Systematisches Testen

Unit Testing

Objektorientierte Entwicklung

Entwicklung von vielen unabhängigen Einheiten (Klassen, Methoden), aus denen das Gesamtprogramm zusammengesetzt wird.

Ziel: Wenn sie nur alle Einheiten korrekt verhalten, dann verhält sich das Programm wie gewünscht.

Unit Testing

- systematisches Testen der einzelnen Einheiten
- zwingt Entwickler zu planen und über das gewünschte Verhalten der Programmeinheiten nachzudenken
- unterstützt Design by Contract: teste stichprobenartig, dass alle Klassen ihre Verträge einhalten

Unit Testing mit JUnit

Was wird getestet

- public/protected-Methoden
- private-Methoden können mit `assert` getestet werden.

Wie wird getestet

- Tests sind vom Programm separate Klassen (man kann Tests als Teil der Dokumentation auffassen).
- Darin wird eine beliebige Anzahl von Testfunktionen implementiert.
- Testfunktionen mit Java-Annotationen gekennzeichnet.
- Die Tests werden dann durch ein bestimmtes Programm ausgeführt und die Ergebnisse werden ausgewertet (üblicherweise durch Tools wie Eclipse oder NetBeans).

Java-Annotationen

Mit Annotationen kann man Java-Programme mit nützlichen Zusatzinformationen versehen werden.

In Java sind sieben Arten von Annotationen eingebaut, z.B. `@Deprecated`, `@Override`, `@SuppressWarnings`

Man kann eigene Annotationen definieren.

In JUnit 4 sind zum Beispiel `@Test`, `@Before`, `@After`, `@BeforeClass`, `@AfterClass` und `@Ignore` definiert.

Annotationen – Beispiel

Kennzeichnung von Klassen/Methoden, die nicht mehr benutzt werden sollen und nur aus Kompatibilitätsgründen noch vorhanden sind.

```
@Deprecated
public class A {
   ...
}
```

Compiler erzeugt Warnung, wenn die Klasse `A` benutzt wird.

Annotationen – Beispiel

Kennzeichne Funktionen, die eine bereits existierende Funktion in einer Oberklasse überschreiben mit `@Override`.

```
public class A {
  public void f(List l) {
    ...
  }
}

public class B extends A {
  @Override
  public void f(List l) {
    ...
  }
}
```

Annotationen – Beispiel

`@Override` hilft häufige Fehler zu finden:

```
public class A {
  public void f(List l) {
    ...
  }
  private void g() {
    ...
  }
}
```

```
public class B extends A {
  @Override
  public void f(LinkedList l) {
      ...
  }
  @Override
  public void g() {
      ...
  }
}
```

Compilerfehler: Überladen statt Überschreiben.

JUnit 4

Tests sind in separat vom eigentlichen Programm in eigenen Klassen implementiert (z.B. eine Testklasse für jede zu testende Klasse)

Testklasse kann eine beliebige Anzahl von Tests implementieren.

Einzelne Tests werden durch Methoden in der Testklasse implementiert:

- durch Annotation `@Test` als Test gekennzeichnet
- beliebiger Methodenname
- `public` und ohne Argumente

Die Tests in einer Testklasse werden automatisiert ausgeführt und ein Gesamtergebnis angezeigt.

Testfunktionen

Allgemeine Funktion von Testfunktionen

1. Konstruktion der zu testenden Situation
   - Objekte der zu testenden Klassen erzeugen
   - Objekte in den gewünschten Zustand versetzen
2. Überprüfung des erwarteten Ergebnis
   - Funktionen `assertTrue(test)`, `assertEquals(x, y)`, `...` in `org.junit.Assert`.
   - Ist das erwartete Ergebnis eine Exception, kann man das in der Annotation angeben:
     `@Test(expected = ExceptionName.class)`

JUnit 4 – Beispiel

```
import org.junit.*;
import static org.junit.Assert.assertEquals;

public class BoardTest {
  @Test
  public void testFood1() {
    BoardServerToClient b = new BoardServerToClient(100, 100);
    int x = 94, y = 23, a = 34;
    b.setFood(x, y, a);
    assertEquals(b.getFood(x,y), a);
  }

  @Test(expected = java.lang.IndexOutOfBoundsException.class)
  public void testFood2() {
    BoardServerToClient b = new BoardServerToClient(100, 100);
    int x = 100, y = 23, a = 34;
    b.setFood(x, y, a);
  }
}
```

JUnit 4 – Ausführen der Tests

Eclipse, NetBeans, ... bieten bequeme Möglichkeiten, Tests ablaufen zu lassen und Ergebnisse anzuzeigen.

Tests direkt ausführen:

- `junit.jar` von `www.junit.org` beziehen und zum Classpath hinzufügen.
- Programm und Testklassen kompilieren.
- Ausführung der Tests mit

```
java org.junit.runner.JUnitCore BoardTest
```

```
JUnit version 4.3.1
..E
Time: 0.028
There was 1 failure:
1) testFood2(BoardTest)
java.lang.AssertionError:  (Stack backtrace)

FAILURES!!!
Tests run: 2,  Failures: 1
```

Weitere Annotationen

Eine Testklasse enthält oft viele ähnliche Testfälle.

Programmtext, der vor und nach jedem Test ausgeführt werden soll, kann in Methoden mit den Annotationen `@Before` und `@After` geschrieben werden.

- Alle mit `@Before` markierten Methoden werden vor *jedem* Test aufgerufen (in unbestimmter Reihenfolge).
- Alle mit `@After` markierten Methoden werden nach *jedem* Test aufgerufen (in unbestimmter Reihenfolge).

Beispiel – @Before

```
public class BoardTest {
  private BoardServerToClient b;

  @Before public void initBoard() {
    b = new BoardServerToClient(100, 100);
  }

  @Test public void testFood1() {
    int x = 94, y = 23, a = 34;
    b.setFood(x, y, a);
    assertEquals(b.getFood(x, y), a);
  }

  @Test(expected = java.lang.IndexOutOfBoundsException.class)
  public void testFood2() {
    b.setFood(100, 23, 34);
  }
}
```

Einmaliges Einrichten der Testumgebung

In manchen Fällen (z.B. teure `@Before`-Funktionen) ist es günstig, eine Testsituation einmal einzurichten und dann für alle Testfälle zu benutzen.

Statische Methoden mit Annotationen `@BeforeClass` und `@AfterClass`.

- Alle mit `@BeforeClass` markierten Methoden werden *einmal* vor den Tests ausgeführt.
- Alle mit `@AfterClass` markierten Methoden werden *einmal* nach den Tests ausgeführt.

Entwurf von Tests

Was sollte man mit Unit-Tests prüfen?

- Methoden sowohl mit erwarteten als auch unerwarteten Eingaben testen
- Rand- und Ausnahmefälle prüfen
- Werden Fehler richtig behandelt?
- Schreibe Tests, um zu verhindern, dass einmal aufgetretene Fehler nochmal auftreten (z.B. durch unvorsichtiges Revert).

Tests sollten einen (gedachten) Vertrag möglichst gut stichprobenartig überprüfen.