



Software Engineering Blog

Menschen. Technik. Prozesse.

<http://www.swe-blog.net>

Test NG – Unit-Test 2.5

Eine Alternative zu JUnit

Oliver Fischer

Java User Group Berlin-Brandenburg

23. März 2009

Überblick, Fahrplan für die nächste Stunde

1. Vorstellung
2. Unit-Tests
3. Junit
4. TestNG
5. Fazit

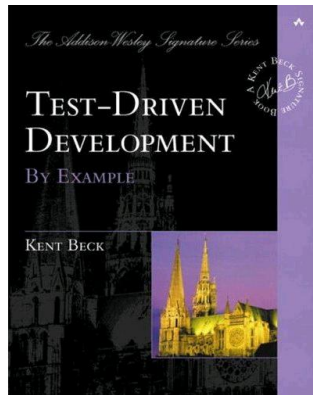


Unit-Tests

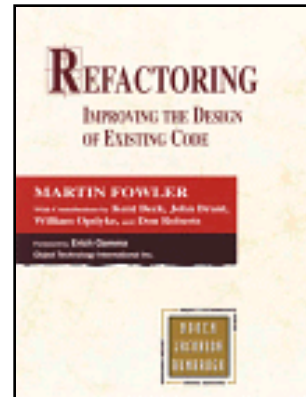
Unit-Tests

Unit-Tests testen die kleinsten Einheiten eines Software-Systems. Ursprünglich nur auf Methoden- und Klassenebene.

- Test Driven Development baut auf Unit-Tests auf: Schreibe erst den Test und dann den Code.
 - Der Testfall beschreibt die Anforderungen.
 - Der Code muß den Testfall erfüllen.
 - Führe nach jeder Änderung die Testfälle aus und passe Test und Code an.
- Testen wird in die Entwicklung integriert und erfolgt parallel zur Codierung



Kent Beck
2002
Addison-Wesley



Martin Fowler
1999
Addison-Wesley





JUnit **Der Klassiker**

JUnit

- Erstes Unit-Test-Framework für Java
- Entwickelt von Kent Beck und Erich Gamma
- Bis einschließlich Version 3 sehr eingeschränkt
- Probleme von JUnit waren Auslöser für die Entwicklung von TestNG
- Version 4 hat wesentliche Probleme behoben

- Muß man mehr sagen?





TestNG

Ein ganz normaler Testfall

```
public class MailReceiverTestCase
{
    @Parameters({"server", "user", "password"})
    @BeforeMethod
    public void connectIMAPServer(String host, String username, String password)
    {
        System.err.println("Connecting " + username + ":" + password + "@" + host);
    }

    @Test
    public void doSomething()
    {...}

    @AfterMethod
    public void disconnectIMAPServer()
    {...}
}
```



Schnellüberblick TestNG

Features von TestNG

- Basiert rein auf Annotationen
- Kann auch mit Java 1.4 verwendet werden
- Unterstützung für Testabhängigkeiten (Führe Test nur aus, wenn alle Voraussetzungen erfolgreich getestet wurden)
- Unterstützung von Testgruppen. Trennung von Implementierung der Testmethoden von deren Ausführung (statisch versus dynamisch).
- Umfassend und leicht erweiterbar über API.
- Unterstützung von Data Driven Testing



Testfälle in TestNG

```
public class Test1 {  
    @Test()  
    public void testMethod1() {  
    }  
}
```

```
@Test()  
public class Test1 {  
    public void testMethod1() {  
    }  
}
```



testng.xml - Die Schaltzentrale

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite thread-count="5" verbose="1" name="Custom suite">
  <test name="Subsystem A">
    <packages>
      <package name="net.sweblog.demo.testng.suite000.suba" />
    </packages>
  </test>

  <test name="Subsystem B">
    <classes>
      <class name="net.sweblog.demo.testng.suite000.subb.TestPreparation" />
      <class name="net.sweblog.demo.testng.suite000.subb.TestCaseA" />
    </classes>
  </test>
</suite>
```



Optionen für @Test

Option	Beschreibung
<code>alwaysRun</code>	Auch bei fehlgeschlagenen Abhängigkeiten ausführen
<code>enabled</code>	Testklasse/-methode ausführen?
<code>description</code>	Testbeschreibung
<code>dataProvider</code>	Name des Datenproviders
<code>dataProviderClass</code>	Klasse des Datenproviders, wenn nicht eigene
<code>expectedExceptions</code>	Liste der erwarteten Exceptions
<code>successPercentage</code>	Notwendige Erfolgsrate bei mehrfacher Ausführung
<code>groups</code>	Liste der Gruppen, zu der die Testklasse/-methode gehört
<code>dependsOnGroups</code>	Liste der Gruppen, von denen die Testklasse/-methode abhängig ist
<code>dependsOnMethods</code>	Liste der Methoden, von denen die Testklasse/-methode abhängig ist
<code>timeOut</code>	Testtimeout in Millisekunden
<code>sequential</code>	Tests in dieser Klasse sequentiell ausführen
<code>invocationCount</code>	Anzahl der Testausführungen für eine Testfall (Methode)
<code>invocationTimeOut</code>	Testtimeout für alle Aufrufe akkumuliert
<code>threadPoolSize</code>	Maximale Threadanzahl für die Testausführung.



Ein Testfall mit Parametern

