JUnit: Einführung in testgetriebenes Entwickeln (engl. Test-Driven Development)

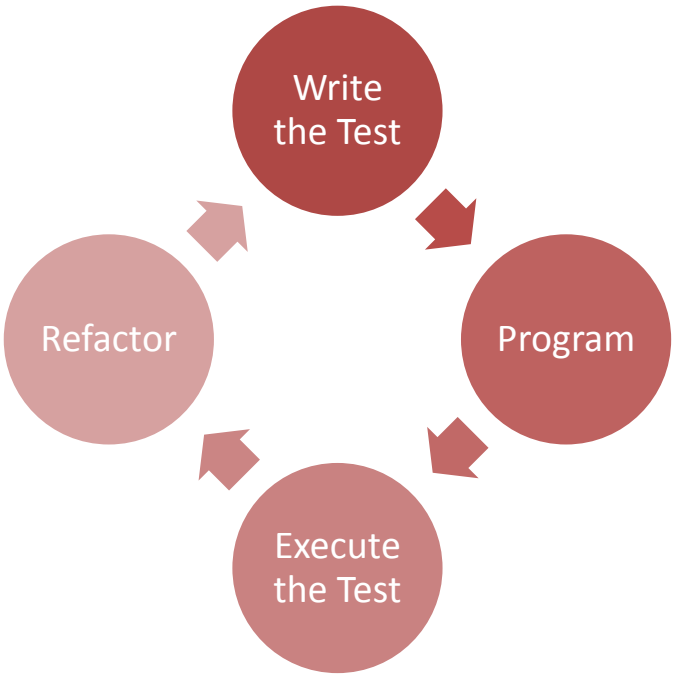
- Die Vorteile automatisierten Testens und testgetriebenen Entwickelns erkennen
- Wichtige Begriffe rund um das Testen kennen und verstehen
- Die Rolle des Testens beim Extreme Programming verstehen
- Einige Eigenschaften des Extreme Programmings einordnen können
- Die Möglichkeiten von JUnit kennen und diese gewinnbringend einsetzen können
- Das Lesen von Testdaten aus Dateien beim Testen simulieren können

Ressourcen

Softwaretests mit JUnit, Johannes Link, dpunkt.verlag, ISBN 3-89864-325-5

`http://junit.org`

„Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken. […] Stellen Sie sich einen Kletterer vor, der jeden seiner Schritte durch einen Haken absichert. Mit jedem gesetzten Sicherheitshaken reduziert er ganz bewusst sein Risiko, wie tief er bei einem Fehltritt fallen kann. […] Der bis zum Haken erkletterte Weg gehört ihm in jeden Fall, selbst wenn ihm ein Fehler unterläuft."[1]



[1] Testgetriebene Entwicklung mit JUnit und FIT, Frank Westphal, dpunkt.verlag

Motivation

- Je früher getestet wird, desto stabiler ist Programm und umso weniger Zeit wird für Fehlersuche und -behebung benötigt
- Frühzeitiges Testen beeinflusst Programmdesign und sorgt für bessere Schnittstellendefinitionen
- Auswirkungen kleiner Programmänderungen können anhand einer Menge automatisierter Tests sofort verifiziert werden
- Tests des zusammengesetzten Systems reichen nicht aus, um Fehler in Komponenten zu entdecken (*Antidecomposition Axiom*[2])

Testarten

Performance- und *Lasttests*

Komponenten- und *Integrationstests*
(Methoden-, Klassen-, Teil-, Gesamtsystemtests)

