JUnit&TDD

JUnit

- JUnit ist ein „Open Source Framework“ zur Automatisierung von Unit-Tests für Java.
- Entwickelt (um 1998) von Kent Beck und Erich Gamma auf der Basis von SUnit (Beck,1994) zum Testen von Smalltalk-Programmen.
- Die aktuellste Version ist JUnit 4.12
- junit.org

Entwicklung eines Testfalls in unit

- Zur Entwicklung eines Testfalls geht man in folgenden Schritten vor:
  - Deklariere eine Testklasse.
  - Schreibe eine setUp() Methode, um die Testobjekte zu initialisieren(annotiert mit @Before->Name der Methode egal) .
  - Schreibe eine tearDown() Methode, um die Testobjekte zu löschen(annotiert mit @After->Name der Methode egal).
  - Deklariere eine oder mehrere public testXXX() Methoden, die die Testobjekte aufrufen und die erwarteten Resultate zusichern.
  - Annotiere die Tests mit @Test

JUnit

- Die Klasse Assert bietet folgende Methoden:
- static void assertTrue(boolean b)
  - Zeigt keinen Fehler an, wenn b wahr ist
- static void assertFalse(boolean b)
  - Zeigt keinen Fehler an, wenn b falsch ist
- static void assertEquals(Object expected, Object actual)
  - Zeigt keinen Fehler an, wenn **expected und actual gleiche Werte haben (falls expected, actual prim. Daten)** expected.equals(actual) == true **(falls expected, actual Objekte)**
- static void assertEquals(double expected, double actual, double upTo)
  - |expected – actual| <= upTo
- static void assertNotNull(Object actual) actual != null

Beispiel

```
public class BankAccount {
    private int balance;
    private int limit;

    public BankAccount(int initialBalance, int l) {
        balance = initialBalance;
        limit = l;
    }

    public void deposit(int amount) {
        if (!(amount > 0))
            throw new IllegalArgumentException("Negativer Betrag");
        balance = balance + amount;
    }

    public void withdraw(int amount) {
        if (!(amount > 0))
            throw new IllegalArgumentException("Negativer Betrag");
        if (!(balance - amount >= limit))
            throw new IllegalArgumentException("Kontolimit ueberschritten");
    }

    public String toString() {
        return "BankAccount[balance = " + balance + "limit = " + limit + "]";
    }

    public int getBal() {
        return balance;
    }

    public int getLimit() {
        return limit;
    }
```

Beispiel

Fortführung vorherige Folie

```
public boolean transferTo(BankAccount other, int amount) {
    try {
        if (!(amount > 0))
            throw new IllegalArgumentException("Negativer Betrag");
        if (!(balance - amount >= limit))
            throw new IllegalArgumentException("Kontolimit ueberschritten");
        withdraw(amount);
        other.deposit(amount);
    } catch (IllegalArgumentException e) {
        return false;
    }
    return true;
  }
}
```

Beispiel

```java
@Test
public void testDeposit() {
    BankAccount b1 = new BankAccount(100, -50);
    b1.deposit(100);
    Assert.assertEquals(200, b1.getBal());
    Assert.assertTrue(b1.getBal() >= b1.getLimit());
}

@Test(expected = IllegalArgumentException.class)
public void testDepositException() {
    BankAccount b1 = new BankAccount(100, -50);
    b1.deposit(-10);
    Assert.assertEquals(200, b1.getBal());
    Assert.assertTrue(b1.getBal() >= b1.getLimit());
}

@Test(expected = IllegalArgumentException.class)
public void testDepositNull() {
    BankAccount b1 = new BankAccount(100, -50);
    b1.deposit(0);
    Assert.assertEquals(200, b1.getBal());
    Assert.assertTrue(b1.getBal() >= b1.getLimit());
}

public void testWithdraw() {
    BankAccount b1 = new BankAccount(100, -50);
    b1.withdraw(100);
    Assert.assertEquals(0, b1.getBal());
    Assert.assertTrue(b1.getBal() >= b1.getLimit());
}
```

Übung

- TDD/Uebung1.txt

10 Gründe warum TDD

If you can't write a test for something you don't understand it

- intensivere Auseinandersetzung mit dem zu lösenden Problem.
- höhere Qualität des Codes.
- Nachvollziehbarkeit des Codes wächst.
- was erwartet der Kunde.
- TDD ist also im Grunde keine Test- sondern eine Designmethode.