```
@Test(description="Ein Testfall", timeOut=1000,  
      dataProvider="FilenameProvider",  
      expectedExceptions={java.io.IOException.class})  
public void justAnotherTestCase(String inputFileName)  
{...}
```

Was macht dieser Testfall?

- Testbeschreibung lautet „Ein Testfall“
- Jede Ausführung darf maximal 1.000 ms dauern
- Die Eingabeparameter liefert der Datenprovider „FilenameProvider“
- Jede Testausführung muß mit einer Exception vom Typ `java.io.IOException` enden



Einen Test parallel ausführen

```
@Test(description="Ein paralleler Testfall", timeOut=1000,  
        dataProvider="URLProvider",  
        invocationCount=50, threadPoolSize=5)  
public void justAnotherTestCase(URL targetURL)  
{...}
```

Was macht dieser Testfall?

- Testbeschreibung lautet „Ein paralleler Testfall“
- Die Ausführung darf maximal 1.000 ms dauern
- Der Test muß 50 mal ausgeführt werden
- 5 Instanzen des Test können parallel ausgeführt werden



Aufbauen und Aufräumen

Parameter	Beschreibung
@BeforeSuite	Führe Methode aus, bevor die Suite ausgeführt wird
@AfterSuite	Führe Methode aus, nachdem alle Tests der Suite ausgeführt wurden
@BeforeTest	Führe Methode vor jedem Test aus
@AfterTest	Führe Methode nach jedem Test aus
@BeforeGroups	Führe Methode vor bestimmten Gruppen aus
@AfterGroups	Führe Methode nach bestimmten Gruppen aus
@BeforeClass	Führe Methode vor allen Testmethoden der eigenen Klasse aus
@AfterClass	Führe Methode nach allen Testmethoden der (eigenen) Klasse aus
@BeforeMethod	Führe Methode vor jeder Testmethode aus
@AfterMethod	Führe Methode nach jeder Testmethode aus



Klasse mit Testfällen initialisieren

@Test

```
public class TestCaseA
```

```
{
```

@BeforeClass

```
public void prepareAllTestsInClass() {
```

```
    System.out.println(getClass().getSimpleName() + ".prepareAllTestsInClass()");
```

```
}
```

@AfterClass

```
public void cleanUpAfterAllTestsInClass() {
```

```
    System.out.println(getClass().getSimpleName() + ".cleanUpAfterAllTestsInClass()");
```

```
}
```

```
public void testCase2() {
```

```
    System.out.println(getClass().getSimpleName() + ".testCase2()");
```

```
}
```

```
}
```



Test in Reihe bringen...

Die Reihenfolge von Tests

- TestNG beachtet `@Before*`- und `@After*`-Annotationen und Gruppenzugehörigkeit
- TestNG garantiert keine darüber hinausgehende Ausführungsreihenfolge
- TestNG Reihenfolge von Methodenausführungen inkl. Testmethoden kann über Option `dependsOnMethods` erzwungen werden
- Es können nur gleich annotierte Methoden mit `dependsOnMethods` geordnet werden.

Ausführungsabhängigkeiten mit `@BeforeMethod`

`@BeforeMethod`

```
public void setUpAAA() {...}
```

`@BeforeMethod(dependsOnMethods = {"setUpAAA"})`

```
public void setUpXYZ() {...}
```



Data-Driven-Tests

Die Reihenfolge von Tests

- Testfälle und Testergebnisse sind (oft) abhängig von den Eingabedaten
- Object-Under-Test oder Method-Under-Test haben datenabhängiges Verhalten
- In vielen Testfällen müssen Grenz- und Wertebereich getestet werden
- TestNG unterstützt Data-Driven-Tests

Data-Driven-Tests sind Testfälle, bei denen die gleiche Code-Basis mit unterschiedlichen Eingabedaten aufgerufen werden.

Unterstützung durch TestNG

- Testmethoden können parametrisiert werden
- Eingabedaten für Testfälle können aus testng.xml kommen
- Standardparameter können im Source-Code vorgegeben werden
- Dabei Beschränkung auf als String-darstellbare Daten
- Flexiblere Lösung durch DataProvider genannte Methoden
- Trennung von Testfall und Testdatum



Testdaten über @Parameter und @Optional (I)

Parameterdefinition als Key/Value-Pair