Akzeptanztests
testen Funktionalität des Systems aus Sicht der Anwender

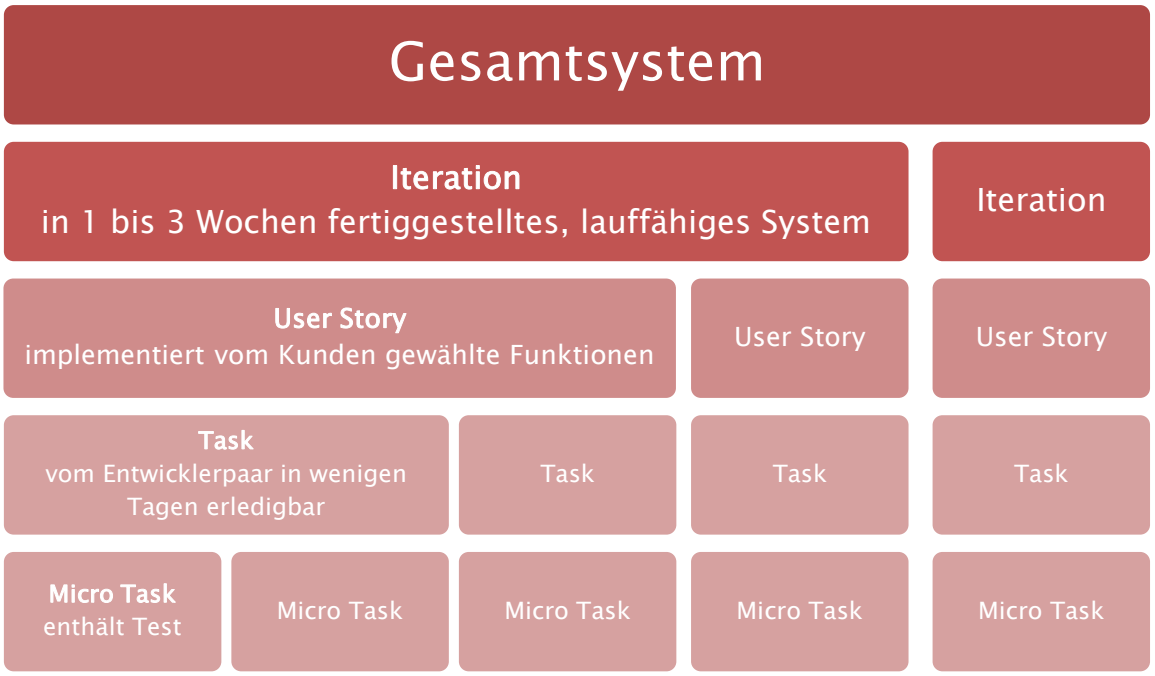
Extreme Programming (XP) und Unit Tests

- *Agiler Softwareentwicklungsprozess* mit kurzen Releasezeiten (Kunde entscheidet über Richtung im Projekt)
- *Pair Programming*
  - Zwei Entwickler arbeiten an einem Computer gemeinsam
  - Einer programmiert, der Andere denkt in größeren Zusammenhängen (Rollen wechseln ständig)
  - Gute Wissensverbreitung im gesamten Team

---

[2] „eine adäquate Testabdeckung eines Moduls garantiert noch nicht die Abdeckung all der Teilmodule, die das Modul aufruft“ (Quelle: `www.wikiservice.at`)

- *Inkrementelle Entwicklung, Entwicklung in kleinen Schritten*



- *Refactoring (dtsch. neu herstellen)*
  ständige Verbesserung der internen Struktur von existierendem und funktionierendem Programmcode
  - Code vereinfachen so dass er alle seine Designkonzepte kommuniziert
  - Ohne automatisierte Tests kaum möglich, da Gefahr sehr groß, dass Umbau unvorhergesehene Nebenwirkungen hervorruft
  - Spätestens nach jedem Task durchführen
- *Unit Tests*
  - Werden vor dem Entwicklungscode erstellt, bei Bedarf geändert und angepasst
  - Alle Tests werden bei Integration von neuem Code ausgeführt
  - Schlägt irgendein Test fehl, muss zuerst Fehler behoben werden, bevor Integration fortgesetzt wird

Vorteile von XP

- Code in XP soll so geschrieben werden, dass er alle Konzepte kommuniziert, die er enthält (Methoden- und Klassennamen)
- Code nur so komplex, wie es momentan nötige Funktionalität verlangt (ignoriert vermutlich zukünftige Funktionalität)
- Schnelles und häufiges Feedback ob Code das tut was er tun soll (häufig ausgeführte automatische Tests)

