ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Unit-Tests mit JUnit

Warum testen Entwickler so ungerne?

Standardausreden von Entwicklern:

- "Ich habe keine Zeit zum Testen."
- "Testen von Software ist langweilig und stupide."
- "Testen ist destruktiv!"
- "Mein Code ist praktisch fehlerfrei, auf jeden Fall gut genug."
- ...?

Automatisiertes Testen in Java

- **JUnit** wurde ursprünglich von **Erich Gamma** und **Kent Beck** entwickelt
- JUnit ist ein kleines, mächtiges Java-Framework zum Schreiben und Ausführen automatischer Unit Tests.
- Das Testen mit JUnit ist so einfach wie das Kompilieren.
- Die Testfälle sind selbstüberprüfend und damit wiederholbar.
- **`http://www.junit.org`**
  - ⇨ 3.8.x altes JUnit noch häufig im Einsatz
  - ⇨ 4.x neues JUnit ab Java 5 (2014: 4.12)
  - ⇨ 5.x Zur Zeit (11/2017) Milestone Release für 5.1

Unit Tests

- **Unit Testing** ist der Test von Programmeinheiten in Isolation von anderen im Zusammenhang eines Programms benötigten, mitwirkenden Programmeinheiten.
- Unit Tests konzentrieren sich dabei auf einzelne Einheiten des Systems.
- Die Größe der unabhängig getesteten Einheit kann dabei von einzelnen Methoden über Klassen bis hin zu Komponenten reichen.

Grundprinzip des Extreme Programming

Test first!

Schreiben Sie Tests, bevor Sie den Code schreiben, der diese Tests erfüllen soll, damit Sie sicherstellen, dass Ihr Code einfach zu testen ist.

Beispiel: KontoTest

Für die Klasse **Konto** ist eine Testklasse zu schreiben.
Folgendes ist zu tun:

1. Der Name der Testklasse setzt sich zusammen aus dem Klassennamen und dem Wort **Test**.
2. Das Package `org.junit` ist ggf. zu importieren. (Bei JUnit 3.* heißt das Package `junit.framework`)
3. Beim älteren JUnit musste die Testklasse eine Unterklasse der Klasse `junit.framework.TestCase` sein
4. Bei JUnit 4.* muss ein `static import` auf die Methoden der Klasse `org.junit.Assert` vorgenommen werden.

Aufbau der Testklasse

```
import org.junit.*;
import static org.junit.Assert.*;
public class KontoTest {
    /** optionaler Konstruktor fuer die Test-Klasse */
    public KontoTest(){  }
    /**
     *  Optionale Methode zum Aufbau des Testgerüsts,
     *  Wird vor jeder Testfall-Methode aufgerufen.
     */
     @Before
     public void setUp() {     }
    /**
     * Optionale Methode zur Freigabe des Testgerüsts
     * Wird nach jeder Testfall-Methode aufgerufen.
     */
     @After
     public void tearDown() {    }
}
```


Annotationen

- Seit Java 5 gibt es in der Programmiersprache Java sogenannte **Annotationen** (engl. annotations).
- Annotationen bieten die Möglichkeit, sogenannte **Meta-Daten** im Code unterzubringen.
- Annotationen beeinflussen nicht direkt die Programmsemantik, sie beeinflussen jedoch die Art und Weise wie Programme von Tools und Bibliotheken behandelt werden.
- Annotationen können aus den Quell-Dateien, aus class-Dateien und zur Laufzeit gelesen werden.

Annotationen in JUnit 4

- **`@Test`**
  kennzeichnet Methoden als Testfälle
- **`@Before`**
  markiert Methoden, die vor Beginn jedes Tests durchlaufen werden müssen
- **`@After`**
  markiert Methoden, die nach Ende jedes Tests ablaufen müssen
- **`@BeforeClass`** und **`@AfterClass`**
  markieren Aufgaben, die nur einmal je Testklasse ausgeführt werden sollen
- **`@Ignore`**
  kennzeichnet temporär nicht auszuführende Testfälle

