Testen mit JUnit

Martin Wirsing

in Zusammenarbeit mit
Michael Barth, Philipp Meier und Gefei Zhang

12/04

Ziele

- Lernen Unit Tests zu schreiben
- Lernen mit Unit-Testen mit JUnit durchzuführen

Testen

Da die meisten Programme Fehler enthalten, ist es notwendig, jedes Programm zu testen und die gefundenen Fehler zu verbessern.

- Testen bedeutet, ein Programm stichprobenartig auszuführen, um Fehler zu finden.
- Ein Test ist „**erfolgreich**“, wenn ein **Fehler gefunden** wurde.
- Durch Testen kann man nicht die Korrektheit eines Programms beweisen, sondern nur seine Fehlerhaftigkeit.

Arten von Tests

Man unterscheidet beim Testen u.a.

Unit-Test	Test der einzelnen Methoden einer Klasse
Modul-Test	Test einer Menge von Klassen mit einer bestimmten Aufgabe
Integrationstest	Test der Integration mehrerer Module
Systemtest	Test des Gesamtsystems (im Labor)
Feldtest	Test während des Einsatzes

Unit-Test

Beim **Unit-Test** wird **jede Methode einer Klasse** systematisch **getestet** und zwar bzgl. der gegebenen (informellen oder formalen) Spezifikation.

Man unterscheidet

Blackbox-Test Test des Verhaltens des Programms, ohne die Implementierung zu berücksichtigen

Whitebox-Test Test des Rumpfes der Methode

- Bei beiden Techniken soll eine **möglichst repräsentative und vollständige Menge von Fällen** getestet werden.

Whitebox-Test

Beim **Whitebox-Test** sind das z.B. alle möglichen unterschiedlichen Fälle von Abläufen eines Programms.

Beispiel:

Das Programmfragment
while(B) {A }
C

besitzt die unterschiedlichen Abläufe C und AC, AAC,...

Man sollte zumindest den **Abbruchfall** testen, sowie **einen Schleifendurchlauf** und dann Abbruch (als Grenzfall) sowie **mehrere Schleifendurchläufe** und dann Abbruch.

Whitebox-Test: Beispiel Quersumme

```
int quersumme(int x)
{   int qs = 0;
    while (x > 0)
    {  qs = qs + x % 10;
       x  = x/10;
    }
    return qs;
}
```

Wahl der Testfälle aufgrund der Codestruktur

Mögliche Testfälle: **Abbruchfall**: x<=0

1 und mehrere Schleifendurchläufe:

x =1, x=9, x= 12, x= 352

Blackbox-Test

- Beim Blackbox-Test werden für jede Methode Spezialfälle der Spezifikation getestet. Die Implementierung der Methoden wird NICHT und darf NICHT berücksichtigt werden.
- Für jeden Methode werden nur die Parameter und deren Datentypen betrachtet. Um eine möglichst vollständige Testabdeckung zu erreichen, sollte für jeden Parameter im Kopf der Methode gewählt werden:
  - Ein Standardwert in der Mitte des Datenbereichs
  - Grenzwerte des Datenbereichs bzw. des Definitionsbereichs
  - Bei induktiven Datentypen Werte für jeden Konstruktor.

  und Kombinationen davon.
- **Beispiel:** Für einen Parameter int p mit Def.bereich {0, ..., 100} testet man z.B. die Werte –1, 0, 38, 100, 101.

  Analog testet man bei einer Reihung int[] a die Werte a[0], a[length/2],, a[length-1]

