
FACHHOCHSCHULE WÜRZBURG SCHWEINFURT
FACHBEREICH INFORMATIK UND WIRTSCHAFTSINFORMATIK

JUnit 4.1

Veranstaltung: **Fortgeschrittenes Programmieren mit Java**

Angefertigt von: **Anatol Zund & Mircea Dumitru**

INHALTSVERZEICHNIS

1	TESTEN.....	3
1.1	Einleitung [Newbies]	3
1.2	Herleitung der Testfälle [Smarties].....	6
1.3	Vorgehensmethoden zur Softwareentwicklung [Smarties].....	7
2	JUNIT 4.1 – EINFÜHRUNG [NEWBIES].....	8
2.1	Was ist JUnit?	8
2.2	Installation von JUnit und Integration in Eclipse.....	9
3	JUNIT 4.1 – BEISPIEL [NEWBIES].....	10
3.1	Importanweisung	10
3.2	Klasse.....	11
3.3	@Before / @ After	11
3.4	Einzelner Testfall	11
3.5	Asserts	12
3.6	Exceptions	13
4	WEITERFÜHRENDE TESTS [SMARTIES].....	14
4.1	Integrationstest (Cactus).....	14
4.2	Funktionale Tests (HTTPUnit).....	14
4.3	Load/Performance Testing (JMeter)	14
4.4	Load Testing (JUnitPerf)	14
5	ANHANG.....	15
5.1	Quellenverzeichnis	15
5.2	Klassen.....	15

1 Testen

JUnit in einem Satz:

JUnit ist ein Java-Framework für die Erstellung von Unit Tests.

Definition JUnit:

JUnit ist ein [Framework](#) zum [Testen](#) von [Java](#)-Programmen, das besonders für automatisierte [Unit-Tests](#) einzelner *Units* (meist [Klassen](#) oder [Methoden](#)) geeignet ist. Es basiert auf Konzepten, die ursprünglich unter dem Namen [SUnit](#) für [Smalltalk](#) entwickelt wurden.[†]

1.1 Einleitung [Newbies]

1.1.1 Warum Testen?

In der heutigen Zeit, wo Softwareprodukte immer komplexer werden, ist vor allem die Qualität einer Software sehr wichtig. Da aber komplexe Software in begrenzter Zeit nur unvollständig erfassbar und realisierbar ist enthält jede Software Fehler.

Um diese Fehler zu aufzudecken werden Tests notwendig, die eine teilweise hohe Softwarequalität sicherstellen können.

Ziel eines Tests ist somit eine Software mit der Absicht zu prüfen, Fehler zu finden.

Ein Test sollte:

- Fehler so früh wie möglich finden
- unabhängig erfolgen
- mehr bringen als kosten
- automatisiert wiederholbar sein
- möglichst zeitnah zur Programmierung sein
- Spaß machen

Ein Test sollte nicht:

- keine Fehler finden

Somit kann ein Test nicht die Korrektheit eines Software beweisen, sondern nur die Anwesenheit von Fehlern.

[†] <http://de.wikipedia.org/wiki/JUnit>

1.1.2 Softwaretestarten

Tests können bzgl. eines Testobjekts in verschiedene Testarten klassifiziert werden:

Unit –bzw. Modul Tests
Integrationstests
Performance Tests
Funktionale –und Akzeptanz Tests

Unit –bzw. Modul Tests

Unit –bzw. Modul Tests dienen zur Verifikation der Korrektheit der einzelnen Softwarebausteine (Module; je nach Programmiersprache z. B. Klassen). Durch Ablauf aller Testfälle soll nach Programmfehlern gesucht werden. Hierzu wird ein ausführbarer Code erzeugt der die einzelnen Testfälle durchspielt und die Ergebnisse mit den erwarteten Werten vergleicht. Dabei prüft jeder Testfall immer nur ein mögliches Verhalten des Moduls ab, um hinterher genau feststellen zu können, wodurch ein Fehler erzeugt wurde. Unit Tests sollten zeitnah zur Programmierung erstellt und nach jeder Programmmodifikation ausgeführt werden.