```
<suite thread-count="5" verbose="1" name="Custom suite">  
  <parameter name="server" value="aServer"/>  
  <parameter name="user" value="aUser"/>  
  <parameter name="password" value="secret"/>  
  ...  
</suite>
```

Auch Setup-Methoden
können parameterisiert
werden!

Parameterdefinition als Key/Value-Pair

```
@Parameters({"server", "user", "password"})
```

```
@BeforeMethod
```

```
public void connectIMAPServer(String host, String username, String password)  
{  
  System.err.println("Connecting " + username + ":" + password + "@" + host);  
}
```



Testdaten über @Parameter und @Optional (II)

Parameterisierung von Testfällen

```
@Parameters({"receiver"})
```

```
@Test
```

```
public void sendMail(@Optional("max@mustermann.de") String receiver)
{
    System.err.println("Mail an " + receiver);
}
```

Standardwerte für Parameter vorgeben

```
@Parameters({"server", "user", "password"})
```

```
@BeforeMethod
```

```
public void connectIMAPServer(@Optional("imap.firma.de") String host,
                              @Optional("peter") String username,
                              @Optional("lustig") String password)
{... }
```



Testdaten über DataProvider (I)

Und wenn es flexibler sein muß?

- Nicht alle Daten lassen sich als String ausdrücken
- Wie kann ich folgendes erreichen:
 - Teste alle Beträge zwischen zehn und zwanzig Euro
 - Lade Testdaten aus der Datenbank (vielleicht ein Spiegel der Produktionsdatenbank)
 - Lese Werte aus der Datei xyz.dat
- @Parameter und @Optional lassen keine Wertemengen zu
- @Parameter und @Optional eignen sich sehr gut zur Konfiguration von Tests

DataProvider sind flexiblere Möglichkeit

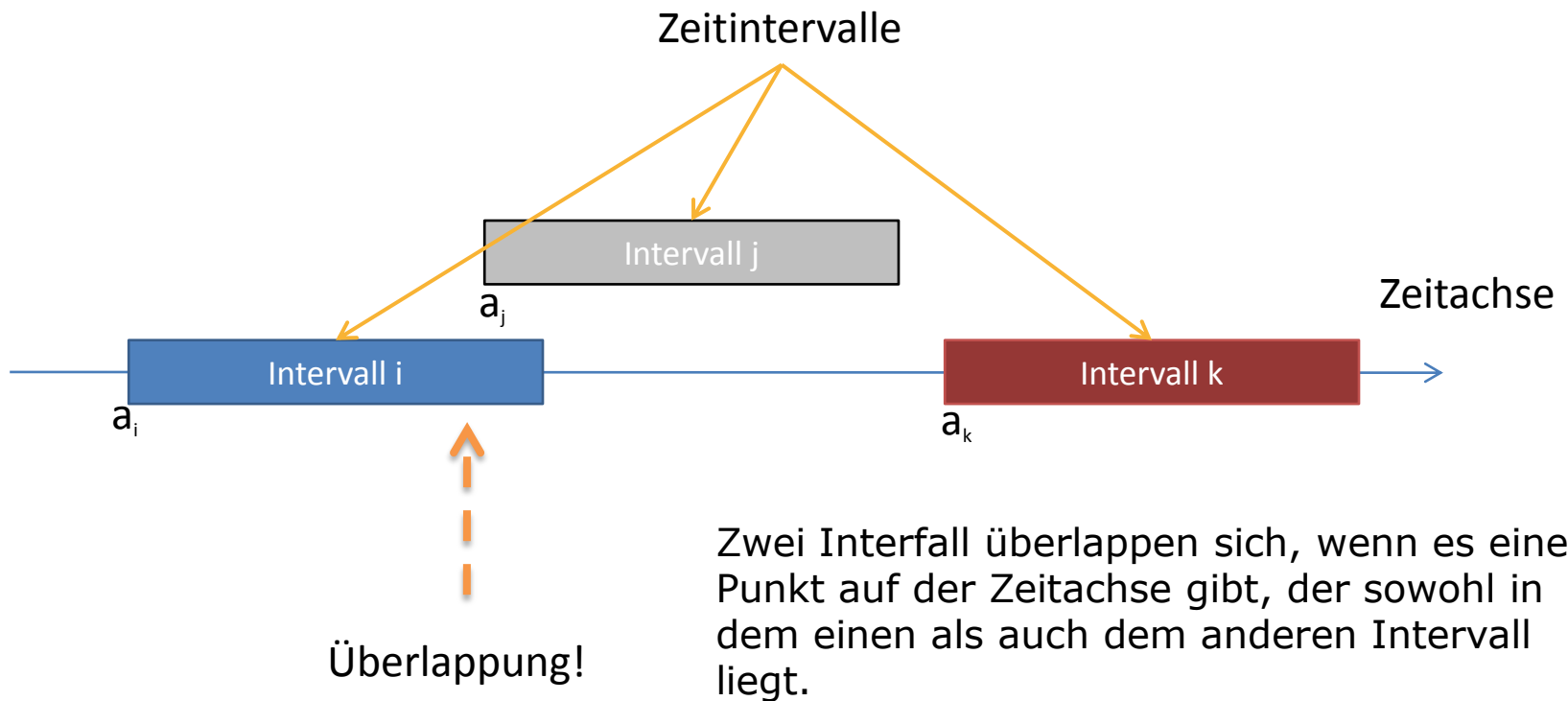
- DataProvider sind statische Methoden die zurückgeben müssen
 - Array-of-Array von Object: `object[][]`
 - Iterator über Array von Objekten: `Iterator<Object[]>`
- DataProvider können ihr Verhalten in Abhängigkeit der Testmethode ändern



Testdaten über DataProvider (II)

Aufgabe Testfalldefinition

Definieren Sie Testfälle für die Methode `boolean overlapsWith(Intervall other)`...



Testdaten über DataProvider (III)

Haben Sie die folgenden Testfälle:

- für **null** als Parameter für other?
- in dem das andere Intervall **vor dem anderen** liegt?
- in dem das andere Intervall **nach dem andern** liegt?
- in dem das andere Intervall **vollständig in dem anderen** liegt?
- in dem das andere Intervall **das andere vollständig umfaßt**?
- in dem das andere Intervall **genau an das andere angrenzt**?
- in dem das andere Intervall **genau auf das andere folgt**?
- in dem beide Intervalle **gleich sind**?
- in dem das eine Intervall **punktförmig ist**?
- in dem beide Intervall **punktförmig sind**?

Wie viele Testmethoden sind hierfür notwendig?



Testgruppen (I)

- Testfälle haben oft einen thematischen Schwerpunkt oder bestimmte Eigenschaft nach denen sie gruppiert werden können. Beispielsweise:

Schwerpunkt	Eigenschaft
Datenbankintegration	schnell
Kundenverwaltung	Speicherplatzintensiv
Nightly Build	
Integrationstest (nach Commit)	

Unterstützung durch TestNG

- Testfälle lassen sich **zu Gruppen zusammenfassen**
- Gruppen können **andere Gruppen** beinhalten (Metagroups)
- Gruppen können **selektiv** ausgeführt werden




Testgruppen (II)