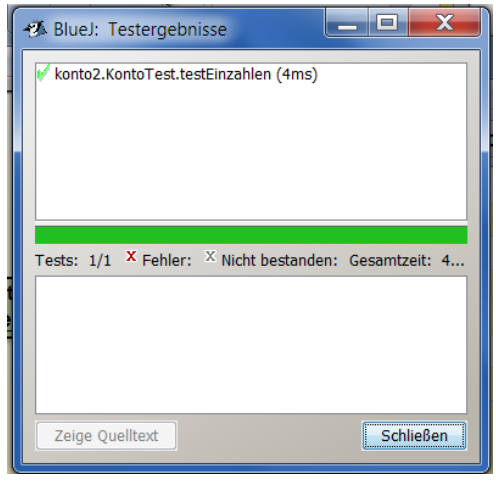
Erster Test

```
public class KontoTest {
    private Konto konto1;

    @Before
    public void setUp() {
        konto1 = new Konto(1234, "Meier", 1000.0);
    }
     /**
     * Einzahlen positiv testen
     */
    @Test
    public void testEinzahlen(){
        konto1.einzahlen(500.0);
        assertEquals("Betrag nicht korrekt gebucht!",
                  konto1.getKontostand(), 1500.0, 0.001);
    }
}
```

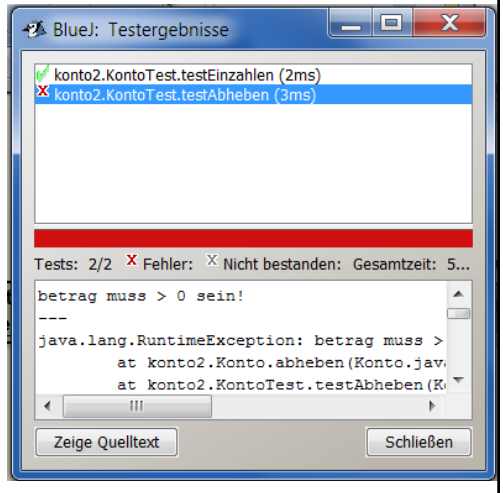
Test ablaufen lassen

- In BlueJ und in Eclipse können JUnit-Tests integriert gestartet werden.
- Sind alle Tests erfolgreich, so erscheint ein grüner Balken.



Fehlerhafte Tests

- Ist ein Test nicht erfolgreich, so erscheint ein roter Balken und Informationen über die Fehler können angezeigt werden.



Die Klasse Assert

- JUnit erlaubt es, Werte und Bedingungen zu testen, die jeweils erfüllt sein müssen, damit der Test okay ist.
- Die Klasse **`Assert`** definiert dazu eine Menge von **`assert`**–Methoden, die von Testklassen verwendet werden können.

Die Klasse Assert (Ausschnitt)

- **`assertTrue(boolean condition)`**
  verifiziert, ob eine Bedingung wahr ist.
- **`assertEquals(Object expected, Object actual)`**
  verifiziert, ob zwei Objekte gleich sind. Der Vergleich der Objekte erfolgt in JUnit über die `equals` Methode.
- **`assertEquals(int expected, int actual)`**
  verifiziert, ob zwei ganze Zahlen gleich sind. Der Vergleich erfolgt für die elementaren Java-Typen über ==
- **`assertEquals(double expected, double actual, double delta)`**
  verifiziert, ob zwei Fließkommazahlen gleich sind. Es wird zusätzlich eine Toleranz erwartet.

Die Klasse Assert (Ausschnitt)

- **`assertNull(Object object)`**
  verifiziert, ob eine Objektreferenz null ist.
- **`assertNotNull(Object object)`**
  verifiziert, ob eine Objektreferenz nicht null ist.
- **`assertSame(Object expected, Object actual)`**
  verifiziert, ob zwei Referenzen auf das gleiche Objekt verweisen.
- Zu all diesen Methoden gibt es jeweils noch eine Variante, bei der eine Fehlermeldung als erster Parameter mitgegeben werden kann
- **`fail(String message)`**
  wird aufgerufen, um explizit einen Test scheitern zu lassen; löst einen `java.lang.AssertionError` aus.

AssertionError

Was passiert, wenn ein Test fehlschlägt?

- Der laufende Testfall wird abgebrochen und eine Ausnahme vom Typ **`AssertionError`** ausgeworfen.
- Dieser Fehler wird in einem speziellen Objekt gesammelt (**`Result`**) und dem sogenannten Runner übergeben, der üblicherweise in der verwendeten IDE integriert ist.

Testen von Exceptions

- Wie testen wir, ob eine Exception wie erwartet geworfen wurde?
  - ⇨ Anwendung der Assert-Methode `fail`.

```
    @Test
    public void testKonstruktor() {
       Konto konto3;
       try {
           konto3 = new Konto(123, "Meier", 0.0);
           fail("Kontonummer nicht 4-stellig");
       } catch (IllegalArgumentException e){}
...
```

  - ⇨ Wird die Ausnahme wie erwartet ausgelöst, so passiert nichts, ansonsten löst `fail` die Ausnahme `AssertionError` aus.