Integrationstests

Integrationstests prüfen die korrekte Interaktion voneinander abhängiger Komponenten in einem komplexen System. Dazu müssen die zu prüfenden Komponenten den Unit-Test erfolgreich abgeschlossen haben um zu zeigen, dass sie isoliert fehlerfrei funktionieren.

Performance Tests

Performance Tests dienen dazu das Verhalten einzelner Module oder Systeme bezüglich einer definierten Hardwareumgebung zu Testen. Im Mittelpunkt steht dabei, ob das System die vom Kunden gewünschte Reaktionszeit einhält und auch unter hoher Auslastung keine Fehler produziert.

Funktionale Tests

Funktionale Tests dienen dazu, die vom Kunden gewünschte Funktionalität des Systems unter Real-Bedingungen zu prüfen. Dabei wird getestet ob das entwickelte System den Anforderungen des Auftraggebers oder späteren Benutzers entspricht.

1.1.3 Unit Testarten

Die weitere Klassifizierung von Unit Tests basiert auf drei verschiedene Unit Testarten:

- Logischer Unit Test
- Integrations Unit Test
- Funktionaler Unit Test

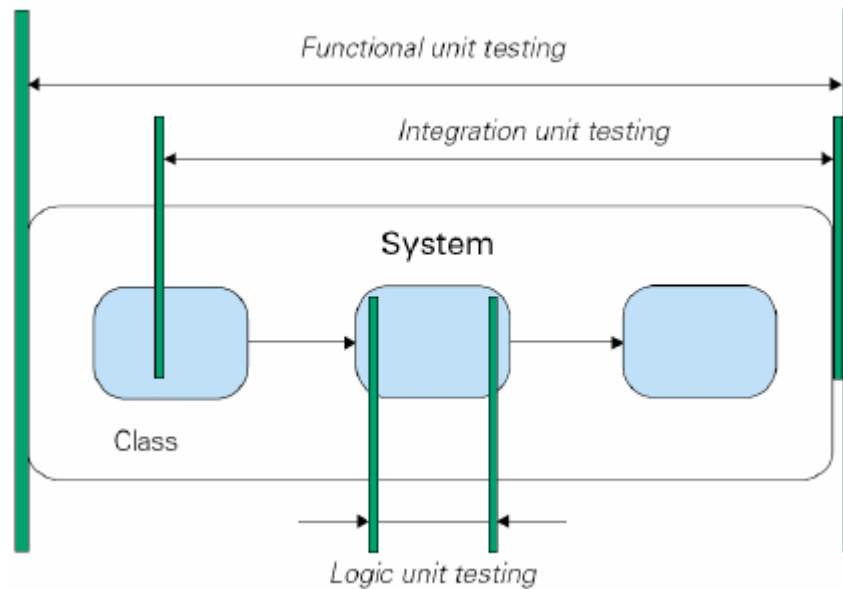


Abbildung 1: Unit Testarten

Logischer Unit Test

Der Logische Unit Test überprüft nur einzelne Methoden bestimmter Klassen eines Softwaresystems.

Integrations Unit Test

Der Integrations Unit-Test überprüft das Zusammenspiel zwischen den einzelnen Komponenten eines Softwaresystems.

Funktionaler Unit Test

Der Funktionaler Unit-Test erweitert die Grenzen des Logischen Unit Tests und Integrations Unit Test so, dass Arbeitsabläufe getestet werden können.

Die einzelnen Arten von Unit Tests können als *White-Box-Test* oder als *Black-Box-Test* durchgeführt werden.

White-Box-Test

Der White-Box-Test wird von den gleichen Programmieren entwickelt und durchgeführt wie das zu testende System selbst. Dabei besteht ein Kenntnis über die innere Struktur des Testobjektes.

Black-Box-Test

Der Black-Box-Test wird von speziellen Testabteilungen entwickelt und durchgeführt. Dabei besteht kein Kenntnis über den inneren Aufbau des zu testenden Objektes.

