

Test Driven Development mit Java

Teil 1

Gesamtinhaltsverzeichnis

1	Einführung	1-3
1.1	Einige Vorurteile	1-4
1.2	Einige Begriffe	1-5
1.2.1	Automatisierung von Tests	1-5
1.2.2	Komponententest (Unit-Test)	1-5
1.2.3	Integrations- / Interaktionstests	1-5
1.2.4	Akzeptanz-Tests	1-6
1.2.5	White-Box / Black-Box-Tests	1-6
1.2.6	Testabdeckung	1-6
1.3	Test-Driven Development	1-7
1.4	Ein kleines Beispiel	1-8
1.5	Resultate und Konsequenzen	1-15
1.6	Grundlegendes zum vorliegenden Skripts	1-16
1.7	Aufbau des Skripts	1-17
1.8	Organisation der Beispiel-Projekte	1-18
2	Testing – Ein Miniframework	2-3
2.1	Manuelles Testen	2-4
2.2	Asserts	2-7
2.3	Runner	2-13
2.4	Reflected Runner	2-14
3	Testing – Das JUnit Test-Framework	3-3
3.1	Start	3-5
3.2	Exceptions	3-9
3.3	Before-After	3-13
3.4	Assertions	3-19
3.5	Test-Suites	3-21
3.6	Parameter	3-22
3.7	Private Methods	3-23
3.8	Assert That	3-25
3.9	Erweiterungen von JUnit	3-28
4	Testing – Übungen	4-3
4.1	Isbn	4-4
4.2	Trimmer	4-7

4.3	Event-Bus.....	4-9
4.4	Typesafe Properties	4-13
4.5	Resultate	4-17
4.5.1	Sprechende Namen.....	4-17
4.5.2	Unabhängigkeit.....	4-17
4.5.3	Strukturelle Einfachheit.....	4-18
4.5.4	Vollständigkeit.....	4-18
4.5.5	Unabhängigkeit von problematischen Ressourcen.....	4-20
4.5.6	Test als Spezifikation und Dokumentation	4-20
5	Mocking – Ein Miniframework.....	5-3
5.1	Start.....	5-5
5.2	Dynamic Proxies.....	5-12
5.3	Call-Objekte.....	5-16
5.4	Assertions.....	5-20
5.5	Arrangements.....	5-23
5.6	Ein Eclipse-Test.....	5-29
6	Mocking – Mockito	6-3
6.1	XMLWriter	6-4
6.2	Pythagoras	6-10
6.3	Spezialitäten.....	6-14
7	Mocking – Übungen	7-3
7.1	AccountService.....	7-4
7.2	GroupChanger.....	7-10
8	Test Driven Development: Beispiel	8-3
8.1	Parser für numerische Expressions.....	8-4
8.2	Entwurf logischer Schaltungen	8-8
8.3	CSV-Dateien	8-12
9	Refactoring: Beispiel Gruppenwechsel	9-3
9.1	Gruppenwechsel.....	9-5
10	Spezielle Test-Werkzeuge	10-3
10.1	Testen von DB-Anwendungen: DbUnit.....	10-4
10.2	Testen von DB-Anwendungen: Simple Tool.....	10-10
10.3	Testen von Swing-Anwendungen: FEST	10-13
10.4	Testen von WEB-Anwendungen: Selenium.....	10-19
10.5	Testen von EJB-Anwendungen: OpenEJB.....	10-28
10.6	Testen mit FIT	10-35

11	Testen von Multithreading-Anwendungen.....	11-3
11.1	Utility-Klassen.....	11-4
11.2	Queue.....	11-6
11.2.1	FixedQueue	11-6
11.2.2	FixedQueueTest	11-7
11.2.3	Future	11-13
11.2.4	FutureImpl.....	11-13
11.2.5	FutureTest	11-14
11.3	Executor	11-15
11.3.1	ExecutorTest.....	11-17
11.4	Nodes	11-20
11.4.1	Node	11-20
11.4.2	NodeTest	11-23
12	Anhang.....	12-3
12.1	Reflection	12-4
12.2	Dynamic Proxy	12-7
13	Literatur	13-3

1

Einführung

1.1	Einige Vorurteile	1-4
1.2	Einige Begriffe	1-5
1.2.1	Automatisierung von Tests	1-5
1.2.2	Komponententest (Unit-Test)	1-5
1.2.3	Integrations- / Interaktionstests	1-5
1.2.4	Akzeptanz-Tests	1-6
1.2.5	White-Box / Black-Box-Tests	1-6
1.2.6	Testabdeckung	1-6
1.3	Test-Driven Development	1-7
1.4	Ein kleines Beispiel	1-8
1.5	Resultate und Konsequenzen	1-15
1.6	Grundlegendes zum vorliegenden Skripts	1-16
1.7	Aufbau des Skripts	1-17
1.8	Organisation der Beispiel-Projekte	1-18

1 Einführung

Die Einführung beginnt mit einigen grundsätzlichen Bemerkungen zum Thema "Test" und "testgetriebener Software-Entwicklung".

Ein kleines Beispiel wird dann in Thema TDD einführen: in mehreren aufeinander aufbauenden Schritten wird eine Datencontainer-Klasse entwickelt: eine Klasse namens `IntArray`.

Schließlich werden der Aufbau dieses Skripts und der Aufbau des Eclipse-Workspaces beschrieben, in denen die Beispielprojekte enthalten sind.

Zunächst aber zwei Zitate von Kent Beck, dem großen Philosophen der testgetriebenen Software-Entwicklung (siehe den Literaturhinweis am Ende des Skripts).

Kent Beck beschreibt den "rhythm of Test-Driven Development":

1. *Quickly add a test.*
2. *Run all tests and see the new one fail.*
3. *Make a little change.*
4. *Run all tests and see them all succeed.*
5. *Refactor to remove duplication.*

Und ein etwas längeres Zitat:

The goal is clean code that works...

Clean code that works is out of the reach of even the best programmers some of the time, and out of the reach of most programmers (like me) most of the time.

Divide and conquer...

First we'll solve the "that works" part of the problem. Then we'll solve the "clean code" part.

This is the opposite of architecture-driven development, where you solve "clean code" first, then scramble around trying to integrate into the design the things you learn as you solve the "that works" problem.

1.1 Einige Vorurteile

Das Thema "Testen von Software" ist mit einer Reihe von Vorurteilen verbunden (die folgende Aufzählung ist dem Buch von Johannes Link entnommen: "Softwaretests mit JUnit"):

- Ich habe keine Zeit zum Testen.
- Testen von Software ist langweilig und stupide.
- Mein Code ist praktisch fehlerfrei, auf jeden Fall gut genug.
- Die Testabteilung testet. Die können das eh viel besser.

Hier einige einfache Klarstellungen:

Fast jeder Code enthält Fehler. Wer keine Zeit zum Testen hat, entdeckt diese Fehler nicht. Der Code ist instabil. Es werden sich Fehlermeldungen häufen. Die Bearbeitung dieser Fehlermeldungen dauert (Debugging etc.). Der Versuch, Zeit zu sparen, führt zum Gegenteil: dass mehr Zeit benötigt wird.

Testen – richtig verstanden – ist alles andere als langweilig und stupide. Testen ist ebenso anspruchsvoll wie die Erstellung des eigentlichen Programmcodes. Außerdem kann man besser schlafen, wenn die Software, die man erstellt hat, die Tests bestanden hat.

Jeder Code wird irgendwann einmal angefasst. Die Funktionalität von Software wird erweitert oder geändert. Wer garantiert, dass solche Änderungen und Erweiterungen keine unerwünschten Seiteneffekte haben?

Der Systemtest – und nur damit kann sich die Testabteilung befassen – kann keine Komponententests (Unit-Tests) ersetzen. Die Komponenten können nur von demjenigen getestet werden, der sie selbst geschrieben hat. Solche Komponenten müssen isoliert vom Gesamtsystem getestet werden können. Wird auf solche Unit-Tests verzichtet, können Fehler recht kostspielig werden – Fehler, die beim Unit-Test frühzeitig und viel billiger behoben werden könnten.

Resultate:

- Die Zeit, die zum Testen benötigt wird, ist keine verlorene Zeit. Testen hilft dabei, Zeit zu sparen.
- Testen von Software ist anspruchsvoll und kreativ.
- Code mag heute fehlerfrei sein; er sollte auch nach einer Änderung / Erweiterung weiterhin fehlerfrei sein.
- Units können nur vom Entwickler selbst getestet werden.

1.2 Einige Begriffe

1.2.1 Automatisierung von Tests

Ein Test soll zeigen, dass die tatsächlich von der Software produzierten Ergebnisse den erwarteten Ergebnissen entsprechen – oder eben: nicht entsprechen. Im ersten Falle sind die Tests erfolgreich, im letzteren Falle sind sie fehlgeschlagen (auch dies ist zunächst ein durchaus "positives" Resultat). Man kann die von der Software produzierten Ergebnisse natürlich bei jedem Test "manuell" mit den erwarteten Ergebnissen vergleichen. Es leuchtet aber sofort ein, dass solche Tests nicht praktikabel sind. U.a. muss man Tests wiederholt ausführen können. Sie sollten also derart geschrieben werden, dass sich automatisch ablaufen können – dass also der Vergleich zwischen den tatsächlichen und den erwarteten Ergebnissen automatisiert ist. Dann können solche Tests natürlich auch ohne großen Aufwand beliebig häufig wiederholt werden.

1.2.2 Komponententest (Unit-Test)

Ein Komponententest bezieht sich auf eine isolierte Komponente. Was dabei als Komponente verstanden wird, ist natürlich Definitionssache. Es kann sich um eine einzelne Methode handeln, um eine Klasse, um ein Subsystem. Diese Einheiten sollten auf jeden Fall unabhängig voneinander getestet werden können.

Ein Interpreter z.B. enthält eine Scanner- und eine Parser-Komponente. Zunächst sollte der Scanner unabhängig von anderen Bestandteilen des Interpreters getestet werden können. Auch der Parser sollte unabhängig von weiteren Bestandteilen getestet werden können. Aber der Parser wird abhängig sein vom Scanner. Also wird der Parser-Test einen erfolgreichen Scanner-Test voraussetzen... (Oder sollte – zum Testen des Parsers – der Scanner nicht eher "simuliert", "gemockt" werden?) Komponenten sollten von dem Entwickler dieser Komponenten getestet werden.

1.2.3 Integrations- / Interaktionstests

Integrationstest beziehen sich auf das Zusammenspiel aller Komponenten eines Systems. Sie setzen erfolgreiche Komponententests voraus. Auch solche Tests sollten natürlich automatisiert ablaufen. Die Abstände, in denen solche Tests laufen, sind zwar größer als beim Unit-Test (dort sollten die Intervalle möglichst klein sein), sollten aber dennoch relativ klein sein (z.B. jeden Tag ein Integrations-Test).

1.2.4 Akzeptanz-Tests

Akzeptanz-Tests geben dem Kunden resp. dem Management das für die Überwachung des Projektfortschritts nötige Feedback. Sie werden vom Kunden spezifiziert – nur dieser weiß, was die Software "nach außen hin" leisten muss. Es geht hier also um die Funktionalität des Gesamtsystems aus Sicht des Benutzers. Auch solche Akzeptanz-Tests können teilweise automatisiert werden. Dabei ist es natürlich wieder der Entwickler, der die Spezifikation des Kunden in solche automatische Tests technisch umsetzen muss.

1.2.5 White-Box / Black-Box-Tests

Beim White-Box-Test wird die Kenntnis der internen Implementierung eines Moduls (einer Unit) vorausgesetzt. Solche Tests werden u.a. dann verwendet, wenn es um das Mocken von Objekten geht, von welchen die zu testende Unit abhängig ist (siehe hierzu die Kapitel zum Thema Mocking). Beim Black-Box-Tests wird dagegen von der Implementierung einer Unit komplett abgesehen. Es wird nur das nach außen sichtbare Verhalten einer Unit getestet.

1.2.6 Testabdeckung

Hier geht's um die Frage, wann ein Test "ausreichend" ist. Ein Test eines komplexen Systems kann – sofern seine Kosten vertretbar sein sollen - niemals komplett vollständig sein. Es geht dann um die Frage, was getestet werden soll. Z.B.: Welche möglichen Verzweigungen des Codes müssen getestet werden (alle können nicht getestet werden). Welche grenzwertigen Situationen müssen getestet werden?

1.3 Test-Driven Development

Das Thema Test-Driven Development ist eng verknüpft mit den Konzepten "Extreme Programming (XP)" resp. "Agile Softwareentwicklung".

XP geht davon aus, dass ein komplexes System ohnehin nicht vollständig entworfen und spezifiziert werden kann. Solche Systeme können nur evolutionär entwickelt werden. Und dabei gewinnt die eigentliche Programmierung wieder einen zentralen Stellenwert: war sie bislang nur als "Hilfstätigkeit" begriffen worden, welche einfach nur eine gegebene Spezifikation umsetzt, so wird sie nun der eigentliche Dreh- und Angelpunkt der Entwicklung.

Das System wird in relative keine Einzelteile zerlegt, welche in einem überschaubaren Rahmen (1 bis 3) Wochen implementiert werden können. Diese Einheiten werden wiederum in kleinere Einheiten zerlegt, welche unmittelbar implementiert werden. Am Resultat dieser Implementierung kann unmittelbar ihr Erfolg oder ihr Misserfolg abgelesen werden. Und zu dieser Implementierung – das ist entscheidend – gehört immer auch das gleichzeitige Schreiben (oder das Anpassen!) von Test-Code. Die Implementierung kann somit immer sofort verifiziert werden. "Ohne diesen Test gilt auch die Implementierung als nicht vorhanden" (Joh. Link).

Die Entwicklung der Komponenten geschieht in folgenden Schritten:

Zunächst wird Test-Code geschrieben. Dieser Code wird genau diejenigen Klassen benutzen, welche zum Schreiben des Test-Codes noch gar nicht existieren. Das erscheint zunächst merkwürdig: Wie kann ein Test einer Klasse geschrieben werden, welche noch nicht existiert? Natürlich kann er geschrieben werden! – er kann nur nicht kompiliert werden. Aber indem der Test-Code geschrieben wird, wird doch überhaupt erst klar, was die zu schreibende Klasse eigentlich leisten soll.

Dann wird die zu testende Klasse geschrieben – mit minimalem Aufwand. Es geht zunächst einmal nur darum, den Compiler zufriedenzustellen. Der Test-Code ist dann übersetzbar. Aber seine Ausführung wird scheitern. Im nächsten Schritt wird die zu testende Klasse derart erweitert (wiederum mit minimalem Aufwand), dass der Test gelingt. Dann wird der Test-Code erweitert – mit dem erklärten Ziel, seine Ausführung wieder scheitern zu lassen. Die zu testende Klasse wird erweitert, damit der Test wieder gelingt. Der Test-Code wird um neue Funktionalitäten erweitert. Die oben beschriebene Schrittfolge wird erneut durchlaufen.

Dieses Konzept sei im Folgenden an einem einfachen Beispiel erläutert (ein Beispiel demonstriert mehr als lange Romane!).

1.4 Ein kleines Beispiel

Es soll eine Klasse `IntArray` entwickelt werden. Der erste Test-Code könnte wie folgt aussehen:

```
IntArray list = new IntArray();  
list.add(20);  
list.add(40);  
assertEquals(2, list.size());  
assertEquals(20, list.get(0));  
assertEquals(40, list.get(1));
```

`assertEquals` sei eine Utility-Methode, welcher zwei Parameter übergeben werden: das erwartete und das tatsächliche Ergebnis. Entspricht das tatsächliche Ergebnis dem erwarteten Ergebnis, kehrt `assertEquals` lautlos zurück. Fall nicht, wirft `assertEquals` eine `Exception` – wir wissen dann, dass der Test fehlgeschlagen ist.

Aufgrund des obigen Tests wissen wir nun, dass ein `IntArray` offenbar ein "Collection"-Objekt ist, welches beliebig viele `int`-Werte speichern können soll. Wir wissen weiterhin, dass mittels des Aufrufs der `add`-Methode ein `int`-Wert zu einem `IntArray` hinzugefügt werden soll. Wir wissen, dass `Add` einen Parameter vom Typ `int` haben muss. Wir wissen weiterhin, dass `size` eine Methode sein wird, welche die Anzahl der Elemente eines `IntArrays` liefert. Und wir wissen schließlich, dass mittels der `get`-Methode die Werte einer `ArrayList` abfragbar sein sollen – und dass dieser `get`-Methode der Index des Elements übergeben werden muss und dass sie einen `int`-Wert zurückliefern muss.

Anders gesagt: indem wir den Test-Code schreiben, spezifizieren(!) wird die zu entwickelnde Klasse. Und diese Spezifikation ist ersten genauer als verbale "Romane". Und zweitens: sie ist nicht "abstrakt", sondern "ausführbar": wenn denn die Klasse `ArrayList` irgendwann existiert, kann ihre Korrektheit sofort mittels dieses Test-Codes verifiziert werden. Der Test-Code dient also nicht nur zum Test bereits fertiger Software, sondern zugleich auch zu deren Spezifikation.

Der nächste Schritt besteht dann darin, den Test-Code kompilierbar zu machen. Es geht hier darum, die einfachste `IntArray`-Klasse zu schreiben, die sich denken lässt – deren Existenz dann aber den Test-Code kompilierbar macht. Also z. B.:

```
public class IntArray {  
    public void add(int element) { }  
    public int size() { return 0; }  
    public int get(int index) { return 0; }  
}
```

Dies ist die einfachste Implementierung. Sie ist natürlich "falsch": die `add`-Methode kehrt einfach tatenlos zurück; die `size`-Methode liefert

immer 0; und die `get`-Methode ignoriert ihren Parameter und liefert ebenfalls einfach immer 0.

Wichtig ist: der Compiler ist zufrieden. Er übersetzt die Klasse und auch den Test-Code. Die Ausführung des Test-Codes wird natürlich scheitern – bereits beim ersten `assertEquals`. Das Scheitern ist beabsichtigt.

Der nächste Schritt könnte nun darin bestehen, mit den einfachsten Mitteln sicherzustellen, dass die Ausführung des Test-Codes gelingt. Hier die zu diesem Zweck erweiterte `IntArray`-Klasse:

```
public class IntArray {
    public void Add(int element) {
    }
    public int size() {
        return 2;
    }
    public int get(int index) {
        return index == 0 ? 20 : 40;
    }
}
```

Das ist natürlich "getürkt". Aber der obige Test-Code läuft einwandfrei durch.

Dann wird der Test-Code erweitert – derart, dass der darauffolgende Testlauf wieder scheitern wird:

```
IntArray list = new IntArray();
list.add(20);
list.add(40);
list.add(80);
assertEquals(3, list.size());
assertEquals(20, list.get(0));
assertEquals(40, list.get(1));
assertEquals(80, list.get(2));
```

Weiterhin "türken" ist nun natürlich sinnlos. Also benötigt man eine einigermaßen "realistische" Implementierung der `IntArray`-Klasse (man beachte, dass die folgende Version aber natürlich noch nicht der Weisheit letzter Schluss sein kann!):

```
public class IntArray {
    private int[] elements = new int[3];
    private int count;
    public void add(int element) {
        this.elements[this.count] = element;
        this.count++;
    }
    public int size() {
        return this.count;
    }
    public int get(int index) {
        return this.elements[index];
    }
}
```

Der neue Testlauf ist wieder erfolgreich.

Und wird wissen, wie der Test-Code derart erweitert werden kann, dass der Test wieder scheitern wird (wir planen also das Scheitern ein!):

```
IntArray list = new IntArray();  
list.add(20);  
list.add(40);  
list.add(80);  
list.add(90);  
assertEquals(4, list.size());  
assertEquals(20, list.get(0));  
assertEquals(40, list.get(1));  
assertEquals(80, list.get(2));  
assertEquals(90, list.get(3));
```

Damit ist klar, dass wir uns dem Problem der Reallokation zuwenden müssen (die in der Klasse `IntArray` erzeugten Tabelle mit der Größe 4 zu dimensionieren, wird letztlich nicht viel helfen...):

```
class IntArray {  
    private int[] elements = new int[3];  
    private int count;  
    public void Add(int element) {  
        if (this.elements.Length == this.count) {  
            int[] newElements = new int[this.count * 2];  
            for (int i = 0; i < this.count; i++)  
                newElements[i] = this.elements[i];  
            this.elements = newElements;  
        }  
        this.elements[this.count] = element;  
        this.count++;  
    }  
    public int size() {  
        return this.count;  
    }  
    public int get(int index) {  
        return this.elements[index];  
    }  
}
```

Der Test-Code wird erfolgreich ausgeführt werden können.

Wir planen das nächste Scheitern:

```
IntArray list = new IntArray();
list.add(20);
list.add(40);
list.add(80);
list.add(90);
assertEquals(4, list.size());
assertEquals(20, list.get(0));
assertEquals(40, list.get(1));
assertEquals(80, list.get(2));
assertEquals(90, list.get(3));
try {
    list.get(4);
    fail();
}
catch(RuntimeException e) {
    // everything okay!
}
```

`fail` sei eine Utility-Methode, die immer eine `RuntimeException` wirft.

`list.get(4)` wird 0 liefern – offenbar ein falsches Ergebnis (das 5-te Element wurde nie hinzugefügt). Dieser Aufruf sollte eigentlich eine `Exception` liefern (und genau das wird im obigen Test-Code verlangt – also: spezifiziert).

Die Klasse `IntArray` wird wieder nur soweit geändert, dass der nächste Test gelingt:

```
public class IntArray {

    // wie gehabt...

    public int Get(int index) {
        if (index >= this.count)
            throw new RuntimeException("...");
        return this.elements[index];
    }
}
```


Der `IntArray` soll mehr können: statt immer nur neue Werte ans Ende des `IntArrays` anzuhängen, möchte man einen Wert an einer beliebigen Stelle einfügen können:

```
IntArray list = new IntArray();
list.add(20);
list.add(40);
list.add(80);
list.add(90);
list.insert(1, 11);
list.insert(1, 22);
list.insert(1, 33);
assertEquals(7, list.Count);
assertEquals(20, list.get(0));
assertEquals(33, list.get(1));
assertEquals(22, list.get(2));
assertEquals(11, list.get(3));
assertEquals(40, list.get(4));
assertEquals(80, list.get(5));
assertEquals(90, list.get(6));
try {
    list.get(7);
    fail();
}
catch(Exception e) {
    // everything okay!
}
```

Zunächst muss der Compiler zufriedengestellt werden:

```
class IntArray {
    // wie gehabt...

    public void insert(int index, int element) {
    }
}
```

Der Compiler übersetzt nur den Test-Code. Aber die Ausführung dieses Codes scheitert (was beabsichtigt ist!).

Also muss `Insert` nun "richtig" implementiert werden:

```
class IntArray {
    // wie gehabt...

    public void insert(int index, int element) {
        // Platz machen
        for (int i = this.count; i > index; i--)
            this.elements[i] = this.elements[i - 1];
        // Element in die "Luecke" einfuegen
        this.elements[index] = element;
    }
}
```

Leider (und das war eigentlich nicht geplant...) scheitert der Test-Code. Wir haben nicht berücksichtigt, dass auch beim `Insert` evtl. eine Reallokation stattfinden muss. Also eine kleine Erweiterung:

```
class IntArray {  
    // wie gehabt...  
  
    public void Insert(int index, int element) {  
        if (this.elements.Length == this.count) {  
            int[] newElements = new int[this.count * 2];  
            for (int i = 0; i < this.count; i++)  
                newElements[i] = this.elements[i];  
            this.elements = newElements;  
        }  
        // Platz machen  
        for (int i = this.count; i > index; i--)  
            this.elements[i] = this.elements[i - 1];  
        // Element in die "Luecke" einfuegen  
        this.elements[index] = element;  
    }  
}
```

Der Test ist nun erfolgreich.

Was aber kein Grund ist, die Hände in den Schoß zu legen. Denn die obige Erweiterung wurde per Copy&Paste erzeugt. Der Code enthält also Redundanz: in der `insert`-Methode befinden sich dieselben Zeilen wie in der `add`-Methode. Ein solcher Code "riecht schlecht". Also ist Refactoring angesagt. Refactoring bedeutet, die interne Struktur des Codes zu verbessern, ohne seine Funktionalität zu ändern.

Hier das Ergebnis des Refactorings:

```
class IntArray {  
    // wie gehabt...  
  
    public void Add(int element) {  
        this.ensureCapacity();  
        this.elements[this.count] = element;  
        this.count++;  
    }  
  
    public void Insert(int index, int element) {  
        this.ensureCapacity();  
        // Platz machen  
        for (int i = this.count; i > index; i--)  
            this.elements[i] = this.elements[i - 1];  
        // Element in die "Luecke" einfuegen  
        this.elements[index] = element;  
    }  
  
    private void ensureCapacity() {  
        if (this.elements.Length == this.count) {  
            int[] newElements = new int[this.count * 2];  
            for (int i = 0; i < this.count; i++)  
                newElements[i] = this.elements[i];  
            this.elements = newElements;  
        }  
    }  
}
```

Der Test-Code läuft weiterhin einwandfrei.

Wir beenden unser Demonstrations-Beispiel – obwohl noch einiges zu tun wäre (sowohl im Test-Code als auch in der zu testenden Klasse). Was wäre denn noch zu tun?...

1.5 Resultate und Konsequenzen

Die testgetriebene Entwicklung besteht aus einem Zyklus von Spezifikation (per "Test-Anwendung") und Implementierung. Dabei stellt sich der Entwickler abwechselnd auf zwei Positionen: auf die Position des Benutzers einer Klasse und auf die Position des Implementierens einer Klasse. Der Entwickler ist also kein "bornierter" Implementierer – er hat zunächst die Benutzung im Auge. Und eine Klasse ist nicht Selbstzweck – sie ist dazu da, benutzt zu werden.

Ein Entwickler, der mit der Implementierung beginnt und die Nutzung der Klasse aus den Augen verliert, wird möglicherweise eine Klasse entwickeln, die nur umständlich zu nutzen ist. Vielleicht ist die Klasse auch unvollständig. Solche Mängel können natürlich am besten vermieden werden, wenn der Entwickler zunächst an die Benutzung der Klasse denkt – und nicht nur an diese Benutzung denkt, sondern sie auch – für den Compiler verständlich und übersetzbar – aufschreibt.

Indem der Entwickler die Benutzung der Klasse demonstriert, spezifiziert er die Klasse. Diese Spezifikationen (also die Testmethoden) erzählen jeweils eine kleine "Story", aus welcher genau hervorgeht, wie die Klasse zu benutzen ist. Jemand, der die Klasse nutzen will, muss dann eigentlich nur die Testmethoden lesen. Anhand dieser Methoden kann er die Art und Weise der möglichen Nutzung exakt verstehen.

Jeder Schritt der Entwicklung einer Klasse ist abgesichert. Es kann also niemals Zweifel daran aufkommen, dass der aktuelle Implementierungsstand korrekt ist. (Was aber nicht notwendigerweise bedeutet, dass die Implementierung vielleicht später zu refaktorisieren ist. Und auch nicht, dass die aktuelle Spezifikation auf ewig Bestand hat...)

Beim Wechsel von "rot" zu "grün" kann der Entwickler sich "zurücklehnen" – die zuletzt vorgenommenen Erweiterungen / Änderungen haben sich definitiv als zielführend herausgestellt. Der Entwickler wird daher das "grün" jeweils auch als Belohnung und Bestätigung seiner Arbeit auffassen können. Und gewöhnlich lieben es Menschen, bestätigt zu werden...

Der Vorteil von Testmethoden erweist sich insbesondere dann, wenn die Klassen später geändert oder erweitert werden müssen – der einfach refakturiert werden sollen. Nach jedem Schritt einer solchen Erweiterung / Änderung / Refaktorisierung können die Tests erneut ausgeführt werden. Ist das Ergebnis dann "grün", kann der Entwickler sicher sein, nichts "kaputt repariert" zu haben. Und nach der Auslieferung der neuen Software kann der Entwickler ruhig schlafen. (Psychologie ist nicht zu unterschätzen...)

Auch ein Test-Client ist genau so ernst zu nehmen wie ein "richtiger" Client einer Klasse. Auch er braucht Pflege (immer dann, wenn die Spezifikation geändert oder erweitert wird).

1.6 Grundlegendes zum vorliegenden Skript

Um testgetrieben entwickeln zu können, benötigen wir Werkzeuge. Wir benötigen als allererstes ein Werkzeug für den Black-Box-Test von "Units" (eine Unit ist im einfachsten Falle eine einzelne Klasse). Weiterhin benötigen wir ein Mocking-Werkzeug – ein Werkzeug, welches das Verhalten einer Klasse simulieren kann, welche von einer anderen, der zu testenden Klasse benutzt wird (es geht um das sog. "endoskopisches Testen"). Und abhängig von der zu erstellenden Anwendung werden wir weitere Spezialwerkzeuge nutzen: Werkzeuge zum Test von GUIs (seien sie nun Web- oder Desktop-basiert), Werkzeuge zum Testen von Datenbank-Zugriffsklassen etc. Die wichtigsten dieser Werkzeuge sind aber diejenigen, die Unit-Tests und Mocking ermöglichen.

Als Werkzeug zum Unit-Test wird in diesem Skript das im Eclipse integrierte JUnit-Testframework vorgestellt (und benutzt). Für das Mocking wird Mockito verwendet werden. Für GUI-Tests wird das FEST-Framework (für Swing-Anwendungen) und Selenium (für Servlet-basierte-Anwendungen) verwendet. Das JUnit-Testframework und Mockito werden recht ausführlich erläutert werden – was die Spezialwerkzeuge angeht, so wird nur deren grundlegende Funktionsweise erläutert. Alles andere kann der offiziellen Dokumentation dieser Werkzeuge entnommen werden (allerdings sind diese Dokumentationen i.d.R. äußerst dürftig...).

Die Funktionsweise von Testwerkzeugen – insbesondere diejenige von Mocking-Werkzeugen – erscheint häufig auf den ersten Blick geheimnisvoll: als "magic". Jemand, der ein Werkzeug benutzt, sollte aber zumindest dessen grundlegende Funktionsweise begriffen haben (und es nicht einfach unverstanden und dumm nutzen) – zumal dann, wenn der Benutzer ein Software-Entwickler ist, welcher Software-Werkzeuge nutzt. Neben dem eigentlichen Thema: "Testgetriebene Entwicklung" geht es im vorliegenden Skript vor allem auch darum, den Werkzeugen ihre Magie zu nehmen – zu zeigen, dass auch die Entwickler dieser Tools nur mit Wasser gekocht haben. M.a.W.: Bevor z.B. das JUnit-Testframework vorgestellt wird, wird dieses Framework zunächst einmal mit den Bordmitteln von Java "nachgebaut" – selbstredend natürlich nur der "harte Kern" dieses Werkzeugs. Bevor Mockito vorgestellt wird, wird auch dieses Werkzeug in seinem Kern nachgebaut. Ein Entwickler, der weiß, wie seine Werkzeuge tatsächlich funktionieren, wird diese Werkzeuge auch sicher und professionell nutzen können – und auch Verhalten dieser Werkzeuge interpretieren können, welches vielleicht zunächst unverständlich ist.

Beim "Nachbau" dieser Werkzeuge werden insbesondere folgende Java-Features genutzt: Annotations, Reflection und Dynamic Proxies. Natürlich werden diese Features nur insoweit vorgestellt, als sie zum Verständnis der Test-Tools unbedingt erforderlich sind – die Darstellung wird sich also auf die Essentials dieser Features beschränken.

1.7 Aufbau des Skripts

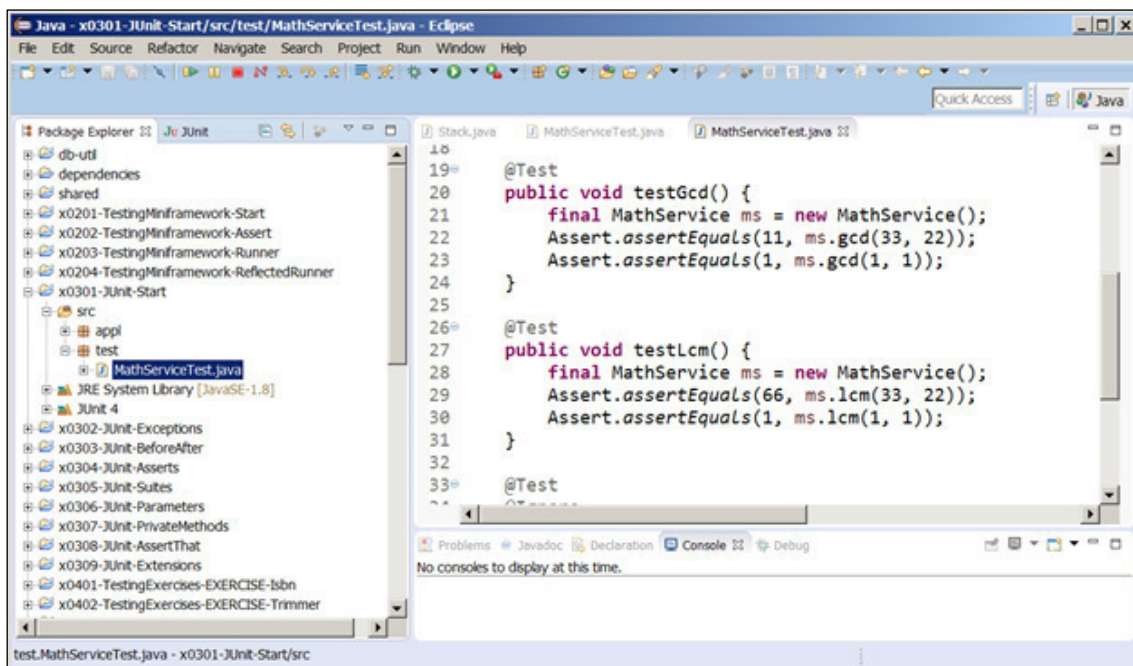
- Im Kapitel 2 wird der Unit-Test vorgestellt. Dabei wird nebenbei ein kleines Werkzeug entwickelt, mit welchem die Tests ausgeführt werden.
- Im Kapitel 3 wird das JUnit-Testframework vorgestellt. (Als Alternative könnte z.B. auch TestNG: auch dieses Werkzeug basiert im Grunde auf denselben Prinzipien.)
- Kapitel 4 enthält einige Übungen, in denen es darum geht, Klassen "im nachhinein" zu testen. Anhand dieser Übungen sollte ein Gespür dafür entwickelt werden können, wie brauchbare Testklassen beschaffen sind.
- Kapitel 5 zeigt, was es mit Mocking auf sich hat. Auch hier wird nebenbei ein kleines Mocking-Werkzeug entwickelt.
- Im Kapitel 6 wird dann eines der professionellen Mocking-Werkzeuge vorgestellt: Mockito. (Man könnte z.B. auch JMock oder EasyMock nutzen.)
- Kapitel 7 enthält einigen Übungen zum Thema Mocking.
- Im Kapitel 8 geht's schließlich um die "Philosophie" und Praxis der testgetriebenen Software-Entwicklung – also um das eigentliche Kernthema des Seminars. Das Kapitel beschreibt drei Aufgaben, die mittels TDD schrittweise gelöst werden können. Im Seminar werden allerdings nicht alle drei Aufgaben gelöst werden können – die Zeit ist zu knapp. Zumindest eine der drei Aufgaben sollte aber ausführlich behandelt werden. Zu allen drei Aufgaben existieren fertige Lösungen, die in einer separaten Workspace implementiert und in einem separaten Skript beschrieben sind.
- Das Kapitel 9 widmet sich dem Thema "Refactoring und Test".
- Im Kapitel 10 geht's um einige Spezialwerkzeuge – zum Testen von Datenbank-basierten Anwendungen, zum Testen von Swing- und Servlet-basierten Anwendungen und zum Testen von EJB-Anwendungen.
- Im Kapitel 11 schließlich geht's um das Testen von Multithreading-Anwendungen.
- Das Kapitel 14 ("Anhang") führt in die Grundlagen von Reflection, Annotations und Dynamic-Proxies ein.
- Das Kapitel 15 schließlich enthält einige Literatur-Hinweise.

1.8 Organisation der Beispiel-Projekte

Die Beispielprogramme befinden sich in der Datei `Java-TDD.zip`. Diese Datei kann an beliebiger Stelle im Dateisystem extrahiert werden. Als Resultat entsteht ein Ordner mit dem Namen `Java-TDD`. Dieser enthält ein Unterverzeichnis `projects`, welcher die Beispielprojekte enthält. Er enthält ein weiteres Unterverzeichnis namens `dependencies`, in welchem sich die verwendeten Tools befinden.

Zu jedem Abschnitt eines jeden Kapitels dieses Skripts existiert im dem `projects`-Verzeichnis ein entsprechendes Projekt: z.B. `x0201-TestingMiniframework-Start`, `x0301-JUnit-Start` etc. Die Namen (und die Nummerierung) dieser Projekte entsprechen der Kapitel-/ Abschnitts-Nummerierung in diesem Skript.

Ein Ausschnitt aus dem Workspace:



2

Testing – Ein Miniframework

2.1	Manuelles Testen	2-4
2.2	Asserts	2-7
2.3	Runner.....	2-13
2.4	Reflected Runner.....	2-14

2 Testing – Ein Miniframeframework

Anhand einer einfach zu testenden Beispielklasse wird in diesem Kapitel schrittweise ein kleines Testwerkzeug entwickelt. Dabei werden bereits zentrale Aspekte des Testens diskutiert werden können.

Die zu testende Klasse `MathService` enthält zwei Methoden: die erste berechnet den größten gemeinsamen Teiler (`gcd`), die zweite das kleinste gemeinsame Vielfache (`lcm`) zweier Ganzzahlen:

```
package appl;

public class MathService {

    public int gcd(int x, int y) {
        if (x <= 0 || y <= 0)
            // throw new IllegalArgumentException("bad arguments");
            return -1;
        while (x != y) {
            if (x > y)
                x -= y;
            else
                y -= x;
        }
        return x;
    }

    public int lcm(final int x, final int y) {
        if (x <= 0 || y <= 0)
            // throw new IllegalArgumentException("bad arguments");
            return -1;
        int a = x;
        int b = y;
        while (a != b) {
            if (a < b)
                a += x;
            else
                b += y;
        }
        return a;
    }
}
```

Beide Methoden prüfen zunächst die übergebenen Argumente. Sind diese Argumente nicht positiv, liefern sie im Augenblick den Wert `-1` zurück. Sie sollten aber eigentlich eine `IllegalArgumentException` liefern. Dieser Fehler in der Implementierung sollte natürlich beim Testen erkannt werden. Ansonsten sind die Algorithmen korrekt (mit einer kleinen Ausnahme: bei `lcm` könnte ein Überlauf stattfinden...)

2.1 Manuelles Testen

Wir können nun eine Testklasse `MathServiceTest` schreiben, welche aus mehreren kleinen Methoden besteht: `testGcd`, `testLcm`, `testGcdException` und `testLcmException`:

```
package test;

import appl.MathService;

public class MathServiceTest {

    public void testGcd() {
        final MathService ms = new MathService();
        System.out.println("\t" + ms.gcd(33, 22));
        System.out.println("\t" + ms.gcd(1, 1));
    }

    public void testLcm() {
        final MathService ms = new MathService();
        System.out.println("\t" + ms.lcm(33, 22));
        System.out.println("\t" + ms.lcm(1, 1));
    }

    public void testGcdException() {
        final MathService ms = new MathService();
        try {
            ms.gcd(0, 1);
            System.out.println("\terror");
        }
        catch (final IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }

    public void testLcmException() {
        final MathService ms = new MathService();
        try {
            ms.lcm(-1, 1);
            System.out.println("\terror");
        }
        catch (final IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }
    // ...
}
```

In jeder dieser Methoden wird zunächst ein neues `MathService`-Objekt erzeugt. Natürlich könnten auch alle Testmethoden auf einem einzigen ("globalen") `MathService`-Objekt operieren – aber nur deshalb, weil `MathService`-Objekte zustandslos sind. Bei zustandsbehafteten Objekten aber muss sichergestellt werden, dass zu Anfang jeder Testmethode das zu testende Objekt seinen Initialzustand besitzt. Sollen also solche "stateful objects" getestet werden, kommt man nicht umhin, in jeder

Testmethode ein neues Objekt zu erzeugen. Und das, was bei stateful objects unumgänglich ist, kann bei stateless objects sicher nicht falsch sein. Unabhängig davon, ob stateful oder stateless objects getestet werden sollen, sollte am Anfang jeder Testmethode also ein neues Testobjekt erzeugt werden.

Hinweis: Davon zu reden, dass "Objekte" getestet werden, ist natürlich strenggenommen falsch. Wir wollen natürlich Klassen testen. Aber zum Testen von Klassen müssen diese gewöhnlich instanziiert werden (es sei denn, es sollen nur statische Methoden von Klassen getestet werden)...

Im Hauptprogramm wird für die Ausführung jeder der Testmethoden ein neues Objekt der Testklasse erzeugt (um somit von vornherein auszuschließen, dass die Testmethoden voneinander abhängig sind – um also etwas auszuschließen, dass eine Testmethode vom Resultat einer anderen Testmethode abhängig ist). Natürlich ist dann auch die Reihenfolge der Testmethoden-Aufrufe völlig gleichgültig:

```
package test;

public class MathServiceTestRunner {
    public static void main(final String[] args) {
        System.out.println("testGcd");
        new MathServiceTest().testGcd();

        System.out.println("testLcm");
        new MathServiceTest().testLcm();

        System.out.println("testGcdException");
        new MathServiceTest().testGcdException();

        System.out.println("testLcmException");
        new MathServiceTest().testLcmException();
    }
}
```

(Hinweis:Das JUnit-Testframework verhält sich genauso: für die Ausführung jeder Testmethode wird ein neues Objekt der Testklasse erzeugt.)

Hier die Ausgaben:

```
testGcd
    11
    1
testLcm
    66
    1
testGcdException
    error
testLcmException
    error
```

Die Ausführung der Methoden `testGcd` und `testLcm` liefern offenbar die erwarteten Ergebnisse (die wir hoffentlich irgendwo auf Papier hinterlegt haben...)

Die Ausführung von `testGcdException` und `testLcmException` zeigt aber, dass sich die zu testenden Methoden (`lcm` und `gcd`) im Falle fehlerhafter (illegaler) Parameter fehlerhaft verhalten.

Werden diese Methoden nun korrigiert (wird also `return -1` durch `throws...` ersetzt), so entsprechen alle Ausgaben den Erwartungen:

```
testGcd
    11
    1
testLcm
    66
    1
testGcdException
    bad arguments
testLcmException
    bad arguments
```

(Wir wissen hoffentlich, wie die Zeilen "bad arguments" zu interpretieren sind – nämlich positiv!...)

Der Nachteil solcher manuellen Tests ist klar: bei jedem Test muss manuell das erwartete Verhalten mit dem tatsächlichen Verhalten verglichen werden.

Der erste Schritt hin zum automatischen Test besteht dann offenbar darin, die erwarteten Ergebnisse nicht irgendwo auf Papier zu hinterlegen, sondern direkt in den Methoden der Testklasse – so dass diese Ergebnisse dann automatisch mit den tatsächlichen Ergebnissen verglichen werden können. Ausgaben sollten nur dann erzeugt werden, wenn das tatsächliche Verhalten vom erwarteten Verhalten abweicht; ansonsten sollten überhaupt keine Ausgaben produziert werden (keine Ausgaben sind gute Ausgaben...).