Testen von Exceptions

- Alternative zum Testen von Exceptions:

```
    @Test (expected = IllegalArgumentException.class)
    public void testEinzahlenNegativerBetrag() {
            konto1.einzahlen(-100.0);
    }
```

- Hier wird also eine IllegalArgumentException erwartet
  - ⇨ Wird die Ausnahme wie erwartet ausgelöst, so passiert nichts, ansonsten wird die Ausnahme `AssertionError` ausgelöst.

Was genau ist das?

Ein sogenanntes Klassenliteral, das die Klasse selbst repräsentiert (später mehr)

Testen von Exceptions

- Was passiert, wenn eine nicht geplante Exception ausgelöst wird?
  - ⇨ Diese Exception wird als Error in dem Result-Objekt gesammelt.

Testfälle gruppieren

- **Testfall**: eine bestimmte Konfiguration von Objekten wird aufgebaut, gegen die der Test läuft.
- Diese Menge von Testobjekten wird auch als **Test-Fixture** oder **Testgerüst** bezeichnet.
- Pro Testfallmethode wird meist nur eine bestimmte Operation und oft sogar nur eine bestimmte Situation im Verhalten der Fixture getestet.

Lebenszyklus eines Testfalls

Was passiert, wenn JUnit die Tests dieser Klasse ausführt?

1. Das Test-Framework durchsucht die Testklasse mit Hilfe der Reflection API nach Test-Methoden
   ⇨ Signatur: mit `@Test` markierte void-Methode ohne Parameter
   `@Test void XXXX() { … }`
2. Für jede Test-Methode wird ein neues Objekt der Testklasse erzeugt.
3. Aufruf der mit `@Before`e markierten Methoden, sofern vorhanden.
4. Ausführen der eigentlichen Test-Methode.
5. Aufruf der mit `@After` markierten Methoden, falls diese redefiniert wurde, und Freigeben des Test-Objekts
6. Dieser Zyklus wird ab Schritt 2 solange wiederholt, bis alle Testfälle jeweils einmal ausgeführt wurden.

Ausführen mehrerer Testklassen

- Wie führen wir eine Reihe von Tests zusammen aus?
- Ziel:
  Testprozess so weit automatisieren, dass man den Test ohne manuellen Eingriff wiederholbar durchführen kann.
- Wichtig:
  die Unit Tests möglichst häufig ausführen, idealerweise nach jedem Kompilieren.

Führen Sie möglichst nach jedem erfolgreichen Kompiliervorgang alle gesammelten Unit Tests aus.

Ausführen mehrerer Testklassen

- In JUnit3: Tests in einer Testsuite zusammenfassen und gemeinsam ausführen.
- In einer statischen `suite` Methode wird definiert, welche Tests zusammen ausgeführt werden sollen.
- Eine Suite von Tests wird dabei durch ein `TestSuite` Objekt definiert, dem wir beliebig viele Tests und selbst andere Testsuiten hinzufügen können.

Junit 3: TestSuite

- pro Package eine Testsuiteklasse definieren
- Testsuiteklassen mit dem Namen `AllTests` benennen
- Hierarchien von Hierarchien von Testsuiten bilden:
  `suite.addTest(konto5.AllTests.suite());`

Junit 3: TestSuite

```
import junit.framework.*;

public class AllTests extends TestCase {
    public static void main(String[] args) {
      junit.swingui.TestRunner.run(AllTests.class);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(KontoTest.class);
        suite.addTestSuite(BankTest.class);
        return suite;
    }
}
```


Ausführen mehrerer Testklassen

- In JUnit4 werden Testklassen wie folgt zusammengefasst:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteC

@RunWith(Suite.class)
@SuiteClasses({KontoTest.class,
               BankTest.class})
public class AllTests  {}
```

bedeutet, dass JUnit diesen Test mit der Klasse Suite startet

listet alle Testklassen auf, die in Suite zusammengefasst werden sollen

Ausführen mehrerer Testklassen

- **`org.junit.runners.Suite`**
  Klasse zum alternativen Starten von Tests, wird verwendet über die Annotation:
  `org.junit.runner.@RunWith`
- **`@RunWith(Suite.class)`**
  eine Klasse, die damit annotiert wird, wird über den Testrunner Suite getestet.
- **`@SuiteClasses( ...})`**
  wird benötigt, um der Klasse Suite die Klassen mitzugeben, die zu testen sind.

Tests von der Shell aus starten

- Mit Hilfe der Klasse
  `org.junit.runner.JUnitCore` können Tests auch von der Shell aus gestartet werden:

```
$ java org.junit.runner.JUnitCore AllTests
JUnit version 4.10
.......I...I.
Time: 0,011

OK (11 tests)
```