1.2 Herleitung der Testfälle [Smarties]

Die wichtigsten Bausteine, um erfolgreich zu Testen, sind die richtigen Testfälle. Es nutzt wenig, wenn man 20 Tests durchführt die im Endeffekt das Selbe Verhalten abprüfen, man dafür jedoch ein anderes Verhalten komplett übersieht.

Für die Herleitung der Testfälle gibt es verschiedene Methoden, die wir hier auszugsweise aufzeigen wollen.

1.2.1 Äquivalenzklassenmethode

Bei der Äquivalenzklassenmethode werden Eingabeparameter eines Moduls oder einer Methode, die das gleiche Verhalten haben, in Klassen eingeteilt. Dazu wird die Spezifikation des Moduls oder der Methode analysiert.

Beispiel:

Man betrachtet ein Modul, das nur Geldbeträge zwischen 0,01€ und maximal 500€ verarbeiten soll. Nun kann man alle Werte zwischen 0,01 und 500 zu einer Klasse zusammenfassen. Man wählt aus diesem Bereich dann einen Wert zum Testen aus. Ist der Test mit diesem Wert erfolgreich kann man davon ausgehen, dass er auch für alle anderen Werte der Klasse erfolgreich verläuft.

In diesem Beispiel kann man 3 Äquivalenzklassen bilden:

- $0,01 \leq x \leq 500$
- $x \leq 0$
- $x > 500$

Prüft man aus diesen Bereichen jeweils einen Wert ab, kann man das Verhalten des Moduls testen.

1.2.2 Grenzwertanalyse

Die Grenzwertanalyse ist eine Erweiterung der Äquivalenzklassenmethode. Zusätzlich wird hier besonderes Augenmerk auf die Grenzen der Spezifikationen gelegt, da die Fehler sich in diesem Bereich häufen.

In unserem Beispiel würde man die 3 Testfälle für die Äquivalenzklassen um folgende Testfälle erweitern:

- -0,01
- 0,00
- 0,01
- 500,00
- 500,01

1.3 Vorgehensmethoden zur Softwareentwicklung

[Smarties]

1.3.1 Testgetriebene Entwicklung

Von testgetriebener Entwicklung spricht man, wenn bei der Softwareentwicklung der Testplan vor der eigentlichen Implementierung erstellt wird. Während bei der klassischen Vorgehensweise der Test erst nach Fertigstellung des Moduls/Systems erstellt wird.

Nachteile wie mangelnde Testbarkeit, Zeitmangel, unzureichende Tests werden bei dieser Methode vermieden.

Testgetriebene Entwicklung arbeitet mit Tests auf zwei Ebenen.

Testen im Kleinen

Diese Tests beziehen sich auf die einzelnen Module des Systems. Hierbei wird der Testplan während der Entwicklung mit entwickelt. Man teilt die Implementierung in kleine Portionen auf z.B. in einzelne Methoden. Vor jeder Implementierung einer Methode werden die dazu gehörenden Testfälle erstellt. Dann wird (noch vor der Implementierung der Methode) ein Testlauf durchgeführt, der fehlschlägt. Nun wird die Methode implementiert und anschließend wieder getestet. Dies wiederholt sich so lange, bis alle Testfälle positiv beendet werden. Danach räumt man den Sourcecode auf und kümmert sich um die Kommentare. Ein abschließender Testlauf stellt sicher, dass man dabei keine Änderungen an der Funktion der Methode vorgenommen hat.

Testen im Großen

Diese Tests beziehen sich auf das gesamte System. Die Testfälle lassen sich aus den Spezifikationen ableiten und werden schon vor der Systemimplementierung aufgestellt. Mit Hilfe dieser Tests kann geprüft werden, ob das System den Anforderungen des Kunden genügt.

Vorteile

Die Vorteile der testgetriebenen Entwicklung liegen in der verringerten Fehleranzahl (besonders bei Änderungen) und einer besseren Messbarkeit der Anforderungen. Da nur korrekte Module gespeichert werden, kann jeder Programmierer meistens auf „korrekte“ Module zugreifen um mit ihnen zu arbeiten. Des Weiteren fördert diese Vorgehensweise das „**Think first – code later**“-Prinzip, da man sich vor der Implementierung schon Gedanken über die Testfälle und damit auch über mögliche Probleme macht.