JUnit

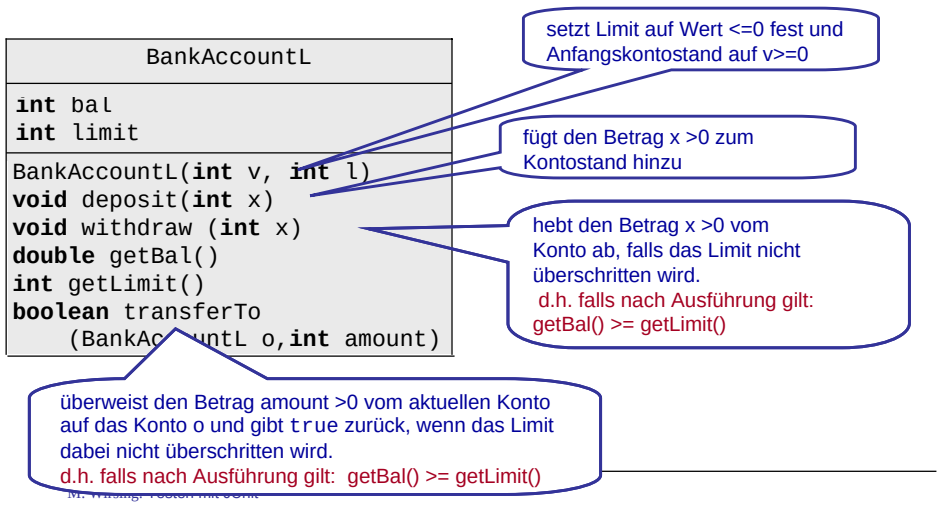
- JUnit ist ein „Open Source Framework“ zur Automatisierung von Unit-Tests für Java.
- Entwickelt (um 1998) von Kent Beck und Erich Gamma auf der Basis von SUnit (Beck,1994) zum Testen von Smalltalk-Programmen.
- Zum Herunterladen unter
  `http://download.sourceforge.net/junit/`

Entwicklung eines Testfalls in JUnit

Zur Entwicklung eines Testfalls geht man in folgenden Schritten vor:

1. Deklariere eine Unterklasse von TestCase.
2. Redefiniere die setUp() Methode, um die Testobjekte zu initialisieren .
3. Redefiniere die tearDown() Methode, um die Testobjekte zu löschen.
4. Deklariere eine oder mehrere public testXXX() Methoden, die die Testobjekte aufrufen und die erwarteten Resultate zusichern.
5. (Optional) Definiere eine main() Methode, die den Testfall startet. (Nicht nötig im Rahmen von Entwicklungsumgebungen wie Eclipse)

Beispiel: Bankkonto mit Überziehungsrahmen



Beispiel: BankAccount (1)

```
public class BankAccount
{   private int balance;
    private int limit;                        //Invariante: balance >= 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;
    }
```

Beispiel: BankAccount (2)

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

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


Beispiel: BankAccount (3)

```
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: Einfacher Testfall für BankAccount

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


Noch ein BankAccounttestfall

```
public void testWithdraw()
{     BankAccount b1 = new BankAccount(100,-50);

      b1.withdraw(100);
      assertEquals(0, b1.getBalance());
      assertTrue(b1.getBalance() >= b1.getLimit());
}
```

Testklasse für BankAccount (1)

```
import junit.framework.*;
public class BankAccountTest extends TestCase
{ private BankAccount b1;
  private BankAccount b2;

  public BankAccountTest(String arg0)
  {     super(arg0);
  }
  public void setUp()
  {     b1 = new BankAccount(100, -50);
        b2 = new BankAccount(100, -50);
  }
  public void tearDown()      //ohne Effekt bei BankAccount,
  {     b1 = null;            //da immer neue Testobjekte erzeugt werden
        b2 = null;
  }
```


Testklasse für BankAccount (2)

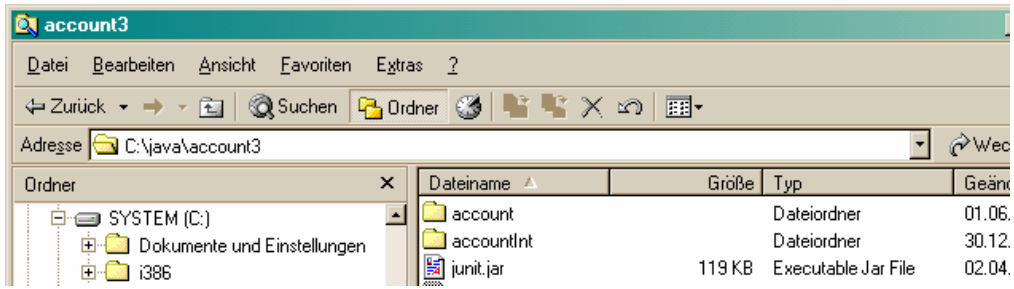
Fortsetzung

```
...

