

Programmierprojekt: Software Tests

Annette Bieniusa
Sommersemester 2017

Testen

- Kernfrage: Erfüllt die Software ihre Anforderungen / Spezifikation?
- Funktionale Anforderungen
 - Korrekte Ergebnisse bei Berechnungen
- Nicht-funktionale Anforderungen
 - Laufzeit von Algorithmen, Look-and-feel der GUI, Skalierbarkeit von Servern, Latenzzeiten ...

Testen

- „Ein Test [...] ist der **überprüfbare** und jederzeit **wiederholbare** Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten **Anforderungen**“ (Denert, 1991)
- Wichtiger Baustein der **Qualitätssicherung** in der Software-Entwicklung
- „Program testing can be used to show the presence of bugs, but never show their absence!“ (Edsger W. Dijkstra)

Klassifizierung nach Umfang

- **Unit-Tests** (Komponententests) für einzelne Komponenten der Software
- **Integrationstests** für die Zusammenarbeit von Komponenten
- **Systemtest** für die Integration in die Anwendungsplattform
- **Abnahmetests** für die Interaktion mit dem Benutzer / Kunden

Klassifizierung von Informationsstand

- **White-Box-Tests** inspizieren auch den inneren Aufbau einer Komponente, häufig vom Entwickler durchgeführt
- **Black-Box-Tests** agieren mit einer wohldefinierten Schnittstelle, ohne Kenntnis über den tatsächlichen Aufbau einer Komponente, häufig von unabhängigen Instanzen auf Basis der Dokumentation durchgeführt

Methodik

- Ziel: Möglichst hohe **Abdeckung** des Java-Codes durch Testfälle
 - Möglichst jede Methode einer Klasse sollte getestet werden
 - Jede Methode sollte mit verschiedenen Parametern getestet werden
 - Randfälle sind wichtig!
- [Randomisierte Tests erzeugen zufällige Testfälle
-> verringert die Chance, dass der Programmierer
Randfälle übersehen hat]

Junit4

- Framework zum Unit-Testing von Java-Klassen
- Einfache Möglichkeit Tests zu spezifizieren und diese automatisiert auszuführen
- Reihenfolge der Ausführung ist nicht spezifiziert
- Testfälle müssen daher **unabhängig** von einander sein

Aufbau eines Tests

- Bei Hinzufügen eines neuen JUnit4- Tests wird folgender Stub erzeugt (und das Junit4 Package zum Build-Pfad hinzugefügt):

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

```
import org.junit.Test;
```

```
public class ProjektTest {
```

```
    @Test
```

```
        public void test() {
```

```
            fail("Not yet implemented");
```

```
        }
```

```
}
```


Assertions - Booleans

- Assertions testen, ob eine Bedingung erfüllt ist, andernfalls wird der Test nicht bestanden
- Um den Fehler zu beschreiben, kann man eine kurze Erklärung hinzufügen

@Test

```
public void testAssertFalse() {  
    org.junit.Assert.assertFalse("failure - should be false", false);  
}
```

@Test

```
public void testAssertTrue() {  
    org.junit.Assert.assertTrue("failure - should be true", true);  
}
```

Assertions - Objekte

```
@Test
```

```
public void testAssertNotNull() {  
    org.junit.Assert.assertNotNull("should not be null", new Object());  
}
```

```
@Test
```

```
public void testAssertNull() {  
    org.junit.Assert.assertNull("should be null", null);  
}
```

```
@Test
```

```
public void testAssertNotSame() {  
    org.junit.Assert.assertNotSame("should not be same Object", new Object(), new Object());  
}
```

```
@Test
```

```
public void testAssertSame() {  
    Integer aNumber = Integer.valueOf(768);  
    org.junit.Assert.assertSame("should be same", aNumber, aNumber);  
}
```

Assertions - Equals

@Test

```
public void testAssertArrayEquals() {  
    byte[] expected = "trial".getBytes();  
    byte[] actual = "trial".getBytes();  
    org.junit.Assert.assertArrayEquals("failure - byte arrays not same",  
        expected, actual);  
}
```

@Test

```
public void testAssertEquals() {  
    org.junit.Assert.assertEquals("failure - strings are not equal", "text", "text");  
}
```

Testen von Exceptions

```
@Test(expected= IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

Test fixtures

```
import org.junit.*;
public class TestFoobar {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // wird zu Beginn einmal ausgeführt
    }
    @Before
    public void setUp() throws Exception {
        // wird vor jedem Test ausgeführt
    }
    @Test
    public void test1() {
        // ein Test
    }
    @After
    public void tearDown() throws Exception {
        // wird nach jedem Test ausgeführt
    }
    @AfterClass
    public static void tearDownClass() throws Exception {
        // wird ganz am Schluss einmal ausgeführt
    }
}
```

- Manchmal ist es sinnvoll, dass sich mehrere Testfälle eine **Testumgebung teilen**
- Um die Unabhängigkeit zu gewährleisten, muss vor/nach dem Test die Umgebung wiederhergestellt werden (***setUp / tearDown***)

Junit für den Stundenverwaltung

- Tests für Anwendungslogik
 - Ist die Berechnung von Überstunden/fehlenden Stunden/ etc. korrekt?
- Tests für das Lesen und Schreiben von Daten
 - Sind nach einem Update alle neuen und alten Einträge vorhanden?
 - Wichtig: Regressionstests, wenn Formate geändert werden!
- Tests für das Erstellen von Übersichten
 - Summe aller Einnahmen / Ausgaben, pro Kategorie
- **NICHT TESTEN mit Junit:**
 - Datenbank funktioniert
 - GUI / Moustache / HTML Seiten etc. lassen sich nur schwer automatisiert testen

Strategien zur Testbarkeit

- Modularisierung
 - Testbare Einheiten identifizieren
 - Interfaces nutzen
 - Berechnung und Modell von GUI trennen!!!
- Einfache Tests mit guter Testabdeckung
 - Randfälle! (leere Collections)
- Dokumentation von Verhalten in Übereinstimmung mit Tests