2.2 Asserts

Wir beginnen nun mit dem Bau unseres kleinen Test-Frameworks. Alle Klassen dieses Mini-Frameworks sind im Package `util.test` angesiedelt.

Wir definieren zunächst eine Klasse `Assert`, die im folgenden Methode für Methode vorgestellt wird (die Klasse enthält ausschließlich statische Methoden):

```
package util.test;  
  
public class Assert {
```

Der `assertEquals`-Methode werden zwei `int`-Werte übergeben: ein erwarteter und ein tatsächlicher Wert. Im Falle, dass diese Werte sich unterscheiden, wird ein `AssertionError` geworfen (mit einer aussagefähigen Message):

```
public static void assertEquals(  
    final int expected, final int actual) {  
    if (expected != actual)  
        throw new AssertionError(  
            "Expected: " + expected + ". But was: " + actual);  
}
```

Natürlich müsste diese Methode in einem realistischen Framework mehrfach überladen sein: für `double`-, `float`-, `char`- etc. Parameter (also für alle primitiven Typen).

Zusätzlich zu all diesen überladenen Methoden für primitive Typen wird eine weitere `assertEquals`-Methode definiert, welche `Object`-Referenzen entgegennimmt:

```
public static void assertEquals(  
    final Object expected, final Object actual) {  
    if (expected == null) {  
        if (actual != null)  
            throw new AssertionError(  
                "Expected: null. But was: " + actual);  
    }  
  
    if (!expected.equals(actual))  
        throw new AssertionError(  
            "Expected: " + expected + ". But was: " + actual);  
}
```

Die übergebenen Objekte werden (sofern `expected` nicht `null` ist) mittels der in `Object` definierten `equals`-Methode verglichen – welche hofentlich in der Klasse der Objekte, die an `Assert.assertEquals` übergeben werden, überschrieben ist...

Die Methode `fail` wirft bedingungslos einen `AssertionError`:

```
public static void fail() {
    throw new AssertionError("Failed");
}
```

Wir werden testen wollen, ob eine Methode der zu testenden Klasse tatsächlich die von ihr erwartete `Exception` wirft. Hierzu werden wir eine der beiden folgenden `assertThrows`-Methoden nutzen.

Hier die erste dieser beiden Methoden:

```
public static <T> void assertThrows(
    final Class<?> exceptionType, final Func<T> func) {
    try {
        func.apply();
        throw new AssertionError(
            "Expected exception (but was not thrown): " +
            exceptionType.getName());
    }
    catch (final AssertionError e) {
        throw e;
    }
    catch (final Exception e) {
        if (! exceptionType.isAssignableFrom(e.getClass()))
            throw new AssertionError("Expected exception: " +
                exceptionType.getName() + ". But was: " +
                e.getClass().getName());
    }
}
```

Der Methode wird zunächst der Typ der erwarteten `Exception` übergeben (in Form einer `Class`-Referenz); als zweiter Parameter wird ein Objekt übergeben, dessen Klasse das Interface `Func` implementiert. Hier das Interface:

```
package util.test;

@FunctionalInterface
public interface Func<T> {
    public abstract T apply() throws Exception;
}
```

In einem `try`-Block wird dann die `apply`-Methode auf das `Func`-Objekt aufgerufen: `func.apply()`. Sofern diese Methode (wider Erwarten!) keine Ausnahme wirft, sondern stattdessen normal zurückkehrt, wird ein `AssertionError` geworfen (der erste `catch`-Zweig dient nur dazu, diesen `AssertionError` als solchen weiterzuwerfen). Ansonsten wird der Typ der geworfenen `Exception` mit dem Typ der erwarteten `Exception` verglichen – und im Falle einer Nicht-Übereinstimmung wiederum ein `AssertionError` geworfen. Ansonsten wird `assertThrows` normal verlassen.

Der zweiten `assertThrows`-Methode wird statt eines `Func`-Objekts ein `Proc`-Objekt übergeben:

```
public static <T> void assertThrows(
    final Class<?> exceptionType, final Proc proc) {
    try {
        proc.execute();
        throw new AssertionError(
            "Expected exception (but was not thrown): " +
            exceptionType.getName());
    }
    catch (final AssertionError e) {
        throw e;
    }
    catch (final Exception e) {
        if (! exceptionType.isAssignableFrom(e.getClass()))
            throw new AssertionError("Expected exception: " +
                exceptionType.getName() + ". But was: " +
                e.getClass().getName());
    }
}
```

`Proc` ist ebenso wie `Func` ein funktionales Interface:

```
package util.test;

@FunctionalInterface
public interface Proc {
    public abstract void execute() throws Exception;
}
```

Damit ist das Ende der `Assert`-Klasse erreicht.

Mit den Methoden der `Assert`-Klasse kann die `MathServiceTest`-Klasse nun wie folgt reformuliert werden:

```
package test;

import appl.MathService;
import util.test.Assert;

public class MathServiceTest {
```

Wir erwarten, dass der GGT von 33 und 22 gleich 11 ist – und der GGT von 1 und 1 gleich 1 ist:

```
public void testGcd() {
    final MathService ms = new MathService();
    Assert.assertEquals(11, ms.gcd(33, 22));
    Assert.assertEquals(1, ms.gcd(1, 1));
}
```


Wir erwarten, dass das KGV von 33 und 22 gleich 66 ist – und das KGV von 1 und 1 gleich 1 ist:

```
public void testLcm() {  
    final MathService ms = new MathService();  
    Assert.assertEquals(66, ms.lcm(33, 22));  
    Assert.assertEquals(1, ms.lcm(1, 1));  
}
```

Wir erwarten, dass `gcd(0, 1)` eine `IllegalArgumentException` wirft (und benutzen für diese Zusicherung einen "umständlichen" `try-catch`):

```
public void testGcdException() {  
    final MathService ms = new MathService();  
    try {  
        ms.gcd(0, 1);  
        Assert.fail();  
    }  
    catch (final IllegalArgumentException e) {  
        // this is okay  
    }  
}
```

Wir erwarten, dass `lcm(-1, 1)` ebenfalls eine `IllegalArgumentException` wirft – formulieren dies nun aber mittels der ersten der beiden `Assert.assertThrows`-Methoden etwas eleganter:

```
public void testLcmException() {  
    final MathService ms = new MathService();  
    Assert.assertThrows(IllegalArgumentException.class,  
        () -> ms.lcm(-1, 1));  
}
```

Damit ist das Ende von `MathServiceTest` erreicht.

Nun zum Hauptprogramm.

Jede der Testmethoden kann prinzipiell einen `AssertionError` liefern. Damit nun alle Testmethoden aufgerufen werden (auch dann, wenn eine von ihnen eine solche `Exception` wirft), müssen natürlich die Aufrufe all dieser Methoden in einem jeweils eigenen `try-catch`-Kontext erfolgen:

```
package test;

public class MathServiceTestRunner {
    public static void main(final String[] args) {
        try {
            System.out.println("testGcd");
            new MathServiceTest().testGcd();
        }
        catch (final AssertionError e) {
            System.out.println(e);
        }
        try {
            System.out.println("testLcm");
            new MathServiceTest().testLcm();
        }
        catch (final AssertionError e) {
            System.out.println(e);
        }

        try {
            System.out.println("testGcdException");
            new MathServiceTest().testGcdException();
        }
        catch (final AssertionError e) {
            System.out.println(e);
        }

        try {
            System.out.println("testLcmException");
            new MathServiceTest().testLcmException();
        }
        catch (final AssertionError e) {
            System.out.println(e);
        }
    }
}
```

Hier die Ausgaben (bei fehlerhafter Implementierung der `MathService`-Methoden):

```
testGcd
testLcm
testGcdException
    java.lang.AssertionError: Failed
testLcmException
    java.lang.AssertionError: Expected exception (but was
not thrown):
        java.lang.IllegalArgumentException
```

Und hier die Ausgaben, nachdem die Fehler im `MathService` beseitigt sind:

```
testGcd  
testLcm  
testGcdException  
testLcmException
```

Das einzige, was im "Erfolgsfall" ausgegeben wird, sind die Namen der aufgerufen Testmethoden (damit der Benutzer nicht fälschlicherweise annehmen kann, dass überhaupt keine Aufrufe stattgefunden haben...)

Hinweis: Neben des `assertEquals`-Methoden könnte die `Assert`-Klasse um `assertSame`-Methoden erweitert werden...

2.3 Runner

Die Formulierung des obigen Hauptprogramms ist sehr aufwendig. Die Einführung einer kleinen `Runner`-Klasse kann die Formulierung wesentlich vereinfachen:

```
package util.test;

public class Runner {
    public static void runTestMethod(
        final String name, final Runnable runnable) {
        try {
            System.out.println(name);
            runnable.run();
        }
        catch (final AssertionError e) {
            System.out.println("\t" + e.getMessage());
        }
    }
}
```

`runTestMethod` verlangt die Übergabe eines Methodennamens und eines `Runnable`s. Im `try`-Block wird der Methodenname ausgegeben und dann das `Runnable` ausgeführt: `run()`. Sofern im Kontext dieses Aufrufs ein `AssertionError` geworfen wird, wird dieser protokolliert.

Die neue `main`-Methode sieht nun viel aufgeräumter aus:

```
package test;

import util.test.Runner;

public class MathServiceTestRunner {
    public static void main(final String[] args) {
        Runner.runTestMethod("testGcd",
            () -> new MathServiceTest().testGcd());
        Runner.runTestMethod("testLcm",
            () -> new MathServiceTest().testLcm());
        Runner.runTestMethod("testGcdException",
            () -> new MathServiceTest().testGcdException());
        Runner.runTestMethod("testLcmException",
            () -> new MathServiceTest().testLcmException());
    }
}
```

Die Ausgaben sind dieselben wie im letzten Abschnitt.

2.4 Reflected Runner

Im Folgenden wird gezeigt, wie das Hauptprogramm noch ein letztes Mal verkleinert werden kann – und zwar derart, dass es sich um die Aufrufe der einzelnen Testmethoden überhaupt nicht mehr kümmern muss.

Angenommen, wir annotieren jede Testmethode der Testklasse mit einer Annotation namens `@Test`:

```
package util.test;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Hier die neue Testklasse:

```
package test;

import appl.MathService;
import util.test.Assert;
import util.test.Test;

public class MathServiceTest {

    @Test
    public void testGcd() {
        final MathService ms = new MathService();
        Assert.assertEquals(11, ms.gcd(33, 22));
        Assert.assertEquals(1, ms.gcd(1, 1));
    }

    @Test
    public void testLcm() {
        final MathService ms = new MathService();
        Assert.assertEquals(66, ms.lcm(33, 22));
        Assert.assertEquals(1, ms.lcm(1, 1));
    }

    @Test
    public void testGcdException() {
        final MathService ms = new MathService();
        try {
            ms.gcd(0, 1);
            Assert.fail();
        }
        catch (final IllegalArgumentException e) {
            // this is okay
        }
    }
}
```

```
@Test
public void testLcmException() {
    final MathService ms = new MathService();
    Assert.assertThrows(IllegalArgumentException.class,
        () -> ms.lcm(-1, 1));
}
```

Wir können nun einen Reflection-basierten `Runner` schreiben, welcher in einer gegebenen Klasse nach all denjenigen Methoden sucht, die mit `@Test` annotiert sind – und diese dann aufruft:

```
package util.test;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class Runner {
    public static void run(final Class<?> testClass) {
        for (final Method method : testClass.getMethods()) {
            if (Runner.isTestMethod(method)) {
                try {
                    System.out.println(method.getName());
                    final Object obj = testClass.newInstance();
                    method.invoke(obj);
                }
                catch (final InvocationTargetException e) {
                    System.out.println("\t" + e.getTargetException());
                }
                catch (final Exception e) {
                    System.out.println("\t" + e);
                }
            }
        }
    }

    private static final boolean isTestMethod(final Method method) {
        if (method.getAnnotation(Test.class) == null)
            return false;
        final String methodName = method.getName();
        if (!Modifier.isPublic(method.getModifiers()))
            throw new RuntimeException(
                methodName + " : must be public");
        if (Modifier.isStatic(method.getModifiers()))
            throw new RuntimeException(methodName +
                " : must not be static");
        if (method.getParameterCount() != 0)
            throw new RuntimeException(
                methodName + " : mustn't have paramters");
        if (method.getReturnType() != void.class)
            throw new RuntimeException(
                methodName + " : must be void");
        return true;
    }
}
```

Runner besitzt nun eine `run`-Methode, welcher der Typ einer Testklasse übergeben wird. Die `run`-Methode iteriert über alle `Methods` der Testklasse. Sofern es sich bei der jeweiligen Methode um eine Testmethode handelt (diese Frage wird von der privaten Methode `isTestMethod` beantwortet – s.u.), wird ein Objekt der Testklasse erstellt (`testClass.newInstance`), um dann schließlich die Testmethode auf dieses Testobjekt aufzurufen (per `method.invoke`). Sofern die derart aufgerufene Testmethode eine `Exception` liefert, wird `invoke` diese `Exception` in eine `TargetInvocationException` einwickeln – wir müssen diese `Target-Exception` dann also auswickeln.

Die Methode `isTestMethod` liefert dann `true`, wenn das ihr übergebene `Method`-Objekt eine Methode repräsentiert, welche folgende Eigenschaften hat: sie muss mit `@Test` annotiert sein; sie muss `public` sein; sie darf nicht `static` sein; und sie muss parameterlos und `void` sein.

Das neue Hauptprogramm sieht nun äußerst schlank aus:

```
package test;

import util.test.Runner;

public class MathServiceTestRunner {
    public static void main(final String[] args) {
        Runner.run(MathServiceTest.class);
    }
}
```

Die Ausgaben sind dieselben wie im letzten Abschnitt.

3

Testing – Das JUnit Test-Framework

3.1	Start.....	3-5
3.2	Exceptions.....	3-9
3.3	Before-After	3-13
3.4	Assertions.....	3-19
3.5	Test-Suites	3-21
3.6	Parameter.....	3-22
3.7	Private Methods.....	3-23
3.8	Assert That	3-25
3.9	Erweiterungen von JUnit	3-28

3 Testing – Das JUnit Test-Framework

Im Folgenden wird das in Eclipse enthaltene JUnit-Test-Framework vorgestellt (das Framework kann natürlich auch außerhalb von Eclipse genutzt werden).

Wir werden sehen, dass dieses Framework in seinem Kern genauso funktioniert wie das im letzten Kapitel entwickelte eigene `TestFramework`.

Die JUnit-Klasse können wie folgt zum CLASSPATH des jeweiligen Projekts hinzugefügt werden:

Properties → Java Build Path → Libraries → Add Library → JUnit → JUnit-4

Die `Assert`-Klasse des JUnit-Frameworks enthält leider keine `Asserts` zum Testen von Exceptions. Deshalb verwenden die folgenden Beispiele zusätzlich zur JUnit-`Assert`-Klasse eine eigene `XAssert`-Klasse (im `shared`-Projekt enthalten). Die Methoden dieser Klasse benutzen folgende funktionale Interfaces:

```
package util.test;

@FunctionalInterface
public interface Proc {
    public abstract void execute() throws Exception;
}
```

```
package util.test;

@FunctionalInterface
public interface Func<T> {
    public abstract T apply() throws Exception;
}
```

(Man könnte versucht sein, statt dieser beiden Interfaces die Standard-Interfaces `Consumer` resp. `Function` zu nutzen – die Methoden dieser Interfaces dürfen aber keine Exceptions werfen und sind daher für unsere Zwecke unbrauchbar.)

Hier nun die XAssert-Klasse:

```
package util.test;

public class XAssert {

    public static <T> void assertThrows(
        final Class<?> exceptionType, final Func<T> func) {
        try {
            func.apply();
            throw new AssertionError(
                "Expected exception (but was not thrown): " +
                exceptionType.getName());
        }
        catch (final AssertionError e) {
            throw e;
        }
        catch (final Exception e) {
            if (! exceptionType.isAssignableFrom(e.getClass()))
                throw new AssertionError("Expected exception: " +
                    exceptionType.getName() + ". But was: " +
                    e.getClass().getName());
        }
    }

    public static void assertThrows(
        final Class<?> exceptionType, final Proc proc) {
        assertThrows(exceptionType,
            () -> { proc.execute(); return null; });
    }
}
```

Diese `assertThrows`-Methoden werden im Folgenden immer wieder verwendet werden.

3.1 Start

Hier die zu testende Klasse (sie ist bereits aus dem letzten Kapitel bekannt):

```
package appl;

public class MathService {

    public int gcd(int x, int y) {
        if (x <= 0 || y <= 0)
            throw new IllegalArgumentException("bad arguments");
        while (x != y) {
            if (x > y)
                x -= y;
            else
                y -= x;
        }
        return x;
    }

    public int lcm(final int x, final int y) {
        if (x <= 0 || y <= 0)
            throw new IllegalArgumentException("bad arguments");
        int a = x;
        int b = y;
        while (a != b) {
            if (a < b)
                a += x;
            else
                b += y;
        }
        return a;
    }
}
```

Und hier die Testklasse:

```
package test;

import org.junit.Assert;
import org.junit.Ignore;
import org.junit.Test;

import appl.MathService;

public class MathServiceTest {

    @Test
    public void testGcd() {
        final MathService ms = new MathService();
        Assert.assertEquals(11, ms.gcd(33, 22));
        Assert.assertEquals(1, ms.gcd(1, 1));
    }

    @Test
    public void testLcm() {
```

```
    final MathService ms = new MathService();
    Assert.assertEquals(66, ms.lcm(33, 22));
    Assert.assertEquals(1, ms.lcm(1, 1));
}
}
```

Auch in JUnit werden alle Testmethoden mittels `@Test` annotiert (`@org.junit.Test`). Auch JUnit besitzt eine `Assert`-Klasse mit statischen `assertEquals`-Methoden, um tatsächliche Resultate mit den erwarteten Resultaten vergleichen zu können.

JUnit besitzt weitere Features, die im `TestFramework` nicht enthalten sind.

Man kann eine Testmethode mit der Annotation `@Ignore` ausstatten:

```
@Test
@Ignore
public void testTomorrow() {
    // ...
}
```

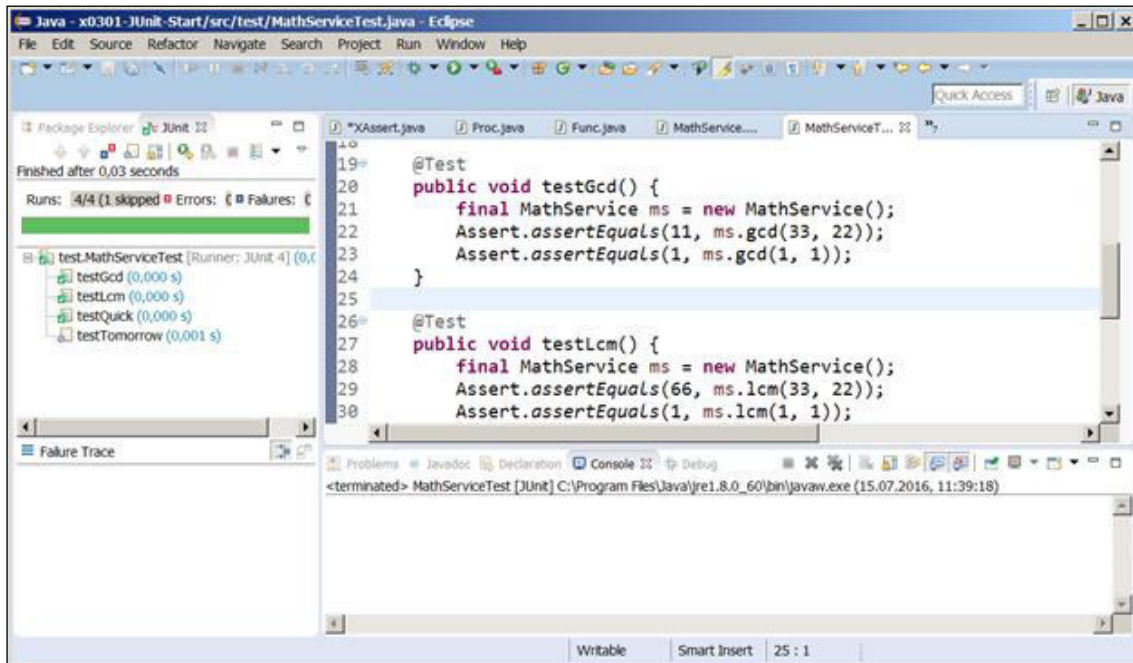
`@Ignore` bewirkt, dass JUnit diese Methode nicht ausführen wird. JUnit wird aber protokollieren, dass diese Methode nicht aufgerufen wird (man sieht dann also, dass noch etwas zu tun ist...).

Man kann eine Testmethode zusätzlich mit der Annotations `@Timeout` ausstatten:

```
@Test(timeout = 1000)
public void testQuick() {
    // ...
}
```

`@Timeout` bewirkt, dass ein Fehler angezeigt wird, wenn die Methode länger als 1000 Millisekunden für ihre Ausführung benötigt. (Auf diese Weise können u.a. Endlosschleifen entdeckt werden.)

Der Testlauf kann über CTRL-11 bzw. über "run as... JUnit-Test" gestartet werden.

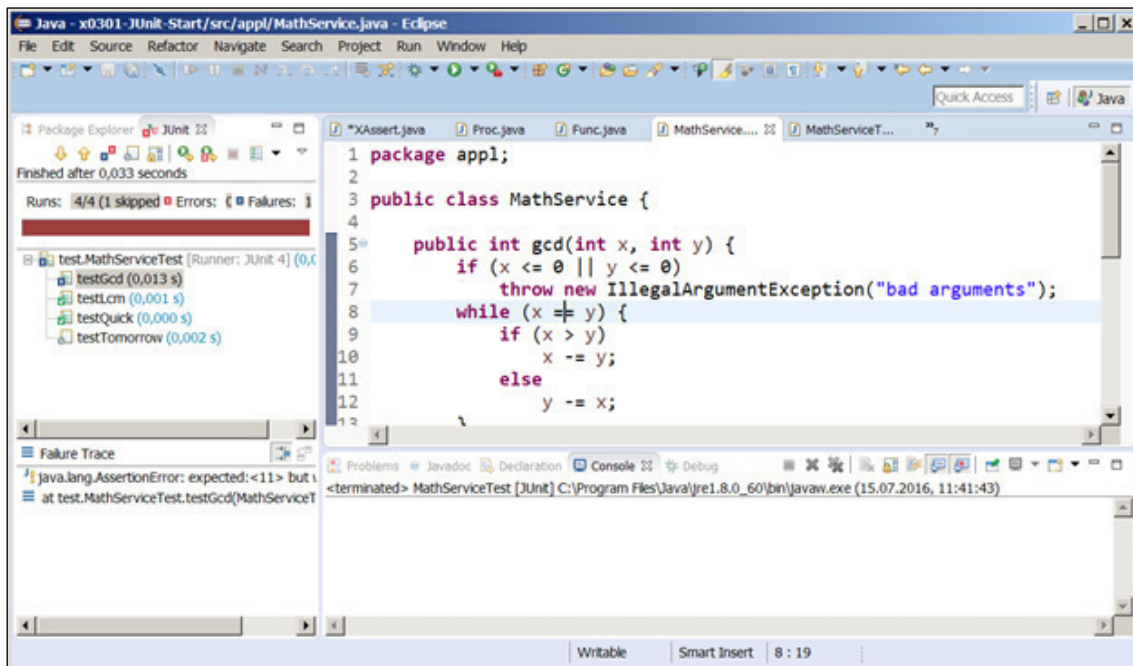


Im Test-Explorer wird das Ergebnis der Tests angezeigt. Ein grünes Symbol steht für Erfolg, ein rotes für Misserfolg. (Auch im Test-Explorer kann der Test (können die Tests) neu gestartet werden.

Angenommen, die `gcd`-Method ist "falsch" implementiert (statt auf Ungleichheit testet die `while`-Schleife auf Gleichheit):

```
public int gcd(int x, int y) {
    if (x <= 0 || y <= 0)
        throw new IllegalArgumentException("bad arguments");
    while (x == y) {
        if (x > y)
            x -= y;
        else
            y -= x;
    }
    return x;
}
```

Hier das Resultat des Testlaufs:



Wir arbeitet der Test-Runner von JUnit?

Der Test-Runner ermittelt diejenigen Methoden der Testklasse, die mit `@Test` annotiert sind. Solche Methoden müssen Instanz-Methoden sein; und sie müssen `public`, parameterlos und `void` sein.

Bevor eine Testmethode gestartet wird, wird jeweils ein neues Objekt der Testklasse erzeugt. Jede Testmethode wird also auf einem jungfräulichen Testklassen-Objekt ausgeführt. Damit ist von vornherein garantiert, dass die Testmethoden voneinander unabhängig sind. (Die Reihenfolge der Ausführung der Testmethoden ist übrigens nicht unbedingt identisch mit der Reihenfolge ihrer textuellen Definition!)

Statt des Eclipse-Runners der Test auch mittels einer einfachen `main`-Methode ausgeführt werden. Die obige Testklasse könnte um folgende `main`-Methode erweitert werden:

```
public static void main(final String[] args) {
    final Result result =
        JUnitCore.runClasses(MathServiceTest.class);
    System.out.println("runs = " + result.getRunCount() +
        " failures = " + result.getFailureCount());
    final List<Failure> failures = result.getFailures();
    for (final Failure failure : failures)
        System.out.println(failure);
}
```

3.2 Exceptions

Im Folgenden wird gezeigt, wie getestet werden, dass Methoden unter bestimmten Bedingungen eine `Exception` werfen müssen. Die Erwartung besteht dann nicht darin, dass solche Methoden ein bestimmtes Resultat liefern, sondern eben darin, dass sie eine `Exception` werfen.

Gegeben sei folgende zu testende Klasse:

```
package appl;

public class Stack<T> {

    private final T[] elements;
    private int size;

    @SuppressWarnings("unchecked")
    public Stack(final int capacity) {
        this.elements = (T[]) new Object[capacity];
    }

    public boolean isEmpty() {
        return this.size == 0;
    }

    public boolean isFull() {
        return this.size == this.elements.length;
    }

    public void push(final T elem) {
        if (this.isFull())
            throw new IllegalStateException("full");
        this.elements[this.size] = elem;
        this.size++;
    }

    public T top() {
        if (this.isEmpty())
            throw new IllegalStateException("empty");
        return this.elements[this.size - 1];
    }

    public T pop() {
        if (this.isEmpty())
            throw new IllegalStateException("empty");
        final T elem = this.elements[this.size - 1];
        this.elements[this.size - 1] = null;
        this.size--;
        return elem;
    }
}
```

Ein `Stack` ist ein Stack mit begrenzter Kapazität (die Kapazität wird dem Konstruktor übergeben). Ein `Stack` kann Objekte des Typs `T` speichern. Im Konstruktor wird ein Array erzeugt, in welchem die Elemente des `Stacks` gespeichert werden. Mittels der Methoden `isEmpty` resp.

`isFull` kann ein `Stack` daraufhin geprüft werden, ob er leer oder voll ist. Mittels `push` kann dem `Stack` ein neues Element hinzugefügt werden. Per `top` kann das oberste Element des `Stacks` besichtigt werden. Und mittels `pop` kann das oberste Element entfernt werden. Die Methoden `push`, `top` und `pop` sind an Preconditions geknüpft: `push` setzt voraus, dass der `Stack` nicht bereits voll ist; und `top` und `pop` setzen voraus, dass er nicht leer ist. Sofern beim Aufruf dieser Methoden die entsprechende Precondition verletzt ist, wird jeweils eine `Exception` geworfen.

Beim Test geht's nun insbesondere darum, zu zeigen, dass beim Verletzen der Preconditions auch die entsprechenden `Exceptions` geworfen werden.

Hier die Testklasse:

```
package test;

import org.junit.Assert;
import org.junit.Test;

import appl.Stack;
import util.test.XAssert;

public class StackTest {
```

Zunächst ein Test, welcher die "Story" einer "normalen" Nutzung eines `Stacks` erzählt:

```
@Test
public void TestNormal() {
    final Stack<String> stack = new Stack<String>(2);
    Assert.assertTrue(stack.isEmpty());
    Assert.assertFalse(stack.isFull());
    stack.push("one");
    Assert.assertFalse(stack.isEmpty());
    Assert.assertFalse(stack.isFull());
    Assert.assertEquals("one", stack.top());
    stack.push("two");
    Assert.assertFalse(stack.isEmpty());
    Assert.assertTrue(stack.isFull());
    Assert.assertEquals("two", stack.top());
    Assert.assertEquals("two", stack.top());
    Assert.assertEquals("two", stack.pop());
    Assert.assertFalse(stack.isEmpty());
    Assert.assertFalse(stack.isFull());
    Assert.assertEquals("one", stack.top());
    Assert.assertEquals("one", stack.pop());
    Assert.assertTrue(stack.isEmpty());
    Assert.assertFalse(stack.isFull());
}
```

Wie man sieht, enthält die `Assert`-Klasse neben den `assertEquals`-Methoden auch solche Methoden wie `assertTrue` und `assertFalse`. (Auch hier hätte man `assertEquals` benutzen können – die Verwen-

derung von `assertTrue` und `assertFalse` ist aber offensichtlich eleganter.)

Im Falle, dass der `Stack` leer ist, darf `top` nicht aufgerufen werden. Wird `top` trotzdem aufgerufen, muss `top` eine `IllegalStateException` werfen:

```
@Test(expected = Exception.class)
public void TestTopWhenEmpty() {
    final Stack<String> stack = new Stack<String>(2);
    stack.top();
}
```

Der `@Test`-Annotation wird hier zusätzlich mit dem Attribut `expected` ausgestattet. Als Wert von `expected` wird dasjenige `Class`-Objekt übergeben, welches die Klasse der erwarteten `Exception` repräsentiert. Dieser Typ kann "ungenau" sein: statt `IllegalStateException.class` kann also auch `Exception.class` übergeben werden.

Wenn der `Stack` leer ist, muss natürlich auch der Aufruf von `pop` eine `Exception` liefern (im Folgenden wird bei `expected` der exakte Typ angegeben):

```
@Test(expected = IllegalStateException.class)
public void TestPopWhenEmpty() {
    final Stack<String> stack = new Stack<String>(2);
    stack.pop();
}
```

Wenn der `Stack` voll ist, muss der Aufruf von `push` eine `Exception` liefern:

```
@Test(expected = RuntimeException.class)
public void testPushWhenFull() {
    final Stack<String> stack = new Stack<String>(2);
    stack.push("one");
    stack.push("two");
    stack.push("three");
}
```

(Da eine `IllegalStateException` eine `RuntimeException` ist, kann als Wert von `expected` auch `RuntimeException.class` angegeben werden.)

Beim genaueren Hinsehen erkennt man, dass dieses Verfahren in bestimmten Fällen leider zu "ungenau" ist: der Test würde z.B. auch dann ein grünes Ergebnis liefern, wenn bereits der erste oder zweite Aufruf von `push` (oder sogar der Konstruktor!) eine `Exception` geworfen hätte. Es ist also nicht sichergestellt, dass erst der dritte Aufruf von `push` die `Exception` wirft.

Im Gegensatz hierzu ist die folgende Variante genau:

```
@Test
public void testPushWhenFullWithLambda() {
    final Stack<String> stack = new Stack<String>(2);
    stack.push("one");
    stack.push("two");
    XAssert.assertThrows(Exception.class, () ->
stack.push("three"));
}
```

Der Aufruf, welche die `Exception` werfen soll, wird als Lambda-Ausdruck an `XAssert.assertThrows` übergeben (siehe die Klasse `XAssert` im `shared-Projekt`). Hier funktioniert sowohl die "genaue" als auch die "ungenau" Variante.

Natürlich könnte man auch den umständlichen `try-catch` verwenden. Er ist zwar umständlich, aber der Aufruf, vom welchem die `Exception` erwartet wird, kann exakt benannt werden:

```
@Test
public void TestPushWhenFullWithTryCatch() {
    final Stack<String> stack = new Stack<String>(2);
    stack.push("one");
    stack.push("two");
    try {
        stack.push("three");
        Assert.fail();
    }
    catch (final IllegalStateException e) {
        Assert.assertEquals("full", e.getMessage());
    }
}
```

Hier ein weiterer `XAssert.assertThrows`-Test für die `top`-Methode:

```
@Test
public void TestTopWhenEmpty2() {
    final Stack<String> stack = new Stack<String>(2);
    XAssert.assertThrows(IllegalStateException.class,
        () -> stack.top());
}
```

Derselbe Test noch einmal in der "ungenauen" Variante:

```
@Test
public void TestTopWhenEmpty3() {
    final Stack<String> stack = new Stack<String>(2);
    XAssert.assertThrows(RuntimeException.class, () ->
stack.top());
}
```

Resultat: die Variante, die bevorzugt genutzt werden sollte, ist wahrscheinlich die `XAssert.assertThrows`-Variante – `expected` ist zu ungenau, `try-catch` zu umständlich.

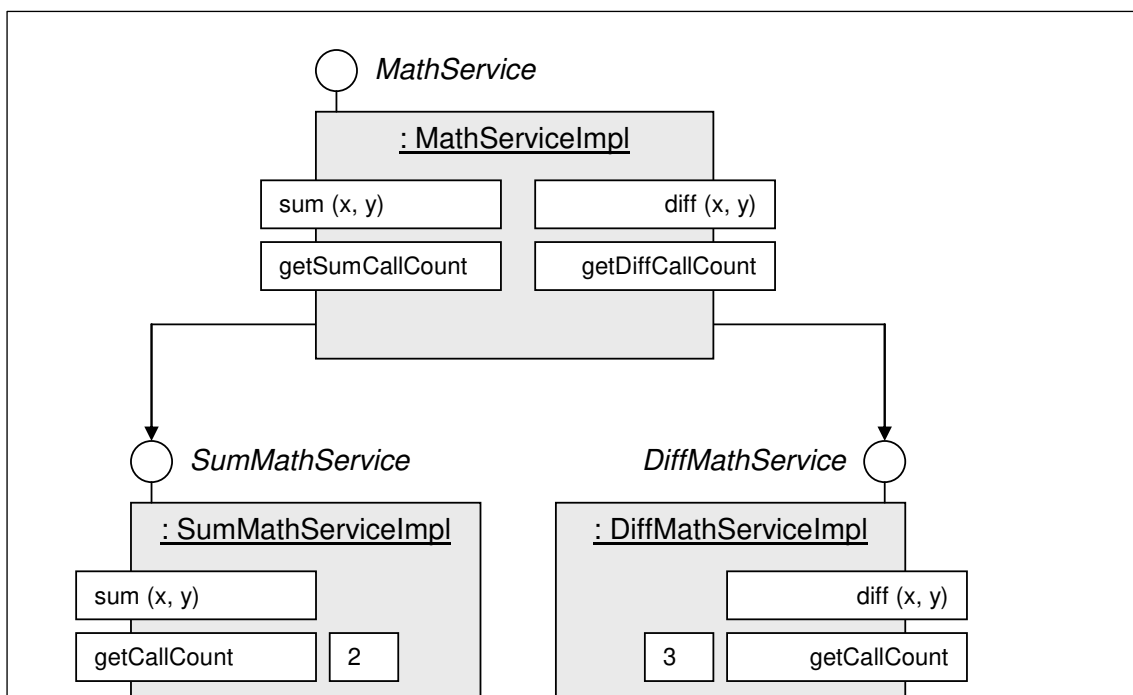
3.3 Before-After

Häufig müssen zunächst mehrere Objekte erzeugt und zusammengesteckt werden, um eines dieser Objekte dann überhaupt testen zu können. Hier ein Beispiel:

Ein Mathematiker kann die Summe und die Differenz zweier Zahlen berechnen: er bietet die Methoden `sum` und `diff` an. Ein Mathematiker kann auch danach gefragt werden, wie häufig er bereits die Summe zweier Zahlen resp. die Differenz zweier Zahlen berechnet hat: `getSumCallCount` und `getDiffCallCount`. Ein Mathematiker ist im Interface `MathService` spezifiziert und in `MathServiceImpl` implementiert.

Ein `MathServiceImpl`-Objekt benutzt zwei Hilfsobjekte: einen Summe-Mathematiker und einen Differenz-Mathematiker. Ein Hilfsmathematiker besitzt jeweils eine Berechnungs-Methode (`sum` resp. `diff`) und eine `getCallCount`-Methode. Jeder Hilfsmathematiker speichert die Anzahl der bereits erfolgten `sum`- resp. `diff`-Aufrufe. Auch die Hilfsmathematiker sind jeweils zunächst in einem Interface spezifiziert. Und der Chef-Mathematiker soll sich auf die Hilfsmathematiker nur über diese Interfaces beziehen – damit nämlich die Implementierung der Hilfsmathematiker jederzeit durch eine andere Implementierung ersetzt werden kann.

Also müssen dem Chef-Mathematiker Referenzen auf die beiden benötigten Hilfsmathematiker per Dependency Injection übergeben werden. Dies geschieht mittels des Konstruktors.



Hier die im folgenden Test benutzten Interfaces – das Interface des Chefmathematikers und seiner beiden Hilfsmathematiker:

```
package appl;

public interface MathService {
    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);
    public abstract int getSumCallCount();
    public abstract int getDiffCallCount();
}
```

```
package appl;

public interface SumMathService {
    public abstract int sum(int x, int y);
    public abstract int getCallCount();
}
```

```
package appl;

public interface DiffMathService {
    public abstract int diff(int x, int y);
    public abstract int getCallCount();
}
```

Hier die Implementierung dieser Interfaces:

```
package appl;

public class MathServiceImpl implements MathService {

    private final SumMathService sumMath;
    private final DiffMathService diffMath;

    public MathServiceImpl(
        final SumMathService sumMath,
        final DiffMathService diffMath) {
        this.sumMath = sumMath;
        this.diffMath = diffMath;
    }

    @Override
    public int sum(final int x, final int y) {
        return this.sumMath.sum(x, y);
    }

    @Override
    public int diff(final int x, final int y) {
        return this.diffMath.diff(x, y);
    }

    @Override
    public int getSumCallCount() {
        return this.sumMath.getCallCount();
    }
}
```

```
@Override
public int getDiffCallCount() {
    return this.diffMath.getCallCount();
}
}
```

```
package appl;

public class SumMathServiceImpl implements SumMathService {

    private int count;

    @Override
    public int sum(final int x, final int y) {
        this.count++;
        return x + y;
    }

    @Override
    public int getCallCount() {
        return this.count;
    }
}
```

```
package appl;

public class DiffMathServiceImpl implements DiffMathService {

    private int count;

    @Override
    public int diff(final int x, final int y) {
        this.count++;
        return x - y;
    }

    @Override
    public int getCallCount() {
        return this.count;
    }
}
```

Soll nun ein Chefmathematiker getestet werden, so muss er zunächst erzeugt werden und mit den Hilfsmathematikern zusammengebracht werden. Dies muss natürlich zu Beginn jeder `@Test`-Methode passieren.

Um nun aber vermeiden zu können, dieses Erzeugen und Zusammenstecken der Objekte in jeder der `@Test`-Methode tatsächlich auch implementieren zu müssen, kann der Aufbau des Objektgraphen in einer einzigen Methode implementiert werden. Diese Methode wird mit `@Before` annotiert. Eine solche mittels mit `@Before` gekennzeichnete Methode wird bei der Ausführung der Testklasse vor jeder `@Test`-Methode erneut aufgerufen.

Neben `@Before` existiert die inverse Annotation `@After`. Eine `@After`-Methode wird nach der Ausführung jeder `@Test`-Methode aufgerufen. Sie kann z.B. bestimmte Ressourcen freigeben, welche in der `@Before`-Methode jeweils angefordert wurden.

Sowohl die `@Before`- als auch die `@After`-Methode muss `public`, parameterlos und vom Typ `void` sein. Die Namen der Methoden sind ohne Belang. Wir nennen die Methoden im Folgenden `before` und `after`.

Neben den `@Before`- und `@After`-Annotationen gibt's noch zwei weitere: `@BeforeClass` und `@AfterClass`. Im Unterschied zu `@Before` und `@After` müssen diese Methoden `static` sein (zum Zeitpunkt ihres Aufrufs gibt's kein Objekt der Testklasse). Sowohl `@BeforeClass` als auch `@AfterClass` werden jeweils nur ein einziges Mal aufgerufen – vor dem ersten Test und nach dem letzten ausgeführten Test.

Hier das Schema eines kompletten Test-Ablaufs:

```
@BeforeClass
    @Before
        @Test
    @After
    @Before
        @Test
    @After
        ...
@AfterClass
```

Hier schließlich die Testklasse für den Mathematiker:

```
package test;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import appl.DiffMathService;
import appl.DiffMathServiceImpl;
import appl.MathService;
import appl.MathServiceImpl;
import appl.SumMathService;
import appl.SumMathServiceImpl;

public class MathServiceTest {

    private MathService ms;

    @BeforeClass
    public static void beforeClass() {
        System.out.println("beforeClass");
    }
}
```

```
@Before
public void before() {
    System.out.println("\tbefore");
    final SumMathService sms = new SumMathServiceImpl();
    final DiffMathService dms = new DiffMathServiceImpl();
    this.ms = new MathServiceImpl(sms, dms);
}

@After
public void after() {
    System.out.println("\tafter");
}

@AfterClass
public static void afterClass() {
    System.out.println("afterClass");
}

@Test
public void testSum() {
    System.out.println("\t\ttestSum");
    Assert.assertEquals(42, this.ms.sum(40, 2));
    Assert.assertEquals(0, this.ms.sum(0, 0));
    Assert.assertEquals(2, this.ms.getSumCallCount());
    Assert.assertEquals(0, this.ms.getDiffCallCount());
}

@Test
public void testDiff() {
    System.out.println("\t\ttestDiff");
    Assert.assertEquals(42, this.ms.diff(44, 2));
    Assert.assertEquals(0, this.ms.diff(0, 0));
    Assert.assertEquals(0, this.ms.getSumCallCount());
    Assert.assertEquals(2, this.ms.getDiffCallCount());
}

@Test
public void testSumDiff() {
    System.out.println("\t\ttestSumDiff");
    Assert.assertEquals(42, this.ms.sum(40, 2));
    Assert.assertEquals(42, this.ms.diff(44, 2));
    Assert.assertEquals(0, this.ms.sum(0, 0));
    Assert.assertEquals(0, this.ms.diff(0, 0));
    Assert.assertEquals(2, this.ms.getSumCallCount());
    Assert.assertEquals(2, this.ms.getDiffCallCount());
}
}
```

In der `@Before`-Methode wird ein Objektgraph erzeugt und an die Instanzvariable `ms` gebunden. Über diese Instanzvariable können dann die `@Test`-Methoden den Chefmathematiker erreichen.

Hier die Demo-Ausgaben des Tests (ein "richtiger" Test sollte natürlich eigentlich keinerlei Ausgaben produzieren):

```
beforeClass
  before
    testSum
  after
  before
    testDiff
  after
  before
    testSumDiff
  after
afterClass
```

3.4 Assertions

Bislang wurde hauptsächlich die Methode `Assert.assertEquals` verwendet, um Erwartungen mit Tatsachen vergleichen zu können. Neben `Assert.assertEquals` existieren eine Reihe weiterer Methoden, die im Folgenden vorgestellt werden.

Um diese weiteren `Assert`-Methoden vorstellen zu können, ist keine eigene zu testende Klasse erforderlich, deren Verhalten getestet wird. Die Tests sind stattdessen "trivial":

```
@Test
public void trueFalse() {
    Assert.assertTrue(true);
    Assert.assertFalse(false);
}
```

`Assert.assertTrue` ist erfolgreich, wenn das ihr übergebene Argument `true` ist; `Assert.assertFalse` ist dann erfolgreich, wenn das übergebene Argument `false` ist.