2 JUnit 4.1 – Einführung [Newbies]

2.1 Was ist JUnit?

JUnit ist ein kleines und mächtiges Open Source Java-Framework zur Durchführung und Automatisierung von Unit Tests als Black-Box-Tests.

Da die Tests direkt in Java programmiert werden, ist das Testen mit JUnit so einfach wie das Kompilieren. Die Testfälle sind selbstüberprüfend und damit wiederholbar.

JUnit wurde 1998 von Erich Gamma und Kent Beck entwickelt und basiert auf SUnit zum Testen von Smalltalk Programmen.

JUnit liegt mittlerweile in der Version 4.1 vor und hat folgende Änderungen zur Version 3.8:

- ist einfacher geworden (bei TestSuites)
- setzt Java-5.0-Features ein und voraus: Static imports, Annotationen.
- beim neuesten Eclipse (3.2, Build Date: 29. Juni 2006) schon dabei

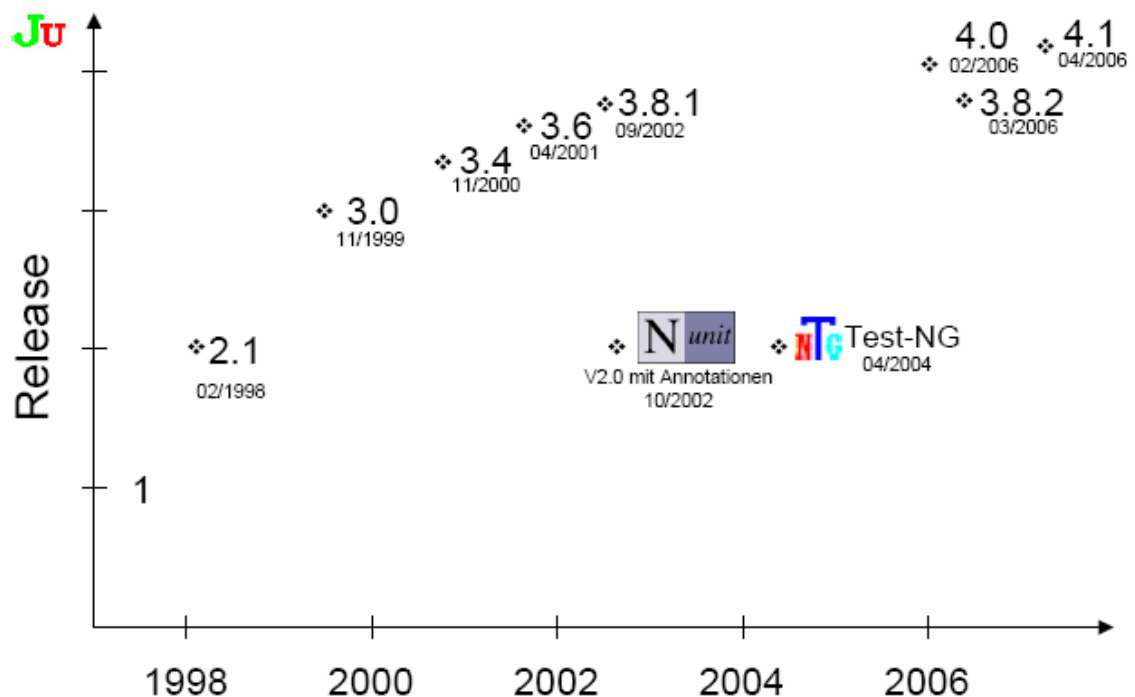


Abbildung 2: JUnit Releases

Ein JUnit-Test kennt nur zwei Ergebnisse:

Entweder der Test gelingt („grün“) oder er misslingt („rot“). Das Misslingen kann als Ursache einen Fehler (*Error*) oder ein falsches Ergebnis (*Failure*) haben.

2.2 Installation von JUnit und Integration in Eclipse

Installation unter Eclipse 3.2

Braucht man nicht; ist wie schon erwähnt drin enthalten.