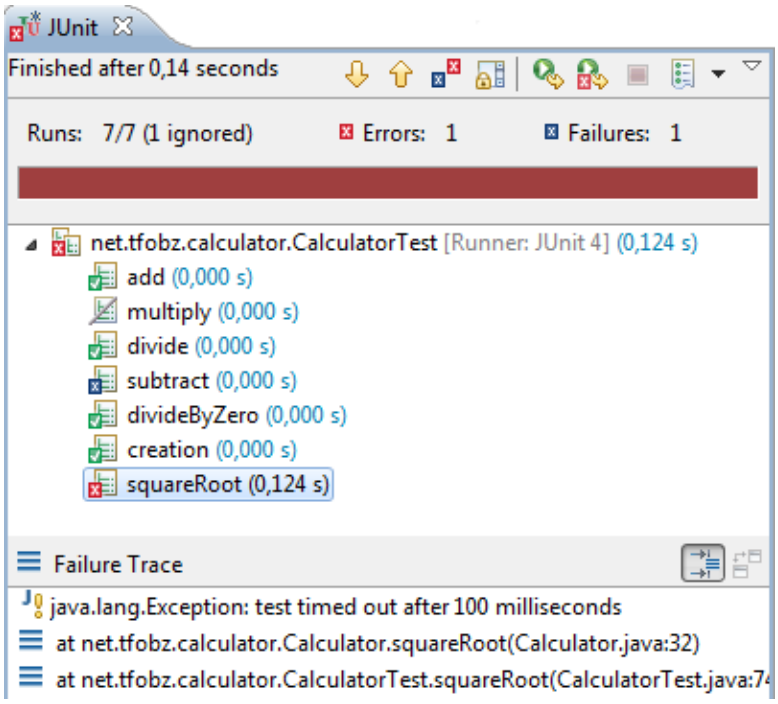
Vorteile von testgetriebenem Entwickeln

- Änderungen die vorhandene Funktionalitäten zerstören werden sofort erkannt
- Tests dokumentieren den Code
- Design des Programms wird maßgeblich von den Tests bestimmt
- Nachträgliches Erstellen von Tests wird vermieden („wieso soll ich Test schreiben für Code der eh schon funktioniert!“)

JUnit Beispiel

```
public class CalculatorTest
{
  private static Calculator c = null;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
    c = new Calculator();
  }

  @AfterClass
  public static void tearDownAfterClass() throws Exception {
    c = null;
  }

  @Before
  public void setUp() throws Exception {
    c.clear();
  }

  @Test
  public void creation() {
    assertEquals(0, c.getResult());
  }

  @Test
  public void add() {
    c.add(2);  c.add(2);
    assertEquals(4, c.getResult());
  }

  @Test
  public void subtract() {
    c.subtract(2); c.subtract(2);
    assertEquals(-4, c.getResult())
  }

  @Test
  @Ignore
  public void multiply() {
    c.add(2);  c.multiply(2);
    assertEquals(4, c.getResult());
  }

  @Test
  public void divide() {
    c.divide(2);
    assertEquals(0, c.getResult());
    c.add(2);  c.divide(2);
    assertEquals(1, c.getResult());
  }

  @Test (expected=ArithmeticException.class)
  public void divideByZero() {
    c.divide(0);
  }

  @Test (timeout=100)
  public void squareRoot() {
    c.squareRoot(4);
    assertEquals(2, c.getResult());
  }
}
```

```
public class Calculator
{
  private static int result = 0;
  public int getResult() {
    return result;
  }
  public void clear() {
    result = 0;
  }
  public void add(int n) {
    result = result + n;
  }
  public void subtract(int n) {
    result = result + 1;  // Bug
  }
  public void multiply(int n) {
    //TODO
  }
  public void divide(int n) {
    result = result / n;
  }
  public void squareRoot(int n) {
    for(;;);  // Bug
  }
}
```



- *Annotationen* (ab Java 5) binden Zusatzinformationen in den Quelltext ein, die beim Kompilieren ausgewertet werden
- Jede Klasse kann Testklasse sein, muss nur Defaultkonstruktor haben
- Jede Methode dieser Klasse kann Testmethode (*@Test*), Initialisierungs- oder Aufräummethode (*@BeforeClass*, *@AfterClass*, *@Before*, *@After*) sein
- Alle Testmethoden werden bei Testdurchführung nacheinander in beliebiger Reihenfolge gestartet
- Schlägt ein `assert` in Testmethode fehl, wird Testmethode abgebrochen
- *@Ignore* Testmethode wird momentan nicht gestartet
- *@Test (expexted=IndexOutOfBoundException.class)* Test nur dann erfolgreich, wenn Testmethode Exception wirft
- *@Test (timeout=100)* Test wird nach 100ms abgebrochen

`assertEquals`	zuerst erwarteter Wert dann tatsächlicher Wert (!)
`assertSame`	Referenzen müssen gleich sein
`assertTrue`	`assertFalse`
`assertNull`	`assertNotNull`
`assertArrayEquals`	Auch Reihenfolge der Elemente muss übereinstimmen

Test Suite

- Zusammenstellung von Testklassen, die gemeinsam ausgeführt werden
- Dient zum Gruppieren von Tests, die gewisse Funktionalitäten, Teil- oder Gesamtsystem testen

```
@RunWith(Suite.class)
@SuiteClasses({ PersonListTest.class, PersonTest.class })
public class AllTestsPerson
{
}
```