```
@Test
public void nullNotNull() {
    Assert.assertNull(null);
    Assert.assertNotNull("hello");
}
```

`Assert.assertNull` ist erfolgreich, wenn das ihr übergebene Argument `null` ist; `Assert.assertNotNull` ist erfolgreich, wenn das Argument nicht `null` ist (also eine gültige Objekt-Referenz ist).

```
@Test
public void equals() {
    Assert.assertEquals(1, 1);
    // assertEquals(long, long)
    Assert.assertEquals("hello", "hello");
    // assertEquals(Object, Object)
    Assert.assertEquals(1.0, 1.0, 0.0);
    // assertEquals(double, double, double)
    Assert.assertEquals(1.0, 1, 0);
    // assertEquals(double, double, double)
}
```

`Assert.assertEquals` gibt's in mehreren überladenen Varianten. Z. B.:

```
Assert.assertEquals(byte v1, byte v2)
Assert.assertEquals(short v1, short v2)
Assert.assertEquals(int v1, int v2)
Assert.assertEquals(long v1, long v2)
Assert.assertEquals(float v1, float v2, float delta)
Assert.assertEquals(double v1, double v2, double delta)
Assert.assertEquals(char v1, char v2);
Assert.assertEquals(boolean v1, boolean v2)
```

Man beachte, dass bei der `float` resp. `double`-Varianten zusätzlich zu den beiden zu vergleichenden Werten eine Genauigkeit angegeben wird. Die `Assert.assertEquals`, die mit primitiven Value-Typen parametrisiert sind, vergleichen die übergebenen Werte per `==`; die mit `Object`-Referenzen parametrisierte `Assert.assertEquals`-Methode vergleicht die übergebenen Objekte per `equals`.

```
@Test
public void same() {
    final String s1 = "Hello";
    String s2 = "H";
    s2 += "ello";
    final String s3 = s1;
    Assert.assertEquals(s1, s2);
    Assert.assertEquals(s1, s3);
    Assert.assertSame(s1, s3);
    Assert.assertNotSame(s1, s2);
}
```

`Assert.assertSame` liefert `true`, wenn die beiden ihr übergebenen Referenzen gleich sind (also auf dasselbe Objekt zeigen). `Assert.assertSame` unternimmt also einen Referenzvergleich. `Assert.assertNotSame` liefert `true`, wenn die übergebenen Referenzen ungleich sind. Man beachte den Unterschied dieser beiden Methoden zu den `Assert.assertEquals`-Methoden!

```
@Test
public void arrays() {
    final Integer[] integerArray1 = { 1, 2, 3 };
    final Integer[] integerArray2 = { 1, 2, 3 };
    final int[] intArray1 = { 1, 2, 3 };
    final int[] intArray2 = { 1, 2, 3 };
    Assert.assertArrayEquals(integerArray1, integerArray2);
    // assertArrayEquals(Object[], Object[])
    Assert.assertArrayEquals(intArray1, intArray2);
    // assertArrayEquals(int[], int[])
}
```

Arrays können mittels `asserArrayEquals` verglichen werden (auch hier gibt's natürlich überladene Varianten).

Zu allen `Assert`-Methoden gibt's jeweils noch eine weitere Variante, der als erster Parameter jeweils ein Text übergeben wird. Dieser Fehlertext wird dann natürlich ins Fehlerprotokoll übernommen. Ein solcher Text ist aber nicht unbedingt hilfreich.

3.5 Test-Suites

Eine Test-Suite ist eine Zusammenfassung mehrerer Testklassen zu einer Klasse, die ihrerseits selbst wieder als Testklasse verwendet werden kann. Wird die Testsuite ausgeführt, so werden alle darin enthalten Testklassen ausgeführt.

Seien z.B. folgende Testklassen gegeben:

```
package test;
// ...
public class ATest {
    @Test
    public void test() {
        assertTrue(true);
    }
}
```

```
package test;
// ...
public class BTest {
    @Test
    public void test() {
        assertFalse(false);
    }
}
```

Und hier eine Testsuite, bestehend aus ATest und BTest (unter Verwendung des neuen JUnit):

```
package test;
// ...
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({ ATest.class, BTest.class })

public class Test {
}
```

Die Sub-Tests werden einfach über eine @Suite.SuiteClasses-Annotation angegeben.

3.6 Parameter

Dieselbe(n) Testmethoden können im Verlauf eines Tests mehrfach aufgerufen werden – und zwar mit unterschiedlichen "Parametern".

Die Klasse definiert zu diesem Zweck einen Konstruktor, der mit den erforderlichen Parametern parametrisiert ist. Die ihm übergebenen Parameter werden in Instanzvariablen gespeichert. Die Test-Methode(n) kann (können) dann auf diese Instanzvariablen zugreifen.

Bereitgestellt werden die Parameter als Resultat einer statischen Methode, die mit `@Parameters` annotiert ist. Diese muss eine Liste von Object-Array erzeugen und zurückliefern (eine `Collection<Object[]>`). Jeder der Object-Arrays muss Einträge besitzen, welche an die Parameter des Konstruktors übergeben werden können.

```
package test;
// ...
@RunWith(value = Parameterized.class)
public class MathServiceTest {

    private final int x;
    private final int y;
    private final int expectedResult;

    public MathServiceTest(
        final int x, final int y, final int expectedResult) {
        this.x = x;
        this.y = y;
        this.expectedResult = expectedResult;
    }

    @Parameters
    public static Collection<Object[]> args() {
        final List<Object[]> list = new ArrayList<Object[]>();
        list.add(new Object[] { 1, 1, 1 });
        list.add(new Object[] { 3, 1, 1 });
        list.add(new Object[] { 4, 2, 2 });
        list.add(new Object[] { 33, 22, 11 });
        return list;
    }

    @Test
    public void testGcd() {
        final MathService ms = new MathService();
        Assert.assertEquals(this.expectedResult, ms.gcd(this.x,
this.y));
    }
}
```

3.7 Private Methods

Sollten nur öffentliche Methoden getestet werden oder auch private? Eigentlich sollte der Test auf öffentliche Methoden beschränkt sein. Aber es mag Ausnahmen geben.

Sei etwa folgende zu testende Klasse gegeben:

```
package appl;

public class Pythagoras1 {
    public double c(final double a, final double b) {
        return this.sqr(this.sqr(a) + this.sqr(b));
    }
    private double sqr(final double x) {
        return x * x;
    }
    private double sqrt(final double x) {
        return Math.sqrt(x);
    }
}
```

Ein `Pythagoras1` kann aufgrund zweier Katheten die Hypotenuse bestimmen (das ist der Sinn der `c`-Methode).

Können wir die privaten `sqr` und `sqrt`-Methoden testen? Erste Antwort: nein. Zweite Antwort: Ja – via Reflection:

```
package test;
// ...
import java.lang.reflect.Method;

public class PythagorasTest {
    @Test
    public void testC1() {
        final Pythagoras1 p = new Pythagoras1();
        Assert.assertEquals(5.0, p.c(3.0, 4.0), 0);
    }
    @Test
    public void testSqr1() throws Exception {
        final Pythagoras1 p = new Pythagoras1();
        final Method m = p.getClass().getDeclaredMethod("sqr",
double.class);
        m.setAccessible(true);
        final Object result = m.invoke(p, 4.0);
        Assert.assertEquals(2.0, result);
    }
}
```

Sollten wir aber nicht besser eine eigene `Calculator`-Klasse mit den öffentlichen Methoden `sqrt` und `sqr` schreiben, die dann auf ganz natürliche Weise getestet werden könnte? Und sollte dann der `Pythagoras` einfach an ein Objekt einer solchen `Calculator`-Klasse delegieren? Dann gäb's erst gar nicht das Bedürfnis, private Methoden testen zu wollen...

Die Klasse Calculator:

```
package appl;

public class Calculator {
    public double sqr(final double x) {
        return x * x;
    }
    public double sqrt(final double x) {
        return Math.sqrt(x);
    }
}
```

Ein Ausschnitt aus einem Test:

```
@Test
public void testCalculator() throws Exception {
    final Calculator c = new Calculator();
    Assert.assertEquals(2.0, c.sqrt(4.0), 0);
}
```

Eine Klasse Pythagoras2:

```
package appl;

public class Pythagoras2 {
    private final Calculator calculator = new Calculator();
    public double c(final double a, final double b) {
        return this.calculator.sqrt(
            this.calculator.sqr(a) + this.calculator.sqr(b));
    }
}
```

Ein Ausschnitt aus einem Test:

```
@Test
public void testC2() {
    final Pythagoras2 p = new Pythagoras2();
    Assert.assertEquals(5.0, p.c(3.0, 4.0), 0);
}
```

3.8 Assert That

Von `org.hamcrest` stammt eine Erweiterung zu JUnit. Es handelt sich um sog. *Macher*, welche der `Assert.assertThat`-Methode übergeben werden können.

Die Verwendung von `assertThat` führt dazu, die Tests quasi umgangssprachlich formulieren zu können. Man könnte auch von einer *domain-specific-language* (DSL) sprechen.

Hier ein kleines Beispiel, dessen Studium dem Leser / der Leserin überlassen bleiben soll:

```
package test;

import static org.hamcrest.CoreMatchers.both;
import static org.hamcrest.CoreMatchers.either;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.hasItem;
import static org.hamcrest.CoreMatchers.hasItems;
import static org.hamcrest.CoreMatchers.instanceOf;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.CoreMatchers.nullValue;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import org.hamcrest.BaseMatcher;
import org.hamcrest.Description;
import org.hamcrest.Factory;
import org.hamcrest.Matcher;
import org.junit.Assert;
import org.junit.Test;

@SuppressWarnings("unchecked")
public class AssertThatTest {

    @Test
    public void testEqualTo() {
        final int expected = 1;
        final int actual = 1;
        Assert.assertEquals(expected, actual);
        Assert.assertThat(actual, equalTo(expected));
        Assert.assertThat(actual, is(expected));
        Assert.assertThat(actual, is(equalTo(expected)));
    }

    @Test
    public void testInstanceOf() {
        Assert.assertTrue("Hello" instanceof String);
        Assert.assertThat("Hello", instanceOf(String.class));
        final Class<?> cls = String.class;
        Assert.assertSame(cls, "Hello".getClass());
        Assert.assertThat("Hello", instanceOf(cls));
    }

    @Test
```



```

public void testHasItem() {
    final List<Integer> list = new ArrayList<Integer>();
    list.add(42);
    list.add(43);
    list.add(44);
    list.add(null);
    Assert.assertTrue(list.contains(42));
    Assert.assertTrue(list.contains(42) && list.contains(43));
    Assert.assertTrue(list.contains(null));
    Assert.assertThat(list, hasItem(42));
    Assert.assertThat(list, hasItems(42, 43));
    Assert.assertThat(list, hasItem(equalTo(42)));
    Assert.assertThat(list, hasItems(equalTo(42), equalTo(43)));
    Assert.assertThat(list, hasItem(nullValue(Integer.class)));
}

@Test
public void testBothWith() {
    final List<Integer> list = new ArrayList<Integer>();
    list.add(42);
    list.add(43);
    list.add(44);
    Assert.assertTrue(
        list.contains(42) && list.contains(43));
    Assert.assertThat(
        list, both(hasItem(42)).and(hasItem(43)));
    Assert.assertThat(
        list,
both(hasItem(42)).and(hasItem(43)).and(hasItem(44)));
}

@Test
public void testBothEither() {
    final List<Integer> list = new ArrayList<Integer>();
    list.add(42);
    list.add(43);
    list.add(44);
    Assert.assertTrue(
        list.contains(42) || list.contains(99));
    Assert.assertThat(
        list, either(hasItem(42)).or(hasItem(99)));
    Assert.assertThat(
        list,
either(hasItem(42)).or(hasItem(43)).or(hasItem(99)));
}

@Test
public void testSum() {
    final List<Integer> list = new ArrayList<Integer>();
    list.add(42);
    list.add(43);
    list.add(44);
    Assert.assertThat(129, Sum.sum(list));
}
}

@SuppressWarnings("rawtypes")
class Sum extends BaseMatcher {

```

```
@Override
public boolean matches(final Object value) {
    return this.sum == (Integer)value;
}

@Override
public void describeTo(final Description description) {
    description.appendText(this.coll + " => " + this.sum);
}

private final Collection<Integer> coll;
private final int sum;
public Sum(final Collection<Integer> coll) {
    int s = 0;
    for (final Integer v : coll)
        s += v;
    this.coll = coll;
    this.sum = s;
}
@Factory
public static Matcher sum(final Collection<Integer> coll) {
    return new Sum(coll);
}
}
```

Wie man sieht, kann man auch eigene `Matcher` schreiben...

3.9 Erweiterungen von JUnit

JUnit lädt zu weiteren eignen Erweiterungen ein. Im Folgenden wird gezeigt, wie mittels JUnit ein kleines Werkzeug entwickelt werden kann, welches für Performance-Messungen verwendet werden kann.

Hier zunächst eine Klasse, deren Methoden nicht die performantesten sind (sowohl `gcd` als auch `lcm` benötigen jeweils 100 ms):

```
package appl;

public class MathService {

    private void sleep(final int millis) {
        try {
            Thread.sleep(millis);
        }
        catch (final InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public int gcd(int x, int y) {
        if (x <= 0 || y <= 0)
            throw new IllegalArgumentException("bad arguments");
        this.sleep(100);
        while (x != y) {
            if (x > y)
                x -= y;
            else
                y -= x;
        }
        return x;
    }

    public int lcm(final int x, final int y) {
        if (x <= 0 || y <= 0)
            throw new IllegalArgumentException("bad arguments");
        this.sleep(100);
        int a = x;
        int b = y;
        while (a != b) {
            if (a < b)
                a += x;
            else
                b += y;
        }
        return a;
    }
}
```

Hier zunächst der Test, der die Performance-Erweiterungen benutzt:

```
package test;

import org.junit.Assert;
import org.junit.Before;
import org.junit.ClassRule;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TestRule;

import appl.MathService;
import util.PerfClassRule;
import util.PerfTestRule;
import util.Repeat;

public class MathServiceTest {

    private MathService mathService;

    @ClassRule
    public static PerfClassRule classRule = new PerfClassRule(10);

    @Rule
    public TestRule testRule = new PerfTestRule(classRule);

    @Before
    public void before() {
        //System.out.println("before");
        this.mathService = new MathService();
    }

    @Test
    @Repeat(10)
    public void testGcd() {
        //System.out.println("\ttestGcd");
        Assert.assertEquals(5, this.mathService.gcd(25, 5));
    }

    @Test
    @Repeat(5)
    public void testLcm() {
        //System.out.println("\ttestLcm");
        Assert.assertEquals(50, this.mathService.lcm(25, 10));
    }
}
```

Die Ausgaben:

testGcd ==> 10043

testLcm ==> 5035

Und hier schließlich der Quellcode der benutzten Erweiterungen (das nähere Studium sei auch hier dem Leser / der Leserin überlassen):

```
package util;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Repeat {
    public abstract int value();
}
```

```
package util;

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

import org.junit.rules.TestRule;
import org.junit.runner.Description;
import org.junit.runners.model.Statement;

public class PerfClassRule implements TestRule {

    private final int times;
    private final Map<String, Long> map = new HashMap<>();
    private long startTime;

    public PerfClassRule(final int times) {
        this.times = times;
    }

    public void begin(final String methodName) {
        // System.out.println(">>");
        final Long duration = this.map.get(methodName);
        if (duration == null) {
            this.map.put(methodName, 0L);
        }
        this.startTime = System.currentTimeMillis();
    }

    public void end(final String methodName) {
        // System.out.println("<<");
        final long stopTime = System.currentTimeMillis();
        final Long duration = this.map.get(methodName);
        if (duration == null)
            throw new AssertionError();
        this.map.put(methodName,
            duration + (stopTime - this.startTime));
    }

    @Override
```

```
public Statement apply(final Statement base,
    final Description description) {
    return new Statement() {
        @Override
        public void evaluate() throws Throwable {
            for (int i = 0; i < PerfClassRule.this.times; i++)
                base.evaluate();
            System.out.println();
            for (final Entry<String, Long> entry :
                PerfClassRule.this.map.entrySet()) {
                System.out.println(entry.getKey() + " ==> " +
                    entry.getValue());
            }
        }
    };
}
```

```
package util;

import org.junit.rules.TestRule;
import org.junit.runner.Description;
import org.junit.runners.model.Statement;

public class PerfTestRule implements TestRule {

    public final PerfClassRule classRule;
    public PerfTestRule(final PerfClassRule classRule) {
        this.classRule = classRule;
    }
    @Override
    public Statement apply(final Statement base,
        final Description description) {
        final String methodName = description.getMethodName();
        final Repeat repeat = description.getAnnotation(Repeat.class);
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
                System.out.print('.');
                PerfTestRule.this.classRule.begin(methodName);
                final int times = repeat == null ? 1 : repeat.value();
                for (int i = 0; i < times; i++) {
                    base.evaluate();
                }
                PerfTestRule.this.classRule.end(methodName);
            }
        };
    }
}
```


4

Testing – Übungen

4.1	Isbn.....	4-4
4.2	Trimmer	4-7
4.3	Event-Bus	4-9
4.4	Typesafe Properties	4-13
4.5	Resultate	4-17
4.5.1	Sprechende Namen	4-17
4.5.2	Unabhängigkeit	4-17
4.5.3	Strukturelle Einfachheit	4-18
4.5.4	Vollständigkeit	4-18
4.5.5	Unabhängigkeit von problematischen Ressourcen	4-20
4.5.6	Test als Spezifikation und Dokumentation	4-20

4 Testing – Übungen

Dieses Kapitel enthält einige Übungen, in denen die zu testenden Klassen bereits vorgegeben sind. Die Aufgabe besteht jeweils darin, diese Klassen nun "im Nachhinein" zu testen.

- In der ersten Übung geht's um den Test einer Klasse namens `Isbn` (Objekte dieser Klassen repräsentieren ISBN-Nummern).
- In der zweiten Übung wird eine `Trimmer`-Klasse getestet. Die `Trimmer`-Klasse liest eine Textdatei ein und gibt diese in komprimierter Form (in "getrimmter" Form) wieder aus.
- In der dritten Übung schließlich geht's um das Testen einer vorgegebenen `EventBus`-Klasse (diese Klasse ist nicht ganz trivial).
- In der vierten Übung geht's um den Test einer Klasse, welche den typsicheren Zugriff auf Properties erlaubt: um das Testen einer vom Autor dieses Skripts entwickelten Klasse namens `SafeProperties`. (Auch diese Klasse ist nicht ganz trivial...)

Natürlich müssen nicht alle Aufgaben gelöst werden – es handelt sich nur um Angebote. Die Lösungen finden sich in einer separaten Workspace.

4.1 Isbn

Objekte der Klasse `Isbn` repräsentieren ISBN-Nummern. Ein `Book`-Objekt z.B. könnte neben dem Title des Buches eine Referenz auf ein ISBN-Objekt besitzen.

Eine ISBN-Nummer besteht aus vier Teilen: Land, Verlag, Nummer und Check-Zeichen. Land, Verlag, Nummer sind `Strings` – zusammengekommen müssen sie die Länge 9 besitzen. Das Check-Zeichen ist ein `char`.

Hier zwei beispielhafte ISBN:

1-111-11111-1

22-2222-222-2

Die `Isbn`-Klasse garantiert, dass ihre Objekte immutable sind.

```
package appl;

public class Isbn {

    public final String country;
    public final String publisher;
    public final String number;
    public final char check;

    public Isbn(
        final String country,
        final String publisher,
        final String number,
        final char check) {
        this.country = country;
        this.publisher = publisher;
        this.number = number;
        this.check = check;
        this.assertValid();
    }

    public Isbn(final String isbn) {
        final String[] tokens = isbn.split("-");
        if (tokens.length != 4)
            throw new IllegalArgumentException(
                "String must contain 4 tokens");
        if (tokens[3].length() != 1)
            throw new IllegalArgumentException(
                "check must contain exactly 1 character");
        this.country = tokens[0];
        this.number = tokens[1];
        this.publisher = tokens[2];
        this.check = tokens[3].charAt(0);
        this.assertValid();
    }

    private void assertValid() {
        assertNumeric(this.country);
        assertNumeric(this.publisher);
    }
}
```

```
        assertNumeric(this.number);
        assertNumeric(String.valueOf(this.check));
        if (this.externalForm().length() != 13)
            throw new IllegalArgumentException("length must be 13");
    }

    public String externalForm() {
        return this.country + "-" + this.publisher + "-" +
            this.number + "-" + this.check;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() +
            " [" + this.externalForm() + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + this.check;
        result = prime * result + this.country.hashCode();
        result = prime * result + this.number.hashCode();
        result = prime * result + this.publisher.hashCode();
        return result;
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj)
            return true;
        if (obj == null || this.getClass() != obj.getClass())
            return false;
        final Isbn other = (Isbn) obj;
        return this.country.equals(other.country)
            && this.publisher.equals(other.publisher)
            && this.number.equals(other.number)
            && this.check == other.check;
    }

    private static void assertNumeric(final String s) {
        if (s == null || s.length() == 0)
            throw new IllegalArgumentException(
                "illegal null or empty argument");
        for (int i = 0; i < s.length(); i++) {
            final char c = s.charAt(i);
            if (!Character.isDigit(c))
                throw new IllegalArgumentException(
                    s + " must be numeric");
        }
    }
}
```

Die Attribute der Klasse sind `public`: dies ist unproblematisch, da sie `final` sind.

Dem ersten Konstruktor werden vier Argumente übergeben, die zur Initialisierung der Attribute verwendet werden. Am Ende des Konstruktors wird `assertValid` aufgerufen.

Dem zweiten Konstruktor wird die ISBN in ihrer "external form" übergeben: als einziger String, dessen Bestandteile durch Bindestriche getrennt sind.

Die Methode `externalForm` liefert die ISBN in ihrer "external form" zurück.

Schließlich überschreibt die Klasse die `Object`-Methoden `toString`, `equals` und `hashCode`.

Testen Sie diese Klasse!

Beginnen Sie mit dem Test der Konstruktoren.

Die Testklasse enthält bereits eine erste Testmethode:

```
package test;
// ...
public class IsbnTest {

    @Test
    public void TestCreationWithEmptyArguments()
    {
        XAssert.assertThrows(IllegalArgumentException.class,
            () -> new Isbn("", "", "", ' '));
        XAssert.assertThrows(IllegalArgumentException.class,
            () -> new Isbn(null, null, null, ' '));
    }

    // TODO...
}
```

4.2 Trimmer

Die `trim`-Methode der folgenden `Trimmer`-Klasse liest eine Eingabedatei und produziert eine Ausgabedatei. Die Ausgabe enthält alle Zeilen der Eingabe – aber jeweils ohne führende Blanks. Leere Zeilen und Zeilen, die mit der Zeichenfolge `"/"` beginnen, werden nicht in die Ausgabe übernommen.

```
package appl;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class Trimmer {

    public void trim(final String inputFileName,
                    final String outputFileName) throws IOException {
        try (final BufferedReader reader = new BufferedReader(
            new InputStreamReader(
                new FileInputStream(inputFileName)))) {
            try (PrintWriter writer =
                new PrintWriter(outputFileName)) {
                String line;
                while ((line = reader.readLine()) != null) {
                    line = line.trim();
                    if (line.length() > 0 && !line.startsWith("/"))
                        writer.println(line);
                }
            }
        }
    }
}
```

Eine mögliche Anwendung:

```
package test;

import appl.Trimmer;

public class TrimmerDemo {
    // Die main-Methode...
    public static void main(final String[] args)
        throws Exception {
        final Trimmer trimmer = new Trimmer();
        trimmer.trim("src/test/TrimmerDemo.java",
                    "src/test/TrimmerDemo.txt");
    }
}
```

Die Ausgabe (Trimmer.txt):

```
package test;
import appl.Trimmer;
public class TrimmerDemo {
    public static void main(final String[] args)
        throws Exception {
        final Trimmer trimmer = new Trimmer();
        trimmer.trim("src/test/TrimmerDemo.java",
            "src/test/TrimmerDemo.txt");
    }
}
```

Wie der (manuelle) Blick in die Ausgabedatei zeigt, funktioniert das Programm wie erwartet...

Testen Sie diese Klasse!

Der Rumpf der Testklasse ist bereits gegeben:

```
package test;
// ...
public class TrimmerTest {

    @Test
    public void TestTrimmer()
    {
        final Trimmer trimmer = new Trimmer();
        // TODO...
    }

}
```

Hinweis: Die Probleme, auf die wir hier stoßen, können leicht vermieden werden, wenn die Parametrisierung der `trim`-Methode geändert wird:

```
public void trim(Reader reader, Writer writer)
```

`Reader` ist eine Basisklasse, von welcher `StringReader` und `InputStreamReader` abgeleitet sind; von `Writer` sind `StringWriter` und `PrintWriter` abgeleitet.

Zeigen Sie dann schließlich auch den Einsatz von `Trimmer.trim` in einer "produktiven" Anwendung.

Welche Lehren lassen sich aus diesem Beispiel ableiten?

4.3 Event-Bus

Ein `EventBus` kann verwendet werden, um die Komponenten eines Systems lose miteinander zu koppeln.

Bei einem `EventBus` können sich "Subscriber" registrieren, welche an bestimmten Events (Politik-Nachrichten, Sport-Nachrichten, Yellow-Nachrichten etc.) interessiert sind. "Publisher" können Events auf den `EventBus` feuern (sie veröffentlichen Politik-Nachrichte, Sport-Nachrichten etc.). Die von den Publishers gefeuerten Events müssen dann vom `EventBus` an die Subscriber zugestellt werden.

Hier die zu testende Klasse `EventBus`:

```
package util;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Consumer;

public class EventBus {

    private final Map<Class<?>, List<Consumer<?>>> map
        = new HashMap<>();

    public <T> void register(
        final Class<T> eventType,
        final Consumer<T> subscriber) {
        this.map.computeIfAbsent(eventType,
            key -> new ArrayList<Consumer<?>>()).add(subscriber);
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public void fire(final Object event) {
        for(Class<?> type = event.getClass();
            type != null; type = type.getSuperclass()) {
            final List<Consumer<?>> subscribers
                = this.map.get(type);
            if (subscribers != null)
                for (final Consumer subscriber : subscribers)
                    subscriber.accept(event);
        }
    }
}
```


Hier einige Event-Klasse, die im Test verwendet werden können:

```
package test;

import java.util.Objects;

public class FooEvent {

    public final String value;

    public FooEvent(final String value) {
        Objects.requireNonNull(value);
        this.value = value;
    }

    @Override
    public int hashCode() {
        return this.value.hashCode();
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj)
            return true;
        if (obj == null || this.getClass() != obj.getClass())
            return false;
        final FooEvent other = (FooEvent) obj;
        return this.value.equals(other.value);
    }

    @Override
    public String toString() {
        return "FooEvent [value=" + this.value + "]";
    }
}
```

```
package test;

public class BarEvent {

    public final int value;

    public BarEvent(final int value) {
        this.value = value;
    }

    @Override
    public int hashCode() {
        return this.value;
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj)
            return true;
        if (obj == null || this.getClass() != obj.getClass())
            return false;
        final BarEvent other = (BarEvent) obj;
```

```
        return this.value == other.value;
    }

    @Override
    public String toString() {
        return "BarEvent [value=" + this.value + "]";
    }
}
```

Hier zunächst eine beispielhafte Anwendung der `EventBus`-Klasse:

```
package test;

import util.EventBus;

public class EventBusDemo {

    public static void main(final String[] args) {
        final EventBus bus = new EventBus();

        bus.register(FooEvent.class, e -> {
            final String v = e.value;
            System.out.println("handler1: " + v);
        });

        bus.register(BarEvent.class, e -> {
            final int v = e.value;
            System.out.println("handler2: " + v);
        });

        bus.register(BarEvent.class, e -> {
            final int v = e.value;
            System.out.println("handler3: " + v);
        });

        bus.register(Object.class, e -> {
            System.out.println("handler4: " + e);
        });

        bus.fire(new FooEvent("Hello"));
        bus.fire(new BarEvent(42));
        bus.fire("Schall und Rauch");
    }
}
```

Die Ausgaben:

```
handler1: Hello
handler4: FooEvent [value=Hello]
handler2: 42
handler3: 42
handler4: BarEvent [value=42]
handler4: Schall und Rauch
```

Der Rumpf der Testklasse ist bereits gegeben:

```
package test;  
  
import org.junit.Test;  
  
public class EventBusTest {  
  
    @Test  
    public void TestStringEventWithOneSubscriber() {  
        // TODO  
    }  
  
    @Test  
    public void TestStringEventsWithOneSubscriber() {  
        // TODO  
    }  
  
    @Test  
    public void TestThreeSubscribers() {  
        // TODO  
    }  
  
    @Test  
    public void TestPolymorphy() {  
        // TODO  
    }  
}
```

4.4 Typesafe Properties

Auf die Einträge eines `Properties`-Objekts soll typsicher zugegriffen werden können. Dazu dient die Klasse `SafeProperties`, die im Folgenden zu testen ist.

Der Zugriff auf die Werte der Properties wird mittels eines Interfaces erfolgen, welches jeweils mittels des Dynamic-Proxies-Mechanismus implementiert wird.

Die zu testende Klasse (`SafeProperties`) solle hier nicht im Einzelnen vorgestellt werden (siehe den Quellcode in den Packages `util` und `util.annotation`). Stattdessen sollen hier zunächst einige Anwendungen dieser Klasse gezeigt werden (`test.SafeDemo`):

Angenommen, es existiert folgendes Interface:

```
@PropertiesInterface
public static interface Alpha {
    String red();
    String green();
    String blue();
}
```

Alle Methoden sind parameterlos und liefern `Strings` zurück.

Dieses Interface könnte nun wie folgt für den sicheren Zugriff auf Properties-Einträge verwendet werden:

```
public static void demoAlpha() {
    final Properties props = new Properties();
    props.setProperty("red", "ROT");
    props.setProperty("green", "GRUEN");
    props.setProperty("blue", "BLAU");
    final Alpha alpha = SafeProperties.load(Alpha.class, props);
    System.out.println(alpha.red());
    System.out.println(alpha.green());
    System.out.println(alpha.blue());
}
```

Die Ausgaben:

```
ROT
GRUEN
BLAU
```

Angenommen, wir definieren ein "intelligenteres" Interface:

```
@PropertiesInterface
public static interface Beta {
    int i();
    double d();
    char c();
    boolean b();
}
```

Die Methoden liefern nun nicht mehr Strings zurück, sondern int-, double-, char- und boolean-Werte.

Hier eine Anwendung:

```
public static void demoBeta() {
    final Properties props = new Properties();
    props.setProperty("i", "42");
    props.setProperty("d", "3.14");
    props.setProperty("c", "x");
    props.setProperty("b", "true");
    final Beta beta = SafeProperties.load(Beta.class, props);
    System.out.println(beta.i());
    System.out.println(beta.d());
    System.out.println(beta.c());
    System.out.println(beta.b());
}
```

Die Ausgaben:

```
42
3.14
x
true
```

Der Rahmen des Testprogramms ist bereits vorgegeben:

```
package test;

import java.util.Properties;

import org.junit.Test;

import util.SafeProperties;
import util.annotations.Default;
import util.annotations.PropertiesInterface;

@SuppressWarnings("unused")
public class SafePropertiesTest {

    @PropertiesInterface
    public static interface Alpha {
        String red();
        String green();
        String blue();
    }

    @Test
    public void testAlpha() {
        final Properties props = new Properties();
        props.setProperty("red", "ROT");
        props.setProperty("green", "GRUEN");
        props.setProperty("blue", "BLAU");
        final Alpha alpha = SafeProperties.load(Alpha.class, props);
        // TODO...
    }

    @Test
    public void testAlphaMissingPropertyException() {
        final Properties props = new Properties();
```

```
        props.setProperty("red", "ROT");
        props.setProperty("blue", "BLAU");
        // TODO...
    }

    @PropertiesInterface
    public static interface Beta {
        int i();
        double d();
        char c();
        boolean b();
    }

    @Test
    public void testBeta() {
        final Properties props = new Properties();
        props.setProperty("i", "42");
        props.setProperty("d", "3.14");
        props.setProperty("c", "x");
        props.setProperty("b", "true");
        final Beta beta = SafeProperties.load(Beta.class, props);
        // TODO...
    }

    @Test
    public void testBetaMissingPropertyException() {
        final Properties props = new Properties();
        props.setProperty("i", "42");
        props.setProperty("c", "x");
        // TODO...
    }

    @Test
    public void testBetaParserException() {
        final Properties props = new Properties();
        props.setProperty("i", "42AAA");
        props.setProperty("d", "3.14");
        props.setProperty("c", "x");
        props.setProperty("b", "true");
        // TODO...
    }

    @PropertiesInterface
    public static interface Gamma {
        @Default("42") int i();
        double d();
        @Default("z") char c();
        @Default("false") boolean b();
    }

    @Test
    public void testGamma() {
        final Properties props = new Properties();
        props.setProperty("i", "77");
        props.setProperty("d", "3.14");
        props.setProperty("b", "true");
        final Gamma gamma = SafeProperties.load(Gamma.class, props);
        // TODO...
    }
}
```

```

@Test
public void testGammaMissingPropertyException() {
    final Properties props = new Properties();
    // TODO...
}
}

```

Neben der Klasse `SafeProperties` existiert die Klasse `SafeMessages`. Auch hier ist der Rahmen des Testprogramms bereits vorgegeben:

```

package test;

import java.util.Properties;

import org.junit.Test;

import util.SafeMessages;
import util.annotations.Default;
import util.annotations.MessagesInterface;

@SuppressWarnings("unused")
public class SafeMessagesTest {

    @MessagesInterface
    public static interface Alpha {
        String greeting();
        String sum(int x, int y, int sum);
    }

    @MessagesInterface
    public static interface Beta {
        @Default("Hallo Welt")
        String greeting();

        @Default("Die Summe von ?0 und ?1 ist ?2")
        String sum(int x, int y, int sum);
    }

    @Test
    public void testAlpha() {
        final Properties props = new Properties();
        props.setProperty("greeting", "Hello World");
        props.setProperty("sum", "?2 is the sum of ?0 and ?1");
        final Alpha alpha = SafeMessages.load(Alpha.class, props);
        // TODO...
    }

    @Test
    public void testAlphaMissingPropertyException() {
        final Properties props = new Properties();
        props.setProperty("greeting", "Hello World");
        // TODO...
    }

    @Test
    public void testBeta() {
        // TODO...
    }
}

```

4.5 Resultate

Bei den bisherigen Tests sind bereits einige wichtige Eigenschaften von Testklassen deutlich geworden:

4.5.1 Sprechende Namen

Testmethoden sollten – wie alle anderen Methoden natürlich auch – sprechende Namen haben (auch wenn das zuweilen dazu führt, dass recht lange Namen erforderlich sind). Testmethoden sollten also nicht aufwendig kommentiert werden müssen.

4.5.2 Unabhängigkeit

Testmethoden müssen voneinander vollständig unabhängig sein. Eine Testmethode darf sich insbesondere nicht auf ein Ergebnis verlassen, welches von einer anderen Testmethode berechnet wurde. Instanzvariablen haben also in Testklassen nichts zu suchen (es sei denn, man benutzt `@Before`-Methoden). Testmethoden sollten also in sich abgeschlossen sein.

Hier ein katastrophales Beispiel:

```
// Ein katastrophales Beispiel!!!

private Stack<String> stack;

@Test
public void testCreate() {
    this.stack = new Stack<String>(2);
    Assert.assertEquals(0, this.stack.Count);
}

@Test
public void testPushTwoElements() {
    this.stack.push("one");
    this.stack.push("two");
    Assert.assertEquals(2, this.stack.size());
}

@Test
public void testPushWithException() {
    try {
        this.stack.push("three");
        Assert.fail();
    }
    catch(IllegalStateException e) {
    }
}
```

Der Entwickler hat sich offenbar gedacht, dass die Testmethoden in exakt derjenigen Reihenfolge ausgeführt werden, in welcher sie textuell hinterlegt sind. Genau darauf aber ist kein Verlass! Würde z.B. die zweite der obigen Testmethode als erste ausgeführt werden, würde einfache eine `NullPointerException` geworfen.

4.5.3 Strukturelle Einfachheit

Testmethoden sollten keine komplexen Kontrollstrukturen enthalten. Beim Testen des `push`-Verhaltens eines `Stacks` hätte man z.B. folgende Methode schreiben können:

```
@Test
public void testPush() {
    Stack<String> stack = new Stack<>(2);
    String[] elements = new String[] { "one", "two", "three" };
    int i = 0;
    try {
        while (i < elements.size()) {
            stack.push(elements[i]);
            i++;
        }
        Assert.fail();
    }
    catch(IllegalStateException e) {
        if (i != 2)
            Assert.fail();
    }
}
```

Diese Testmethode ist wesentlich komplexer als die zu testende Methode – sie enthält wahrscheinlich eher Fehler als die `push`-Methode. Folgende Testmethode ist wesentlich verständlicher:

```
@Test
public void testPush() {
    Stack<String> stack = new Stack<>(2);
    stack.push("one");
    stack.push("two");
    try {
        stack.push("three");
        Assert.fail();
    }
    catch(IllegalStateException e) {
    }
}
```

4.5.4 Vollständigkeit

Im strengen Wortsinn kann ein Test natürlich niemals "vollständig" sein. Angenommen, wir wollen eine Methode testen, welche die Wurzel einer Gleitkommazahl berechnet. Leider gibt's unendlich viele solcher Zahlen – die Wurzel-Methode kann also niemals mit allen Zahlen getestet werden.

In der Regel bildet man sog. Äquivalenzklassen. Eine Äquivalenzklasse ist ein Bereich von Eingabe-Werten für eine zu testende Methode. Diese Bereiche werden derart festgelegt, dass man annehmen kann, dass eine Methode für alle Werte eines jeweiligen Bereichs sich gleichartig verhalten wird. Bei der Wurzel-Methode wird man folgende Bereiche festlegen: den Bereich der negativen Zahlen (hier muss die Methode eine `ArgumentException` werfen); den Bereich mit dem einzigen Wert 0

– hier muss 0 zurückgeliefert werden; den Bereich der Werte größer als 0 aber kleiner als 1 – für alle Zahlen dieses Bereichs muss ein Wert zurückgeliefert werden, der größer ist als der Eingabe-Wert; der Bereich mit dem Wert 1 – hier muss 1 geliefert werden; und schließlich den Bereich aller Werte größer als 1 – hier muss ein Wert zurückgeliefert werden, welcher kleiner als der Eingabewert ist. Für alle diese Bereiche wird dann ein Stellvertreter-Wert ausgewählt, mit welchem der Test ausgeführt wird.

Hat eine Klasse eine Vielzahl von Methoden, die auf bestimmte Weise zusammenspielen (wie z.B. im `Stack`-Beispiel), muss natürlich dieses Zusammenspiel getestet werden (`pop` nimmt die Wirkung von `push` zurück etc.).

Insbesondere sollte das Verhalten bei illegalen Eingaben (nicht positive Werte für `MathService.gcd`) resp. illegalen Sequenzen von Methoden-Aufrufen (`Stack.pop` im Zustand `Stack.isEmpty` etc.) getestet werden.

Triviale Tests sollten nach Meinung des Autors vermieden werden. Sei etwa folgende Klasse gegeben:

```
class Point {  
    public int x;  
    public int y;  
}
```

Folgender Test ist überflüssig:

```
@Test  
public void TestPointX() {  
    Point p = new Point();  
    p.x = 42;  
    Assert.assertEquals(42, p.x);  
}
```

Solche Tests sind langweilig – sie verleiden einem den Spaß am Testen. Und man sollte sich schließlich nicht dümmer stellen als man ist...

4.5.5 Unabhängigkeit von problematischen Ressourcen

Ein Test, der z.B. von externen Dateien abhängt, ist problematisch. Die Eingabedatei, aus welcher die zu testende Methode liest, mag versehentlich verändert worden sein – oder gelöscht worden sein. Dann wird die Methode natürlich alles andere liefern als das erwartete Ergebnis. Sofern das Resultat der Methode eine Ausgabe ist, muss die Ausgabedatei im Testprogramm wieder eingelesen werden...

In solchen Fällen sollte man sich überlegen, ob die Eingaben nicht direkt im Testprogramm hinterlegt werden können (z.B. als `String`, welcher dann von einem `StringReader` gelesen werden kann, der seinerseits als `Reader` der zu testenden Methode übergeben wird; die Ausgaben können mittels eines `StringWriter` im Hauptspeicher abgelegt werden, wobei der `StringWriter` der zu testenden Methode als `Writer` übergeben wird (siehe das `Trimmer`-Beispiel). Beim produktiven Einsatz der Methode können dann `InputStreamReader` bzw. `PrintWriter` übergeben werden.

Man möchte beim Testen auch gern von Datenbanken unabhängig sein – auch dies sind problematische externe Ressourcen. Oder von Socket-Verbindungen etc. Eine solche Unabhängigkeit kann aber erst dann realisiert werden, wenn Mocking-Werkzeuge eingesetzt werden.

4.5.6 Test als Spezifikation und Dokumentation

Eine Testklasse sollte man als Spezifikation der in ihr getesteten Klasse(n) ansehen können – und damit zugleich auch als deren Dokumentation...

5

Mocking – Ein Miniframeframework

5.1	Start.....	5-5
5.2	Dynamic Proxies.....	5-12
5.3	Call-Objekte.....	5-16
5.4	Assertions.....	5-20
5.5	Arrangements.....	5-23
5.6	Ein Eclipse-Test.....	5-29

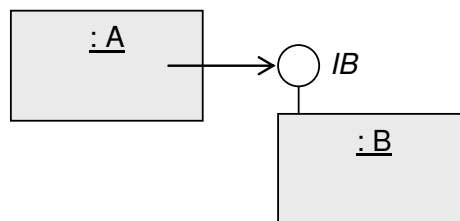
5 Mocking – Ein Miniframework

Worum geht's beim Mocking?

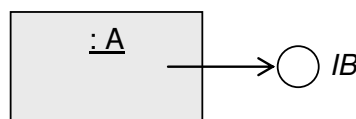
Seien zwei Klassen `A` und `B` gegeben. Die Klasse `A` benötigt ein Objekt, dessen Klasse ein Interface `IB` implementiert. Aus irgendwelchen Gründen steht beim Testen aber eine solche Klasse (z.B. `B`) nicht zur Verfügung (sie ist noch nicht fertig; oder sie benötigt den Zugriff auf eine Datenbank, deren Inhalt dem Wechsel unterliegt und also zur wiederholten Produktion von Testergebnissen nicht brauchbar ist; oder `B` beruht auf Socket-Verbindungen, deren Problematik aber beim Test der Klasse `A` nicht gewünscht ist...)

Dann stellt sich die Frage, wie die Klasse `A` getestet werden kann, ohne dass eine "richtige" Implementierung von `IB` genutzt wird. Es soll getestet werden, ob `A` mit einem `IB`-Objekt korrekt zusammenspielt (Interaktionstest). Es muss also das "Innere" von `A` getestet werden: ruft `A` Methoden von `IB` in erwarteter Reihenfolge und mit den erwarteten Argumenten auf? (Endoskopischer Test). Die Klasse `A` wird bei solchen Test somit als White-Box betrachtet.

Hier das Objektdiagramm zur oben beschriebenen Problematik:



Was tun, wenn `B` fehlt?:



Man benötigt einen Mock für `IB`.

Solche Mock-Objekte können "automatisch" produziert werden. Es gibt eine Vielzahl von Mock-Werkzeugen, welche eine solche automatische Produktion ermöglichen: EasyMock, JMock, Mockito...

All diese Werkzeugen zeichnen sich auf den ersten Blick dadurch aus, dass sie offenbar "Magic" enthalten. Tatsächlich haben deren Entwickler aber allesamt natürlich nur mit Wasser gekocht.

Um diese Magic aufzuklären und ein tieferes Verständnis dieser Mock-Werkzeuge zu ermöglichen, wird in diesem Kapitel ein kleines eigenes Mock-Werkzeug entwickelt, welches sich an Mockito orientiert – Mockito ist eines der "moderneren" Werkzeuge. Mockito implementiert das Muster "Arrange-Act-Assert" (ältere Mock-Werkzeuge orientieren sich am sog. "Record-Replay"-Muster). Das Werkzeug wird in einer Reihe von sechs Schritten implementiert werden.

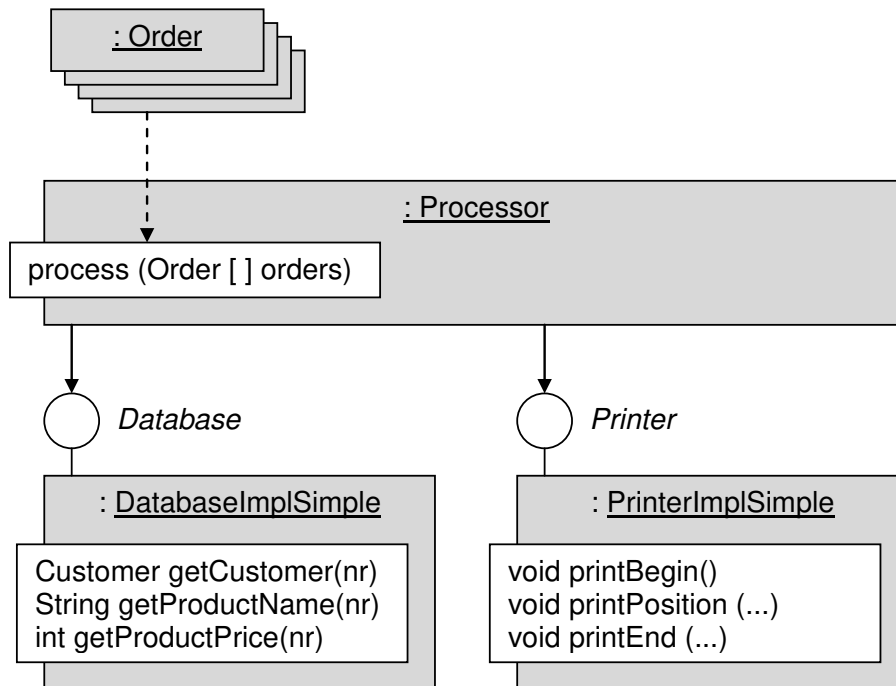
Um ein solches Werkzeug zu entwickeln, benötigt man natürlich ein Beispiel mit einer Testklasse, in welcher gemockte Objekte genutzt werden. Als Beispiel dient ein kleines Stapelverarbeitungs-Programm, welches aufgrund einer Auftragsliste eine Druckausgabe erzeugt – und dabei referenzielle Zugriffe auf eine Datenbank ausführt.

Das Mock-Werkzeug benutzt insbesondere die Technik der "Dynamic Proxies". Es handelt sich dabei um eine grundlegende Java-Technik, die natürlich auch in anderen Kontexten sinnvoll verwendet werden kann. Für diejenigen der Leser / Leserinnen, die diese Technik noch nicht kennen oder noch niemals verwendet haben, bereichert dieses Kapitel daher auch deren Java-Kenntnisse – ein kleiner Seiteneffekt...

5.1 Start

In diesem ersten Abschnitt wird das komplette ablauffähige Programm vorgestellt. Für dieses Programm existieren zunächst keinerlei automatischer Tests.

Zunächst ein kleines Objektdiagramm, welches das im Folgenden vorgestellte Programmsystem veranschaulicht:



Das Programm benötigt eine Datenbasis, welche Kunden- und Produkt-Informationen enthält.

Für die Kunden-Informationen gibt's eine eigene Klasse: `Customer`:

```
package domain;

public class Customer {

    public final int nr;
    public final String name;

    public Customer(final int nr, final String name) {
        this.nr = nr;
        this.name = name;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() +
            " [" + this.nr + ", " + this.name + "];"
    }

}
```


Für Produkte gibt's merkwürdigerweise keine(!) eigene Klasse. (Der Grund: wir wollen später bestimmte Probleme demonstrieren, die wir nicht demonstrieren können, wenn's analog zur Klasse `Customer` auch eine Klasse `Product` geben würde...)

Wir definieren ein Interface `Database`, welches die Zugriffsmethoden auf diese Tabellen spezifiziert:

```
package services.ifaces;

import domain.Customer;

public interface Database {

    public abstract Customer getCustomer(int nr);
    public abstract String getProductName(int nr);
    public abstract int getProductPrice(int nr);
}
```

Das Programm benutzt eine sehr einfache Implementierung dieses Interfaces – die Datenbank existiert einfach im Hauptspeicher:

```
package services.impl;

import java.util.HashMap;
import java.util.Map;

import domain.Customer;
import services.ifaces.Database;

public class DatabaseImplSimple implements Database {

    private static class ProductData {
        public final String name;
        public final int price;
        public ProductData(final String name, final int price) {
            this.name = name;
            this.price = price;
        }
    }

    private final Map<Integer, Customer> customers
        = new HashMap<>();
    private final Map<Integer, ProductData> products
        = new HashMap<>();

    private void addCustomer(final int nr, final String name) {
        this.customers.put(nr, new Customer(nr, name));
    }

    private void addProduct(final int nr,
        final String name, final int price) {
        this.products.put(nr, new ProductData(name, price));
    }

    public DatabaseImplSimple() {
        this.addCustomer(1000, "Nowak");
        this.addCustomer(2000, "Rueschenpoehler");
    }
}
```

```
this.addCustomer(3000, "Kreuzer");
this.addProduct(100, "Jever", 20);
this.addProduct(200, "Veltins", 40);
this.addProduct(300, "Bitburger", 60);
}

@Override
public Customer getCustomer(final int nr) {
    final Customer c = this.customers.get(nr);
    if (c == null)
        throw new RuntimeException("Customer not found");
    return c;
}

@Override
public String getProductName(final int nr) {
    final ProductData p = this.products.get(nr);
    if (p == null)
        throw new RuntimeException("Product not found");
    return p.name;
}

@Override
public int getProductPrice(final int nr) {
    final ProductData p = this.products.get(nr);
    if (p == null)
        throw new RuntimeException("Product not found");
    return p.price;
}
}
```

Man sieht: Kunden sind als `Customer` repräsentiert, Produkte nur `ProductData`-Objekte, die als solche außerhalb der Klasse nicht sichtbar sind.

Es existieren also folgende Kunden resp. Produkte:

1000 Nowak
2000 Rueschenpoehler
3000 Kreuzer

100 Jever 20
200 Veltins 40
300 Bitburger 60

Das Programm benutzt als Eingabe eine Liste von `Order`-Objekten. Eine `Order` besitzt die Attribute `customerNr`, `productNr` und `amount`:

```
package domain;

public class Order {

    public final int customerNr;
    public final int productNr;
    public final int amount;

    public Order(final int customerNr,
                 final int productNr, final int amount) {
        this.customerNr = customerNr;
        this.productNr = productNr;
        this.amount = amount;
    }

    @Override
    public String toString() { ... }
}
```

Die Liste dieser `Order`-Objekte ist nach Kunden gruppiert. Das erste Beispiel-Programm benutzt folgende beispielhafte Eingabe:

```
Order[] orders = new Order[] {
    new Order(1000, 100, 1),
    new Order(1000, 200, 1),
    new Order(2000, 100, 2),
    new Order(3000, 300, 3),
};
```

Die Aufgabe des zu testenden Algorithmus besteht nun darin, diese Liste in "verständlicher" Form auszudrucken. Das vorliegende Programm z.B. erzeugt aufgrund der obigen `Order`-Liste und aufgrund der oben vorgestellten beispielhaften Datenbasis folgende Console-Ausgaben:

Orders

Nowak	Jever	20	1	20
Nowak	Veltins	40	1	40
Rueschenpoehler	Jever	20	2	40
Kreuzer	Bitburger	60	3	180
				=====
				280

Die Ausgabe ist unschwer zu interpretieren.

Sie beginnt mit einer Kopfzeile ("Orders") und endet mit der Summe aller Positionswerte. Ein Positionszeile besteht aus dem Namen des Kunden, dem Namen des Produkts, der Bestellmenge, dem Einzelpreis des Produkts und dem Positionswert (Menge * Einzelpreis). Die Ausgabe wird abgeschlossen mit der Summe aller Positionswerte.

Für die Produktion der Ausgabe benutzt das Programm einen Printer, welcher über das Interface `Printer` spezifiziert ist:

```
package services.ifaces;  
  
import domain.Customer;  
  
public interface Printer {  
  
    public abstract void printBegin();  
    public abstract void printPosition(Customer customer,  
        String productName, int price, int amount, int value);  
    public abstract void printEnd(int sum);  
}
```

Die Bedeutung der Methoden dieses Interfaces sollte klar sein.

Das Programm benutzt eine einfache Implementierung dieses Interfaces namens `PrinterImplSimple`:

```
package services.impl;  
  
import domain.Customer;  
import services.ifaces.Printer;  
  
public class PrinterImplSimple implements Printer {  
  
    @Override  
    public void printBegin() {  
        System.out.printf("Orders\n");  
    }  
  
    @Override  
    public void printPosition(final Customer customer,  
        final String productName, final int price,  
        final int amount, final int value) {  
        System.out.printf("\t%-20s %-10s %5d %5d %5d\n",  
            customer.name, productName, price, amount, value);  
    }  
  
    @Override  
    public void printEnd(final int totalSum) {  
        System.out.printf("\t\t\t\t\t\t\t\t====\n");  
        System.out.printf("\t\t\t\t\t\t\t\t%5d\n", totalSum);  
    }  
}
```

Der Verarbeitungs-Algorithmus ist in der Klasse `Processor` implementiert:

```
package appl;

import domain.Customer;
import domain.Order;
import services.ifaces.Database;
import services.ifaces.Printer;

public class Processor {

    private final Database database;
    private final Printer printer;

    public Processor(final Database database, final Printer printer) {
        this.printer = printer;
        this.database = database;
    }

    public void process(final Order[] orders) {
        this.printer.printBegin();
        int sum = 0;
        for (int index = 0; index < orders.length; index++) {
            final Order order = orders[index];
            final Customer customer =
                this.database.getCustomer(order.customerNr);
            final String productName =
                this.database.getProductName(order.productNr);
            final int productPrice =
                this.database.getProductPrice(order.productNr);
            final int value = order.amount * productPrice;
            this.printer.printPosition(customer, productName,
                productPrice, order.amount, value);
            sum += value;
        }
        this.printer.printEnd(sum);
    }
}
```

Dem Konstruktor von `Processor` muss eine `Database` und ein `Printer` übergeben werden. Wir haben es also mit Constructor-Injection zu tun.

Der `process`-Methode wird ein Array von `Orders` übergeben. Für jede `Order` wird der `Customer`, der Name und der Preis des Produkts ermittelt und dann jeweils eine Positionszeile gedruckt. Am Ende wird die Gesamtsumme aller Positionswerte ausgegeben.

Und hier schließlich das Hauptprogramm (dies erst entscheidet darüber, mit welchem `Printer` und welcher `Database` tatsächlich gearbeitet wird – mit einem `PrinterImplSimple` und einer `DatabaseImplSimple`):

```
package appl;

import domain.Order;
import services.ifaces.Database;
import services.ifaces.Printer;
import services.impl.DatabaseImplSimple;
import services.impl.PrinterImplSimple;

public class Application {

    public static void main(final String[] args) {

        final Order[] orders = new Order[] {
            new Order(1000, 100, 1),
            new Order(1000, 200, 1),
            new Order(2000, 100, 2),
            new Order(3000, 300, 3),
        };

        final Database database = new DatabaseImplSimple();
        final Printer printer = new PrinterImplSimple();
        final Processor processor = new Processor(database, printer);

        processor.process(orders);
    }
}
```

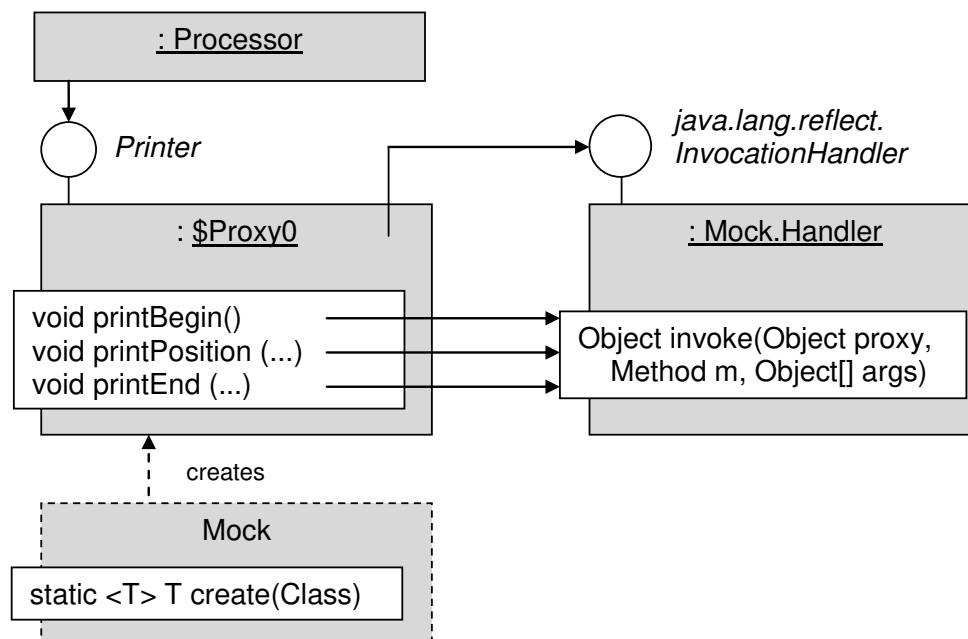
5.2 Dynamic Proxies

Im Folgenden wird der erste Schritt hin zu einem allgemein verwendbaren Mock-Framework vorgestellt. Alle Klassen dieses Frameworks existieren im package `uti.mock`.

Das Ziel der folgenden Bemühungen besteht darin, die `Processor`-Klasse zu testen. Ein `Processor` ist abhängig von zwei Objekten: von einem Objekt, dessen Klasse das Interface `Printer` implementiert, und von einem zweiten Objekt, dessen Klasse `Database` implementiert. Diese beiden Objekte sollen durch Mocks ersetzt werden. Wir beginnen mit dem Interface `Printer`. Wir werden also auf den `PrinterImplSimple` verzichten wollen.

Das erste Teilziel besteht darin, die Aufrufe des `Processors` an einen `Printer` aufzuzeichnen – und zwar mittels eines allgemein nutzbaren Verfahrens (welches dann auch für beliebige andere Interfaces genutzt werden kann).

Zu diesem Zweck benutzen wir ein Dynamic-Proxy:



Mittels des Dynamic-Proxies-Mechanismus können wir zur Laufzeit den Bytecode einer Klassen generieren, welche ein beliebiges, vorgegebenes Interface implementiert.

Aufgrund des Interfaces `Printer` wird in unserem Falle eine Klasse namens `$Proxy0` generiert.

Die Methoden dieser Proxy-Klasse (also in unserem Falle: `printBegin`, `printPosition` und `printEnd`) sind allesamt nach demselben Schema implementiert:

- Jede der Proxy-Methoden berechnet zunächst dasjenige `Method`-Objekt, welches die aufgerufene Proxy-Methode beschreibt.
- Dann werden die Parameter, die an die Proxy-Methode übergeben worden sind, "flachgeklopft" zu einem `Object`-Array.
- Schließlich wird die `invoke`-Methode eines Objekts aufgerufen, dessen Klasse das Java-Interface `InvocationHandler` implementiert (dabei wird das `Method`-Objekt und der `Object`-Array als Parameter übergeben). Bei der Instanziierung der generierten Proxy-Klasse muss dann natürlich eine Referenz auf einen solchen `InvocationHandler` übergeben werden.

Die `invoke`-Methode eines `InvocationHandlers` ist somit eine zentrale Methode, an die jeder Aufruf einer Proxy-Methode delegiert. Und eben in dieser Methode wird dann schließlich die Protokollierung der Methodenaufruf erfolgen können.

Wir implementieren das `InvocationHandler`-Interface in einer inneren Klasse der statischen Klasse `Mock`. Die Klasse enthält neben dieser inneren Klasse zudem eine statische `create`-Methode, mittels derer die Generierung der Proxy-Klasse und deren Instanziierung veranlasst werden kann.

```
package util.mock;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

public class Mock {

    private static class Handler implements InvocationHandler {
        @Override
        public Object invoke(
            final Object proxy,
            final Method method,
            final Object[] args) throws Throwable {
            final Object[] arguments =
                args == null ? new Object[0] : args;
            System.out.println(method.getName() + " " +
                Arrays.toString(arguments));
            return null;
        }
    }

    @SuppressWarnings("unchecked")
    public static <T> T create(final Class<T> iface) {
        return (T) Proxy.newProxyInstance(
            ClassLoader.getSystemClassLoader(),
            new Class<?>[] { iface },
```



```
        new Handler());  
    }  
}
```

Die Klasse, die das `InvocationHandler`-Interface implementiert, heißt `Handler`. In ihrer `invoke`-Methode gibt sie den Namen der Methode und die Parameter aus.

Der statischen `create`-Methode der Klasse `Mock` wird als ein Class-Objekt übergeben, welches das zu implementierende Interface beschreibt – dasjenige Interface, für welches die Proxy-Klasse generiert werden soll. `create` ruft die Java-Methode `Proxy.newProxyInstance` auf, welcher u.a. das zu implementierende Interface und ein `InvocationHandler` übergeben wird.

`create` liefert die Referenz auf das erzeugte Proxy-Objekt zurück. Da die generierte Proxy-Klasse das Interface `T` (in unserem Falle: `Printer`) implementiert, kann die `create`-Methode auch eine Referenz vom Typ `T` (in unserem Falle: `Printer`) zurückliefern.

Wir können nun den im ersten Projekt benutzten `PrinterImplSimple` ersetzen durch einen gemockten `Printer`:

```
package appl;  
  
import domain.Order;  
import services.ifaces.Database;  
import services.ifaces.Printer;  
import services.impl.DatabaseImplSimple;  
import util.mock.Mock;  
  
public class Application {  
  
    public static void main(final String[] args) {  
  
        final Order[] orders = new Order[] {  
            new Order(1000, 100, 1),  
            new Order(1000, 200, 1),  
            new Order(2000, 100, 2),  
            new Order(3000, 300, 3),  
        };  
  
        final Database database = new DatabaseImplSimple();  
  
        final Printer printer = Mock.create(Printer.class);  
  
        System.out.println(printer.getClass().getName());  
  
        final Processor processor = new Processor(database, printer);  
  
        processor.process(orders);  
    }  
}
```

Hier die Ausgaben:

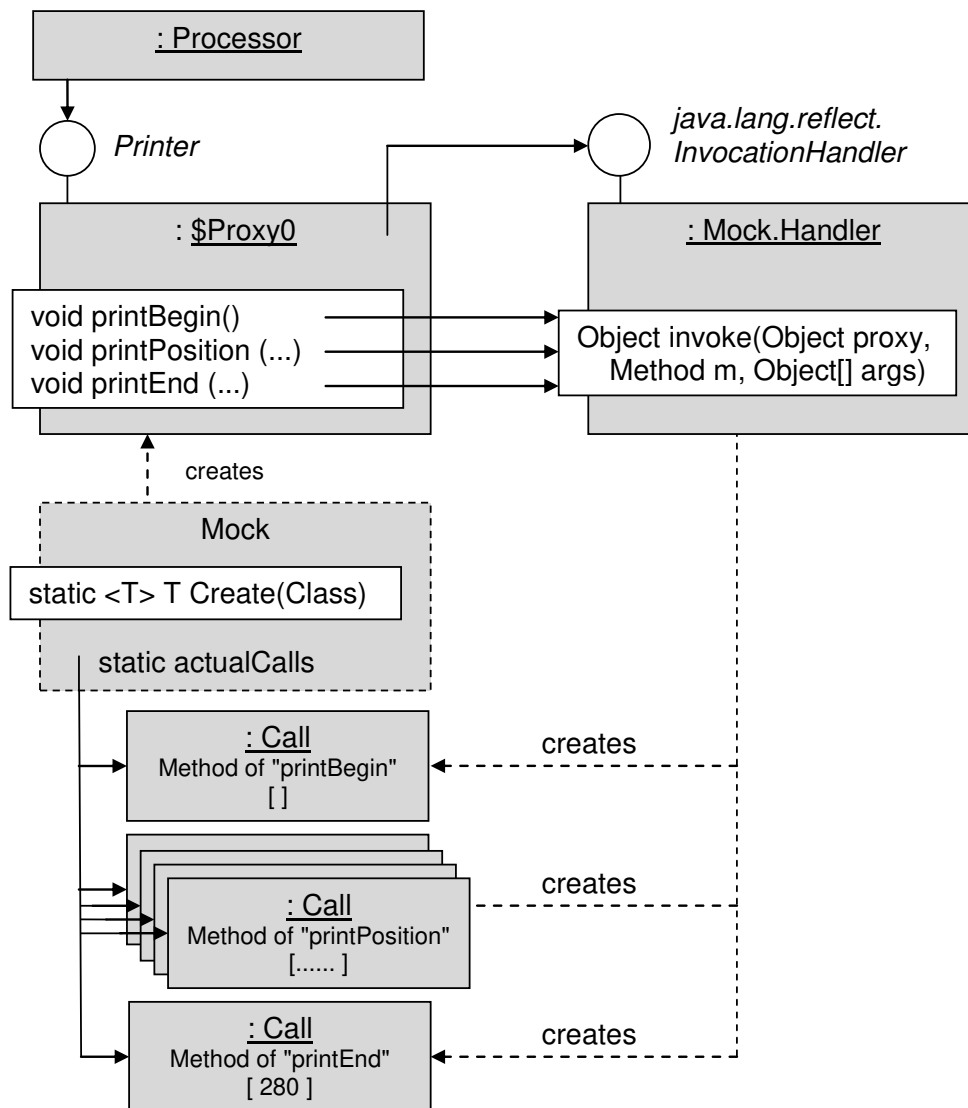
```
com.sun.proxy.$Proxy0  
printBegin []  
printPosition [Customer [1000, Nowak], Jever, 20, 1, 20]  
printPosition [Customer [1000, Nowak], Veltins, 40, 1, 40]  
printPosition [Customer [2000, Rueschenpoehler], Jever, 20, 2, 40]  
printPosition [Customer [3000, Kreuzer], Bitburger, 60, 3, 180]  
printEnd [280]
```

Wir wissen nun, wie die Aufrufe individueller, in einem Interface spezifizierter Methoden an eine "generische" Methode delegiert werden – und damit ist unser Problem bereits zur Hälfte gelöst...

5.3 Call-Objekte

Die Ausgaben, die im letzten Programm erzeugt wurden, sind schön und gut. Besser wäre es, wenn die Protokollierung ein bleibendes Resultat hätte.

Bei jedem Aufruf der `Intercept`-Methode kann ein `Call`-Objekt erzeugt werden, welches den Aufruf repräsentiert. Diese `Call`-Objekte können zu einer Liste hinzugefügt werden- eine Liste, welche dann die aktuellen Aufrufe der Proxy-Methoden repräsentiert.



Ein Aufruf kann repräsentiert werden durch ein `Method`-Objekt und die aktuellen Parameter (in Form eines `Object`-Arrays). Und glücklicherweise werden der `invoke`-Methode eines `InvocationHandler`s eben genau diese Informationen übergeben.

Hier die Essentials der `Call`-Klasse:

```
public class Call {  
    public final Method method;  
    public final Object[] args;  
    // ...  
}
```

Und hier schließlich die vollständige `Call`-Klasse (man beachte, dass sie `equals` und `hashCode` überschreibt – denn später werden `Call`-Objekte miteinander verglichen werden müssen):

```
package util.mock;  
  
import java.lang.reflect.Method;  
import java.util.Arrays;  
import java.util.Objects;  
  
public class Call {  
  
    public final Method method;  
    public final Object[] args;  
  
    public Object returnValue;  
  
    public Call(final Method method, final Object[] args) {  
        Objects.requireNonNull(method);  
        Objects.requireNonNull(args);  
        this.method = method;  
        this.args = args;  
    }  
  
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;  
        result = prime * result + Arrays.hashCode(this.args);  
        result = prime * result + this.method.hashCode();  
        return result;  
    }  
  
    @Override  
    public boolean equals(final Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null || this.getClass() != obj.getClass())  
            return false;  
        final Call other = (Call) obj;  
        return this.method.equals(other.method)  
            && Arrays.equals(this.args, other.args);  
    }  
  
    @Override  
    public String toString() {  
        return this.getClass().getSimpleName() + " [" +  
            this.method.getName() + ", " +  
            Arrays.toString(this.args) + ", " +  
            this.returnValue + "];"  
    }  
}
```

Die Bedeutung des `returnValue`-Attributs wird erst später deutlich werden...

Und hier die geänderte `Mock`- und `Mock.Handler`-Klassen:

```
package util.mock;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Mock {

    private final static List<Call> actualCalls
        = new ArrayList<>();

    public static List<Call> getActualCalls() {
        return Collections.unmodifiableList(Mock.actualCalls);
    }

    private static class Handler implements InvocationHandler {
        @Override
        public Object invoke(
            final Object proxy,
            final Method method,
            final Object[] args) throws Throwable {
            final Object[] arguments =
                args == null ? new Object[0] : args;
            final Call call = new Call(method, arguments);
            Mock.actualCalls.add(call);
            return null;
        }
    }

    @SuppressWarnings("unchecked")
    public static <T> T create(final Class<T> iface) {
        return (T) Proxy.newProxyInstance(
            ClassLoader.getSystemClassLoader(),
            new Class<?>[] { iface },
            new Handler());
    }
}
```

Die `Call`-Objekte sind nun in der statischen Liste `actualCalls` gesammelt worden.

Diese Liste könnte am Ende von `main` ausgegeben werden:

```
package appl;

import domain.Order;
import services.ifaces.Database;
import services.ifaces.Printer;
import services.impl.DatabaseImplSimple;
import util.mock.Mock;

public class Application {

    public static void main(final String[] args) {

        final Order[] orders = new Order[] {
            new Order(1000, 100, 1),
            new Order(1000, 200, 1),
            new Order(2000, 100, 2),
            new Order(3000, 300, 3),
        };

        final Database database = new DatabaseImplSimple();

        final Printer printer = Mock.create(Printer.class);

        final Processor processor = new Processor(database, printer);

        processor.process(orders);

        Mock.getActualCalls().forEach(System.out::println);
    }
}
```

Hier die Ausgaben:

```
printBegin []
printPosition [Customer [1000, Nowak], Jever, 20, 1, 20]
printPosition [Customer [1000, Nowak], Veltins, 40, 1, 40]
printPosition [Customer [2000, Rueschenpoehler], Jever, 20, 2, 40]
printPosition [Customer [3000, Kreuzer], Bitburger, 60, 3, 180]
printEnd [280]
```

5.4 Assertions

Wie könnten wir nun testen, ob die von uns erwarteten `Printer`-Aufrufe den tatsächlichen in der `Mock`-Klasse nun bereits gespeicherten Aufrufen entsprechen?

```
package appl;

import domain.Customer;
import domain.Order;
import services.ifaces.Database;
import services.ifaces.Printer;
import services.impl.DatabaseImplSimple;
import util.mock.Mock;

public class Application {

    public static void main(final String[] args) {

        final Order[] orders = new Order[] {
            new Order(1000, 100, 1),
            new Order(1000, 200, 1),
            new Order(2000, 100, 2),
            new Order(3000, 300, 3),
        };

        final Database database = new DatabaseImplSimple();

        final Printer printer = Mock.create(Printer.class);

        final Processor processor = new Processor(database, printer);

        processor.process(orders);

        Mock.verify(() -> printer.printBegin());
        Mock.verify(() -> printer.printPosition(
            new Customer(1000, "Nowak"), "Jever", 20, 1, 20));
        Mock.verify(() -> printer.printPosition(
            new Customer(1000, "Nowak"), "Veltins", 40, 1, 40));
        Mock.verify(() -> printer.printPosition(
            new Customer(2000, "Rueschenpoehler"), "Jever", 20, 2,
40));
        Mock.verify(() -> printer.printPosition(
            new Customer(3000, "Kreuzer"), "Bitburger", 60, 3, 180));
        Mock.verify(() -> printer.printEnd(280));

        System.out.println("Green!!!");
    }
}
```

Nachdem der `Processor` seine Arbeit getan hat – und via `Printer`-Aufrufe die `actualCall`-Liste der `Mock`-Klasse gefüllt hat – rufen wir unsererseits genau diejenigen Methoden auf den gemockten `Printer` auf, deren Aufrufe wir erwartet haben.

Wir rufen diese Methoden aber nicht direkt auf, sondern übergeben sie als `Runnable`s (in Form von Lambdas) an eine statische `verify`-Methode der `Mock`-Klasse.

Diese `verify`-Methode wird dann jeweils die an ihr übergebene Methode ausführen. Diese Ausführung wird ein `Call`-Objekt erzeugen – und wir müssen dann nur mehr testen, ob es innerhalb der `actualCall`-Liste ein `Call`-Objekt gibt, welches dem gerade produzierten `Call`-Objekt gleicht.

Hier die erweiterte `Mock`-Klasse:

```
package util.mock;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Mock {

    private final static List<Call> actualCalls
        = new ArrayList<>();

    public static List<Call> getActualCalls() {
        return Collections.unmodifiableList(Mock.actualCalls);
    }

    private static boolean isVerifying;

    private static class Handler implements InvocationHandler {
        @Override
        public Object invoke(
            final Object proxy,
            final Method method,
            final Object[] args) throws Throwable {
            final Object[] arguments
                = args == null ? new Object[0] : args;
            final Call call = new Call(method, arguments);

            if (Mock.isVerifying) {
                if (!Mock.actualCalls.contains(call))
                    throw new AssertionError(
                        "expected, but not called: " + call);
                return null;
            }

            Mock.actualCalls.add(call);
            return null;
        }
    }

    @SuppressWarnings("unchecked")
    public static <T> T create(final Class<T> iface) {
        return (T) Proxy.newProxyInstance(
            ClassLoader.getSystemClassLoader(),
```



```
        new Class<?>[] { iface },
        new Handler());
    }

    public static void verify(final Runnable runnable) {
        Mock.isVerifying = true;
        runnable.run();
        Mock.isVerifying = false;
    }
}
```

Wir müssen nun unterscheiden können: wurde die `invoke`-Methode des `Handlers` im Kontext der Ausführung der zu testenden Methode aufgerufen oder im Kontext der `verify`-Methode. Hierzu existiert die statische `Mock-Variable` `verifying`.

In `verify` wird `verifying` auf `true` gesetzt. Dann wird die Interface-Methode (mittels des `Runnables`) aufgerufen – die wiederum die `invoke`-Methode des `Handlers` aufrufen wird. Und abschließend wird `verifying` wieder auf `false` gesetzt.

In der `invoke`-Methode des `Handlers` muss dann unterschieden werden. Ist `verifying` nicht gesetzt, wird (wie gehabt) ein `Call`-Objekt erzeugt und in `actualCalls` eingetragen; ist `verifying` gesetzt, wird ebenfalls ein `Call`-Objekt erzeugt – und nun aber nur geprüft, ob ein äquivalentes `Call`-Objekt in `actualCalls` enthalten ist. Falls nicht, wird eine `Exception` geworfen.

Angenommen nun, die zu testende `process`-Methode der `Processor`-Klasse "vergisst" den Aufruf von `printEnd(280)`. Dann wird folgende Meldung ausgegeben:

```
expected, but not called: printEnd(280)
```

Versuchen Sie, weiteren möglichen Fehlern des `Processors` auf die Schliche zu kommen!

5.5 Arrangements

Im Folgenden wird das zweite Interface gemockt, welches der `Processor` nutzt: das Interface `Database`:

```
package services.ifaces;

import domain.Customer;

public interface Database {

    public abstract Customer getCustomer(int nr);
    public abstract String getProductName(int nr);
    public abstract int getProductPrice(int nr);
}
```

Während `Printer` ausschließlich `void`-Methoden spezifiziert, spezifiziert `Database` Methoden, die Rückgaben liefern: eine `Customer`-Referenz, eine `String`-Referenz und einen `int`-Wert. Zusätzlich zur Verifizierung der erwarteten Methodenaufrufe wird hier die sog. "Stub"-Funktionalität benötigt (ein Sub dient als Lieferant von Werten, welche der Aufrufer für seine Berechnungen benötigt).

Um ein solches Interface mocken zu können, muss irgendeine Form gefunden werden, in welche Rückgabewerte hinterlegt werden können. Wenn z.B. `getCustomer` mit dem Wert 1000 aufgerufen wird, muss das gemockte Interface den folgenden `Customer` liefern: `new Customer(1000, "Nowak")`; wenn `getProductName` mit der Nummer 100 aufgerufen wird, muss "Jever" geliefert werden; wenn `getProductPrice` mit der Nummer 100 aufgerufen wird, muss der `int`-Wert 20 geliefert werden etc.

Hier die erweiterte Testklasse:

```
package appl;

import domain.Customer;
import domain.Order;
import services.ifaces.Database;
import services.ifaces.Printer;
import util.mock.Mock;

public class Application {

    public static void main(final String[] args) {

        final Order[] orders = new Order[] {
            new Order(1000, 100, 1),
            new Order(1000, 200, 1),
            new Order(2000, 100, 2),
            new Order(3000, 300, 3),
        };

        final Database database = Mock.create(Database.class);

        final Printer printer = Mock.create(Printer.class);
```

```
Mock.when(() -> database.getCustomer(1000))
    .thenReturn(new Customer(1000, "Nowak"));
Mock.when(() -> database.getCustomer(2000))
    .thenReturn(new Customer(2000, "Rueschenpoehler"));
Mock.when(() -> database.getCustomer(3000))
    .thenReturn(new Customer(3000, "Kreuzer"));
Mock.when(() -> database.getProductName(100))
    .thenReturn("Jever");
Mock.when(() -> database.getProductPrice(100))
    .thenReturn(20);
Mock.when(() -> database.getProductName(200))
    .thenReturn("Veltins");
Mock.when(() -> database.getProductPrice(200))
    .thenReturn(40);
Mock.when(() -> database.getProductName(300))
    .thenReturn("Bitburger");
Mock.when(() -> database.getProductPrice(300))
    .thenReturn(60);

final Processor processor = new Processor(database, printer);

processor.process(orders);

// wie gehabt...
}
```

Wir benötigen eine weitere Phase: die "Arrange"-Phase. Diese muss der Act-Phase vorangehen. In der Act-Phase werden vom `Processor` die Methoden `getCustomer`, `getProductName` und `getProductPrice` aufgerufen – und dann muss natürlich bereits klar sein, was diese Aufrufe jeweils liefern sollen.

Betrachten wird die erste Zeile dieser Arrange-Phase:

```
Mock.when(() -> database.getCustomer(1000))
    .thenReturn(new Customer(1000, "Nowak"));
```

Diese Zeile kann wie folgt gelesen werden: Wenn irgendwann in der Act-Phase die Methode `getCustomer` auf das gemockte `Database` aufgerufen wird, soll dieser Aufruf einen `Customer` mit den Properties `1000` und `"Nowak"` liefern.

Die `Mock`-Klasse muss also um eine `when`-Methode erweitert werden. Dieser wird im Gegensatz zur `verify`-Methode kein `Runnable`, sondern ein `Supplier<T>` übergeben (wobei `T` den Return-Typ meint). `when` muss eine Referenz zurückliefern, auf welche anschließend eine `thenReturn`-Methode aufgerufen werden können muss. Dieser wird der vom gemockten Interface zurückzuliefernde Wert übergeben. Schließlich verlangen wir Typsicherheit: Wenn an `when` eine `Supplier<Customer>` übergeben wird, muss `thenReturn` als Parameter auch eine `Customer`-Referenz verlangen.

Hier zunächst die erforderlichen Erweiterungen der `Mock`-Klasse:

```
package util.mock;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Optional;
import java.util.function.Supplier;

public class Mock {

    private final static List<Call> actualCalls
        = new ArrayList<>();
    private final static List<Call> arrangedCalls
        = new ArrayList<>();

    public static List<Call> getActualCalls() {
        return Collections.unmodifiableList(Mock.actualCalls);
    }

    private static boolean isVerifying;
    private static boolean isArranging;

    private static class Handler implements InvocationHandler {
        @Override
        public Object invoke(
            final Object proxy,
            final Method method,
            final Object[] args) throws Throwable {
            final Object[] arguments
                = args == null ? new Object[0] : args;
            final Call call = new Call(method, arguments);

            if (Mock.isArranging) {
                Mock.arrangedCalls.add(call);
                final Class<?> returnType = method.getReturnType();
                return returnType.isPrimitive() ?
                    SimpleTypes.defaultValue(returnType) : null;
            }

            if (Mock.isVerifying) {
                if (!Mock.actualCalls.contains(call))
                    throw new AssertionError(
                        "expected, but not called: " + call);
                return null;
            }

            Mock.actualCalls.add(call);
            if (call.method.getReturnType() != void.class) {
                final Optional<Call> arrangedCall =
                    Mock.arrangedCalls.stream().filter(
                        c -> c.equals(call)).findFirst();
                if (!arrangedCall.isPresent())
                    throw new AssertionError(
                        "actual, but not arranged: " + call);
            }
        }
    }
}
```

```

        return arrangedCall.get().returnValue;
    }

    return null;
}

@SuppressWarnings("unchecked")
public static <T> T create(final Class<T> iface) {
    arrangedCalls.clear();
    actualCalls.clear();
    return (T) Proxy.newProxyInstance(
        ClassLoader.getSystemClassLoader(),
        new Class<?>[] { iface },
        new Handler());
}

public static void verify(final Runnable runnable) {
    Mock.isVerifying = true;
    runnable.run();
    Mock.isVerifying = false;
}

public static class Arrangement<T> {
    public void thenReturn(final T value) {
        Mock.arrangedCalls.get(Mock.arrangedCalls.size() - 1)
            .returnValue = value;
    }
}

public static <T> Arrangement<T> when(
    final Supplier<T> supplier) {
    Mock.isArranging = true;
    supplier.get();
    Mock.isArranging = false;
    return new Arrangement<T>();
}
}

```

Neben `actualCalls` existiert nun eine weitere Liste: die Liste derjenigen Calls, die in der Arrange-Phase aufgezeichnet werden (`arrangedCalls`).

`verifying` zeigt bekanntlich an, ob die `invoke`-Methode des `Handlers` in der Assert-Phase aufgerufen wird; `arranging` wird anzeigen, dass `invoke` in der Arrange-Phase aufgerufen wird.

Die generische `when`-Methode ist mit `T` parametrisiert. Ihr wird ein `Supplier<T>` übergeben und sie liefert ein `Arrangement<T>` zurück. `when` operiert ähnlich wie `verify`: statt der `run`-Methode eines `Runnable`s wird hier allerdings die `get`-Methode eines `Suppliers` aufgerufen. `get` wird also `getCustomer`, `getProductName` oder `getProductPrice` auf das gemockte Interface aufrufen – und diese Methoden rufen natürlich jeweils wieder die `invoke`-Methode des `Handlers` auf. Damit `invoke` nun weiß, dass ihr Aufruf in der Arrange-Phase stattfindet.

det, wird vor dem `get`-Aufruf `arranging` auf `true` gesetzt. Schließlich wird ein `Arrangement<T>` erzeugt und zurückgeliefert.

Der Aufruf von `when` führt zunächst dazu, dass ein weiteres `Call`-Objekt der Liste `arrangedCalls` hinzugefügt wird.

Die `when`-Methode liefert ein `Arrangement<T>` zurück – ein Objekt, auf welches dann `thenReturn` aufgerufen werden. Und eben diese `thenReturn`-Methode ruft setzt das `returnValue`-Attribut des letzten in der `arrangedCall`-Liste eingetragenen `Calls`.

Die `Handler`-Klasse sieht nun etwas komplizierter aus. In dieser Methode wurde bislang nur eine einfache Unterscheidung gemacht: erfolgt ihr Aufruf in der `Act`- oder `Assert`-Phase? Diese Unterscheidung muss nun um einen dritten Fall erweitert werden: erfolgt der Aufruf von `invoke` in der `Arrange`-Phase?

Egal, in welcher Phase `invoke` aufgerufen wird: `invoke` erzeugt zunächst ein `Call`-Objekt.

Wird `invoke` in der `Arrange`-Phase aufgerufen, wird dieses `Call`-Objekt der Liste `arrangedCalls` hinzugefügt. Zu diesem Zeitpunkt hat das `Call`-Objekt noch keinen `returnValue` – bzw. `returnValue` ist `null`. Erst mittels `Arrangement.thenReturn` wird in das `Call`-Objekt der `returnValue` eingetragen.

Aber `invoke` muss bereits einen Wert zurückliefern – wenngleich dieser Wert nirgendwo benutzt wird. Sofern die `Proxy`-Methode, über die `invoke` aufgerufen wird, eine Referenz zurückliefert, kann `invoke` einfach `null` liefern. Liefert die `Proxy`-Methode aber einen Wert eines primitiven Typs zurück (`int`, `double` etc.), muss der Default-Wert dieses Typs (`0`, `0.0` etc.) geliefert werden. Dieser default-Value wird mittels `SimpleTypes.defaultValue` ermittelt (s. weiter unten).

Wird `invoke` in der `Assert`-Phase aufgerufen, wird – wie gehabt – geprüft, ob der ihr übergebene `Call` auch tatsächlich stattgefunden hat.

Wird `invoke` schließlich in der `Act`-Phase aufgerufen, wird zunächst – wie gehabt – das erzeugte `Call`-Objekt in die `actualCalls` eingetragen. Sofern es sich um den Aufruf einer `void`-Methode handelte, war's das. Ansonsten wird in der Liste `arrangedCalls` nach einem `Call`-Objekt gesucht, welches dem an `invoke` übergebenen `Call`-Objekts äquivalent ist. Wird kein solches Objekt gefunden, wird eine `Exception` geworfen. Ansonsten wird von `invoke` nun der im gefundenen `Call` enthaltene `returnValue` zurückgeliefert.

Die `Mock.create`-Methode wird schließlich derart erweitert, dass bei jedem Aufruf die Listen `arrangedCalls` und `actualCalls` geleert werden (eine Testklasse besteht gewöhnlich aus mehreren Testmethoden, in deren Setup jeweils Interfaces per Aufruf von `Mock.create` gemockt werden).

Damit sei die Entwicklung des Mocker-Werkzeugs abgeschlossen. Wir haben gesehen, dass ein solches Werkzeug mit recht einfachen Mitteln implementiert werden kann.

Allerdings hat das Werkzeug noch eine Reihe von Mängeln:

- Das Werkzeug erkennt zwar erwartete Aufrufe, die nicht stattgefunden haben; es erkennt aber keine tatsächlichen Aufrufe, die nicht erwartet wurden.
- Das Werkzeug erkennt keine "ungenutzten" Arrangements.
- In der Arrange-Phase wird auf die Resultate von `when` die `thenReturn`-Methode aufgerufen. Das muss vervollständigt werden: es sollte auch `thenThrow` aufgerufen werden können (um eine Methode zu mocken, die eine Exception liefert).

Etc. etc. – aber immerhin.

Zur Vervollständigung hier noch die in der `Mock`-Klasse genutzte Klasse `SimpleTypes`:

```
package util.mock;

import java.util.HashMap;
import java.util.Map;

public class SimpleTypes {
    private static final Map<Class<?>, Object> defaultValues
        = new HashMap<>();
    static {
        SimpleTypes.defaultValues.put(byte.class, (byte)0);
        SimpleTypes.defaultValues.put(short.class, (short)0);
        SimpleTypes.defaultValues.put(int.class, 0);
        SimpleTypes.defaultValues.put(long.class, (long)0);
        SimpleTypes.defaultValues.put(float.class, (float)0);
        SimpleTypes.defaultValues.put(double.class, (double)0);
        SimpleTypes.defaultValues.put(char.class, (char)0);
        SimpleTypes.defaultValues.put(boolean.class, false);
    }

    public static Object defaultValue(final Class<?> type) {
        return SimpleTypes.defaultValues.get(type);
    }
}
```

5.6 Ein Eclipse-Test

Schließlich sei gezeigt, wie das Mocking-Framework einem Eclipse-Test genutzt werden kann:

```
package test;

import org.junit.Before;
import org.junit.Test;

import appl.Processor;
import domain.Customer;
import domain.Order;
import services.ifaces.Database;
import services.ifaces.Printer;
import util.mock.Mock;

public class ProcessorTest {

    Database database;
    Printer printer;
    Processor processor;

    @Before
    public void setUp() {
        this.database = Mock.create(Database.class);
        this.printer = Mock.create(Printer.class);
        this.processor = new Processor(this.database, this.printer);
    }

    @Test
    public void testEmptyInput() {
        // Arrange

        // Act
        final Order[] orders = new Order[] {};
        this.processor.process(orders);

        // Assert
        Mock.verify(() -> this.printer.printBegin());
        Mock.verify(() -> this.printer.printEnd(0));
    }

    @Test
    public void testOneOrder() {
        // Arrange
        Mock.when(() -> this.database.getCustomer(1000))
            .thenReturn(new Customer(1000, "Nowak"));
        Mock.when(() -> this.database.getProductName(100))
            .thenReturn("Jever");
        Mock.when(() -> this.database.getProductPrice(100))
            .thenReturn(20);

        // Act
        final Order[] orders = new Order[] {
            new Order(1000, 100, 1)
        };
        this.processor.process(orders);

        // Assert
        Mock.verify(
            () -> this.printer.printBegin());
        Mock.verify(
```



```
        () -> this.printer.printPosition(
            new Customer(1000, "Nowak"), "Jever", 20, 1, 20));
Mock.verify(
    () -> this.printer.printEnd(20));
}

@Test
public void testManyOrders() {
    // Arrange
    Mock.when(() -> this.database.getCustomer(1000))
        .thenReturn(new Customer(1000, "Nowak"));
    Mock.when(() -> this.database.getCustomer(2000))
        .thenReturn(new Customer(2000, "Rueschenpoehler"));
    Mock.when(() -> this.database.getCustomer(3000))
        .thenReturn(new Customer(3000, "Kreuzer"));
    Mock.when(() -> this.database.getProductName(100))
        .thenReturn("Jever");
    Mock.when(() -> this.database.getProductPrice(100))
        .thenReturn(20);
    Mock.when(() -> this.database.getProductName(200))
        .thenReturn("Veltins");
    Mock.when(() -> this.database.getProductPrice(200))
        .thenReturn(40);
    Mock.when(() -> this.database.getProductName(300))
        .thenReturn("Bitburger");
    Mock.when(() -> this.database.getProductPrice(300))
        .thenReturn(60);

    // Act
    final Order[] orders = new Order[] {
        new Order(1000, 100, 1),
        new Order(1000, 200, 1),
        new Order(2000, 100, 2),
        new Order(3000, 300, 3),
    };
    this.processor.process(orders);

    // Assert
    Mock.verify(
        () -> this.printer.printBegin());
    Mock.verify(
        () -> this.printer.printPosition(
            new Customer(1000, "Nowak"), "Jever", 20, 1, 20));
    Mock.verify(
        () -> this.printer.printPosition(
            new Customer(1000, "Nowak"), "Veltins", 40, 1, 40));
    Mock.verify(
        () -> this.printer.printPosition(
            new Customer(2000, "Rueschenpoehler"), "Jever", 20, 2, 40));
    Mock.verify(
        () -> this.printer.printPosition(
            new Customer(3000, "Kreuzer"), "Bitburger", 60, 3, 180));
    Mock.verify(
        () -> this.printer.printEnd(280));
}
}
```

6

Mocking – Mockito

6.1	XMLWriter.....	6-4
6.2	Pythagoras	6-10
6.3	Spezialitäten	6-14

6 Mocking – Mockito

Im Folgenden geht's um die Benutzung eines professionellen Mocking-Werkzeugs.

Wie in der Einleitung bereits notiert gibt es eine Reihe von Mocking-Werkzeugen: EasyMock, JMock, Mockito etc. Einige dieser Werkzeuge sind in die Jahre gekommen; einige andere sind "modern". Eines der modernen Werkzeuge ist Mockito.

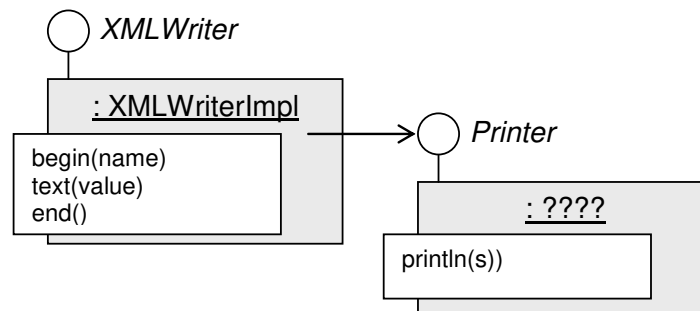
Natürlich existieren alle benötigten Ressourcen bereits im `Dependencies`-Ordner.

Ein Hinweis zur Implementierung:

Grob gesehen funktioniert Mockito genauso wie das im letzten Kapitel entwickelte Mini-Framework. Die Parametrisierung der `when-` resp. `verify`-Methoden sieht allerdings anders aus – Mockito benutzt noch nicht die neuen Möglichkeiten von Java 8 (Lambdas). Daher ist die Form der Mockito-Aufruf etwas gewöhnungsbedürftig (Lambdas sind schöner)...

6.1 XMLWriter

Mittels eines `XMLWriter`s kann auf bequeme Weise ein XML-Text erzeugt werden. Der `XMLWriter` benutzt einen Ausgabe-Schreiber, welcher über das Interface `IPrinter` spezifiziert ist. Dieser Ausgabe-Schreiber wird gemockt.



Ein `XMLWriter` kann z.B. wie folgt verwendet werden:

```
Printer printer = ...
XMLWriter xmlWriter = new XMLWriterImpl(printer, 2)

xmlWriter.Begin("a");
xmlWriter.Begin("b");
xmlWriter.Text("hello");
xmlWriter.End();
xmlWriter.Begin("c");
xmlWriter.Text("world");
xmlWriter.End();
xmlWriter.End();
```

Er muss dann folgende Ausgabe erzeugen:

```
<a>
  <b>
    hello
  </b>
<c>
  world
</c>
</a>
```

Dem Konstruktor von `XMLWriterImpl` wird ein `Printer` und eine Einrückungstiefe übergeben.

Der Nutzen eines `XMLWriters` besteht offenbar darin, dass man weder an die spitzen Klammern noch an die Einrückungen denken muss. Und beim Abschluss eines Elements muss nur mehr `end` aufgerufen werden – parameterlos.

In diesem Beispiel geht es nur um die Verifizierung erwarteter Aufrufe – wir benötigen hier noch keine Stub-Funktionalität (die `println`-Methode von `Printer` ist `void`).

Die Interfaces:

```
package appl;

public interface XMLWriter {
    public abstract void begin(final String name);
    public abstract void text(final String value);
    public abstract void end();
}
```

```
package appl;

public interface Printer {
    public abstract void println(String s);
}
```

Die Implementierung von `XMLWriter` benutzt einen Stack. Beim Aufruf von `begin` wird der an `begin` übergebene Element-Name auf den Stack gelegt; beim Aufruf von `end` wird der oberste Name vom Stack entfernt und zur Ausgabe des Ende-Tags verwendet. Aufgrund der Anzahl der auf dem Stack liegenden Namen kann die jeweils erforderliche Einrückung berechnet werden.

```
package appl;

import java.util.Stack;

public class XMLWriterImpl implements XMLWriter {

    private final Printer printer;
    private final Stack<String> stack = new Stack<>();
    private final int indent;

    public XMLWriterImpl(final Printer printer, final int indent) {
        this.printer = printer;
        this.indent = indent;
    }

    public XMLWriterImpl(final Printer printer) {
        this(printer, 2);
    }

    @Override
    public void begin(final String name) {
        this.printer.println(this.spaces() + "<" + name + ">");
    }
}
```

```

        this.stack.push(name);
    }

    @Override
    public void text(final String value) {
        this.printer.println(this.spaces() + value);
    }

    @Override
    public void end() {
        final String name = this.stack.pop();
        this.printer.println(this.spaces() + "</" + name + ">");
    }

    private String spaces() {
        final StringBuilder buf = new StringBuilder();
        final int n = this.indent * this.stack.size();
        for (int i = 0; i < n; i++)
            buf.append(' ');
        return buf.toString();
    }
}

```

In der Testklasse wird eine `@Before`-Methode definiert, welche einen gemockten `Printer` und einen `XMLWriterImpl` erzeugt und diese an Instanzvariablen bindet:

```

package test;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;

import org.junit.Before;
import org.junit.Test;

import appl.Printer;
import appl.XMLWriter;
import appl.XMLWriterImpl;

public class XMLWriterTest {

    private Printer printer;
    private XMLWriter xmlWriter;

    @Before
    public void before() {
        this.printer = mock(Printer.class);
        this.xmlWriter = new XMLWriterImpl(this.printer);
    }

    // ...
}

```

`mock` ist eine statische Methode der Klasse `Mockito`. Sie erzeugt ein Mock-Objekt einer generierten Mock-Klasse. Das Interface, welches diese Mock-Klasse implementiert (hier: `Printer`), wird an `mock` in Form einer `Class`-Referenz übergeben.

Das von `mock` erzeugte Mock-Objekt wird dann an den Konstruktor von `XMLWriterImpl` übergeben (zusammen mit der gewünschten Einrückungstiefe).

Der erste Test:

```
@Test
public void TestNestedElement() {
    this.xmlWriter.begin("a");
    this.xmlWriter.text("hello");
    this.xmlWriter.end();

    verify(this.printer).println("<a>");
    verify(this.printer).println("  hello");
    verify(this.printer).println("</a>");
}
```

Zunächst werden drei Methoden auf den `XMLWriterImpl` aufgerufen. Der Aufruf dieser Methoden wird dazu führen, dass pro Aufruf jeweils einmal die `println`-Methode des gemockten `Printers` aufgerufen wird. Diese wird jeweils delegieren an einen `InvocationHandler`, welcher den Aufruf als "Call-Objekt" repräsentiert und dieses in einer Liste der "tatsächlichen Aufrufe" einträgt.

Dann spezifizieren wird, welche Aufrufe wird erwartet haben. Wir haben erwartet, dass im Kontext der drei Aufrufe an den `XMLWriterImpl` dreimal die `println`-Methode von `Printer` aufgerufen wurde – mit den Werten "<a>", " hello" und "".

Diese Erwartungen werden in Form von Aufrufen der `Mockito.verify`-Methode formuliert. Dieser Methode wird das gemockte Objekt (`printer`) übergeben. Sie liefert eine Referenz auf dieses ihr übergebene Objekt zurück, auf welches dann jeweils diejenige Methode aufgerufen werden kann, deren Aufruf in der "Act"-Phase wir erwartet haben.

Jeder Aufruf von `verify(...)` ... erzeugt wiederum ein Call-Objekt – ein Call-Objekt, welches jeweils einen erwarteten Aufruf repräsentiert. Wird dann in der Liste der tatsächlichen Aufrufe ein Call-Objekt gefunden, dessen Zustand dem via `verify(...)` ... erzeugten Call-Objekts gleicht, so ist alles okay. Ansonsten wird eine Fehler protokolliert – der erwartete Aufruf hat dann nicht stattgefunden.

Die Reihenfolge, in welcher die Erwartungen formuliert werden, spielt keine Rolle. Auch folgende Testmethode würde grün anzeigen:

```
@Test
public void TestNestedElement() {
    this.xmlWriter.begin("a");
    this.xmlWriter.text("hello");
    this.xmlWriter.end();

    verify(this.printer).println("</a>");
    verify(this.printer).println("<a>");
    verify(this.printer).println("  hello");
}
```

Und der Test zeigt auch dann grün an, wenn nicht alle tatsächlichen Aufrufe auch erwartet wurden:

```
@Test
public void TestNestedElement() {
    this.xmlWriter.begin("a");
    this.xmlWriter.text("hello");
    this.xmlWriter.end();

    verify(this.printer).println("<a>");
    verify(this.printer).println("  hello");
}
```

Wird allerdings ein Aufruf erwartet, der tatsächlich nicht stattgefunden hat, wird rot angezeigt:

```
@Test
public void TestNestedElement() {
    this.xmlWriter.begin("a");
    this.xmlWriter.text("hello");
    this.xmlWriter.end();

    verify(this.printer).println("<a>");
    verify(this.printer).println("  world");
    verify(this.printer).println("</a>");
}
```

Wir haben die Ausgabe von "world" erwartet – aber eine solche Ausgabe fand nicht statt.

Eine zweite, etwas komplexere Testmethode:

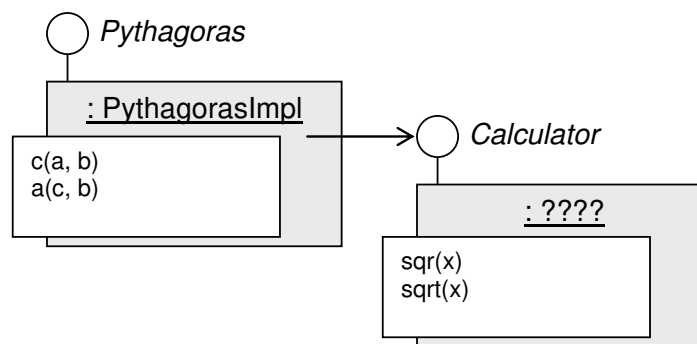
```
@Test
public void TestMoreNestedElements() {
    this.xmlWriter.begin("a");
    this.xmlWriter.begin("b");
    this.xmlWriter.text("hello");
    this.xmlWriter.end();
    this.xmlWriter.begin("c");
    this.xmlWriter.text("world");
    this.xmlWriter.end();
    this.xmlWriter.end();

    verify(this.printer).println("  <b>");
    verify(this.printer).println("<a>");
    verify(this.printer).println("    hello");
    verify(this.printer).println("  </b>");
    verify(this.printer).println("  <c>");
    verify(this.printer).println("    world");
    verify(this.printer).println("  </c>");
    verify(this.printer).println("</a>");
}
```

6.2 Pythagoras

Das zweite Beispiel zeigt, wie Stub-Funktionalität mit Mockito implementiert werden kann. Es geht also darum, wie ein gemocktes Objekt Werte an den Aufrufer zurückliefern kann, welcher dieser zur weiteren Berechnungen benötigt.

Ein `PythagorasImpl` kann aufgrund zweier Katheten die Hypotenuse eines rechtwinkligen Dreiecks berechnen – und aufgrund einer Hypotenuse und einer Kathete die zweite Kathete. Bei diesen Berechnungen müssen immer wieder Quadrate und Wurzeln berechnet werden. Zur Erledigung dieser Trivial-Aufgaben benutzt der `PythagorasImpl` ein Objekt, dessen Klasse das Interface `Calculator` implementiert:



Die Interfaces:

```
package appl;

public interface Pythagoras {
    public abstract double c(double a, double b);
    public abstract double a(double c, double b);
}
```

```
package appl;

public interface Calculator {
    public abstract double sqr(double x);
    public abstract double sqrt(double x);
}
```

Die Implementierung des Pythagoras:

```
package appl;

public class PythagorasImpl implements Pythagoras {

    private final Calculator calculator;

    public PythagorasImpl(final Calculator calculator) {
        this.calculator = calculator;
    }

    @Override
    public double c(final double a, final double b) {
        return this.calculator.sqrt(
            this.calculator.sqr(a) + this.calculator.sqr(b));
    }

    @Override
    public double a(final double c, final double b) {
        return this.calculator.sqrt(
            this.calculator.sqr(c) - this.calculator.sqr(b));
    }
}
```

Ein PythagorasImpl könnte wie folgt genutzt werden:

```
Calculator calculator = ...
Pythagoras pythagors = new PythagorasImpl(calculator);

System.out.println(pythagors.c(3.0, 4.0));
System.out.println(pythagors.a(5.0, 4.0));
```

Die berechnete Hypotenuse hat die Größe 5.0, die berechnete Kathete die Größe 3.0.

Zunächst zur @Before-Methode der Testklasse:

```
package test;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import appl.Calculator;
import appl.Pythagoras;
import appl.PythagorasImpl;

public class PythagorasTest {

    private Calculator calculator;
    private Pythagoras pythagoras;

    @Before
    public void before() {
        this.calculator = mock(Calculator.class);
        this.pythagoras = new PythagorasImpl(this.calculator);
    }

    // ...
}
```

Dem PythagorasImpl wird ein gemockter Calculator injiziert.

Die erste Testmethode testet die c-Methode von PythagorasImpl:

```
@Test
public void testHypotenuse() {
    when(this.calculator.sqr(3.0)).thenReturn(9.0);
    when(this.calculator.sqr(4.0)).thenReturn(16.0);
    when(this.calculator.sqrt(25.0)).thenReturn(5.0);

    Assert.assertEquals(5.0, this.pythagoras.c(3.0, 4.0), 0.0);
}
```

Die erste Zeile der obigen Testmethode lässt sich wie folgt lesen: Wenn der PythagorasImpl mit der Berechnung von `c(3.0, 4.0)` beauftragt wird, wird er irgendwann auf den gemockten Calculator die `sqr`-Methode mit den Wert 3.0 aufrufen. Wenn er dies tut, liefert ihm das gemockte Objekt den Wert 9.0 zurück (`thenReturn(9.0)`).

Entsprechend können auch die weiteren Zeilen interpretiert werden. Jeder Aufruf von `when` erzeugt ein Call-Objekt, welches in eine Liste erwarteter Aufrufe aufgenommen wird. Parallel hierzu wird der jeweils zurückzuliefernde Wert gespeichert.

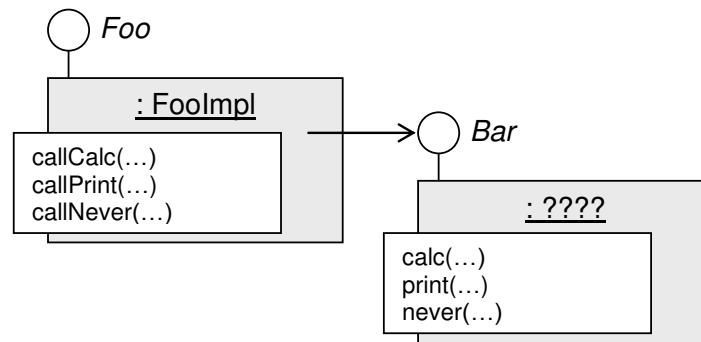
Hier eine Testmethode für den Test der `PythagorasImpl`-eigenen `a`-Methode:

```
@Test
public void testKathete() {
    when(this.calculator.sqr(5.0)).thenReturn(25.0);
    when(this.calculator.sqr(4.0)).thenReturn(16.0);
    when(this.calculator.sqrt(9.0)).thenReturn(3.0);

    Assert.assertEquals(3.0, this.pythagoras.a(5.0, 4.0), 0.0);
}
```

6.3 Spezialitäten

Im Folgenden werden einige zusätzlich Features von Mockito vorgestellt. Als Beispiel dient ein "sinnloses" Foo-Bar-Beispiel:



Die Interfaces:

```
package appl;

public interface Foo {
    public abstract int callCalc(int x);
    public abstract void callPrint(String s);
    public abstract void callNever(int v);
}
```

```
package appl;

public interface Bar {
    public abstract int calc(int x);
    public abstract void print(String s);
    public abstract void never(int v);
}
```

Die FooImpl-Klasse:

```
package appl;

public class FooImpl implements Foo {

    private final Bar bar;

    public FooImpl(final Bar bar) {
        this.bar = bar;
    }

    @Override
    public int callCalc(final int x) {
        // do something...
        final int result = this.bar.calc(x);
        // do something...
        return result;
    }

    @Override
```

```
public void callPrint(final String s) {  
    // do something...  
    this.bar.print(s);  
    // do something...  
}  
  
@Override  
public void callNever(final int v) {  
    // do something...  
    // this.bar.never(v);  
    // do something...  
}  
}
```

`callCalc` und `callPrint` rufen die `call`- resp. die `print`-Methode des mit dem `Foo`-Objekt assoziierten `Bar`-Objekts auf; `callNever` dagegen delegiert nicht an `Bar.never`.

Die `@Before`-Methode der Testklasse erzeugt ein gemocktes `Bar`-Objekt und übergibt dieses dem Konstruktor von `FooImpl`:

```
package test;  
  
import static org.mockito.Mockito.doThrow;  
import static org.mockito.Mockito.mock;  
import static org.mockito.Mockito.never;  
import static org.mockito.Mockito.verify;  
import static org.mockito.Mockito.when;  
  
// ...  
  
public class FooTest {  
  
    private Bar bar;  
    private Foo foo;  
  
    @Before  
    public void before() {  
        this.bar = mock(Bar.class);  
        this.foo = new FooImpl(this.bar);  
    }  
  
    // ...  
}
```


Der erste Test zeigt, dass `callCalc` die `Bar.calc`-Methode aufruft, welche genau denjenigen Wert liefert, der ihr als Parameter übergeben wird (42):

```
@Test
public void test1() {
    when(this.bar.calc(42)).thenReturn(42);
    Assert.assertEquals(42, this.foo.callCalc(42));
    verify(this.bar).calc(42);
}
```

Im zweiten Test wird `callCalc` mit dem Wert `-42` aufgerufen. `callCalc` delegiert an `calc` und reicht die `-42` weiter. Die `calc`-Methode sei nun so spezifiziert, dass sie bei der Übergabe eines negativen Parameters eine `RuntimeException` wirft. Es soll gezeigt werden, dass `callCalc` eben diese von `calc` geworfene Exception nicht auffängt, sondern weiterwirft (so sei die Funktionalität von `calc` spezifiziert):

Wir müssen also simulieren, dass `calc` beim Aufruf mit dem Wert `-42` keinen Return-Wert liefert, sondern eine `RuntimeException` wirft. Statt `thenReturn`s rufen wir `thenThrows` auf:

```
@Test
public void test2() {
    when(this.bar.calc(-42)).thenThrow(RuntimeException.class);
    XAssert.assertThrows(RuntimeException.class,
        () -> this.foo.callCalc(-42));
}
```

Im nächsten Test rufen wir `callPrint` mit einem gültigen String auf. Wir testen, ob dieser String – wie verlangt – an die `Bar.print`-Methode weitergereicht wird:

```
@Test
public void test3() {
    this.foo.callPrint("Hello");
    verify(this.bar).print("Hello");
}
```

Im folgenden Test nun wird `callPrint` mit `null` aufgerufen. `callPrint` reicht diese `null` an `Bar.print` weiter. Wir wollen nun simulieren, dass `Bar.print` eine `RuntimeException` wirft – eine Exception, die von `callPrint` weitergeworfen werden soll:

```
@Test
public void test4() {
    this.bar.print(null);
    doThrow(RuntimeException.class);
    XAssert.assertThrows(RuntimeException.class,
        () -> this.foo.callPrint(null));
}
```

Da `Bar.print` nichts liefert (die Methode ist `void`), kann ihr Aufruf auch nicht im Kontext des Aufrufs von `when` erfolgen. Der Aufruf muss einfach "als solcher" formuliert werden. Dann erfolge der Aufruf von `doThrow` (statt: `thenThrows`).

Schließlich wollen wir zeigen, dass im Kontext des Aufrufs der `callNever`-Methode die `Bar.never`-Methode niemals aufgerufen wird:

```
@Test
public void test5() {
    this.foo.callNever(42);
    verify(this.bar, never()).never(42);
}
```

Als zweites Argument kann an `verify` ein sog. `VerificationMode` übergeben werden. Hier wird `Mockito.never()` übergeben.

7

Mocking – Übungen

7.1	AccountService.....	7-4
7.2	GroupChanger.....	7-10

7 Mocking – Übungen

Dieses Kapitel enthält zwei Übungen zum Thema Mocking:

- In der ersten Übung geht's um einen `AccountService`, der auf ein `AccountDAO` angewiesen ist. Der `AccountService` enthält die Fachlogik der Konto-Anwendung, der `AccountDAO` enthält die Persistenzlogik. Der `AccountService` soll getestet werden – ohne das reale Datenbankzugriffe erfolgen. Das `AccountDAO`-Objekt muss also gemockt werden.
- In der zweiten Anwendung geht's um den Test eines Gruppenwechsel-Algorithmus, der in einer Klasse `GroupChanger` implementiert ist. Die Ausgaben des `GroupChangers` erfolgen über ein Interface. Dieses Ausgabe-Interface wird gemockt werden.

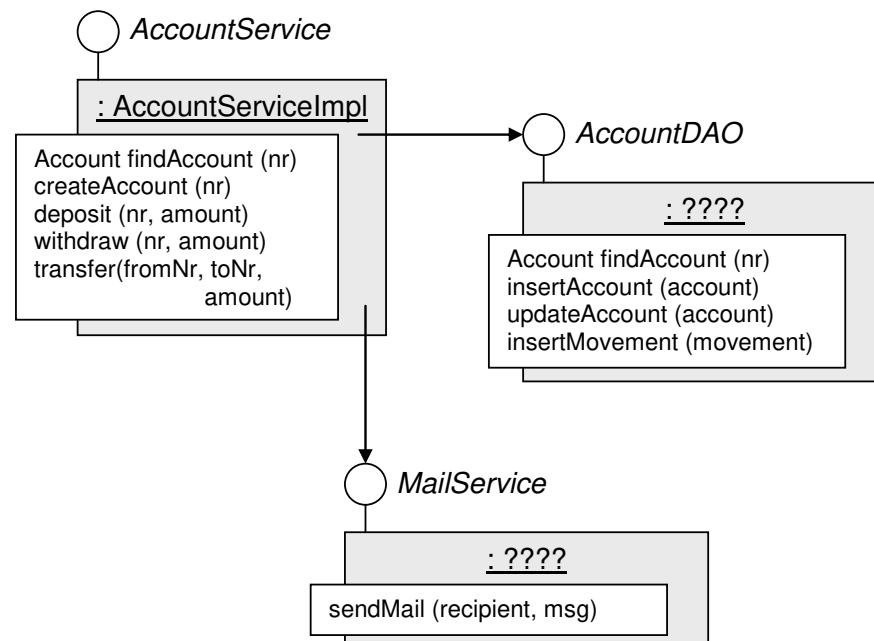
(Da die Zeit knapp ist, wird es i.d.R. reichen müssen, eine der beiden Übungen durchzuführen. By the way: die erste Übung ist komplex, die zweite relativ einfach...)

7.1 AccountService

Die im Folgenden zu implementierende Kontoverwaltung besteht hauptsächlich aus zwei Komponenten. Eine Klasse `AccountServiceImpl` enthält die Fachlogik für das Verwalten von Konten. Die Persistenzlogik der Kontoverwaltung wird in einer zweiten Klasse implementiert werden, welche das Interface `AccountDAO` implementiert. Die Fachlogik-Klasse benutzt diese Persistenz-Klasse.

Die Fachlogik (`AccountServiceImpl`) stellt z.B. eine Methode `withdraw` zur Verfügung. Dieser Methode wird die Nummer eines Kontos und der abzuhebende Betrag übergeben. Diese Methode wird dann zunächst auf die DAO-Methode `findAccount` aufrufen. Diese wird ein Objekt der Klasse `Account` zurückliefern (oder `null`). Dann wird geprüft, ob der Bestand des Kontos noch ausreicht, um den gewünschten Betrag auszahlen zu können. Wenn nicht, wird eine email an die Programmier-Abteilung geschickt (um darauf hinzuweisen, dass der Aufrufer von `withdraw` gefälligst zunächst eine Bestandsprüfung hätte vornehmen müssen) und `Exception` geworfen. Ansonsten wird der gewünschte Betrag vom Bestand des Kontos abgezogen und dieses über den Aufruf der DAO-Methode `updateAccount` wieder persistiert werden. Zudem wird die DAO-Methode `insertMovement` aufgerufen, um ein `Movement`-Objekt zu persistieren (eine Objekt, welches die aktuelle Kontobewegung repräsentiert.)

Der (zu testende!) `AccountServiceImpl` hängt also von zwei weiteren Objekte ab – die gemockt werden sollen: von einem Objekt, dessen Klasse `AccountDAO` implementiert, und einem zweiten Objekt, dessen Klasse `MailService` implementiert:



Die Klassen der persistenten Objekte (`Account` und `Movement`) sind bereits vorgegeben:

```
package appl;

public class Account
{
    public final int nr;
    public int balance;

    public Account(final int nr, final int balance) {
        this.nr = nr;
        this.balance = balance;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + this.balance;
        result = prime * result + this.nr;
        return result;
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj)
            return true;
        if (obj == null || this.getClass() != obj.getClass())
            return false;
        final Account other = (Account) obj;
        return this.nr == other.nr && this.balance == other.balance;
    }

    @Override
    public String toString() { ... }
}
```

```
package appl;

import java.util.Date;
import java.util.Objects;

public class Movement {

    public final int accountNr;
    public final Date date;
    public final int amount;

    public Movement(final int accountNr, final Date date, final int
amount) {
        Objects.requireNonNull(date);
        this.accountNr = accountNr;
        this.date = date;
        this.amount = amount;
    }

    @Override
```



```

public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + this.accountNr;
    result = prime * result + this.amount;
    result = prime * result + this.date.hashCode();
    return result;
}

@Override
public boolean equals(final Object obj) {
    if (this == obj)
        return true;
    if (obj == null || this.getClass() != obj.getClass())
        return false;
    final Movement other = (Movement) obj;
    return this.accountNr == other.accountNr &&
        this.date.equals(other.date) &&
        this.amount == other.amount;
}

@Override
public String toString() { ... }
}

```

Man beachte, dass beide Klasse `equals` und `hashCode` überschreiben. Dies wird für die folgende Verwendung der Klassen entscheidend sein...

Das Interface `AccountService` spezifiziert die fachliche Logik der Kontoverwaltung:

```

package appl;

import java.util.Date;

public interface AccountService {
    public abstract Account findAccount(int nr);
    public abstract void createAccount(int nr);
    public abstract void deposit(int nr, int amount, Date date);
    public abstract void withdraw(int nr, int amount, Date date);
    public abstract void transfer(
        int fromNr, int toNr, int amount, Date date);
}

```

Das (zu mockende) Interface `AccountDAO` spezifiziert die Persistenzlogik:

```

package appl;

public interface AccountDAO {
    public abstract void insertAccount(Account account);
    // may throw RuntimeException (Duplicate key)
    public abstract Account findAccount(int nr);
    // may return null;
    public abstract void updateAccount(Account account);
    // may throw RuntimeException (Account not found)
    public abstract void insertMovement(Movement movement);
}

```

Und das (zu mockende) Interface `MailService` spezifiziert den email-Versender:

```
package appl;

public interface MailService {
    public abstract void sendMail(String recipient, String message);
}
```

Das grobe Gerüst der `AccountServiceImpl`-Klasse ist bereits vorgegeben:

```
package appl;

import java.util.Date;

public class AccountServiceImpl implements AccountService {

    public static final String MAIL_RECEIVER
        = "Programmers";
    public static final String MAIL_MESSAGE
        = "Check balance before calling withdraw";

    private final AccountDAO dao;
    private final MailService mailService;

    public AccountServiceImpl(
        final AccountDAO dao,
        final MailService mailService) {
        this.dao = dao;
        this.mailService = mailService;
    }

    @Override
    public Account findAccount(final int nr) {
        return this.dao.findAccount(nr);
    }

    @Override
    public void createAccount(final int nr) {
        // TODO
    }

    @Override
    public void deposit(
        final int nr, final int amount, final Date date) {
        // TODO
    }

    @Override
    public void withdraw(
        final int nr, final int amount, final Date date) {
        // TODO
    }

    @Override
    public void transfer(
        final int fromNr, final int toNr,
        final int amount, final Date date) {
```

```
        // TODO
    }
}
```

Auch der Rahmen der Testklasse ist bereits vorgegeben:

```
package test;
// ...
public class AccountServiceTest {

    private AccountService accountService;
    private AccountDAO accountDAO;
    private MailService mailService;

    @Before
    public void before() {
        this.accountDAO = mock(AccountDAO.class);
        this.mailService = mock(MailService.class);
        this.accountService = new AccountServiceImpl(
            this.accountDAO, this.mailService);
    }

    @Test
    public void testFindAccount()
    {
        when(this.accountDAO.findAccount(4711))
            .thenReturn(new Account(4711, 42));

        final Account account
            = this.accountService.findAccount(4711);

        Assert.assertEquals(new Account(4711, 42), account);
    }

    @Test
    @Ignore
    public void testCreateAccount() {
        // TODO...
        this.accountService.createAccount(4711);
    }

    @Test
    @Ignore
    public void testCreateAccountDuplicateKey() {
        // TODO...
        XAssert.assertThrows(Exception.class,
            () -> this.accountService.createAccount(4711));
    }

    @Test
    @Ignore
    public void testDeposit() {
        final Date date = new Date();
        // TODO...
        this.accountService.deposit(4711, 2000, date);
    }

    @Test
    @Ignore
```

```
public void testWithdrawWhenAvailable() {
    final Date date = new Date();
    // TODO...
    this.accountService.withdraw(4711, 4000, date);
}

@Test
@Ignore
public void testWithdrawWhenNotAvailable() {
    // TODO...
    XAssert.assertThrows(Exception.class,
        () -> this.accountService.withdraw(
            4711, 4000, new Date()));
}

@Test
@Ignore
public void testWithdrawWhenAccountNotFound() {
    // TODO...
    XAssert.assertThrows(Exception.class,
        () -> this.accountService.withdraw(
            4711, 4000, new Date()));
}

@Test
@Ignore
public void testTransfer() {
    final Date date = new Date();
    // TODO...
    this.accountService.transfer(4711, 4712, 4000, date);
}
}
```

Gehen Sie bei der Implementierung in kleinen Schritten vor. Implementieren Sie jeweils zuerst eine Testmethode. Der folgende Testlauf sollte dann rot sein. Dann implementieren Sie die im Test aufgerufene `AccountServiceImpl`-Methode. Dann sollte der folgende Testlauf grün sein. Wiederholen Sie diese Abfolge, bis alle Test-Methoden und alle `AccountService`-Methoden implementiert sind.

7.2 GroupChanger

Ein Gruppenwechsel-Prozessor transformiert eine Eingabe, welche implizit gruppenförmig strukturiert ist, in eine Ausgabe, in welcher diese Gruppenstruktur explizit wird.

Ein recht abstrakte Definition...

Angenommen, die Eingabe beinhaltet `Order`-Objekte:

```
package appl;

public class Order {

    public final int customerNr;
    public final int productNr;
    public final int amount;

    public Order(final int customerNr,
                 final int productNr, final int amount) {
        this.customerNr = customerNr;
        this.productNr = productNr;
        this.amount = amount;
    }

    @Override
    public String toString() { ... }
}
```

Angenommen, die Eingabe bestehe aus folgenden Objekten:

```
new Order(1000, 100, 1)
new Order(1000, 200, 1)
new Order(2000, 100, 2)
new Order(3000, 300, 3)
new Order(3000, 200, 2)
new Order(3000, 100, 1)
```

Diese Eingabe enthält drei Gruppen: die erste Gruppe enthält die Aufträge für den Kunden 1000, die zweite Gruppe enthält die Aufträge für den Kunden 2000 und die dritte Gruppe enthält die Aufträge für den Kunden 3000. Die erste Gruppe enthält 2 Objekte, die zweite Gruppe enthält ein einziges Objekt und die dritte Gruppe umfasst drei Objekte.

Aufgrund dieser Eingabe könnte z.B. folgende Druckausgabe erzeugt werden müssen:

```
Begin
    GroupBegin 1000
        Position Order[1000, 100, 1]
        Position Order[1000, 200, 1]
    GroupEnd 1000
    GroupBegin 2000
        Position Order[2000, 100, 2]
    GroupEnd 2000
    GroupBegin 3000
        Position Order[2000, 300, 3]
        Position Order[2000, 200, 2]
        Position Order[2000, 100, 1]
    GroupEnd 3000
End
```

Der zu testende Gruppenwechsel-Algorithmus wird die eigentliche Ausgabe an ein Objekt delegieren, dessen Klasse das folgende Interface implementiert:

```
package appl;

public interface Processor<T, K> {
    public abstract void processBegin();
    public abstract void processGroupBegin(K key);
    public abstract void processPosition(T obj);
    public abstract void processGroupEnd(K key);
    public abstract void processEnd();
}
```

T steht für den Typ der Objekte, die prozessiert werden (hier: `Order`); **K** steht für den Schlüssel (**K** bezeichnet den Typ eines in **T** enthaltenen Elements (hier: `customerNr`)).

Eine konkrete Klasse, welche dieses Interfaces implementiert und die obige Ausgabe erzeugt, könnte wie folgt ausschauen (man beachte, dass T hier durch Order und K durch Integer ersetzt sind):

```
package appl;

public class PrintProcessor implements Processor<Order, Integer> {
    @Override
    public void processBegin() {
        System.out.println("Begin");
    }

    @Override
    public void processGroupBegin(final Integer customerNr) {
        System.out.println("\tGroupBegin " + customerNr);
    }

    @Override
    public void processPosition(final Order order) {
        System.out.println("\t\tPosition " + order);
    }

    @Override
    public void processGroupEnd(final Integer customerNr) {
        System.out.println("\tGroupEnd " + customerNr);
    }

    @Override
    public void processEnd() {
        System.out.println("End");
    }
}
```

Hier nun die zu testende Klasse, welche den eigentlichen Gruppenwechsel-Algorithmus implementiert:

```
package appl;

import java.util.Iterator;
import java.util.function.Function;

public class GroupChanger {
    public static <T, K> void run(
        final Iterable<T> input,
        final Function<T, K> keyExtractor,
        final Processor<T, K> processor) {
        final Iterator<T> iterator = input.iterator();
        T current = next(iterator);
        processor.processBegin();
        while (current != null) {
            final K key = keyExtractor.apply(current);
            processor.processGroupBegin(key);
            while (current != null &&
                keyExtractor.apply(current).equals(key)) {
                processor.processPosition(current);
                current = next(iterator);
            }
            processor.processGroupEnd(key);
        }
    }
}
```

```
    }  
    processor.processEnd();  
}  
  
private static <T> T next(final Iterator<T> iterator) {  
    return iterator.hasNext() ? iterator.next() : null;  
}  
}
```

Die statische `run`-Methode hat zwei Typ-Parameter: `T` und `K`. `T` steht für den Typ der Eingabe-Objekte (hier also für `Order`), `K` steht für den "Schlüssel", der die Gruppenstruktur definiert (hier also für `Integer`-den Typ von `customerNr`).

Der `run`-Methode werden drei Parameter übergeben:

- Ein `Iterable<T>`, über welchen das jeweils nächste Eingabeobjekt gelesen werden kann
- Eine `Function<T,K>`, welches als Key-Extractor fungieren wird (aufgrund eines `T`-Objekts einen darin enthaltenen Schlüssel von Typ `K` liefert)
- Ein `Processor<T, K>`, der für die Ausgabe der Resultate genutzt wird.

Das genaue Studium der `run`-Methode sei dem Leser / der Leserin überlassen.

Zum Zwecke des Tests der `GroupChanger`-Klasse (also für den Test der `run`-Methode dieser Klasse) wird das Interface `Processor` gemockt werden.

Hier der bereits vorgegebene Rahmen der Testklasse:

```
package test;  
// ...  
public class GroupChangerTest {  
  
    private Processor<Order,Integer> processor;  
  
    @Before  
    public void before() {  
        this.processor = mock(Processor.class);  
    }  
  
    @Test  
    public void testEmptyInput() {  
        final Order[] orders = new Order[] {  };  
  
        GroupChanger.run(  
            Arrays.asList(orders),  
            (final Order order) -> order.customerNr,  
            this.processor);  
  
        // TODO...  
    }  
}
```



```
@Test
public void testGroupWithOnePosition() {
    final Order[] orders = new Order[] {
        new Order(1000, 100, 10)
    };

    GroupChanger.run(
        Arrays.asList(orders),
        (final Order order) -> order.customerNr,
        this.processor);

    // TODO...
}

@Test
public void testThreeGroups() {
    final Order[] orders = new Order[] {
        new Order(1000, 100, 1),
        new Order(1000, 200, 1),
        new Order(2000, 100, 2),
        new Order(3000, 300, 3),
        new Order(3000, 200, 2),
        new Order(3000, 100, 1),
    };

    GroupChanger.run(
        Arrays.asList(orders),
        (final Order order) -> order.customerNr,
        this.processor);