Installation unter Eclipse 3.1--

Die aktuelle Version (JUnit 4.1) kann von SourceForge bezogen werden :
<http://sourceforge.net/projects/junit/>.

Die Installation ist einfach:

Das heruntergeladene Zip-File junit4.1.zip entpacken und dann das darin enthaltene Jar-Archiv junit4.1.jar zum CLASSPATH hinzufügen.

Prüfen ob alles geklappt hat:

Menü: Window → Preferences → weiter siehe Abbildung 3

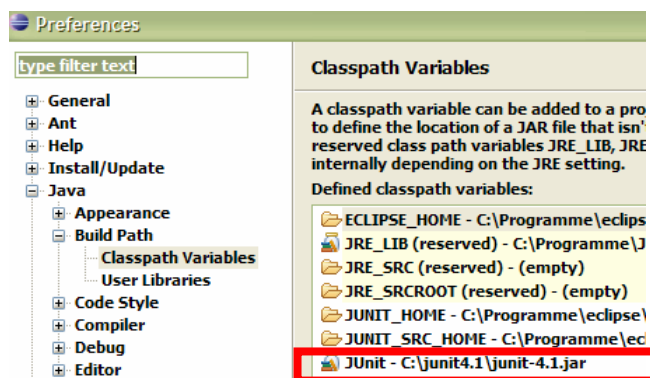


Abbildung 3: Jar-Archiv junit4.1.jar in Eclipse

Integration in Eclipse

In Eclipse hat JUnit eine hervorragende Integration gefunden. Es wurde die Testklasse im *Package Explorer* selektiert und anschließend »Run As« und »JUnit Test« im Kontextmenü auswählen.

In jedem Fall sollte sich der JUnit-View zeigen:

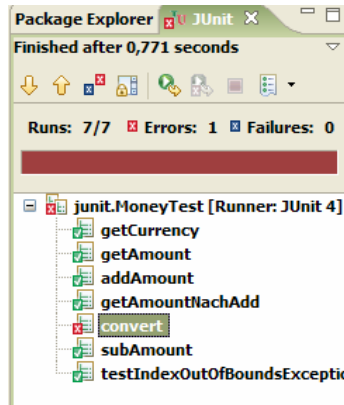


Abbildung 4: Integration von JUnit in Eclipse

3 JUnit 4.1 – Beispiel [Newbies]

Das folgende einführende Beispiel verwendet JUnit 4.1. Im Internet werden aber die meisten Beispiele noch auf der Basis von JUnit 3.8 durchgeführt. Da sich die Syntax zwischen den Versionen etwas unterscheidet kann es hier zu Missverständnissen kommen. Man kann jedoch in 3.8 entwickelte Tests auch auf einer 4.0 Umgebung ausführen.

Für das einführende Beispiel wird die Testklasse *MoneyTes* (s. Anhang) und die dazugehörige Klasse *Money* (s. Anhang) verwendet.

Eine Testklasse enthält eine Menge von Testmethoden, die die verschiedenen Methoden der Ausgangsklasse testet.

In der Testklasse *MoneyTest* werden nun die grundlegenden Funktionen, welche von der Klasse *Money* angeboten werden, abgeprüft.

3.1 Importanweisung

Jede Testklasse muss die benötigten JUnit Klassen importieren. Dies geschieht durch

```
import static org.junit.Assert.*;
import org.junit.*;
```

Natürlich kann man statt der * auch die einzelnen Klassen angeben die man wirklich benötigt (z.B. Test, After, Before, etc.)

3.2 Klasse

Die Testklasse ist grundlegend genau so aufgebaut wie jede andere Javaklasse auch.

```
public class MoneyTest
{
    //Definition der verwendeten Objekte

    // Testfälle
}
```

3.3 @Before / @ After

- @Before-Methoden werden vor jedem Testfall ausgeführt
- @After-Methoden nach jedem Testfall

Benötigen die Tests größere Initialisierungsschritte, so kann man diese mit @Before Auslagern anstatt sie in jedem einzelnen Testfall zu schreiben.

Zusätzlich wird dann jedoch noch ein **import org.junit.Before;** benötigt (entfällt wenn man org.junit.* verwendet).