```
@Test(groups = {"database", "postgreSQL", "linux.database", "windows.database"})
public class TestCasePostgreSQL
{
    public void testMethod001() {
        ...
    }

    public void testMethod002() {
        ...
    }

    public void testMethod003() {
        ...
    }
}
```

*Jeder Test kann
zu mehreren Gruppen
gehören*



Testgruppen (III)

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">  
<suite thread-count="5" verbose="1" name="Custom suite">  
  <test name="database">  
    <groups>  
      <run>  
        <include name="windows.*"/>  
      </run>  
    </groups>  
    <packages>  
      <package name="net.sweblog.demo.testng.groups000"/>  
    </packages>  
  </test>  
</suite>
```

*Reguläre Ausdrücke sind auch
Möglich bei der Auswahl von Gruppen*



Was liegen geblieben ist

- Ausführung von JUnit-Tests ist möglich über testng.xml oder programmatisch
- Annotation-Transformer, die jede TestNG-Annotation manipulieren können vor Ihrer Auswertung
- API
- Reportgenerierung
- Testselektion über BeanShell in testng.xml
- Dependency Injection über ITestContext-Parameter
- ...



Fazit

- TestNG hat eine Reihe von Vorteilen:
 - Non-Intrusive, da konfigurierbar über Annotationen und XML-Datei
 - Laufzeitverhalten (welche Tests auszuführen sind) ist unabhängig von der Testdefinition im Code
 - Möglichkeit Abhängigkeiten zu definieren erlaubt eine klarere Strukturierung von Testfällen und bessere Fehlerlokalisierung
 - Komplizierte Szenarien sind auch mit Hausmitteln beschreibbar
 - API erlaubt Erweiterung über Interceptoren und Listener
 - Unterstützt die Ausführung von JUnit-Tests



Danke