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

}
```


Übersetzen und Ausführen von Tests

- **Installiere JUnit in einem Verzeichnis oberhalb der zu testenden Klasse:**

  Seien z.B. `BankAccount.java` und `BankAccountTest.java`
  im Verzeichnis `accountInt` und
  sei `accountInt` im Verzeichnis `account3`.
  Dann wird `junit.jar` im Verzeichnis `account3` installiert.



Übersetzen und Ausführen von Tests

- **Übersetzung der Testklasse:**

unter Windows:

```
javac -source 1.4 -classpath .;junit.jar
  accountInt\BankAccountTest.java
```

unter UNIX:

```
javac -source 1.4 -classpath .:junit.jar
  accountInt/BankAccountTest.java
```

- **Ausführen der Testklasse**

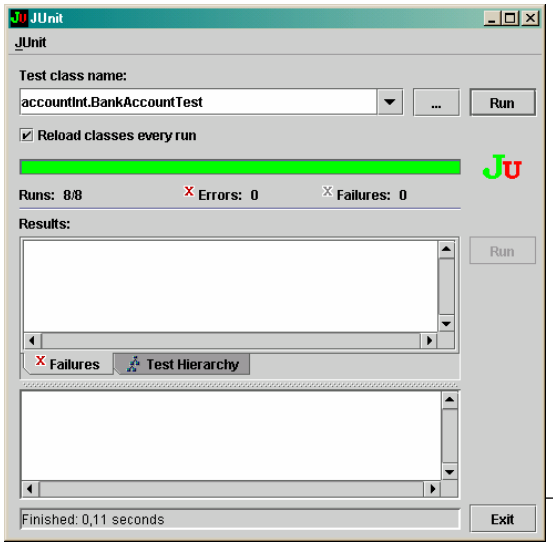
unter Windows:

```
>java -ea -classpath .;junit.jar accountInt.BankAccTest
```

unter UNIX:

```
>java -ea -classpath .:junit.jar accountInt.BankAccTest
```

Ausgabefenster von JUnit



- Erfolgreicher Testlauf von BankAccount: Kein Fehlgeschlagener Test gefunden!

Schema für eine Testklasse der Klasse ZZZ (1)

```
import junit.framework.*;

public class ZZZTest extends TestCase
   {    private ZZZ z;

   public ZZZTest(String name)
   {    super(name);
   }

   protected void setUp()
   {    z = new ZZZ();
   }

   protected void tearDown()
   {    z = null;
   }
```


Schema für eine Testklasse der Klasse ZZZ (2)

Fortsetzung:

```
public void testZZZMethod()
   {    assertTrue(<Boole'scher Wert>);
        assertEquals(expected, actual);
   }

public static void main(String args[])
   {    junit.swingui.TestRunner.run(ZZZTest.class);
   }
}
```


Testoperationen

Die Klasse **Assert** bietet folgende Methoden:

Zeigt keinen Fehler an, wenn

- `static void assertTrue(boolean b)` — b wahr ist
- `static void assertFalse(boolean b)` — b falsch ist
- `static void assertEquals(Object expected, Object actual)`
  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`
- ...

Testfixture

- Eine **Fixture** (JUnit-Jargon) ist eine Menge von Objekten, die den gemeinsamen Ausgangszustand für die Testfälle einer Testklasse darstellt.
- Durch eine **Testfixture vermeidet** man **Codeduplikation** der gemeinsamen Testobjekte mehrerer Testmethoden einer Testklasse..
- Tests können die Objekte einer Testfixture gemeinsam benutzen, wobei jeder Test unterschiedliche Methoden aufrufen und unterschiedliche Resultate erwarten kann. **Jeder Test** einer Testklasse **verwendet seine eigene Fixture**, um die Tests von den Änderungen anderer Tests zu isolieren. Deshalb können die Tests einer Testklasse in jeder Reihenfolge ausgeführt werden.
- Eine Testfixture wird durch die **setUp() Methode** erzeugt; durch die tearDown() Methode werden diese Objekte wieder beseitigt. JUnit ruft die setUp-Methode automatisch vor jedem Test und die tearDown-Methode automatisch nach jedem Test auf.

Testschema für Ausnahmen

Ein Ausnahme im getesteten Code wird folgendermaßen in der Testklasse abgefangen und behandelt