Beispiel:

```
@Before public void setUp()
{
    usa = new Money("USD", 100);
    europa = new Money("EUR", 100);
}
```

Natürlich müssen die im @Before-Block initialisierten Objekte der ganzen Klasse bekannt sein.

Mit @After kann man Befehle zusammenführen die nach jedem Test ausgeführt werden sollen. Z.B. Aufräumen der Datenbanken etc.

3.4 Einzelner Testfall

Jeder Test muss einzeln implementiert werden. Dies erfolgt durch die einleitende Zeile

```
@Test public void hierDerTestName ()
```

Es hat sich dabei als nützlich erwiesen den Namen des Tests an die zu testende Methode anzupassen.

Die einzelnen Tests sind voneinander Unabhängig.
Für jeden der Tests werden die Testumgebungen neu initialisiert.

Beispiel:

Das Money-Objekt europa wird über die @Before Methode (s.u.) mit dem Betrag 100 initialisiert.

```
/** Prüfen ob der Geldbetrag richtig addiert wird*/
@Test public void addAmount ()
{
    europa.addAmount (50.00);
    assertEquals("addAmount 50 -> 150",europa.getAmount (),
                150.00);
}

/** Testen nach Add im Test zuvor der Betrag wieder 100 ist*/
@Test public void getAmountNachAdd ()
{
    assertEquals("Test auf Unabhängigkeit", europa.getAmount (),
                100.00);
}
```

Beide Testfälle passieren ohne Probleme, da nach dem ersten Test das Objekt wieder mit 100 neu initialisiert wurde.

3.5 Asserts

Um nun die einzelnen Ergebnisse zu überprüfen muss jede zu prüfende Bedingung mit Asserts definiert werden. Mit einem Assert wird geprüft ob z.B. ein Attribut der zu testenden Klasse mit vorher festgelegten Werten übereinstimmt.

Je nach dem Ergebnis gilt der Test dann als Bestanden oder nicht Bestanden.

Folgende Asserts stehen in JUnit zur Verfügung:

- assertEquals
- assertTrue - assertFalse
- assertNull - assertNotNull
- assertSame - assertNotSame

Neben dem zu prüfenden Wert kann den einzelnen Asserts noch einen Namen gegeben werden. Dies erleichtert die Analyse anschließend, da als Rückgabe der Name des fehlgeschlagenen Tests erhalten wird.

Beispiel:

```
@Test public void getAmount () {
    assertEquals("getValue 100", europa.getAmount (), 100.00);
}
```

Hier wird geprüft ob das Money-Objekt europa den Betrag 100.00 zurückgibt, den es bei der Initialisierung erhalten hat.

3.6 Exceptions

Um zu Testen, ob die zu testende Klasse auch mit Fehlern umgehen kann, wurde die Möglichkeit geschaffen die Klasse auf zu erwartende Exceptions zu testen. Es wird dabei im Testcode absichtlich eine Exception erzeugt und geprüft, ob die zu testende Klasse auch die entsprechende Exception zurückgibt. Sollte keine oder ein nicht erwartete Exception zurückkommen gilt der Test als nicht bestanden.

Beispiel:

```
/** Prüft ob Exception geworfen wird.Hat nichts mir der Money Klasse
zu tun zeigt aber für die Verwendung von Exceptions.
*/
@Test( expected = IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException()
{ ArrayList emptyList = new ArrayList();
  @SuppressWarnings("unused")
  Object o = emptyList.get(0);
}

/*Dieser Testfall schlägt fehl, da wir eine andere Exception erwarten
als wir bekommen haben*/

@Test (expected = IllegalArgumentException.class)
public void testWrongException()throws IndexOutOfBoundsException
{
  ArrayList emptyList = new ArrayList();
  @SuppressWarnings("unused")
  Object o = emptyList.get(0);
}
```

4 Weiterführende Tests [Smarties]

Dieses Kapitel widmet sich weiterführenden Tests, die auch über den Rahmen von JUnit hinausgehen.