Without a regression test you can’t clean the code

- schnelles Testen ermöglicht, lauffähigen Code zu ändern
- Reduzierung überflüssigen Codes

Fast Feedback Cycles save time and money

- Wenn man zuerst den Test schreibt, lässt sich Korrektheit der Implementierung eines Features sofort überprüfen.
- hilft dabei, ein großes Problem in kleinere, überschaubare und testbare Probleme zu zerlegen.

Without Tests the next Developer will need a word doc and a lot of luck.

- Unit-Tests sind auch eine gute Form der Dokumentation, da sie die Inputs und erwarteten Outputs anzeigen. TDD bringt den Testprozess und damit diese Form der Dokumentation ins Zentrum der Aufmerksamkeit.

Evolutionary Design ist possible without fear

- Angst bestehende Code-Basis zu verändern ohne TDD
- Mit TDD wird die Möglichkeit der Änderungen an der Codebasis quasi zum Prinzip erhoben, was einen wirklich evolutionären Entwicklungsprozess ermöglicht.

You will actually write less code

- Mit Hilfe moderner IDEs und Refactoring-Werkzeugen schreibt man mit TDD weniger Code als ohne.
- Autovervollständigung etc.

Job Security

- Bug-freie Software ist auch das Ziel eines jeden Chefs!

It actually makes my work more enjoyable.

- Bei jedem erfolgreichen Test hat man das Gefühl, etwas Wertvolles geschaffen zu haben: ein Stück lauffähiger Software, die man stolz auch anderen Entwicklern zeigen kann.

Fazit

- TDD stellt die Qualität und Lauffähigkeit von Software sicher. In Verbindung mit Akzeptanztests kommt man so zu einem System, welches den Anforderungen der Kunden tatsächlich entspricht. Und dies führt nicht zuletzt zu Zufriedenheit und Eintracht auf allen Seiten:

TDD praktisch

- Überlege, welche Klasse und Methode geschrieben werden soll. Lege Quellcode für die Klasse und Variablen/Methoden/Konstruktoren an, sodass sich die Compilationsheit übersetzen lässt.
- Schreibe die API-Dokumentation für die Methoden/Konstruktoren und überlege, welche Parameter, Rückgaben, Ausnahmen nötig sind.
- Teste die API an einem Beispiel, ob sich die Klasse mit Eigenschaften „natürlich“ anfühlt. Falls nötig wechsele zu Punkt 1 und passe die Eigenschaften an.

TDD praktisch

- Implementiere eine Testklasse.
- Implementiere die Logik des eigentlichen Programms.
- Gibt es durch die Implementierung neue Dinge, die ein Testfall testen sollte? Wenn ja, erweitere den Testfall.
- Führe die Tests aus und wiederhole Schritt 5 bis alles fehlerfrei läuft.

Asserts

- static void assertTrue(boolean condition)
- static void assertTrue(String message, boolean condition)
- static void assertFalse(boolean condition)
- static void assertFalse(String message, boolean condition)

Asserts

- vstatic void assertNotNull(Object object)
- vstatic void assertNotNull(String message, Object object)
- vstatic void assertNull(Object object)
- vstatic void assertNull(String message, Object object)

Asserts

- static void assertNotSame(Object unexpected, Object actual)
- static void assertNotSame(String message, Object unexpected, Object actual)
- static void assertSame(Object expected, Object actual)
- static void assertSame(String message, Object expected, Object actual)

Assert

- static void assertEquals(Object expected, Object actual)
- static void assertEquals(String message, Object expected, Object actual)

Tests deklarativer schreiben

assertXXX() ohne Matcher	**assertThat()**
assertNotNull(new Object());	assertThat(new Object(), is(notNullValue()));
assertEquals("", StringUtils.reverse(""));	assertThat("", is(equalTo(StringUtils.reverse(""))));
assertSame("", "");	assertThat("", is(sameInstance("")));
assertNotSame("", "a");	assertThat("a", is(not(sameInstance(""))));

Beispiel für gute Lesbarkeit

```
assertThat( list, hasSize(3) );
assertThat( list, both( hasItems( "a", "c", "e" ) ).and( not( hasItems( "b", "d" )
```

Live Coding FizzBuzz

Übung

- TDD/Uebung2.txt