    // TODO...
}
}
```

8

Test Driven Development: Beispiel

8.1	Parser für numerische Expressions.....	8-4
8.2	Entwurf logischer Schaltungen	8-8
8.3	CSV-Dateien.....	8-12

8 Test Driven Development: Beispiel

Wie lernt man programmieren? Indem man programmiert.

Wie lernt man testgetriebene Softwareentwicklung? Indem man testgetrieben Software entwickelt.

Wir brauchen daher Beispiele. Solche Beispiele dürfen einerseits nicht trivial sein – andererseits aber müssen sie überschaubar sein (die Zeit ist knapp...)

Das prinzipielle Vorgehen bei der testgetriebenen Entwicklung ist bereits im Einleitung-Kapitel am Beispiel einer `IntArray`-Klasse beschrieben worden. Das Beispiel war im Gegensatz zu den Beispielen dieses Kapitels recht trivial – trotzdem sollte man sich an diesem Beispiel orientieren.

Im Folgenden werden drei nicht triviale Beispiele beschrieben. Eines dieser Beispiele sollte im Seminar ausführlich behandelt werden – alle können nicht behandelt werden. Lösungen dieser Beispiel-Aufgaben sind in separaten Workspace enthalten (und in einer separaten Dokumentation beschrieben).

- Im ersten Beispiel geht's um die Entwicklung eines Scanners und eines Parsers für numerische Expressions.
- Im zweiten Beispiel geht's um die Entwicklung eines Systems zum Entwurf logischer Schaltungen.
- Im dritten Beispiel geht's um die Entwicklung eines Systems, welches Inhalte von CSV-Dateien abbildet auf Listen von Java-Objekten.

Das erste Beispiel ist fokussiert auf das Konzept einzelner, kleiner Schritte. Im zweiten Beispiel steht eher der Spezifikations-Aspekt im Vordergrund. Im dritten Beispiel schließlich geht's u.a. um die Verwendung Java-spezifischer Features (Reflection).

Im Folgenden werden die Aufgaben grob umrissen – konkrete Entwicklungsschritte werden aber nicht weiter vorgegeben.

Bei der Implementierung sollte natürlich immer auch sofort refaktoriert werden – sobald sich solche Refaktorisierungen anbieten.

8.1 Parser für numerische Expressions

Eine String-förmige Eingabe enthält numerische Ausdrücke. Die numerischen Werte solcher Ausdrücke sollen berechnet werden.

Sei etwa folgende Eingabe gegeben:

```
"10 * (200 - 150) + 3 * (5 - (3 - 1))"
```

Der Wert dieses Ausdrucks ist 59 (natürlich unter der Berücksichtigung der Vorrang-Regel: Punkt- geht vor Strich-Rechnung).

Zunächst geht es darum, die Folge der in der Eingabe enthaltenen Zeichen in eine Folge von "Symbolen" zu transformieren – dies ist die Aufgabe des `Scanners`. Diese vom Scanner erzeugte Symbolfolge wird dann vom eigentlichen Parser weiterverarbeitet werden.

Die obige Zeichenfolge z.B. enthält folgende Symbole

```
10
*
(
200
-
150
)
+
3
etc....
```

Die Spezifikation des `Scanners` ist bereits vorgegeben.

Der `Scanner` liefert Symbole vom Typ `Symbol`:

```
package scanner;

public enum Symbol {
    PLUS, MINUS, TIMES, DIV, OPEN, CLOSE, NUMBER
}
```

`OPEN` steht für die öffnende, `CLOSE` für die schließende Klammer.

Der `Scanner` ist über ein Interface spezifiziert (damit er gemockt werden kann):

```
package scanner;

public interface Scanner {
    public abstract void next();
    public abstract Symbol currentSymbol();
    public abstract double getNumber();
}
```

Der Aufruf von `next` treibt den Scanner jeweils zum nächsten Symbol; `currentSymbol` liefert das aktuelle Symbol zurück (oder null, wenn die Eingabe kein weiteres Symbol mehr enthält); `getNumber` liefert den numerischen Wert des aktuellen Symbols zurück (sofern `currentSymbol` vom Typ `NUMER` ist).

Die (noch unfertige) Implementierung:

```
package scanner;

import java.io.Reader;
import java.util.HashMap;
import java.util.Map;

public class ScannerImpl implements Scanner {

    private static Map<Character, Symbol> operators
        = new HashMap<Character, Symbol>();

    static {
        operators.put('+', Symbol.PLUS);
        operators.put('-', Symbol.MINUS);
        operators.put('*', Symbol.TIMES);
        operators.put('/', Symbol.DIV);
        operators.put('(', Symbol.OPEN);
        operators.put(')', Symbol.CLOSE);
    }

    private final Reader reader;
    private int currentChar;
    private Symbol currentSymbol;

    public ScannerImpl(final Reader reader) {
        this.reader = reader;
        this.nextChar();
    }

    @Override
    public void next() {
        if (this.currentChar == -1) {
            this.currentSymbol = null;
            return;
        }
        this.currentSymbol = operators.get((char) this.currentChar);
        if (this.currentSymbol == null)
            throw new RuntimeException("unexpected char: " +
                (char) this.currentChar +
                " code = " + this.currentChar);
        this.nextChar();
    }

    @Override
    public Symbol currentSymbol() {
        return this.currentSymbol;
    }

    @Override
    public double getNumber() {
```

```
        return 0;
    }

    private void nextChar() {
        try {
            this.currentChar = this.reader.read();
        }
        catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Eine erste Version der Testklasse ist ebenfalls bereits vorgegeben:

```
package test;

import java.io.StringReader;

import org.junit.Assert;
import org.junit.Test;

import scanner.Scanner;
import scanner.ScannerImpl;
import scanner.Symbol;
import util.test.XAssert;

public class ScannerTest {

    @Test
    public void testEOF() {
        final Scanner scanner
            = new ScannerImpl(new StringReader(""));
        scanner.next();
        Assert.assertNull(scanner.currentSymbol());
    }

    @Test
    public void testOperators() {
        final Scanner scanner
            = new ScannerImpl(new StringReader("+-*/"));
        scanner.next();
        Assert.assertSame(
            Symbol.PLUS, scanner.currentSymbol());
        scanner.next();
        Assert.assertSame(
            Symbol.MINUS, scanner.currentSymbol());
        scanner.next();
        Assert.assertSame(
            Symbol.TIMES, scanner.currentSymbol());
        scanner.next();
        Assert.assertSame(
            Symbol.DIV, scanner.currentSymbol());
        scanner.next();
        Assert.assertNull(scanner.currentSymbol());
    }

    @Test
    public void testIllegalChar() {
```

```
final Scanner scanner
    = new ScannerImpl(new StringReader("?"));
XAssert.assertThrows(RuntimeException.class,
    () -> scanner.next());
}
```

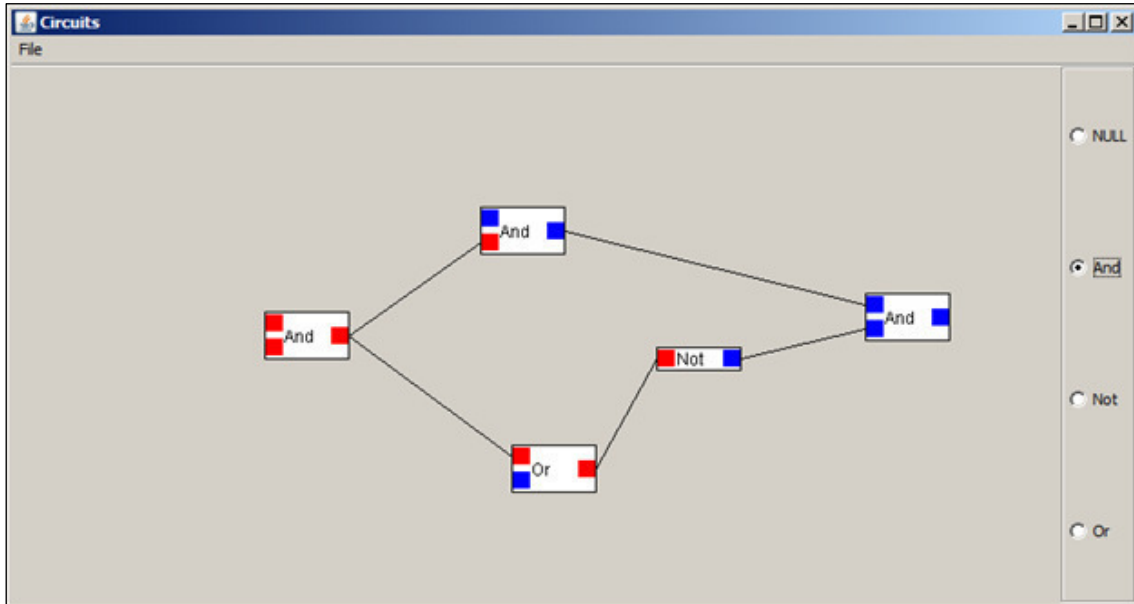
Hinweise für mögliche Entwicklungsschritte:

- Der Scanner soll folgende Eingabe korrekt scannen: "+-/*"
- Der Scanner soll Whitespaces überlesen: " + - /* "
- Der Scanner soll ganze Zahlen erkennen: " 123 + 44 */+ 77 "
- Der Scanner soll Gleitkommazahlen erkennen: " 12.3 + 44 */+ 7.7 "
- Der Parser sollte eine Zahl parsen können: " 12.3 "
- Der Parser sollte eine multiplikative Verknüpfung parsen können: " 2 * 30/10"
- Der Parser sollte eine additive Verknüpfung parsen können (welche ihrerseits multiplikative Verknüpfungen enthält): " 2 * 30 / 10 + 20 - 10 * 5 "
- Der Parser sollte geklammerte Expressions parsen können: " (1 + 2) * 3 "
- Der Parser sollte statt eines numerischen Ergebnisses eine Baum von Expression-Objekten liefern, welche den Ausdruck repräsentieren. Solche Expressions sollten berechnet werden können.
- Wie könnte man Expressions mit `DoubleBinaryOperators` ausstatten, welche "Operatoren" repräsentieren?

Viel Spaß && Erfolg bei der Entwicklung!!

8.2 Entwurf logischer Schaltungen

Es geht um die Entwicklung eines Systems, welches den Entwurf beliebig komplexer logischer Schaltungen ermöglicht.



Hier soll allerdings keine(!) grafische Oberfläche entwickelt werden – sondern "nur" der "Kern", der sich hinter einer solchen GUI verbirgt.

Eine logische Schaltung (ein `Circuit`) kann beliebig viele Eingänge und Ausgänge enthalten. And- und Or-Schaltungen haben jeweils zwei Eingänge (`Inputs`) und einen Ausgang (`Output`); eine Not-Schaltung hat einen Eingang und einen Ausgang; ein Halbaddierer hätte drei Eingänge und zwei Ausgänge.

Welche wichtigen Use-Cases können unterschieden werden?

- Eine Schaltung muss erzeugt werden können. Nach der Erzeugung einer And- und einer Not-Schaltung sind alle Eingänge und der Ausgang jeweils "blau" (also `false`). Nach der Erzeugung einer Not-Schaltung ist auch deren Eingang blau – der Ausgang aber "rot" (also `true`).
- Sofern ein Eingang nicht mit einem Ausgang einer anderen Schaltung verbunden ist – sofern er also "frei" ist – , kann dieser Eingang "getogglet" werden: aus "rot" wird "blau" und umgekehrt. Das hat zur Folge, dass möglicherweise auch der Ausgang (die Ausgänge) dieser Schaltung ihren Zustand ändern müssen – und auch die Eingänge, die von diesen Ausgängen "versorgt" werden etc.
- Ein Ausgang kann mit dem Eingang einer anderen Schaltung verbunden werden – aber nur dann, wenn dieser Eingang "frei" ist (anschließend ist dieser Eingang dann nicht mehr "frei"). Ein Ausgang kann mit mehreren Eingängen verbunden werden (welche

er dann "versorgt") – aber ein Eingang kann nur von einem einzigen Ausgang versorgt werden.

- Man möchte den Typ einer Schaltung ändern können – aus einer And-Schaltung z.B. eine Or-Schaltung machen. (Das setzt voraus, dass die neue Schaltung dieselbe Anzahl Eingänge und Ausgänge hat wie die alte Schaltung.)

An diesen Use-Cases könnte sich die Entwicklung orientieren.

Welche Klassen werden benötigt? Im obigen Text wurden bereits folgende Begriffe genannt: `Circuit`, `Input`, `Output`. Vielleicht kann sich die Entwicklung an diesen "Substantiven" orientieren.

Hier eine mögliche Schrittfolge:

- Schaltungen sollten nach ihrer Erzeugung einen korrekten Zustand haben
- Die Eingänge von Schaltungen sollten getogglet werden können.
- Ausgänge sollten mit Eingängen verbunden werden können (sofern die Eingänge frei sind).
- Eingänge sollten nicht getogglet werden können, wenn sie bereits von einem Ausgang versorgt werden.
- Sind Schaltungen bereits mit anderen Schaltungen verbunden, so hat das Togglen möglicherweise eine "weitreichende" Wirkung.
- Angenommen, eine And-Schaltung ist bereits mit einer Vielzahl anderer Schaltungen verbunden. Dann bemerkt der Benutzer, dass die And-Schaltung eigentlich eine Or-Schaltung sein sollte. Wie kann der Typ einer Schaltung geändert werden?
- Zirkuläre Verbindungen müssen ausgeschlossen werden. Ein Ausgang einer Schaltung A darf nicht auf den Eingang einer Schaltung B zeigen, über welche wiederum die Schaltung A erreicht werden kann.

Hier ein möglicher Einstieg:

```
package test;

import org.junit.Assert;
import org.junit.Test;

import circuits.AndCircuit;
import circuits.Circuit;
import circuits.NotCircuit;
import circuits.OrCircuit;

public class CircuitTest {

    @Test
    public void testAnd() {
        final Circuit c = new AndCircuit();
        Assert.assertEquals(2, c.getInputCount());
    }

    @Test
    public void testOr() {
        final Circuit c = new OrCircuit();
        Assert.assertEquals(2, c.getInputCount());
    }

    @Test
    public void testNot() {
        final Circuit c = new NotCircuit();
        Assert.assertEquals(1, c.getInputCount());
    }
}
```

```
package circuits;

public abstract class Circuit {
    public abstract int getInputCount();
    public abstract int getOutputCount();
}
```

```
package circuits;

public class AndCircuit extends Circuit {

    @Override
    public int getInputCount() {
        return 2;
    }

    @Override
    public int getOutputCount() {
        return 1;
    }
}
```

```
package circuits;  
  
public class OrCircuit extends Circuit {  
  
    @Override  
    public int getInputCount() {  
        return 2;  
    }  
  
    @Override  
    public int getOutputCount() {  
        return 1;  
    }  
}
```

```
package circuits;  
  
public class NotCircuit extends Circuit {  
  
    @Override  
    public int getInputCount() {  
        return 1;  
    }  
  
    @Override  
    public int getOutputCount() {  
        return 1;  
    }  
}
```

Viel Spaß && Erfolg bei der Entwicklung!!

8.3 CSV-Dateien

Wir bauen eine Klasse `CSVReader`, dessen `readLine`-Methode jeweils die nächste Zeile einer CSV-Datei liefert. Die Methode gibt einen String-Array zurück: den Array der Tokens der jeweils nächsten Zeile.

Der `CSVReader` muss neben der Eingabe denjenigen String kennen, der die Tokens einer Zeile trennt (z.B.: ";"). Zusätzlich sollte er die erwartete Anzahl der Tokens pro Zeile kennen (um evtl. bereits Exceptions werfen zu können). Die Eingabe wird dem `CSVReader` in Form eines `Readers` übergeben.

Hier der erste Test:

```
package test;

import java.io.StringReader;
import org.junit.Assert;
import org.junit.Test;
import util.CSVReader;
import util.test.XAssert;

public class CSVReaderTest {

    @Test
    public void testEmptyInput() {
        final String s = "";
        final CSVReader reader
            = new CSVReader(new StringReader(s), 2, ";");
        Assert.assertNull(reader.readLine());
    }

    @Test
    public void testOneLine() {
        final String s = "111;222\n";
        final CSVReader reader
            = new CSVReader(new StringReader(s), 2, ";");
        Assert.assertArrayEquals(
            new String[] { "111", "222" },
            reader.readLine());
        Assert.assertNull(reader.readLine());
    }

    @Test
    public void testThreeLines() {
        final String s = "111;222\n333;444\n555;666\n";
        final CSVReader reader
            = new CSVReader(new StringReader(s), 2, ";");
        Assert.assertArrayEquals(
            new String[] { "111", "222" },
            reader.readLine());
        Assert.assertArrayEquals(
            new String[] { "333", "444" },
            reader.readLine());
        Assert.assertArrayEquals(
            new String[] { "555", "666" },
            reader.readLine());
        Assert.assertNull(reader.readLine());
    }
}
```

```
}

@Test
public void testException() {
    final String s = "111\n";
    final CSVReader reader
        = new CSVReader(new StringReader(s), 2, ";");
    XAssert.assertThrows(
        RuntimeException.class, () -> reader.readLine());
}
}
```

Und die erste Implementierung:

```
package util;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.Reader;

public class CSVReader {

    private final BufferedReader reader;
    private final int columnCount;
    private final String seperator;

    public CSVReader(
        final Reader reader,
        final int columnCount,
        final String seperator) {
        this.reader = new BufferedReader(reader);
        this.columnCount = columnCount;
        this.seperator = seperator;
    }

    public String[] readLine() {
        try {
            final String line = this.reader.readLine();
            if (line == null)
                return null;
            final String[] tokens = line.split(this.seperator);
            if (tokens.length != this.columnCount)
                throw new RuntimeException("illegal line: " + line);
            return tokens;
        }
        catch (final IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Dieser `CSVReader` soll wie folgt erweitert werden:

- Er sollte auch mit Leerzeilen umgehen können.
- Die Tokens einer Zeile sollten in getrimmter Form zurückgeliefert werden.
- Der `CSVReader` sollte auch mit einer Header-Zeile klarkommen, in welcher die Namen der Tokens hinterlegt sind.
- Eine vom `CSVReader` gelieferte Zeile sollte automatisch in eine Java-Objekt transformiert werden.

Viel Spaß && Erfolg bei der Entwicklung!!

9

Refactoring: Beispiel Gruppenwechsel

9.1	Gruppenwechsel.....	9-5
-----	---------------------	-----

9 Refactoring: Beispiel Gruppenwechsel

Nicht immer startet der Entwickler auf der grünen Wiese – oft genug passiert es, dass er sich in existierenden, "gewachsenen" Quellcode einarbeiten muss, um seine Funktionalität zu erweitern oder einfach um Fehler zu beseitigen. Bei einer solchen Einarbeitung kann es sinnvoll sein, schrittweise zu refaktorisieren – durch den Versuch einer Refaktorisierung jeweils einzelner Teile werden diese Teile häufig überhaupt erst verständlich. Man könnte sagen: Verstehen heißt Refaktorisieren. Refaktorisieren ist risikolos, wenn bereits Tests vorliegen. Diese können dann entweder nach erfolgter Refaktorisierung direkt genutzt werden (um zu zeigen, dass die Funktionalität durch die Refaktorisierung nicht beeinträchtigt wurde) oder aber sukzessive erweitert werden.

Bei der Refaktorisierung geht es zunächst natürlich einmal darum, Stellen in der Software zu entdecken, die "schlecht riechen". Hier eine kleine Liste von Kriterien für schlechten Code (die Reihenfolge der Punkte ist eher zufällig):

- Code-Duplikation (Copy & Paste)
- keine klaren Verantwortlichkeiten
- Eierlegende Wollmilchsau
- "langweiliger" Code (stattdessen vielleicht Reflection? Iteration?)
- Exzessive dynamische Typabfrage (`instanceof`)
- Exzessiver Gebrauch von Downcasts
- Exzessiver Gebrauch von "switch"-Konstrukten (mit `instanceof` und `downcast`)
- Exzessiver Gebrauch von `static`
- inkonsistente Namen (Finder beginnen mit `find`, `get`, `read`...)
- inkonsistente Groß/Kleinschreibung (`class c`, `void M()`)
- fehlendes `final`
- Return-Werte statt Exceptions
- viele `catch`-Blöcke für einen `try`, die alle dieselbe Implementierung haben
- Definition lokaler Variablen ohne Initialisierung
- Definition lokaler Variablen, weit bevor sie benötigt werden
- Definition einer Instanzvariablen, wo auch eine lokale Variable ausreicht
- Änderung von Parametern

- Benutzung von `ArrayList` o.ä., wenn ein einfacher Array angemessen ist
- Im Konstruktor: Aufruf von Methoden, die überschrieben werden können
- festverdrahtete Parameter
- Exzessiver Gebrauch von Klassenvererbung(=> Objektkomposition via Interfaces vs. klassischer Vererbung)
- Instanzvariablen namens "typ"
- "Schalter"-Parameter
- tief geschachtelte Kontrollstrukturen
- zu große Methoden
- zu große Klassen
- zu lange Parameterlisten
- lange Message-Chains
- Benutzung von Gruppen von primitiven Daten statt expliziter Objekte
- Direkter Zugriff auf den internen Zustand von Objekten
- Unvollständige Library-Klassen
- Geschwätziige, überflüssige oder schlichtweg falsche Kommentare

Im Folgenden soll in Form einer größeren Übung ein einfaches, aber nicht triviales Programm refaktoriert werden. Das zu refaktorisierende Programm ist vorgegeben – allerdings in einigermaßen grausamer Gestalt. In seiner anfänglichen Form ist das Programm auch kaum testbar. Deshalb soll es in mehreren Schritten refaktoriert werden – u.a. mit dem Ziel, dass seine Teile immer besser testbar werden. Für jede neu entwickelte Klasse soll also auch eine Testklasse erstellt werden. Ein weiteres Ziel ist natürlich ein verbessertes Design – womöglich auch die Erstellung wiederverwendbarer Komponenten.

9.1 Gruppenwechsel

Es handelt sich bei dem vorliegenden Programm um einen einfachen Gruppenwechsel. Eine Eingabedatei enthält nach Gruppen sortierte Auftrags-Sätze. Diese in der Eingabe implizite Gruppenstruktur wird transformiert in eine Druckliste, welche die Gruppenstruktur explizit ausweist.

Die Eingabe-Dateien:

`customers.txt` enthält für jeden Kunden dessen Nummer und dessen Namen:

```
1000,Nowak
2000,Rueschenpoehler
3000,Montjoe
```

`products.txt` enthält für jedes Produkt dessen Nummer, Namen und dessen Einzelpreis:

```
100,Jever,11
200,Veltins,12
300,Krombacher,13
400,Bitburger,14
```

`orders.txt` enthält die Aufträge. Jeder Auftrag hat eine Kundennummer, eine Produktnummer und eine Menge. Die Aufträge eines Kunden sind jeweils zu einer Gruppe zusammengefasst.

```
1000;100;10
1000;200;20
2000;200;20
2000;300;30
2000;400;40
3000;100;10
```

Das folgende Programm erstellt eine Liste aller Aufträge. Sie hat einen Kopf und einen Fuß (im Fuß wird der Gesamtwert aller Aufträge ausgegeben). Für jede Gruppe von Aufträgen wird ein Gruppenkopf (mit der Nummer und dem Namen des Kunden) und ein Gruppenfuß ausgegeben (mit der Summe aller Auftragswerte dieses Kunden). Und für jede Auftragsposition wird eine Zeile mit Produkt-Nummer, Produkt-Namen, Einzelpreis, Menge und Positionswert ausgegeben.

Hier die aufgrund der obigen Eingaben produzierte Ausgabe (`result.txt`):

Orders

1000 Nowak

100 Jever 10 11 110

200 Veltins 20 12 240

350

2000 Rueschenpoehler

200 Veltins 20 12 240

300 Krombacher 30 13 390

400 Bitburger 40 14 560

1190

3000 Montjoe

100 Jever 10 11 110

110

Total: 1650

Hier das "vorgefundene" Programm:

```
package appl;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class Application {

    public static void main(final String[] args) {

        final ArrayList<String[]> customers = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(
            new FileReader("customers.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                final String[] customer = line.split(",");
                if (customer.length != 2)
                    continue;
                customers.add(customer);
            }
        }
        catch(final IOException e) {
            throw new RuntimeException(e);
        }

        final ArrayList<String[]> products = new ArrayList<>();
        try (final BufferedReader reader = new BufferedReader(
            new FileReader("products.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                final String[] product = line.split(",");
                if (product.length != 3)
                    continue;
                products.add(product);
            }
        }
        catch(final IOException e) {
            throw new RuntimeException(e);
        }

        try (PrintWriter writer = new PrintWriter("result.txt")) {
            try (BufferedReader reader = new BufferedReader(
                new FileReader("orders.txt"))) {
                writer.println("Orders");
                writer.println();
                String[] order = readOrder(reader);
                int sum = 0;
                while (order != null) {
                    final String customerNumber = order[0];
                    final String[] customer =
                        getCustomer(customerNumber, customers);
                    writer.println(
                        customerNumber + " " + customer[1]);
                    int groupSum = 0;
                    while (order != null
```

```

        && order[0].equals(customerNumber)) {
            final String productNumber = order[1];
            final String[] product =
                getProduct(productNumber, products);
            final String productName = product[1];
            final int productPrice =
                Integer.parseInt(product[2]);
            final int positionPrice =
                productPrice *
                Integer.parseInt(order[2]);
            writer.println("\t" +
                productNumber + " " +
                productName + " " +
                order[2] + " " +
                productPrice + " " +
                positionPrice);
            groupSum += positionPrice;
            order = readOrder(reader);
        }
        sum += groupSum;
        writer.println("\t-----");
        writer.println("\t" + groupSum);
        writer.println();
    }
    writer.println("Total:\t" + sum);
    System.out.println("Done. See results.txt");
}

catch(final IOException e) {
    throw new RuntimeException(e);
}

}

private static String[] readOrder(final BufferedReader reader)
    throws IOException {
    while (true) {
        final String line = reader.readLine();
        if (line == null)
            return null;
        final String[] order = line.split(";");
        if (order.length != 3)
            continue;
        System.out.println(line);
        return order;
    }
}

private static String[] getCustomer(final String number,
    final ArrayList<String[]> customers) {
    for (final String[] customer : customers) {
        if (customer[0].equals(number))
            return customer;
    }
    throw new RuntimeException(
        "customer " + number + " not found");
}

private static String[] getProduct(final String number,
    final ArrayList<String[]> products) {

```

```
    for (final String[] product : products) {
        if (product[0].equals(number))
            return product;
    }
    throw new RuntimeException(
        "product " + number + " not found");
}
```


10

Spezielle Test-Werkzeuge

10.1	Testen von DB-Anwendungen: DbUnit.....	10-4
10.2	Testen von DB-Anwendungen: Simple Tool.....	10-10
10.3	Testen von Swing-Anwendungen: FEST	10-13
10.4	Testen von WEB-Anwendungen: Selenium.....	10-19
10.5	Testen von EJB-Anwendungen: OpenEJB	10-28
10.6	Testen mit FIT	10-35

10 Spezielle Test-Werkzeuge

Im Folgenden werden einige Spezialwerkzeuge vorgestellt: Werkzeuge zum Testen von Datenbank-basierten Anwendungen, von GUI-Anwendungen, von Web-Anwendung und EJB-Anwendungen. Das Kapitel endet mit einer kleinen Einführung in das FIT-Framework.

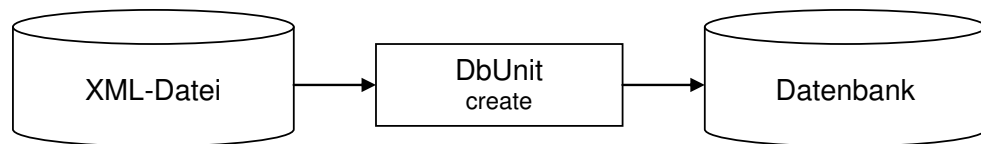
- Im ersten Abschnitt wird DbUnit vorgestellt – ein Werkzeug zum Testen von Datenbank-Klassen.
- Im zweiten Abschnitt wird ein Db-Testwerkzeug vorgestellt, welches vom Autor dieses Skripts entwickelt wurde. Nach Auffassung des Autors ist dieses Werkzeug einfacher zu handhaben als DbUnit.
- Im dritten Abschnitt wird FEST vorgestellt – ein Werkzeug zum Testen von Swing-Anwendungen.
- Im vierten Abschnitt werden die Grundlagen von Selenium erläutert – einem Werkzeug zum Testen von WEB-Anwendungen.
- Im fünften Abschnitt geht's um OpenEJB – ein Werkzeug zu Testen von EJBs.
- Im letzten Abschnitt schließlich wird ein Ausblick auf FIT vorgestellt – ein Werkzeug, welches dem End-Anwender die Spezifikation von Tests erlaubt.

In allen Abschnitten wird jeweils nur ein grundlegender Einblick in die Werkzeuge vermittelt. Um eines dieser Werkzeuge produktiv einzusetzen, ist natürlich ein tieferes Verständnis des jeweiligen Werkzeugs unumgänglich.

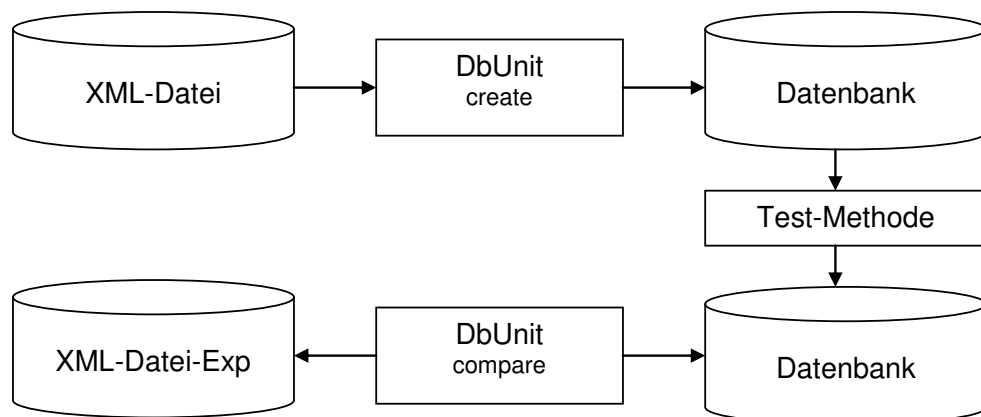
10.1 Testen von DB-Anwendungen: DbUnit

DbUnit (<http://www.dbunit.org>) ist eine Erweiterung von JUnit, die für das Testen von Datenbankanwendungen geeignet ist.

Sollen Datenbankzugriffe getestet werden, ist es erforderlich, dass vor der Ausführung einer jeden Testmethode die Datenbank in einen initialen Zustand versetzt wird. Dieser Zustand wird in Form einer XML-Datei beschrieben. DbUnit kann dann dafür sorgen, dass vor der Ausführung einer jeden Test-Methode die Datenbank gemäß dieser XML-Datei eingerichtet wird.



Die jeweilige Testmethode wird die auf diese Weise eingerichtete Datenbank in irgendeiner Weise manipulieren. Dann sollte man spezifizieren können, wie die Datenbank anschließend aussehen wird (wieder in Form einer XML-Datei). Und diese Erwartung sollte verglichen werden können mit den tatsächlichen Auswirkungen.



dataset.xml

```
<dataset>
  <table name="ACCOUNT">
    <column>NUMBER</column>
    <column>BALANCE</column>
    <column>CREDIT</column>
    <row>
      <value>4711</value>
      <value>5000</value>
      <value>1000</value>
    </row>
    <row>
      <value>4712</value>
      <value>3000</value>
      <value>500</value>
    </row>
  </table>
  <!-- weitere table-Elemente -->
</dataset>
```

Die erste Testmethode wird eine Einzahlung auf 4711 vornehmen. Danach sollte die Datenbank wie folgt ausschauen:

expectedAfterDeposit.xml

```
<dataset>
  <table name="ACCOUNT">
    <column>NUMBER</column>
    <column>BALANCE</column>
    <column>CREDIT</column>
    <row>
      <value>4711</value>
      <value>7000</value>
      <value>10</value>
    </row>
    <row>
      <value>4712</value>
      <value>10000</value>
      <value>20</value>
    </row>
  </table>
</dataset>
```

Die zweite Testmethode wird eine Auszahlung von 4711 vornehmen. Danach sollte die Datenbank wie folgt ausschaun:

expectedAfterWithdraw.xml

```
<dataset>
  <table name="ACCOUNT">
    <column>NUMBER</column>
    <column>BALANCE</column>
    <column>CREDIT</column>
    <row>
      <value>4711</value>
      <value>3000</value>
      <value>1000</value>
    </row>
    <row>
      <value>4712</value>
      <value>3000</value>
      <value>500</value>
    </row>
  </table>
  <!-- weitere table-Elemente -->
</dataset>
```

Die dritte Testmethode schließlich wird eine Überweisung von 4711 nach 4712 vornehmen. Danach sollte die Datenbank wie folgt ausschaun:

expectedAfterWithdraw.xml

```
<dataset>
  <table name="ACCOUNT">
    <column>NUMBER</column>
    <column>BALANCE</column>
    <column>CREDIT</column>
    <row>
      <value>4711</value>
      <value>3000</value>
      <value>1000</value>
    </row>
    <row>
      <value>4712</value>
      <value>5000</value>
      <value>500</value>
    </row>
  </table>
  <!-- weitere table-Elemente -->
</dataset>
```

Hier die Testklasse:

```
package test;
// ...
import services.AccountService;

import org.dbunit.Assertion;
import org.dbunit.IDatabaseTester;
import org.dbunit.JdbcDatabaseTester;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.Column;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.ITable;
import org.dbunit.dataset.ITableMetaData;
import org.dbunit.dataset.xml.XmlDataSet;

import db.util.Jdbc;
import db.util.JdbcProperties;
import db.util.JdbcUtil;

public class AccountServiceTest {

    private IDatabaseTester databaseTester;

    private Connection con;

    private AccountService accountService;

    @Before
    public void before() throws Exception {
        JdbcProperties props = new JdbcProperties("db.properties");

        this.databaseTester = new JdbcDatabaseTester(
            props.getDriver(), props.getUrl(),
            props.getUser(), props.getPassword());
        IDataSet dataSet = new XmlDataSet(
            ClassLoader.getResourceAsStream("dataset.xml"));
        IDatabaseConnection dbcon =
this.databaseTester.getConnection();
        this.con = dbcon.getConnection();

        this.databaseTester.setDataSet(dataSet);
        this.databaseTester.onSetup();
        this.accountService = new AccountService(this.con);
    }

    @After
    public void after() throws Exception {
        this.databaseTester.onTearDown();
    }

    @Test
    public void testDeposit() {
        try {
            this.accountService.deposit(4711, 2000);
            Jdbc.commit(this.con);
        }
        catch (Exception e) {
            Jdbc.rollback(this.con);
        }
    }
}
```



```
        throw e;
    }
    this.compareWith("expectedAfterDeposit.xml");
}

@Test
public void testWithdraw() {
    try {
        this.accountService.withdraw(4711, 2000);
        Jdbc.commit(this.con);
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
        throw e;
    }
    this.compareWith("expectedAfterWithdraw.xml");
}

@Test
public void testTransfer() {
    try {
        this.accountService.transfer(4711, 4712, 2000);
        Jdbc.commit(this.con);
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
        throw e;
    }
    this.compareWith("expectedAfterTransfer.xml");
}

@Test
public void testDepositWithIllegalNumber() {
    try {
        this.accountService.deposit(4713, 2000);
        fail();
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
    }
}

@Test
public void testWithdrawWithIllegalAmount() {
    try {
        this.accountService.withdraw(4711, 20000);
        fail();
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
    }
}

@Test
public void testTransferWithIllegalAmount() {
    try {
        this.accountService.transfer(4711, 4712, 20000);
        fail();
    }
}
```

```

        catch (Exception e) {
            Jdbc.rollback(this.con);
        }
    }

    private void compareWith(String filename) {
        try {
            IDataset actualDataSet =
                this.databaseTester.getConnection().createDataSet();
            IDataset expectedDataSet = new XmlDataSet(
                ClassLoader.getResourceAsStream(filename));
            Assertion.assertEquals(expectedDataSet, actualDataSet);

            ITable actualTable = actualDataSet.getTable("ACCOUNT");
            ITable expectedTable =
                expectedDataSet.getTable("ACCOUNT");

            Assertion.assertEquals(expectedTable, actualTable);
            this.print(actualTable);
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private void print(ITable table) throws Exception {
        ITableMetaData md = table.getTableMetaData();
        System.out.println("tableName = " + md.getTableName());
        this.print("columns:", md.getColumns());
        this.print("primaryKeys:", md.getPrimaryKeys());
        Column[] columns = md.getColumns();
        System.out.println("-----");
        for (int row = 0; row < table.getRowCount(); row++) {
            for (int col = 0; col < columns.length; col++) {
                Object value = table.getValue(row,
                    columns[col].getColumnName());
                if (col > 0)
                    System.out.print(", ");
                System.out.print(value);
            }
            System.out.println();
        }
    }

    private void print(String text, Column[] columns) {
        System.out.println(text);
        for (Column c : columns) {
            System.out.println("\t" + c.getColumnName() + " (" +
                c.getSqlTypeName() + ")");
        }
    }
}

```

10.2 Testen von DB-Anwendungen: Simple Tool

DbUnit erscheint recht "umständlich". Im Folgenden wird daher kurz ein einfacheres Tool vorgestellt, welches vom Autor dieses Skripts entwickelt wurde. Das Tool ist ausbaufähig...

Das genaue Studium dieses Tools sei dem Leser / der Leserin überlassen...