4.1 Integrationstest (Cactus)

Cactus erlaubt es einem Integrationstests von Servlets und JSPs durchzuführen. Dabei wird das zu testende System auf dem Zielsystem ausgeführt um sein Verhalten in der Laufzeitumgebung zu prüfen. Es erweitert JUnit und verwendet im Moment noch JUnit 3.8.1.

Die Testfälle sind gleich aufgebaut wie bei JUnit selbst. Man kann die Tests sowohl manuell über die Komandozeile, als Plugin in einer IDE, oder im Webbrowser ausführen. Als Ergebnis bekommt man eine Zusammenfassung als XML-Datei die fehlgeschlagene Tests aufzeigt, oder aber eine Erfolgsmeldung zurückgibt.

4.2 Funktionale Tests (HTTPUnit)

HttpUnit ist eine Erweiterung für JUnit um automatisierte Website-Tests durchzuführen. Es ermöglicht die zurückgegebene Seite zu analysieren und z.B. dort enthaltenen Links zu folgen. Somit kann das Surf-Verhalten von Usern simuliert werden. Zusätzlich kann man auch Formulare oder JavaScript ausführen.

4.3 Load/Performance Testing (JMeter)

Bei JMeter handelt es sich um ein umfangreiches Testtool um Systeme unter großer Last zu testen. Hierzu erzeugt das Tool je nach Vorgabe eine große Anzahl von Threads, die alle auf das gleiche System zugreifen und misst dabei z.B. die Latenzzeit oder den Datendurchsatz. Dabei unterstützt es unter Anderem folgende Tests:

- http, ftp
- Datenbanken über JDBC
- Java
- JUnit
- Webservices
- LDAP

Da dieses Tool die Möglichkeit hat die Tests über mehrere Rechner zu verteilen kann damit auch ein Lasttest gegen größere Maschinen durchgeführt werden.

4.4 Load Testing (JUnitPerf)

Mit JUnitPerf kann man erweitert man JUnit um die Funktion Performance-Parameter abtesten zu können. So kann man vorgegebene Reaktionszeiten messen und testen. Zusammen mit einem Lastentest mit JMeter kann das System unter realen bzw. extremen Bedingungen getestet und somit verifiziert werden.

5 Anhang

5.1 Quellenverzeichnis

Prof. Dr. Spielmann: Skript zur Vorlesung Software-Qualitätsmanagement

<http://de.wikipedia.org/wiki/Softwareetest>

http://junit.sourceforge.net/doc/faq/faq.htm#tests_9

<http://www.frankwestphal.de/UnitTestingmitJUnit.html>

<http://www.frankwestphal.de/JUnit4.0.html>

<http://www.frankwestphal.de/ftp/Einleitung.pdf>

<http://www.frankwestphal.de/ftp/UnitTestsmitJUnit.pdf>

<http://www.instrumentalservices.com/media/articles/java/junit4/JUnit4.pdf>

www.pst.ifi.lmu.de/lehre/SS06/infoII/folien/Folien12aTestJUnitSS06.pdf

www.java-forum-stuttgart.de/folien/D5_Hiller.pdf

<http://ebus.informatik.uni-leipzig.de/www/media/lehre/seminar-javatools05/semtools05-meder-text.pdf>

http://www.pst.informatik.uni-muenchen.de/lehre/SS06/infoII/zentralfolien/kw28_junit4.pdf