```
public void testForException()
{ try
      {   YYY o = z.m();
          fail("Should raise an ZZZException");
      } catch (ZZZException success)
      { <assert über den Zustand vor dem Auslösen der Ausnahme
              in  z.m()>
      }
}
```

Signalisiert den Fehlschlag des Tests, d.h. dass KEINE Ausnahme ausgelöst wurde

Beispiel: Test auf überzogenes Kontolimit

Eine nichterfüllte Vorbedingung von withdraw in `BankAccount` wird in `BankAccountTest` folgendermaßen abgefangen und behandelt:

```
public void testWithdrawOverLimit()
{     int balance = b1.getBalance();
      try
      {      b1.withdraw(200);
             fail("IllegalArgumentException expected");
      } catch (IllegalArgumentException e)
      {      assertEquals(balance, b1.getBalance());
      }
}
```

Signalisiert, dass KEINE Ausnahme ausgelöst wurde, obwohl das Kontolimit überzogen ist

Testen von behandelten Ausnahmen

Tests für Ausnahmen, die im getesteten Code abgefangen werden, benötigen keine spezielle Form (siehe `BankAccountTest`):

```
public void testTransferToNegativeAmount()
{
      int balance1 = b1.getBalance();
      int balance2 = b2.getBalance();
      assertFalse(b1.transferTo(b2,-30));
      assertEquals(balance1,b1.getBalance());
      assertEquals(balance2,b2.getBalance());
}
```

Aufruf von transferTo führt zu Behandlung der Ausnahme in transferTo mit Resultat `false` . Siehe `BankAccount`

Testsuite

- Eine Testsuite ist eine Menge von Testfällen, die gemeinsam ausgeführt und betrachtet werden.
- Typischerweise testet man in einer Testsuite mehrere Klassen oder ein gesamtes Package.
- Wichtige Operationen der Klasse TestSuite:

`TestSuite(ZZZTest.class)`	Konstruktor konvertiert Testklasse in Testsuite
`static Test suite()`	gibt eine Instanz von TestSuite oder von TestCase zurück
`addTestSuite(ZZZTest.class)`	fügt eine Testklasse zu einer Suite hinzu

Testsuite für BankAccount und SavingsAccount

Die Klasse AllTests konstruiert eine Testsuite aus den Testklassen für BankAccount und SavingsAccount und führt alle diese Tests aus:

```
import junit.framework.*;
public class AllTests
{   public static Test suite()
    {    TestSuite suite = new TestSuite(BankAccountTest.class);
         suite.addTestSuite(SavingsAccountTest.class)
         return suite;
    }
    public static void main(String args[])
    {    junit.swingui.TestRunner.run(AllTests.class);
    }
}
```


Test-gesteuerter Entwurf

- Neue Software-Enwurfstechniken stellen das Testen vor das Implementieren des Programms:

  Exteme Programming, Test-first Programming, Agile Software Development
- Schritte des Test-gesteuerten Entwurfs:
  1. Erstelle UML-Diagramm
  2. Entwerfe einen Test für eine Methode
  3. Schreibe möglichst einfachen Code, bis der Test nicht mehr fehlschlägt
  4. Wiederhole 2. und 3. bis alle Methoden des Klassendiagramms implementiert sind.
- Dabei wird häufig der Code (und manchmal der Test) restrukturiert („Refactoring“). Jedes Mal werden alle Tests durchgeführt, um sicher zu stellen, dass die Coderestrukturierung nicht zu Fehlern im „alten“ Code geführt hat.

Kurze Iterationen, in denen abwechselnd Code und Test geschrieben wird.

Zusammenfassung

- Beim Testen unterscheidet man zwischen Unit-, Modul-, Integrations-, System- und Feldtest.
- Beim Whitebox-Test werden Tests anhand der Programmstruktur entworfen, beim Blackbox-Test zählt nur die Spezifikation.
- JUnit ist ein Framework zur automatischen Ausführung von Unit-Tests
- Beim Test-gesteuerten Entwurf werden zuerst die UML-Diagramme und die Tests entworfen und dann die Programme geschrieben.