```
package test;
// ...
import services.AccountService;

import db.util.Jdbc;
import db.util.JdbcProperties;
import db.util.JdbcTest;

public class AccountServiceTest {

    private static JdbcProperties props = new
JdbcProperties("db.properties");

    private JdbcTest test;

    private Connection con;

    private AccountService accountService;

    @Before
    public void before() throws Exception {
        this.test = new JdbcTest(AccountServiceTest.class, props,
System.out);
        this.con = this.test.getConnection();
        this.accountService = new AccountService(this.con);
        final String[] prepareStrings = new String[] {
            "create table account"
            + "("
            + "    number integer,"
            + "    balance integer,"
            + "    credit integer,"
            + "    primary key(number)"
            + ")",
            "insert into account values(4711, 5000, 1000)",
            "insert into account values(4712, 3000, 500)"
        };
        this.test.prepare(prepareStrings);
    }

    @After
    public void after() throws Exception {
        this.test.close();
    }

    @Test
    public void testDeposit() {
        try {
            this.accountService.deposit(4711, 2000);
        }
    }
}
```

```

        Jdbc.commit(this.con);
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
        throw e;
    }
    final String[] deltaStrings = new String[] {
        "update account set balance = 7000 where number = 4711"
    };

    this.test.assertDelta(deltaStrings);
}

@Test
public void testWithdraw() {
    try {
        this.accountService.withdraw(4711, 2000);
        Jdbc.commit(this.con);
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
        throw e;
    }
    final String[] deltaStrings = new String[] {
        "update account set balance = 3000 where number = 4711"
    };

    this.test.assertDelta(deltaStrings);
}

@Test
public void testTransfer() {
    try {
        this.accountService.transfer(4711, 4712, 2000);
        Jdbc.commit(this.con);
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
        throw e;
    }
    final String[] deltaStrings = new String[] {
        "update account set balance = 3000 where number = 4711",
        "update account set balance = 5000 where number = 4712"
    };

    this.test.assertDelta(deltaStrings);
}

@Test
public void testDepositWithIllegalNumber() {
    try {
        this.accountService.deposit(4713, 2000);
        this.test.assertFail();
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
    }
    this.test.assertDelta(null);
}

```

```
@Test
public void testWithdrawWithIllegalAmount() {
    try {
        this.accountService.withdraw(4711, 20000);
        this.test.assertFail();
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
    }
    this.test.assertDelta(null);
}

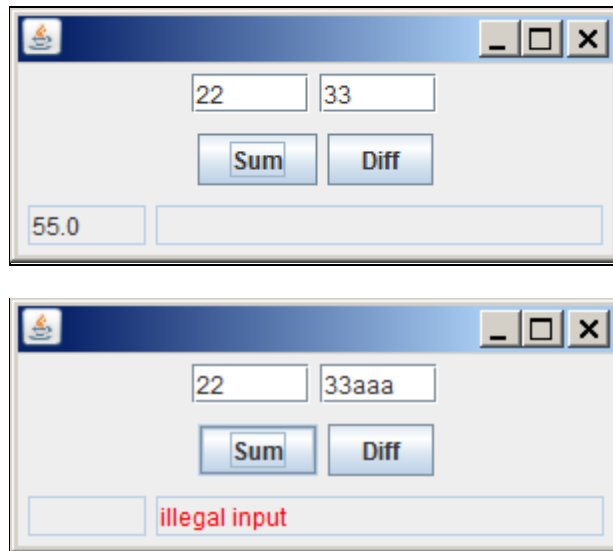
@Test
public void testTransferWithIllegalAmount() {
    try {
        this.accountService.transfer(4711, 4712, 20000);
        this.test.assertFail();
    }
    catch (Exception e) {
        Jdbc.rollback(this.con);
    }
    this.test.assertDelta(null);
}
}
```

10.3 Testen von Swing-Anwendungen: FEST

Im Folgenden wird ein Tool zum Testen von Swing-Anwendungen vorgestellt: FEST. Hier die URL, über welche die erforderlichen jar-Dateien bezogen werden können:

<http://docs.codehaus.org/display/FEST/Swing+Module>

Als Beispiel wird ein kleiner Kalkulator verwendet:



Um eine GUI zu testen, benötigt man ein Tool, welches Benutzereingaben simuliert. Im Folgenden werden sowohl "korrekte" als auch "fehlerhafte" Eingaben simuliert werden.

Hier die zu testende Klasse:

```
package appl;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.util.function.DoubleBinaryOperator;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class MathFrame extends JFrame {

    private static final long serialVersionUID = 1L;

    public static void main(final String[] args) {
        new MathFrame();
    }

    private final JTextField textFieldX = new JTextField(5);
    private final JTextField textFieldY = new JTextField(5);
    private final JButton buttonSum = new JButton("Sum");
    private final JButton buttonDiff = new JButton("Diff");
    private final JTextField textFieldResult = new JTextField(5);
    private final JTextField textFieldError = new JTextField(20);
}
```

```
public MathFrame() {

    final JPanel panelNorth = new JPanel();
    final JPanel panelCenter = new JPanel();
    final JPanel panelSouth = new JPanel();

    this.textFieldX.setName("textFieldX");
    this.textFieldY.setName("textFieldY");
    this.buttonSum.setName("buttonSum");
    this.buttonDiff.setName("buttonDiff");
    this.textFieldResult.setName("textFieldResult");
    this.textFieldError.setName("textFieldError");

    this.setLayout(new BorderLayout());
    panelNorth.setLayout(new FlowLayout());
    panelCenter.setLayout(new FlowLayout());
    panelSouth.setLayout(new FlowLayout());

    this.add(panelNorth, BorderLayout.NORTH);
    this.add(panelCenter, BorderLayout.CENTER);
    this.add(panelSouth, BorderLayout.SOUTH);

    panelNorth.add(this.textFieldX);
    panelNorth.add(this.textFieldY);
    panelCenter.add(this.buttonSum);
    panelCenter.add(this.buttonDiff);
    panelSouth.add(this.textFieldResult);
    panelSouth.add(this.textFieldError);

    this.textFieldResult.setEditable(false);
    this.textFieldError.setEditable(false);
    this.textFieldError.setForeground(Color.red);

    this.buttonSum.addActionListener(e -> this.onCalc((x, y) -> x + y));
    this.buttonDiff.addActionListener(e -> this.onCalc((x, y) -> x - y));

    this.pack();
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setVisible(true);
}

private void onCalc(final DoubleBinaryOperator operator) {
    this.textFieldResult.setText("");
    this.textFieldError.setText("");
    try {
        final double x = Double.parseDouble(this.textFieldX.getText());
        final double y = Double.parseDouble(this.textFieldY.getText());
        final double result = operator.applyAsDouble(x, y);
        this.textFieldResult.setText(String.valueOf(result));
    }
    catch (final NumberFormatException e) {
        this.textFieldError.setText("illegal input");
    }
}
```

Man beachte, dass allen Komponenten via `setName` jeweils ein Name zugewiesen wird (konsequenterweise wird immer der Name der Referenzvariablen verwendet, welche auf die Komponente verweist). Über diese Namen werden die Komponenten dann im Testprogramm angesprochen werden.

Die Testklasse enthält eine statische innere Klasse namens `MathFixture`:

```
private static class MathFixture extends FrameFixture {

    public final JTextComponentFixture textFieldX;
    public final JTextComponentFixture textFieldY;
    public final JButtonFixture buttonSum;
    public final JButtonFixture buttonDiff;
    public final JTextComponentFixture textFieldResult;
    public final JTextComponentFixture textFieldError;

    public MathFixture(final MathFrame target) {
        super(target);
        this.textFieldX = this.textBox("textFieldX");
        this.textFieldY = this.textBox("textFieldY");
        this.buttonSum = this.button("buttonSum");
        this.buttonDiff = this.button("buttonDiff");
        this.textFieldResult = this.textBox("textFieldResult");
        this.textFieldError = this.textBox("textFieldError");
    }
}
```

Ein `MathFixture`-Objekt enthält für jede (interaktive) Komponente der Oberfläche eine "Fixture"-Referenz: für `JTextFields` jeweils ein `JTextComponentFixture`, für `JButtons` jeweils ein `JButtonFixture`.

Dem Konstruktor der Klasse wird der eigentliche `MathFrame` übergeben. Dieser wird an den Konstruktor der Basisklasse `FrameFixture` übergeben. Dann werden im Konstruktor die einzelnen `Fixture`-Objekte erzeugt – mittels Methoden der Basisklasse `FrameFixture` (`textBox`, `button`). Diesen Methoden wird jeweils der Name der "eigentlichen" Komponenten übergeben.

Nach Aufruf des Konstruktors sind also eine Reihe von `Fixture`-Objekten erzeugt worden, welche über den Namen der Swing-Komponenten mit diesen verbunden sind.

Hier nun die eigentliche Testklasse:

```
package test;
// ...
import org.fest.swing.edt.FailOnThreadViolationRepaintManager;
import org.fest.swing.edt.GuiActionRunner;
import org.fest.swing.edt.GuiQuery;
import org.fest.swing.fixture.FrameFixture;
import org.fest.swing.fixture.JButtonFixture;
import org.fest.swing.fixture.JTextComponentFixture;

public class MathFrameTest {

    private static class MathFixture extends FrameFixture { ... }

    private MathFixture fixture;

    @BeforeClass
```



```
public static void beforeClass() {
    FailOnThreadViolationRepaintManager.install();
}

@Before
public void before() {
    final MathFrame frame = GuiActionRunner.execute(
        new GuiQuery<MathFrame>() {
            @Override
            protected MathFrame executeInEDT() {
                return new MathFrame();
            }
        }
    );
    this.fixture = new MathFixture(frame);
    this.fixture.robot.settings().delayBetweenEvents(100);
    this.fixture.show();
}

@After
public void after() {
    this.fixture.cleanUp();
}

@Test
public void testSum() {
    this.fixture.textFieldX.enterText("40");
    this.fixture.textFieldY.enterText("2");
    this.fixture.buttonSum.click();
    this.fixture.textFieldResult.requireText("42.0");
    this.fixture.textFieldError.requireText("");
}

@Test
public void testDiff() {
    this.fixture.textFieldX.enterText("40");
    this.fixture.textFieldY.enterText("2");
    this.fixture.buttonDiff.click();
    this.fixture.textFieldResult.requireText("38.0");
    this.fixture.textFieldError.requireText("");
}

@Test
public void testBadInput() {
    this.fixture.textFieldX.enterText("aaa");
    this.fixture.textFieldY.enterText("bbb");
    this.fixture.buttonSum.click();
    this.fixture.textFieldResult.requireText("");
    this.fixture.textFieldError.requireText("illegal input");
}

@Test
public void testProperties() {
    this.fixture.textFieldError.foreground()
        .requireEqualTo(Color.RED);
    this.fixture.textFieldResult.requireNotEditable();
    this.fixture.textFieldError.requireNotEditable();
}
}
```

Der `MathFrame` wird jeweils in der `@Before`-Methode neu erzeugt:

```
final MathFrame frame = GuiActionRunner.execute(
    new GuiQuery<MathFrame>() {
        @Override
        protected MathFrame executeInEDT() {
            return new MathFrame();
        }
    }
);
```

Wenn Methoden auf Swing-Komponenten aufgerufen werden, müssen diese stets im Event Dispatch Thread (EDT) aufgerufen werden. Dies ist aber nicht derjenige Thread, in welchem die Testmethoden laufen. Also müssen alle Aufrufe an den EDT "delegiert" werden (`invokeLater...`). Daher die etwas komplizierte Erzeugung des `MathFrame`-Objekts...

Dann wird ein `FrameFixture` für diesen Frame erzeugt:

```
this.fixture = new MathFixture(frame);
```

Auf die `Fixture`-Objekte des `MathFixture`-Objekts können dann Methoden aufgerufen werden, welche die Benutzeraktionen simulieren: `enterText`, `click` etc. Z. B.:

```
this.fixture.textFieldX.enterText("40");
this.fixture.textFieldY.enterText("2");
this.fixture.buttonSum.click();
this.fixture.textFieldResult.requireText("42.0");
this.fixture.textFieldError.requireText("");
```

Um die Eigenschaften der Komponenten zu testen, können dann auf die `Fixture`-Objekte Methoden aufgerufen werden wie `requireText`, `requireEditable`, `foreground` etc.:

```
this.fixture.textFieldError.foreground()
    .requireEqualTo(Color.RED);
this.fixture.textFieldResult.requireNotEditable();
this.fixture.textFieldError.requireNotEditable();
```

Es ist immer umständlich, den GUI-Komponenten per `setName` einen Namen verpassen zu müssen. Das kann auch automatisch erledigt werden. Hier eine kleine Hilfsklasse:

```
package util;

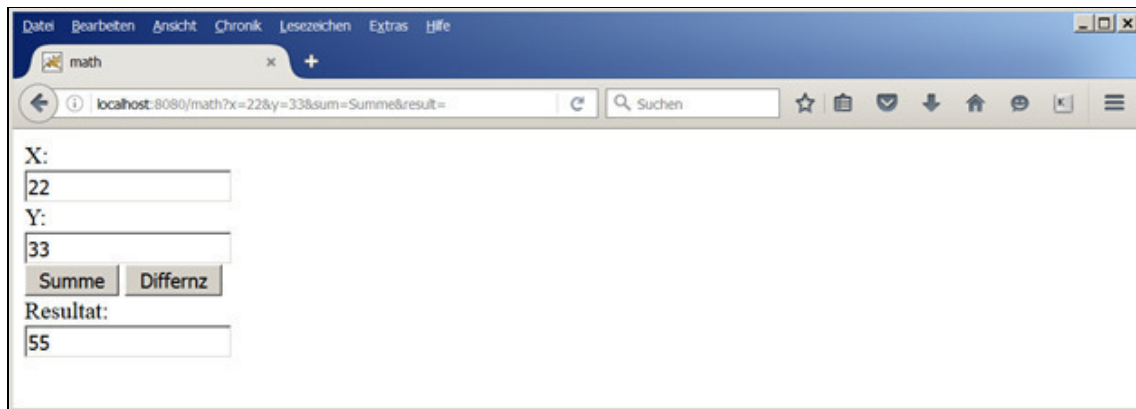
import java.awt.Component;
import java.lang.reflect.Field;

public class SwingUtils {
    public static void setNames(Object container) {
        try {
            for (Field field :
                container.getClass().getDeclaredFields()) {
                field.setAccessible(true);
                Object obj = field.get(container);
                if (!(obj instanceof Component))
                    continue;
                String name = field.getName();
                ((Component) obj).setName(name);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Statt auf jede Komponente `setName` aufzurufen, kann einfach die oben dargestellte `setNames`-Methode aufgerufen werden. Die Namen, die dabei vergeben werden, sind identisch mit den Namen der Referenzvariablen, die auf diese Komponenten verweisen.

10.4 Testen von WEB-Anwendungen: Selenium

Die Oberfläche der zu testenden Anwendung präsentiert sich wie folgt:



Und hier der Quellcode des Servlets (der Einfachheit halber wird nur ein nacktes Servlet verwendet – kein JSP, JSF etc.):

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import java.util.Map;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MathServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    public void service(
        final HttpServletRequest request,
        final HttpServletResponse response)
        throws ServletException, IOException {

        System.out.println("\tHeader-Entries:");
        final Enumeration<?> names = request.getHeaderNames();
        while (names.hasMoreElements()) {
            final String name = (String) names.nextElement();
            final String value = request.getHeader(name);
            System.out.println("\t\t" + name + " ==> " + value);
        }
        System.out.println("\tRequest-Parameters:");
        final Map<?,?> parameters = request.getParameterMap();
        for (final Object obj : parameters.entrySet()) {
            final Map.Entry<?,?> e = (Map.Entry<?,?>) obj;
            final String key = (String)e.getKey();
            final String[] values = (String[])e.getValue();
            System.out.println("\t\t" + key + " ==> " + values[0]);
        }
    }
}
```

```

    }

    final boolean doSum = request.getParameter("sum") != null;
    final boolean doDiff = request.getParameter("diff") != null;
    if (doSum || doDiff) {
        int x = 0;
        try {
            x =
Integer.parseInt(request.getParameter("x").trim());
        }
        catch(final Exception e) {
        }
        int y = 0;
        try {
            y =
Integer.parseInt(request.getParameter("y").trim());
        }
        catch(final Exception e) {
        }
        int result;
        if (doSum)
            result = x + y;
        else if (doDiff)
            result = x - y;
        else
            result = 0;
        request.setAttribute("x", x);
        request.setAttribute("y", y);
        request.setAttribute("result", result);
    }
    this.renderPage(request, response);
}

private String stringValueOf(
    final HttpServletRequest request,
    final String name) {
    final Object obj = request.getAttribute(name);
    return obj == null ? "" : obj.toString();
}

private void renderPage(
    final HttpServletRequest request,
    final HttpServletResponse response)
    throws IOException {

    final String x = this.stringValueOf(request, "x");
    final String y = this.stringValueOf(request, "y");
    final String result =
        this.stringValueOf(request, "result");

    response.setContentType("text/html");
    final PrintWriter out = response.getWriter();
    out.println("<html><head><title>math</title></head>");
    out.println("<body>");
    out.println("<form name='f'>");
    out.println("X:<br>");
    out.println("<input type='text' name='x'
        value='" + x + "'/>");
    out.println("<br>");

```

```

        out.println("Y:<br>");
        out.println("<input type='text' id='y' name='y'
            value='" + y + "'/>");
        out.println("<br>");
        out.println("<input type='submit' name='sum'
            value='Summe'/>");
        out.println("<input type='submit' id='diff' name='diff'
            value='Differenz'/>");
        out.println("<br>");
        out.println("Resultat:<br>");
        out.println("<input type='text' id='result' name='result'
            readonly value='" + result + "'/>");
        out.println("</form></body></html>");
    }
}

```

(Man beachte: einige `input`-Elemente haben eine `id`, andere nicht. Das ist für die Selenium-Demo im übernächsten Kapitel interessant...).

Die `web.xml`:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app ...>

    <servlet>
        <servlet-name>MathServlet</servlet-name>
        <servlet-class>servlets.MathServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>MathServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>

```

Zum Bau und zum Deployment der Anwendung kann ant genutzt werden (`build.xml`).

Das im Folgenden genutzte Selenium-Testwerkzeug stellt ein Firefox-Plugin bereit, mittels dessen in einer Selenium-eigenen IDE ein Selenium-Driver direkt mit dem Browser interagieren kann. Die Interaktionen des Anwenders mit einer WEB-Anwendung können vom Selenium-Driver aufgezeichnet werden. Das Ergebnis dieser Aufzeichnung kann um Verifikationen erweitert werden und dann als `html`-Datei gespeichert werden. Aufgrund einer solchen Datei können dann jederzeit die Benutzer-Interaktionen automatisch erneut ausgeführt werden. Weiterhin kann eine Java-Klasse generiert werden, die mittels JUnit ausgeführt werden kann. Die Ausführung dieser Klasse ist semantisch gleichwertig zur Ausführung des Selenium-Skripts in der Selenium-IDE.

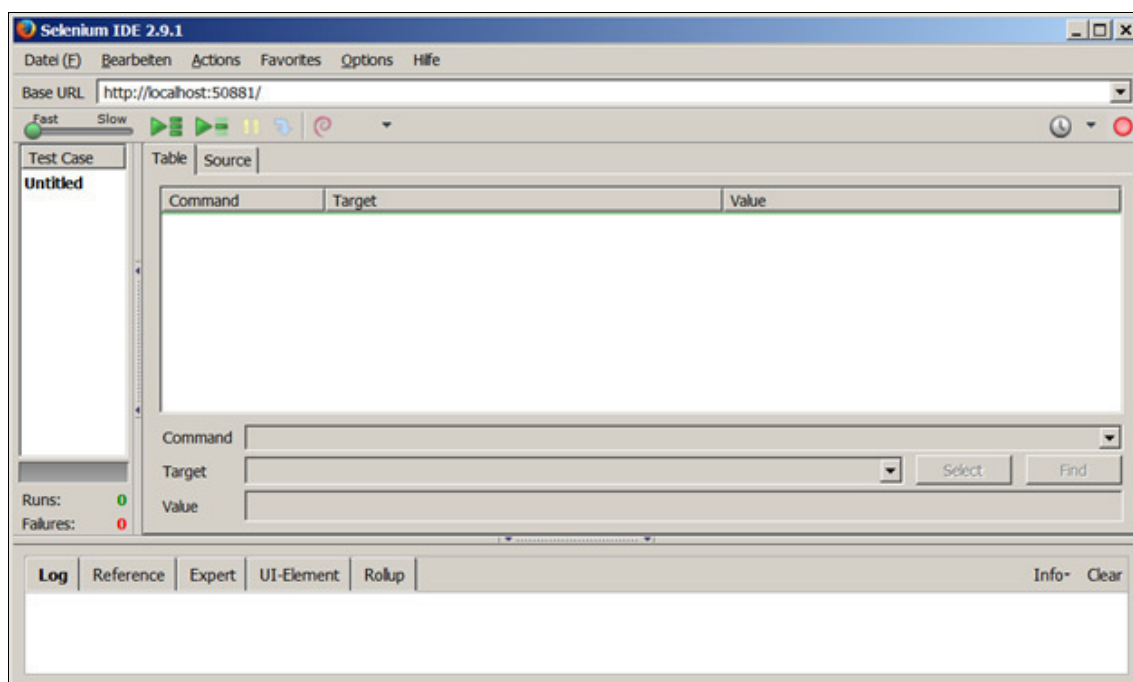
<https://addons.mozilla.org/en-US/firefox/addon/selenium-ide/>

<http://docs.seleniumhq.org/download/>

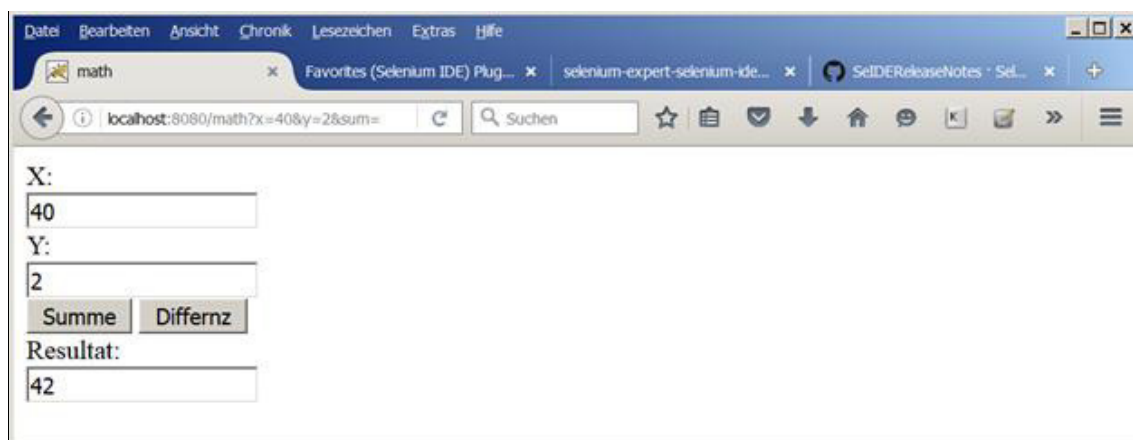
Der erste Download dient zum Installieren des Selenium-Plugins in Firefox. Dieses Plugin enthält bereits die komplette IDE.

Der zweite Download enthält zur Ausführung von JUnit-Tests erforderliche Java-Klassen. Das Ergebnis dieses Downloads liegt bereits entpackt im `dependencies`-Projekt.

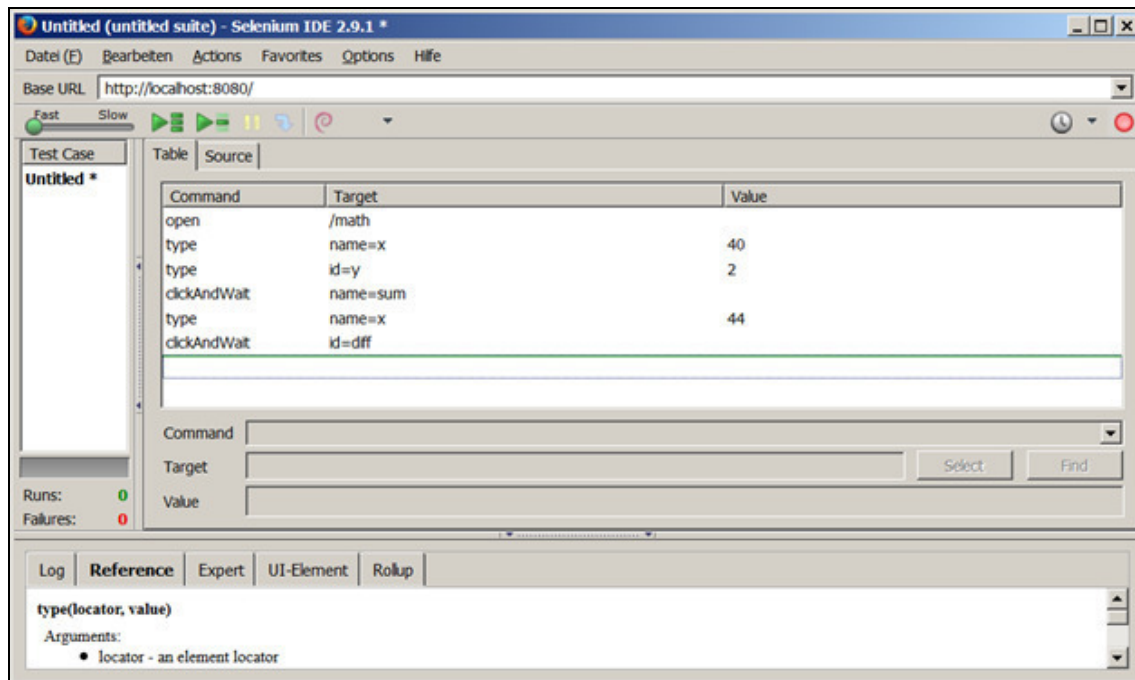
Im Firefox kann nun zunächst die Selenium-IDE gestartet werden: Menu > Web Developer > Selenium IDE. Die IDE befindet sich dann bereits automatisch in der Aufzeichnungs-Phase (erkennbar am roten Knopf oben rechts...):



Dann kann der Benutzer mit der WEB-Anwendung interagieren (z.B. die Summe von 40 und 2 und dann die Differenz von 44 und 2 berechnen...):

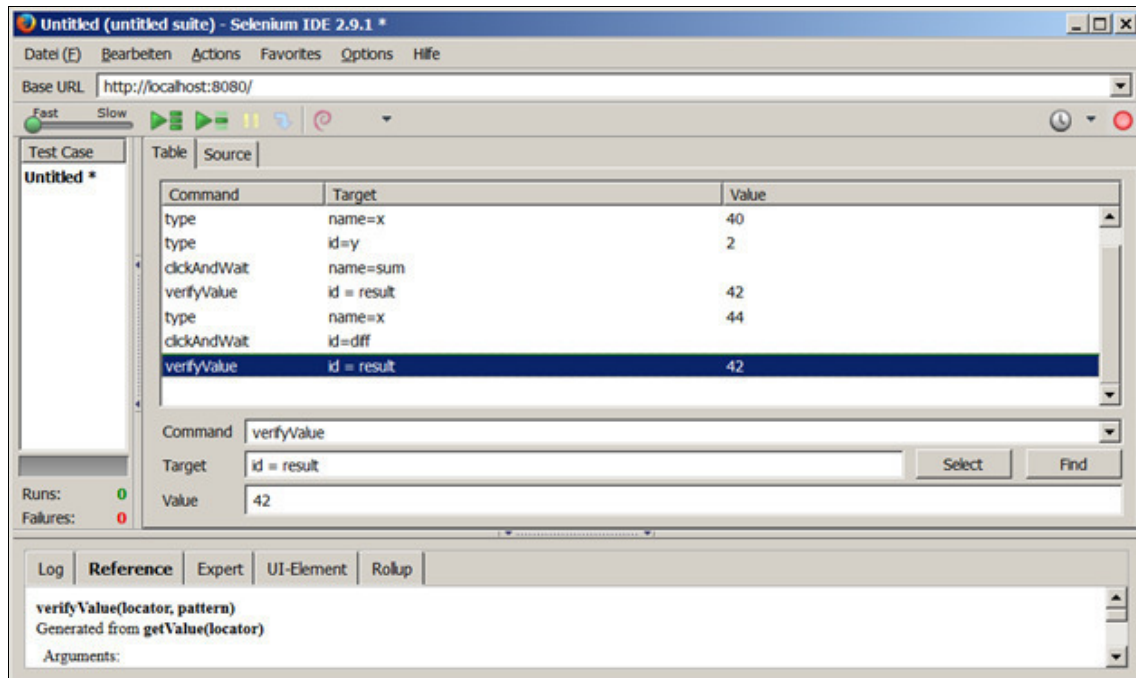


Die Aktionen sind von Selenium aufgezeichnet worden:



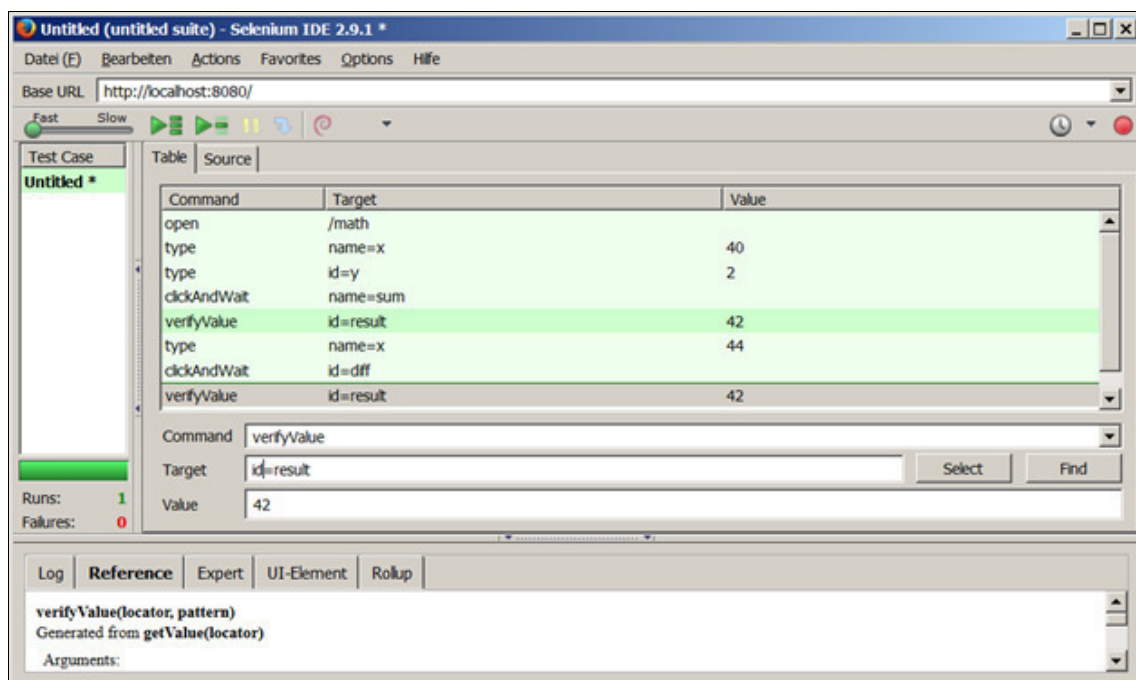
Wie man sieht, hat der Benutzer 40 und 2 eingegeben und dann den "Summe"-Button betätigt. Dann hat der Benutzer in das erste Eingabefeld 44 eingegeben und den "Differenz"-Button betätigt. (Dabei werden Elemente, welche ein `id`-Attribut besitzen, über den Wert eben dieses `id`-Attributs angesprochen; Elemente, welche nur ein `name`-Attribut haben, werden über den Wert dieses `name`-Attributs angesprochen.)

Zu den "Commands" können nun Assertions hinzugefügt werden (eingefügt resp. angefügt werden). Es soll zugesichert werden, dass nach Betätigung des "Summe"-Buttons das Resultat-Feld den Wert 42 hat; und auch nach der anschließenden Betätigung des "Differenz"-Buttons soll dieses Feld den Wert 42 haben. Zu den Commands werden zwei `verifyValue`-Kommandos hinzugefügt:

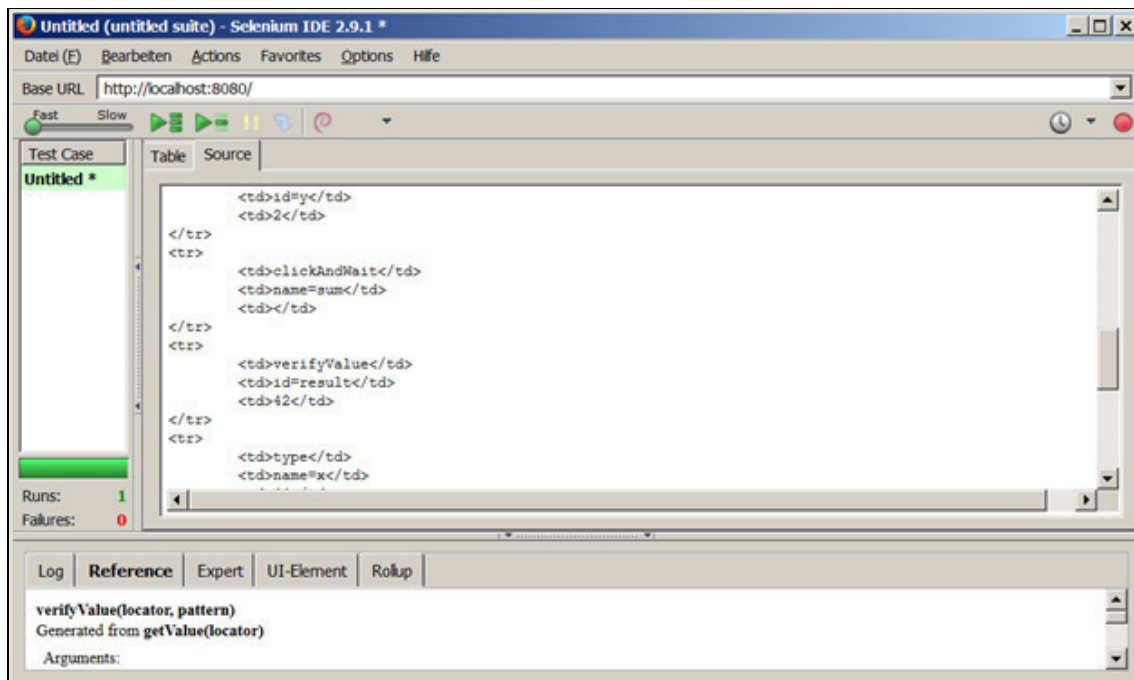


Dann kann ein Replay gestartet werden (über den rechten grünen Pfeil in der Toolbar der IDE). Dabei werden dann natürlich auch die hinzugefügten Assertions geprüft. Hier das Resultat (alles ist natürlich grün – wenn aber die Assertions nicht gehalten hätte, wäre der Replay bei dem ersten Error beendet worden – mit einer roten Anzeige).

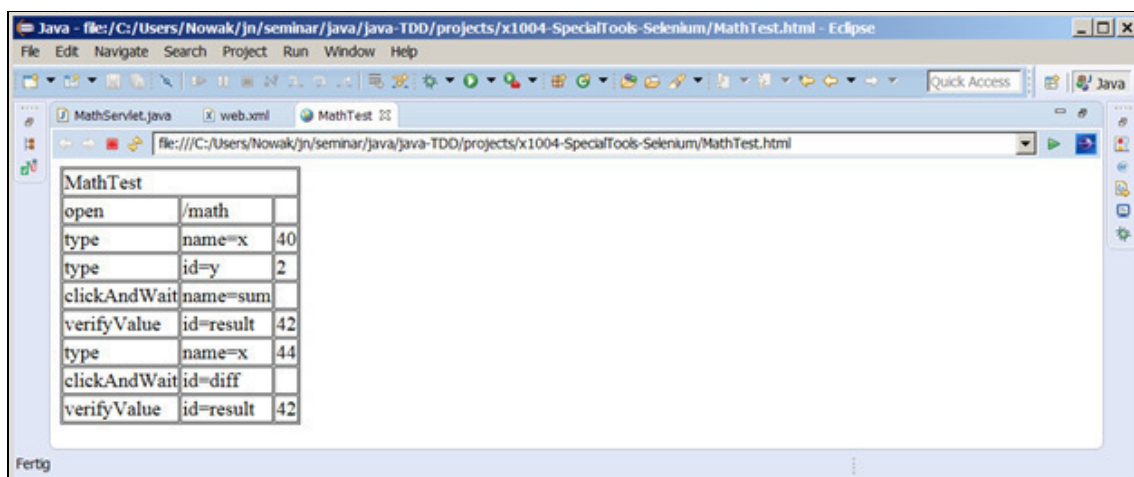
Hier das Resultat des Replays:



Der "Source"-Reiter (neben dem "Table"-Reiter) zeigt den generierten HTML-Text:



Dieser Test kann dann natürlich gespeichert werden (File > Save Test Case). Hier die erzeugte HTML-Datei:



Nach Erweiterungen / Refaktorisierungen des `MathServlets` kann dann diese Datei immer erneut ausgeführt werden.

Die HTML-Datei kann auch transformiert werden in eine Java-Klasse: File > Export Test Case As ... > Java JUnit4 WebDriver. Hier die generierte Java-Klasse:

```
package test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import java.util.concurrent.TimeUnit;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.Alert;
import org.openqa.selenium.By;
import org.openqa.selenium.NoAlertPresentException;
import org.openqa.selenium.NoSuchElementException;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class MathTest {
    private WebDriver driver;
    private String baseUrl;
    private boolean acceptNextAlert = true;
    private final StringBuffer verificationErrors = new
StringBuffer();

    @Before
    public void setUp() throws Exception {
        this.driver = new FirefoxDriver();
        this.baseUrl = "http://localhost:8080/";
        this.driver.manage().timeouts().implicitlyWait(30,
TimeUnit.SECONDS);
    }

    @Test
    public void testMath() throws Exception {
        this.driver.get(this.baseUrl + "/math");
        this.driver.findElement(By.name("x")).clear();
        this.driver.findElement(By.name("x")).sendKeys("40");
        this.driver.findElement(By.id("y")).clear();
        this.driver.findElement(By.id("y")).sendKeys("2");
        this.driver.findElement(By.name("sum")).click();
        try {
            assertEquals("42",
                this.driver.findElement(
                    By.id("result")).getAttribute("value"));
        } catch (final Error e) {
            this.verificationErrors.append(e.toString());
        }
        this.driver.findElement(By.name("x")).clear();
        this.driver.findElement(By.name("x")).sendKeys("44");
        this.driver.findElement(By.id("diff")).click();
        try {
            assertEquals("42",
                this.driver.findElement(
                    By.id("result")).getAttribute("value"));
        } catch (final Error e) {
```

```

        this.verificationErrors.append(e.toString());
    }
}

@After
public void tearDown() throws Exception {
    this.driver.quit();
    final String verificationErrorString =
        this.verificationErrors.toString();
    if (!"".equals(verificationErrorString)) {
        fail(verificationErrorString);
    }
}

private boolean isElementPresent(final By by) {
    try {
        this.driver.findElement(by);
        return true;
    } catch (final NoSuchElementException e) {
        return false;
    }
}

private boolean isAlertPresent() {
    try {
        this.driver.switchTo().alert();
        return true;
    } catch (final NoAlertPresentException e) {
        return false;
    }
}

private String closeAlertAndGetItsText() {
    try {
        final Alert alert = this.driver.switchTo().alert();
        final String alertText = alert.getText();
        if (this.acceptNextAlert) {
            alert.accept();
        } else {
            alert.dismiss();
        }
        return alertText;
    } finally {
        this.acceptNextAlert = true;
    }
}
}

```

Und diese Klasse kann nun einfach per JUnit ausgeführt werden – also ganz ohne Selenium-IDE...

(Natürlich hätte man diese Java-Klasse auch manuell erstellen können. Aber es ist natürlich einfacher, eine User-Session automatisch aufzeichnen zu lassen, die Aufzeichnungen durch Assertions zu bereichern und dann die entsprechende Java-Klasse automatisch generieren zu lassen.)

10.5 Testen von EJB-Anwendungen: OpenEJB

OpenEJB (<http://openejb.apache.org>) ist ein einfacher, kleiner EJB-Container. Der Container kann - wie übliche EJB-Container - als separate VM laufen. Dann ist ein Deployment der Anwendungen erforderlich. OpenEJB kann aber auch lokal laufen - in derselben VM, in welcher auch die (Test-) Anwendung läuft. Ein Deployment ist dann nicht erforderlich. Der Testaufwand wird dadurch natürlich erheblich vereinfacht.

Im Folgenden wird eine Klasse getestet, welche einige Operationen zur Manipulation einer Konto-Tabelle enthält.

Als Datenbank wird eine In-Memory-HSQLDB-Datenbank verwendet.

Zum Zwecke des Testens benötigen wir zunächst eine Möglichkeit, via EJB irgendein beliebiges SQL-Kommando abzusetzen. Hierzu verwenden wir das Interface `Db`:

```
package common;

import javax.ejb.Remote;

@Remote
public interface Db {
    public abstract void execute(String sql) throws Exception;
}
```

Die eigentliche Funktionalität der Anwendung ist in folgendem Interface spezifiziert:

```
package common;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface AccountDao {
    public abstract void create(Account account) throws Exception;
    public abstract void update(Account account) throws Exception;
    public abstract void delete(Account account) throws Exception;
    public abstract Account findByNumber(String number);
    public abstract List<Account> findAll();
}
```

Die Klasse `Account` ist eine einfache Bean-Klasse, die wie folgt implementiert ist:

```
package common;

import java.io.Serializable;

public class Account implements Serializable {
    private static final long serialVersionUID = 1L;

    private String number;
    private int balance;

    public Account() {
    }

    public Account(final String number) {
        this.setNumber(number);
    }

    public String getNumber() {
        return this.number;
    }

    public void setNumber(final String number) {
        Objects.requireNonNull(number);
        this.number = number;
    }

    public int getBalance() {
        return this.balance;
    }

    public void setBalance(final int balance) {
        this.balance = balance;
    }

    @Override
    public String toString() { ... }

    @Override
    public int hashCode() {
        return this.number.hashCode();
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj)
            return true;
        if (obj == null || this.getClass() != obj.getClass())
            return false;
        final Account other = (Account) obj;
        return this.number.equals(other.number);
    }
}
```

Nun zu den Implementierungen.

Zunächst zur Implementierung des Db-Interfaces:

```
package server;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

import javax.annotation.Resource;
import javax.ejb.EJBException;
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.sql.DataSource;

import common.Db;

@Stateless(name = "DbEJB")
@TransactionManagement(TransactionManagementType.CONTAINER)

public class DbImpl implements Db {

    // See: openejb/conf/openejb.xml:
    // <Resource id="My DataSource" ...

    @Resource(name = "bank", type = javax.sql.DataSource.class)
    private DataSource dataSource;

    @Override
    public void execute(final String sql) {
        try(final Connection con = this.dataSource.getConnection()) {
            try (final Statement stmt = con.createStatement()) {
                stmt.execute(sql);
            }
        }
        catch (final SQLException e) {
            throw new EJBException(e);
        }
    }
}
```

DbImpl ist eine Stateless Session Bean. Das Transactions-Management wird dem Container überlassen. Der Container wird eine Resource injizieren: die DataSource (die dann in der execute-Methode genutzt wird).

Hier die Implementierung des DAO-Interfaces:

```
package server;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import javax.annotation.Resource;
import javax.ejb.EJBException;
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.sql.DataSource;

import common.Account;
import common.AccountDao;

@Stateless(name = "AccountDaoEJB")
@TransactionManagement(TransactionManagementType.CONTAINER)

public class AccountDaoImpl implements AccountDao {

    @Resource(name = "bank", type = javax.sql.DataSource.class)
    private DataSource dataSource;

    @Override
    public void create(final Account account) throws Exception {
        try (Connection con = this.dataSource.getConnection()) {
            final String sql =
                "INSERT INTO ACCOUNT (NUMBER, BALANCE) VALUES (?, ?)";
            try (PreparedStatement ps = con.prepareStatement(sql)) {
                ps.setString(1, account.getNumber());
                ps.setInt(2, account.getBalance());
                if (ps.executeUpdate() != 1)
                    throw new EJBException("error create " + account);
            }
        }
        catch (final SQLException e) {
            if (e.getErrorCode() == -104) // HSQLDB !!
                throw new Exception("duplicate key error");
            throw new EJBException(e);
        }
    }

    @Override
    public void update(final Account account) throws Exception {
        try (Connection con = this.dataSource.getConnection()) {
            final String sql =
                "UPDATE ACCOUNT SET BALANCE = ? WHERE NUMBER = ?";
            try (PreparedStatement ps = con.prepareStatement(sql)) {
                ps.setInt(1, account.getBalance());
                ps.setString(2, account.getNumber());
                if (ps.executeUpdate() != 1)
                    throw new Exception("error update " + account);
            }
        }
    }
}
```



```

    }
    catch (final SQLException e) {
        throw new EJBException(e);
    }
}

@Override
public void delete(final Account account) throws Exception {
    try (Connection con = this.dataSource.getConnection()) {
        final String sql =
            "DELETE FROM ACCOUNT WHERE NUMBER = ?";
        try (PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setString(1, account.getNumber());
            if (ps.executeUpdate() != 1)
                throw new Exception("error delete " + account);
        }
    }
    catch (final SQLException e) {
        throw new EJBException(e);
    }
}

@Override
public Account findByNumber(final String number) {
    try (Connection con = this.dataSource.getConnection()) {
        final String sql =
            "SELECT NUMBER, BALANCE FROM ACCOUNT WHERE NUMBER =
?";

        try (PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setString(1, number);
            try (ResultSet rs = ps.executeQuery()) {
                if (!rs.next())
                    return null;
                return this.createAccount(rs);
            }
        }
    }
    catch (final SQLException e) {
        throw new EJBException(e);
    }
}

@Override
public List<Account> findAll() {
    try (Connection con = this.dataSource.getConnection()) {
        final List<Account> accountList = new
ArrayList<Account>();
        final String sql =
            "SELECT NUMBER, BALANCE FROM ACCOUNT";
        try (PreparedStatement ps = con.prepareStatement(sql)) {
            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next())
                    accountList.add(this.createAccount(rs));
            }
            return accountList;
        }
    }
    catch (final SQLException e) {
        throw new EJBException(e);
    }
}

```

```

    }
}

private Account createAccount(final ResultSet rs) throws
SQLException {
    final Account account = new Account();
    account.setNumber(rs.getString("NUMBER"));
    account.setBalance(rs.getInt("BALANCE"));
    return account;
}
}

```

Auch `AccountDaoImpl` ist eine Stateless Session Bean mit Container managed Transactions. Auch hier wird der Container die `DataSource` injizieren.

Hier der JUnit-Test:

```

package test;

// ...

import java.util.List;
import java.util.Properties;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NameClassPair;
import javax.naming.NamingEnumeration;

import common.Account;
import common.AccountDao;
import common.Db;

public class AccountDaoTest {

    private InitialContext initialContext;

    @Before
    public void before() throws Exception {
        final Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.apache.openejb.client.LocalInitialContextFactory");
        properties.put("bank", "new://Resource?type=DataSource");
        properties.put("bank.JdbcDriver", "org.hsqldb.jdbcDriver");
        properties.put("bank.JdbcUrl", "jdbc:hsqldb:mem:bank");
        this.initialContext = new InitialContext(properties);
        this.list(this.initialContext);
        final Db db = (Db)this.initialContext.lookup("DbEJBRemote");
        db.execute("create table account(" +
            " number varchar(10), balance integer, " +
            "primary key(number))");
    }

    @After
    public void after() throws Exception {
        final Db db = (Db)this.initialContext.lookup("DbEJBRemote");
        db.execute("drop table account");
    }
}

```

```
@Test
public void testCreateFindByNumber() throws Exception {
    final AccountDao dao = (AccountDao)
        this.initialContext.lookup("AccountDaoEJBRemote");
    dao.create(new Account("4711"));
    final Account a1 = dao.findByNumber("4711");
    assertNotNull(a1);
    final Account a2 = dao.findByNumber("4712");
    assertNull(a2);
}

@Test
public void testCreateFindAll() throws Exception {
    final AccountDao dao = (AccountDao)
        this.initialContext.lookup("AccountDaoEJBRemote");
    dao.create(new Account("4711"));
    dao.create(new Account("4712"));
    final List<Account> list = dao.findAll();
    assertNotNull(list);
    assertEquals(2, list.size());
    assertTrue(list.contains(new Account("4711")));
    assertTrue(list.contains(new Account("4712")));
    assertFalse(list.contains(new Account("4713")));
}

private void list(final Context ctx) throws Exception {
    final NamingEnumeration<NameClassPair> e = ctx.list("");
    while (e.hasMore()) {
        final NameClassPair p = e.next();
        System.out.println("\t" + p);
    }
}
}
```

10.6 Testen mit FIT

FIT ist im Gegensatz zu JUnit ein Framework für Integrations- oder Akzeptanz-Tests.

