fehler einbaut, muß den Fehler auch beheben, natürlich kann man sich ggf. Hilfe holen, aber die Verantwortung liegt beim Verursacher!

2.3.4 Wiederverwertung von Tests

Wie bereits oben erwähnt, sollen die Tests sicherstellen, daß noch alles funktioniert, wie es soll. Wenn also eine Methode geändert wird - etwa clone(), weil es eine zusätzliche Instanzvariable in der Klasse gibt - hat man eine einfache Möglichkeit zu testen, ob sich alle Methoden mit den Änderungen vertragen (z.B. equals()).