5.2 Klassen

5.2.1 Klasse Money.java

```
package junit;

public class Money {

    private String currency;
    private double wert;

    //Umrechnungsfaktoren
    private final double EURUSD = 1.2833;
    private final double USDEUR = 0.7792;

    // Konstruktor
    // c = Währungskürzel, w = Betrag
    public Money(String c, double w)
    {
        this.currency = c;
        this.wert = w;
    }
}
```

```
/** gibt das Währungskürzel zurück
 * @return String Währungskürzel */
public String getCurrency()
{
    return this.currency;
}

/** gibt den aktuellen Geldbetrag zurück
 * @return double*/
public double getAmount()
{
    return this.wert;
}

//      BERECHNUNGEN
/** fügt dem Wert den Betrag a hinzu
 * @param a*/
public void addAmount(double a)
{
    this.wert = this.wert + a;
}

/** zieht vom Betrag den Wert a ab.*/
public void subAmount(double a) throws MoneyException
{
    double t = this.wert - a;

    if (t < 0.00)
    {
        MoneyException e = new MoneyException();
        throw e;
    }
    else
    {
        this.wert = this.wert - a;
    }
}

/** Rechnet den aktuellen Währungsbetrag in eine anderes um und gibt
 *diesen Wert zurück. Das Währungskürzel des Objekts wird dabei NICHT
 *geändert. @param z @return
 */
public double convert(String z)
{
    double w = 0.00;
    double f = 0.00;

    if (z == "USD")
        f = EURUSD;
    else if (z == "EUR")
        f = USDEUR;

    w = this.wert * f;
    w = this.roundScale2(w);

    return w;
}
```

```
//    UMWANDLUNGEN für Kürzel
/** Wandelt die aktuelle Währung in das neue WKürzel um und gibt es
zurück */
public String currencyString(String c)
{    String cs = null;

    if (c.equals("EUR"))
        cs = "USD";
    else if (c.equals("USD"))
        cs = "EUR";

    return cs;
}

//    RUNDUNG
/** Rundet auf 2 Nachkommastellen
 * @param d
 * @return */
public double roundScale2( double d )
{    return Math rint( d * 100 ) / 100.00;
}

}
```

5.2.2 Klasse TestMoney.java

```
package junit;

import static org.junit.Assert.*;
import org.junit.*;

import java.util.ArrayList;

public class MoneyTest
{    private Money usa;
    private Money europa;

    /** Anlegen eines Money-Objekts für jede Währung
     * Dieser Part wird vor jedem einzelnen Test ausgeführt.
     * Somit hat jeder Test die gleichen Ausgangsbedingungen.*/
    @Before public void setUp()
    {    usa = new Money("USD",100);
        europa = new Money("EUR",100);
    }

    // Hier fangen die eigentlichen Testfälle an
    /** Prüfen ob das Währungskürzel richtig zurückgegeben wird*/
    @Test public void getCurrency()
    {    String c = europa.getCurrency();
        String d = usa.getCurrency();

        assertEquals("getCurrency EUR",c,"EUR");
        assertEquals("getCurrency USD",d,"USD");
    }
}
```

```
/**Prüft ob der Geldbetrag (Amount) richtig zurückgegeben wird
 * Da es double Werte sind muss 100.00 geprüft werden*/
@Test public void getAmount()
{
    assertEquals("getValue 100", europa.getAmount(), 100.00);
}

/** Prüfen ob der Geldbetrag richtig addiert wird*/
@Test public void addAmount()
{
    europa.addAmount(50.00);
    assertEquals("addAmount 50-> 150", europa.getAmount(), 150.00);
}

/** Prüfen ob der Geldbetrag richtig subtrahiert wird*/
@Test public void subAmount()
{
    try
    {
        europa.subAmount(50);
    }
    catch (MoneyException e)
    {
        e.printStackTrace();
    }

    assertEquals("Ziehe 50 von 100
ab", europa.getAmount(), 50.00);
}

/** Prüft ob die Umrechnung der einzelnen Währungen richtig
durchgeführt wird*/
@Test public void convert()
{
    assertEquals("convert Euro to
USD", europa.convert("USD"), 128.33);
    assertEquals("convert USD to
Euro", usa.convert("EUR"), 77.92);
}

/** Prüft ob Exception geworfen wird.
 * Hat nichts mit der Money Klasse zu tun zeigt aber für
 * die Verwendung von Exceptions. */
@Test( expected = IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException()
{
    ArrayList emptyList = new ArrayList();
    @SuppressWarnings("unused")
    Object o = emptyList.get(0);
}

/*Dieser Testfall schlägt fehl, da wir eine andere Exception
erwarten als
wir bekommen haben*
@Test ( expected = IllegalArgumentException.class)
public void testWrongException()throws IndexOutOfBoundsException
{
    ArrayList emptyList = new ArrayList();
    @SuppressWarnings("unused")
    Object o = emptyList.get(0);
}
*/
}
```