Der Auftraggeber kann mit FIT auf einfache Weise Testfälle formulieren – und zwar in Form einer HTML-Tabelle (die etwa mit WinWord oder Excel erzeugt werden kann).

Diese Tabellen enthalten Eingaben und die erwarteten Ausgaben. FIT stellt nun einen Prozessor (einen `FileRunner`) zur Verfügung, der diese Tabellen liest und Ausgaben erzeugt – wiederum in Form von Tabellen. Die Ausgabetabellen haben im Prinzip immer dasselbe Aussehen wie die Eingabetabellen – allerdings wird dort vermerkt, ob der Test jeweils positiv (grün!) oder negativ (rot!) verlaufen ist.

Der Auftraggeber muss also von Java nichts wissen (er weiß von Java nichts!). Aber der Entwickler muss sich natürlich darum kümmern, die Struktur der vom Auftraggeber auszufüllenden Tabellen auf Java-Klassen abzubilden, welche ihrerseits dann an die eigentlich zu testenden Klassen delegieren. Er muss sog. Fixtures schreiben.

Dabei bietet das FIT-Framework bereits einige vorgefertigte Fixture-Klassen an: `ColumnFixture`, `ActionFixture` und `RowFixture` (und einige weitere) – allesamt abgeleitet von der Basisklasse `Fixture`. Auf diesen Klassen kann der Entwickler aufbauen.

Das FIT-Framework ist ein sehr kleines, aber elegantes Werkzeug. Es kann auf einfache Weise erweitert werden, um somit auch Tests ausführen zu können, an welche der Entwickler des Frameworks (Ward Cunningham) nicht gedacht hat.

Im Folgenden werden zwei Klassen getestet: eine `MathService`- und eine `Calculator`-Klasse.

Die zu testenden Klassen sind wie folgt implementiert (im Package `cut` – Class Under Test):

```
package cut;

public class MathService {

    public int gcd(int x, int y) {
        if (x <= 0 || y <= 0)
            throw new RuntimeException();
        int a = x;
        int b = y;
        while (a == b) {
            if (a > b)
                a -= b;
            else
                b -= a;
        }
        return a;
    }

    public int lcm(int x, int y) {
        int a = x;
        int b = y;
        while (a != b) {
            if (a < b)
                a += x;
            else
                b += y;
        }
        return a;
    }
}
```

```
package cut;

public class Calculator {
    private int value;
    private int count;
    public void add(int v) {
        value += v;
        count++;
    }
    public void subtract(int v) {
        value -= v;
        count++;
    }
    public int getValue() {
        return this.value;
    }
    public int getCount() {
        return this.count;
    }
}
```

`MathService` ist stateless, `Calculator` ist statefull.

Die `gcd`-Methode von `MathService` enthält einen Fehler (der in einem der folgenden Tests erkannt werden wird...)

Um die `MathService`-Klasse zu testen, erstellt der Anwender folgende HTML-Tabelle (man könnte alternativ auch z.B. eine Excel-Tabelle nutzen):

```
<html>

<body>

Test fuer den Groessten Gemeinamen Teiler
<p></p>
<table border cellspacing="0" cellpadding="6">
  <tr> <td colspan="3">fixtures.GcdColumnFixture</td> </tr>

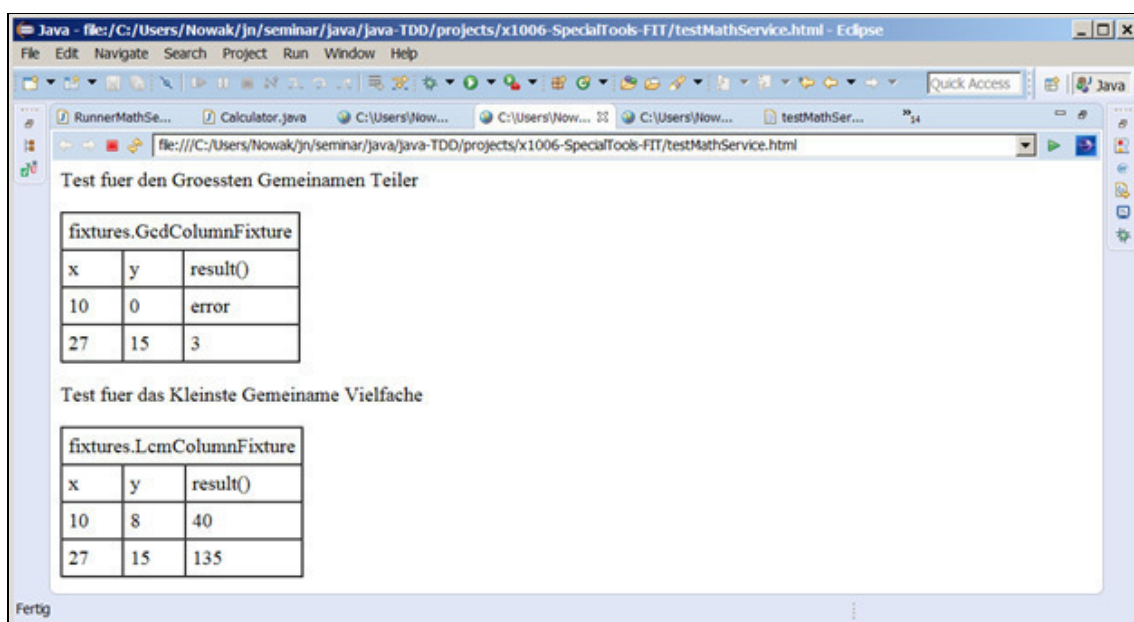
  <tr> <td>x</td> <td>y</td> <td>result()</td> </tr>

  <tr> <td>10</td> <td>0</td> <td>error</td> </tr>
  <tr> <td>27</td> <td>15</td> <td>3</td> </tr>
</table>
<p></p>
Test fuer das Kleinste Gemeiname Vielfache
<p></p>
<table border cellspacing="0" cellpadding="6">
  <tr> <td colspan="3">fixtures.LcmColumnFixture</td> </tr>

  <tr> <td>x</td> <td>y</td> <td>result()</td> </tr>

  <tr> <td>10</td> <td>8</td> <td>40</td> </tr>
  <tr> <td>27</td> <td>15</td> <td>135</td> </tr>
</table>
</body>
</html>
```

Hier die Darstellung des HTML-Dokuments im Browser:



Der Anwender hat offensichtlich Testfälle erzeugt, in denen er die erwarteten Resultate hinterlegt hat. Dabei bezieht er sich offenbar auf zwei Klassen: `fixtures.GcdColumnFixture` und `fixtures.LcmColumnFixture`. Die Spalten sind überschrieben mit `x`, `y` und `result()`.

Was hat es mit diesen Klassen und den Überschriften auf sich?

Die `Fixture`-Klassen bilden die Tabellen-Inhalte, die der Benutzer spezifiziert hat, auf die entsprechende Methode der zu testenden Klassen ab. Man könnte sagen: die `Fixture`-Klassen sind Adapter.

Hier die `GcdColumnFixture`-Klasse:

```
package fixtures;

import cut.MathService;
import fit.ColumnFixture;

public class GcdColumnFixture extends ColumnFixture {
    private final MathService mathService = new MathService();
    public int x;
    public int y;
    public int result() {
        return this.mathService.gcd(this.x, this.y);
    }
}
```

Die Klasse ist von der FIT-Klasse `ColumnFixture` abgeleitet.

Wenn der FIT-Runner nun die erste der beiden obigen Tabellen interpretiert, wird folgendes passieren:

Es wird ein Objekt vom Typ `GcdColumnFixture` erzeugt. Dieses erzeugt seinerseits ein `MathService`-Objekt.

Dann geht FIT zur ersten "Datenzeile" über. Der Wert der `x`-Spalte (10) wird dem `x`-Attribut des gerade erzeugten `GcdColumnFixture`-Objekts zugewiesen. Der Wert der `y`-Spalte (0) wird dem `y`-Attribut dieses Objekts zugewiesen. Dann wird die `result()`-Methode auf dieses `Fixture`-Objekt aufgerufen. Diese ruft dann die eigentlich zu testende `MathService`-Methode auf – und übergibt ihr genau diejenigen Werte, die zuvor an ihre `x`- und `y`-Attribute zugewiesen wurden. Die `gcd`-Methode wirft eine `Exception` – und da in der Tabelle per "error"-Eintrag genau das erwartet wurde, ist für FIT der Test erfolgreich. Bei der Abarbeitung der zweiten Tabellenzeile wird `result()` natürlich genau das erwartete Resultat (3) zurückliefert – also ist für FIT auch der zweite Test erfolgreich.

Entsprechendes gilt dann natürlich auch für die Ausführung der zweiten Tabelle. Hier ist die folgende `Fixture`-Klasse definiert:

```
package fixtures;

import cut.MathService;
import fit.ColumnFixture;

public class LcmColumnFixture extends ColumnFixture {
    private final MathService mathService = new MathService();
    public int x;
    public int y;

    public int result() {
        return this.mathService.lcm(this.x, this.y);
    }
}
```

Um den Test auszuführen, benutzen wir eine kleine Helper-Klasse:

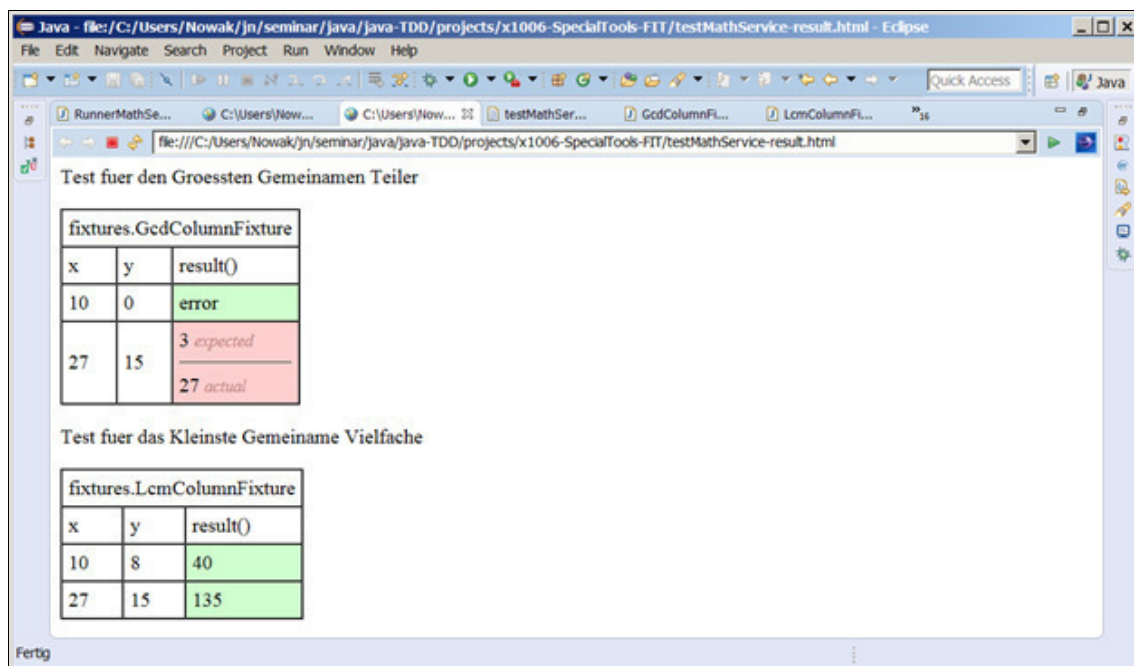
```
package runner;

import fit.FileRunner;

public class RunnerMathService {
    public static void main(final String[] args) {
        FileRunner.main(new String[] {
            "testMathService.html", "testMathService-result.html"
        });
    }
}
```

Die `main`-Methode ruft die `main`-Methode des FIT-Runners auf und übergibt ihr dabei zwei Dateinamen: der erste Name ist der Name der Eingabedatei, der zweite Name bezeichnet die vom Runner zu erzeugende Ausgabedatei.

Nach der Ausführung dieses Runners enthält die Ausgabedatei folgendes Resultat:

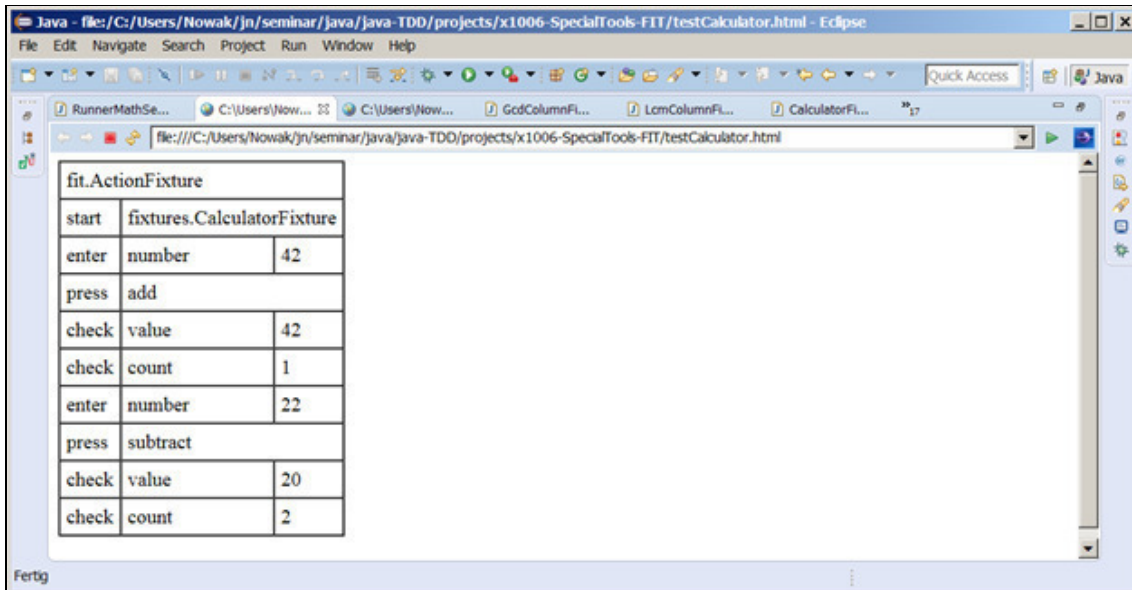


Für jede Tabelle der Eingabedatei ist also eine entsprechende Tabelle in der Ausgabedatei erzeugt wurde. Die grünen Zellen zeigen an, dass die entsprechenden Resultate (`result()`) den Erwartungen entsprechen haben; rote Zellen zeigen an, dass ein Fehler erkannt wurde.

Natürlich sind auch in der Ausgabe die "Überschriften" der Eingabe eingeblendet (sind im obigen Schaubild nur gelöscht worden). Die Texte, die in der Eingabe außerhalb von Tabellen formuliert sind, werden also 1:1 in die Ausgabe übernommen.

Der Fehler in der `gcd`-Methode konnte somit nachgewiesen werden...

Für den Test der `Calculator`-Klasse spezifiziert der Anwender folgenden Test:



fit.ActionFixture		
start	fixtures.CalculatorFixture	
enter	number	42
press	add	
check	value	42
check	count	1
enter	number	22
press	subtract	
check	value	20
check	count	2

In der Tabelle wird ein "Dialog" nachgespielt – ein Dialog mit einem Eingabefeld (`number`), zwei Buttons (`add`, `subtract`) und zwei Ausgabefeldern (`value`, `count`):

Der Benutzer gibt 42 ein (`enter`) und betätigt (`press`) dann den `add`-Button. Dann müssen in den beiden Ausgabefeldern die Werte 42 (`value`) und 1 (`count`) erscheinen. Er gibt 22 ein und betätigt `subtract` – dann müssen die beiden Ausgabefelder die Werte 20 (`value`) und 2 (`count`) beinhalten.

Man beachte, dass als Fixture die FIT-Klasse `ActionFixture` verwendet wird. Die "eigene" Fixture-Klasse (`CalculatorFixture`) wird in der `start`-Zeile angegeben. (Über eine weitere mit `start` markierten Zeile könnte also das benutzte Fixture gewechselt werden.)

Die Begriffe der ersten Spalte (`start`, `enter`, `press` und `check`) sind "Schlüsselwörter" eines `ActionFixture`s. Dabei muss das dem Schlüsselwort `enter` folgende Wort (hier: `number`) in dem zu schreibenden Fixture (`CalculatorFixture`) auf eine Methode abgebildet werden, welche genau einen einzigen Parameter besitzt. Das Wort `press` muss auf eine parameterlose Methode abgebildet werden. Und `check` muss ebenfalls auf eine parameterlose Methode abgebildet werden.

Die CalculatorFixture-Klasse sieht wie folgt aus:

```
package fixtures;

import cut.Calculator;
import fit.Fixture;

public class CalculatorFixture extends Fixture {

    private final Calculator calculator = new Calculator();
    private int number;

    public void number(int number) {
        this.number = number;
    }

    public void add() {
        this.calculator.add(this.number);
    }

    public void subtract() {
        this.calculator.subtract(this.number);
    }

    public int value() {
        return this.calculator.getValue();
    }

    public int count() {
        return this.calculator.getCount();
    }

}
```

CalculatorFixture ist von der FIT-Klasse Fixture abgeleitet. Es sollte klar sein, wie die Zeilen der Eingabe-Tabelle auf Aufrufe der CalculatorFixture-Methode abgebildet werden.

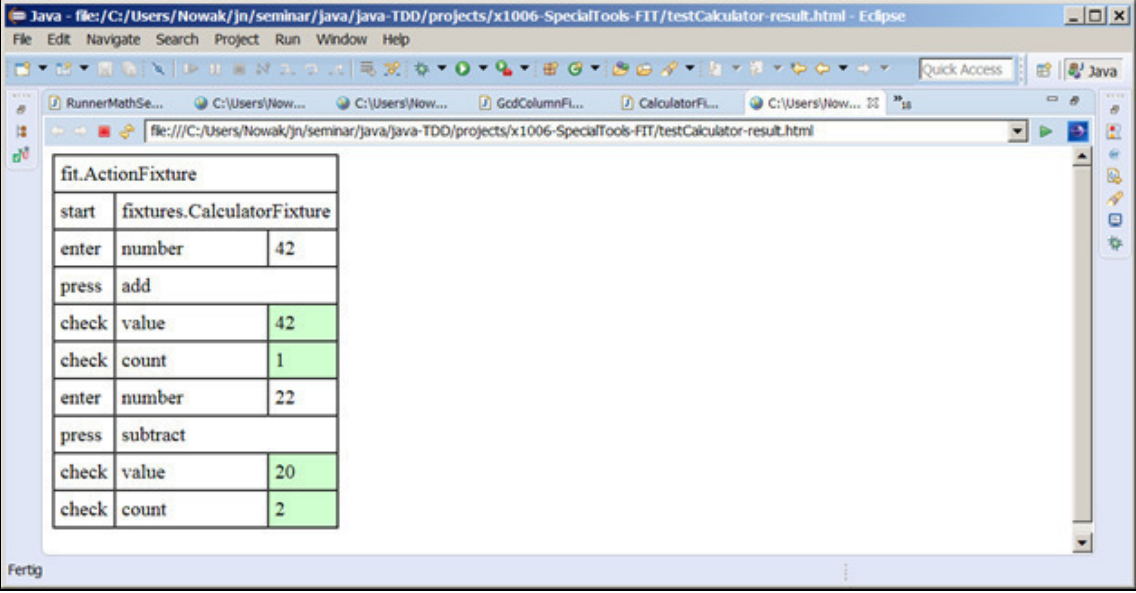
Der Test kann mittels folgender Klasse ausgeführt werden:

```
package runner;

import fit.FileRunner;

public class RunnerCalculator {
    public static void main(final String[] args) {
        FileRunner.main(new String[] {
            "testCalculator.html", "testCalculator-result.html"
        });
    }
}
```

Hier die erzeugte Ausgabedatei:



fit.ActionFixture		
start	fixtures.CalculatorFixture	
enter	number	42
press	add	
check	value	42
check	count	1
enter	number	22
press	subtract	
check	value	20
check	count	2

Fertig

11

Testen von Multithreading-Anwendungen

11.1	Utility-Klassen.....	11-4
11.2	Queue.....	11-6
11.2.1	FixedQueue	11-6
11.2.2	FixedQueueTest	11-7
11.2.3	Future	11-13
11.2.4	FutureImpl.....	11-13
11.2.5	FutureTest	11-14
11.3	Executor	11-15
11.3.1	ExecutorTest.....	11-17
11.4	Nodes	11-20
11.4.1	Node	11-20
11.4.2	NodeTest	11-23

11 Testen von Multithreading-Anwendungen

Im Folgenden wird ein "Node"-System getestet. Das Node-System erlaubt die Spezifikation (und Ausführung) von sequentiellen und parallelen Berechnungsschritten. (In den Grundlagen – aber eben nur in den Grundlagen – ähnelt es dem `CompletableFuture`-Konzept von Java 8.)

Dazu werden eine Reihe von Hilfsklassen entwickelt und getestet – Hilfsklassen, die man natürlich in ähnlicher Form der Standardbibliothek hätte entnehmen können: `FixedQueue`, `Future` und `Executer`. Aber wir wollen ja testen...

11.1 Utility-Klassen

Alle zu diesem Kapitel gehörigen Projekte benutzen zwei zentrale Hilfsklassen.

Die Klasse `Threading` erlaubt "kürzere" Aufrufe blockierender Thread-bezogener Methoden. Die Methoden fangen die `InterruptedExceptions` ab, die bei solchen Aufrufen auftreten können, und wickeln sie in `RuntimeExceptions` ein:

```
package util;

public class Threading {

    public static void sleep(final int millis) {
        try {
            Thread.sleep(millis);
        }
        catch (final InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void sleepRandom(final int maxMillis) {
        try {
            Thread.sleep((int) (Math.random() * maxMillis));
        }
        catch (final InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void wait(final Object monitor) {
        try {
            monitor.wait();
        }
        catch (final InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void join(final Thread thread) {
        try {
            thread.join();
        }
        catch (final InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Und mache der folgenden Tests benötigen eine kleine Stoppuhr – sie benutzen die Klasse `Stopwatch`:

```
package util;

public class Stopwatch {

    private long start;
    private long stop;

    public Stopwatch() {
        this.start();
    }

    public void start() {
        this.start = System.nanoTime();
    }

    public void stop() {
        this.stop = System.nanoTime();
    }

    public long elapsedMilliseconds() {
        return (this.stop - this.start) / 1_000_000;
    }
}
```

11.2 Queue

Queue ist ein Interface für Queue-Klassen. Mittels `enqueue` wird ein Objekt in die Queue eingestellt; mittels `dequeue` wird ihr ein Element entnommen. `size()` liefert die Anzahl der momentan in der Queue befindlichen Objekte. Die Elemente der Queue sind vom Typ `T`:

```
package util;

public interface Queue<T> {
    public abstract void enqueue(T element);
    public abstract T dequeue();
    public abstract int size();
}
```

11.2.1 FixedQueue

Eine `FixedQueue` ist eine `Queue` mit begrenzter Kapazität (die über den Konstruktor vorgegeben wird).

Sowohl `enqueue` als auch `dequeue` können blockieren: `enqueue` blockiert, wenn die Queue voll ist; `dequeue` blockiert, wenn die Queue leer ist.

Die Elemente einer `FixedQueue` werden in einer `LinkedList` enthalten (die FIFO-Operationen sind dann recht performant).

```
package util;

import java.util.LinkedList;

public class FixedQueue<T> implements Queue<T> {

    private final LinkedList<T> elements = new LinkedList<>();
    private final int capacity;

    public FixedQueue(final int capacity) {
        this.capacity = capacity;
    }

    @Override
    public void enqueue(final T element) {
        synchronized (this.elements) {
            while (this.elements.size() == this.capacity)
                Threading.wait(this.elements);
            this.elements.add(element);
            this.elements.notifyAll();
        }
    }

    @Override
    public T dequeue() {
        synchronized (this.elements) {
            while (this.elements.size() == 0)
                Threading.wait(this.elements);
            final T element = this.elements.removeFirst();
            this.elements.notifyAll();
            return element;
        }
    }
}
```

```
    }  
}  
  
@Override  
public int size() {  
    return this.elements.size();  
}  
}
```

Wie man sieht, ist die Queue dafür ausgelegt, dass n Schreiber gleichzeitig schreiben und m Leser gleichzeitig lesen können (`while...`, `notifyAll`).

11.2.2 FixedQueueTest

Im folgenden ersten Test greift nur ein einziger Thread auf eine `FixedQueue` zu. Nachdem die Queue gefüllt wurde, wird jeweils ein Element entnommen – um dann wieder ein weiteres Element einstellen zu können. Am Ende werden alle noch vorhandenen Elemente entnommen. Hier wird `enqueue` und `dequeue` niemals blockieren.

```
@Test  
public void testSingleThreadedNonBlocking() {  
    final Queue<String> queue = new FixedQueue<>(2);  
    Assert.assertEquals(0, queue.size());  
    queue.enqueue("spring");  
    Assert.assertEquals(1, queue.size());  
    queue.enqueue("summer");  
    Assert.assertEquals(2, queue.size());  
    Assert.assertEquals("spring", queue.dequeue());  
    Assert.assertEquals(1, queue.size());  
    queue.enqueue("autumn");  
    Assert.assertEquals(2, queue.size());  
    Assert.assertEquals("summer", queue.dequeue());  
    Assert.assertEquals(1, queue.size());  
    queue.enqueue("winter");  
    Assert.assertEquals(2, queue.size());  
    Assert.assertEquals("autumn", queue.dequeue());  
    Assert.assertEquals(1, queue.size());  
    Assert.assertEquals("winter", queue.dequeue());  
    Assert.assertEquals(0, queue.size());  
}
```

Im folgenden Test wird gezeigt, dass `dequeue` blockiert. `dequeue` wird in einem neuen Thread auf eine leere Queue aufgerufen. Der Hauptthread wartet eine kleine Zeit und ruft dann `Interrupt` auf den abgespaltenen Thread auf. Dies führt dazu, dass die `wait`-Operation, die in `dequeue` aufgerufen wurde, eine Exception wirft – `dequeue` wird auf jeden Fall nicht auf natürliche Weise zurückkehren.

```
@Test
public void testDequeueBlocking() {
    final Queue<String> queue = new FixedQueue<>(2);
    final AtomicBoolean running = new AtomicBoolean();
    final Thread t = new Thread(() -> {
        running.set(true);
        XAssert.assertThrows(Exception.class,
            () -> queue.dequeue());
    });
    t.start();
    Threading.sleep(500);
    t.interrupt();
    Assert.assertTrue(running.get());
}
```

Im folgenden Test wird gezeigt, dass `enqueue` blockiert. Der Hauptthread füllt die Queue (so dass sie voll ist). Ein abgespalten Thread versucht, ein weiteres Element per `enqueue` hinzuzufügen. Es wird gezeigt, dass diese `enqueue`-Operation blockiert (`enqueue` ist nach 500ms noch nicht zurückgekehrt):

```
@Test
public void testEnqueueBlocking() {
    final Queue<String> queue = new FixedQueue<>(2);
    queue.enqueue("red");
    queue.enqueue("green");
    final AtomicBoolean running = new AtomicBoolean();
    final Thread t = new Thread(() -> {
        running.set(true);
        XAssert.assertThrows(Exception.class,
            () -> queue.enqueue("blue"));
    });
    t.start();
    Threading.sleep(500);
    t.interrupt();
    Assert.assertTrue(running.get());
}
```

Im Folgenden werden zwei Threads abgespalten – ein Producer-Thread und ein Consumer-Thread. Der Producer-Thread schreibt in eine Queue (mit der Kapazität 2), der Consumer-Thread liest.

Eine `send`-Liste enthält die Strings, die der Producer schreiben wird; eine `received`-Liste wird die Werte enthalten, die der Consumer lesen wird. Abschließend können beide Listen miteinander verglichen werden – wenn alles mit rechten Dingen zugeht, sind die Listen gleich.

Der Produzent benötigt zwischen 0ms und 200ms Zeit, um jeweils einen Wert zu schreiben (random); auch der Consumer benötigt zwischen 0ms und 200ms. Der Producer-Thread wird verlassen, wenn alle Element der send-Liste geschrieben wurden; der Consumer Thread terminiert dann, wenn er ein null-Element liest.

Der Hauptthread startet Consumer und Producer und wartet dann auf das Ende des Producers. Dann schreibt er eine null in die Queue, damit auch der Consumer-Thread terminieren wird (worauf dann gewartet wird).

```
@Test
public void testOneProducerOneConsumer() throws Exception {
    final int max = 200;

    final Queue<String> queue = new FixedQueue<>(2);

    final List<String> send = new ArrayList<>();
    for (int i = 0; i < 10; i++)
        send.add(String.valueOf(i));

    final List<String> received = new ArrayList<>();

    final Thread consumer = new Thread(() -> {
        String value = queue.dequeue();
        while (value != null) {
            Threading.sleepRandom(max);
            received.add(value);
            value = queue.dequeue();
        }
    });

    final Thread producer = new Thread(() -> {
        for (final String element : send) {
            Threading.sleepRandom(max);
            queue.enqueue(element);
        }
    });

    consumer.start();
    producer.start();
    producer.join();
    queue.enqueue(null);
    consumer.join();

    Assert.assertEquals(send, received);
    Assert.assertEquals(0, queue.size());
}
```

Im Folgenden werden zwei Producer und ein Consumer verwendet. Der erste Producer schreibt die Elemente einer send1-Liste, der zweite die Elemente einer send2-Liste. Die Elemente der send1-Liste sind kleiner als "10", die Elemente der send2-Liste sind größer oder gleich "10".

Die `received`-Liste muss dann alle Werte von `send1` und `send2` enthalten. Sammelt man alle Werte von `received`, die kleiner als "10" sind und alle die größer gleich "10" sind, so müssen die so entstandenen Listen den `send1`- und `send2`-Listen gleichen.

```
@Test
public void testTwoProducerOneConsumer() throws Exception {
    final int max = 200;

    final Queue<String> queue = new FixedQueue<>(2);
    final List<String> send1 = new ArrayList<>();
    for (int i = 0; i < 10; i++)
        send1.add(String.valueOf(i));
    final List<String> send2 = new ArrayList<>();
    for (int i = 10; i < 20; i++)
        send2.add(String.valueOf(i));

    final List<String> received = new ArrayList<>();

    final Thread consumer = new Thread(() -> {
        String value = queue.dequeue();
        while (value != null) {
            Threading.sleepRandom(max);
            received.add(value);
            value = queue.dequeue();
        }
    });

    final Thread producer1 = new Thread(() -> {
        for (final String element : send1) {
            Threading.sleepRandom(max);
            queue.enqueue(element);
        }
    });

    final Thread producer2 = new Thread(() -> {
        for (final String element : send2) {
            Threading.sleepRandom(max);
            queue.enqueue(element);
        }
    });

    consumer.start();
    producer1.start();
    producer2.start();
    producer1.join();
    producer2.join();
    queue.enqueue(null);
    consumer.join();
    Assert.assertEquals(
        send1.size() + send2.size(), received.size());

    final List<String> l1 = received.stream()
        .filter(s -> Integer.parseInt(s) < 10)
        .collect(Collectors.toList());
    final List<String> l2 = received.stream()
        .filter(
            s -> Integer.parseInt(s) >= 10)
        .collect(Collectors.toList());
    Assert.assertEquals(send1, l1);
```

```
    Assert.assertEquals(send2, 12);  
    Assert.assertEquals(0, queue.size());  
}
```

Im Folgenden lesen nun zwei Consumer (die Interpretation des Tests sei dem Leser / der Leserin überlassen):

```
@Test  
public void testTwoProducerTwoConsumer() throws Exception {  
    final int max = 200;  
  
    final Queue<String> queue = new FixedQueue<>(2);  
  
    final List<String> send1 = new ArrayList<>();  
    for (int i = 0; i < 10; i++)  
        send1.add(String.valueOf(i));  
  
    final List<String> send2 = new ArrayList<>();  
    for (int i = 10; i < 20; i++)  
        send2.add(String.valueOf(i));  
  
    final List<String> received1 = new ArrayList<>();  
    final List<String> received2 = new ArrayList<>();  
  
    final Thread consumer1 = new Thread(() -> {  
        String value = queue.dequeue();  
        while (value != null) {  
            Threading.sleepRandom(max);  
            received1.add(value);  
            value = queue.dequeue();  
        }  
    });  
  
    final Thread consumer2 = new Thread(() -> {  
        String value = queue.dequeue();  
        while (value != null) {  
            Threading.sleepRandom(max);  
            received2.add(value);  
            value = queue.dequeue();  
        }  
    });  
  
    final Thread producer1 = new Thread(() -> {  
        for (final String element : send1) {  
            Threading.sleepRandom(max);  
            queue.enqueue(element);  
        }  
    });  
  
    final Thread producer2 = new Thread(() -> {  
        for (final String element : send2) {  
            Threading.sleepRandom(max);  
            queue.enqueue(element);  
        }  
    });  
  
    consumer1.start();  
    consumer2.start();  
    producer1.start();  
}
```



```
producer2.start();
producer1.join();
producer2.join();
queue.enqueue(null);
queue.enqueue(null);
consumer1.join();
consumer2.join();

Assert.assertEquals(
    send1.size() + send2.size(),
    received1.size() + received2.size());

// We cannot assert,
// but the following output gives us some hint
// (behavior is non deterministic)
System.out.println(received1.size());
System.out.println(received2.size());

final List<String> send = new ArrayList<>(send1);
send.addAll(send2);
send.sort((s1, s2) -> s1.compareTo(s2));

final List<String> received = new ArrayList<>(received1);
received.addAll(received2);
received.sort((s1, s2) -> s1.compareTo(s2));

Assert.assertEquals(send, received);
Assert.assertEquals(0, queue.size());
}
```

11.2.3 Future

`Future` ist ein Interface, welches von Future-Klassen implementiert werden kann. Ein `Future` repräsentiert einen Wert, der erst in Zukunft zur Verfügung stehen wird.

Mittels `get` kann der Wert (vom Typ `T`) abgeholt werden – wobei `get` möglicherweise blockiert (wenn der Wert noch nicht verfügbar ist). Mittels `isAvailable` kann getestet werden, ob der Wert bereits verfügbar ist.

```
package util;

public interface Future<T> {
    public abstract T get();
    public abstract boolean isAvailable();
}
```

11.2.4 FutureImpl

`FutureImpl` ist eine Implementierung von `Future`:

Nach der Erzeugung eines `FutureImpl`-Objekts liefert `isAvailable` den Wert `false`. Der Aufruf der `get`-Methode wird blockieren. Erst dann, wenn der Wert der `Futures` mittels `set` gesetzt wird, wird `get` zurückkehren (und `isAvailable` wird den Wert `true` liefern).

`get` und `isAvailable` können wiederholt aufgerufen werden; `set` darf nur ein einziges Mal aufgerufen werden. Ein wiederholter Aufruf von `set` liefert eine `RuntimeException`.

```
package util;

public class FutureImpl<T> implements Future<T> {

    private boolean isAvailable;
    private T value;
    private final Object lock = new Object();

    @Override
    public T get() {
        synchronized (this.lock) {
            if (!this.isAvailable)
                Threading.wait(this.lock);
            return this.value;
        }
    }

    @Override
    public boolean isAvailable() {
        synchronized (this.lock) {
            return this.isAvailable;
        }
    }

    public void set(final T value) {
        synchronized (this.lock) {
            if (this.isAvailable)
                throw new RuntimeException("Future already set");
            this.value = value;
            this.isAvailable = true;
        }
    }
}
```

```
        throw new RuntimeException("Future already set");
        this.value = value;
        this.isAvailable = true;
        this.lock.notifyAll();
    }
}
```

11.2.5 FutureTest

In dem folgenden ersten Test erzeugt der Hauptthread ein `Future` und spaltet einen Thread ab, in welchem nach 100ms das `Future` gesetzt wird. Der Hauptthread wartet derweil via `get` auf das Resultat.

Es wird zugesichert, dass unmittelbar nach dem Starten des Threads das Resultat des `Future`-Objekts noch nicht verfügbar ist. Weiterhin wird zugesichert, dass `get` das erwartete Resultat liefert. Es wird zugesichert, dass `get` mehrfach aufgerufen werden kann. Schließlich wird zugesichert, dass nach der Rückkehr von `get` die `isAvailable` Methode den Wert `true` liefert.

```
@Test
public void testSimple() {
    final FutureImpl<Integer> f = new FutureImpl<>();
    new Thread(() -> {
        Threading.sleep(100);
        f.set(42);
    }).start();
    Assert.assertFalse(f.isAvailable());
    Assert.assertEquals(Integer.valueOf(42), f.get());
    Assert.assertEquals(Integer.valueOf(42), f.get());
    Assert.assertTrue(f.isAvailable());
}
```

Der zweite Test sichert zu, dass auf ein `Future` die `Set`-Methode nur ein einziges Mal aufgerufen werden kann:

```
@Test
public void testMultipleSet() {
    final FutureImpl<Integer> f = new FutureImpl<>();
    f.set(42);
    XAssert.assertThrows(RuntimeException.class, () -> f.set(77));
    Assert.assertEquals(Integer.valueOf(42), f.get());
}
```

11.3 Executor

Ein `Executor` ist ein Thread-Pool, der `Runnable`s und `Supplier`s ausführen kann – die Ergebnissen der `Supplier` werden in Form von `Future`-Objekte bereitgestellt.

Ein `Executor` besitzt eine `Queue`, in welcher `Task`-Objekte eingestellt werden können. Ein `Task` kann sich ausführen: das Interface spezifiziert eine parameterlose `execute`-Methode).

Von `Task` sind die Klasse `RunnableTask` und `SupplierTask` abgeleitet. Eine `RunnableTask` besitzt ein `Runnable` – in `execute` wird dieses `Runnable` ausgeführt. Eine `SupplierTask<T>` besitzt einen `Supplier<T>` und ein `Future<T>` – in `execute` wird der `Supplier` ausgeführt und das Ergebnis via `get` der `Future` zugewiesen (und damit letztere "available" gemacht).

Ein `Executor` startet eine einstellbare Anzahl Threads. Die `Runnable`s dieser Threads laufen in einer "Endlosschleife" – sie warten allesamt auf die Rückkehr der `dequeue`-Methode der `Queue`. Kehrt die Methode zurück, so wird die von ihr gelieferte `Task` ausgeführt und anschließend auf eine neue `Task` gewartet.

Die Klasse `Executor` besitzt zwei `put`-Methoden: Der ersten `put<T>`-Methode kann ein `Supplier<T>` übergeben werden. Aufgrund dieses `Supplier<T>` und eines neu erzeugten `Future<T>`-Objekts wird ein `SupplierTask<T>` erzeugt, welcher in der `Queue` eingestellt wird. Das erzeugte `Future<T>`-Objekt wird als `Future<T>` zurückgeliefert. Der zweiten `put`-Methode wird ein `Runnable` übergeben, welche in Form einer `RunnableTask` in die `Queue` eingestellt wird. Diese zweite `put`-Methode liefert `void`.

Mittels der `shutdown`-Methode kann ein `Executor` kontrolliert heruntergefahren werden. Alle noch in der `Queue` befindlichen `Task`-Objekte werden abgearbeitet – erst dann kehrt `shutdown` zurück. (Die Threads werden dabei jeweils durch einen `null`-`Task` "gestoppt".)

```
package util;

import java.util.function.Supplier;

public class Executor {

    public static abstract class Task {
        public abstract void execute();
    }

    public static class RunnableTask extends Task {
        final Runnable runnable;

        RunnableTask(final Runnable runnable) {
            this.runnable = runnable;
        }
    }
}
```

```

        @Override
        public void execute() {
            this.runnable.run();
        }
    }

    public static class SupplierTask<T> extends Task {
        final Supplier<T> supplier;
        final Future<T> future;

        SupplierTask(final Supplier<T> supplier, final Future<T>
future) {
            this.supplier = supplier;
            this.future = future;
        }

        @Override
        public void execute() {
            ((FutureImpl<T>) this.future).set(this.supplier.get());
        }
    }

    private final Queue<Task> queue;

    private final Thread[] threads;

    private boolean shutdownCalled = false;

    public Executor(final int maxThreads, final Queue<Task> queue) {
        this.threads = new Thread[maxThreads];
        this.queue = queue;
        for (int i = 0; i < maxThreads; i++) {
            this.threads[i] = new Thread(() -> {
                while (true) {
                    final Task task = queue.dequeue();
                    if (task == null)
                        break;
                    task.execute();
                }
            });
            this.threads[i].start();
        }
    }

    public <T> Future<T> put(final Supplier<T> supplier) {
        if (this.shutdownCalled)
            throw new RuntimeException("illegal put after shutdown");
        final Future<T> future = new FutureImpl<>();
        this.queue.enqueue(new SupplierTask<T>(supplier, future));
        return future;
    }

    public void put(final Runnable runnable) {
        if (this.shutdownCalled)
            throw new RuntimeException("illegal put after shutdown");
        this.queue.enqueue(new RunnableTask(runnable));
    }

```

```
public void shutdown() {
    this.shutdownCalled = true;
    for (int i = 0; i < this.threads.length; i++)
        this.queue.enqueue(null);
    for (int i = 0; i < this.threads.length; i++)
        Threading.join(this.threads[i]);
}
```

11.3.1 ExecutorTest

Im ersten Test wird ein `Executor` erzeugt, der mit einem einzigen `Thread` und einer `FixedQueue` der Kapazität 1 operiert. Dem `Executor` werden via `put` 5 `Runnables` zur Ausführung übergeben. Die Ausführung jedes `Runnable` dauert 200ms. Die Ausführung jeder dieser 5 `Runnables` inkrementiert einen Zähler.

Es wird zugesichert, dass nach dem `shutdown` der Zähler den Wert 5 hat (jedes dem `Executor` übergebene `Runnable` also ausgeführt wurde). Ebenso wird zugesichert, dass die Ausführungszeit größer ist als $5 * 200\text{ms}$ (denn die `Runnables` wurden allesamt von einem einzigen `Thread` ausgeführt – also sequentiell).

```
@Test
public void testWithOneThreadAndQueueWithOneElement() {
    final Stopwatch watch = new Stopwatch();
    final Executor pool =
        new Executor(1, new FixedQueue<Executor.Task>(1));
    final AtomicInteger count = new AtomicInteger();
    final int n = 5;
    final int sleepTime = 200;

    watch.start();
    for (int i = 0; i < n; i++) {
        pool.put(() -> {
            Threading.sleep(sleepTime);
            count.incrementAndGet();
        });
    }
    pool.shutdown();
    watch.stop();
    Assert.assertEquals(n, count.get());
    Assert.assertTrue(
        watch.elapsedMilliseconds() >= sleepTime * n);
}
```

Der `Executor` des zweiten Tests startet 3 `Threads` (und benutzt wiederum eine `FixedQueue` mit der Kapazität 1). Hier werden dem `Executor` 6 `Runnables` übergeben. Jede `Action` benötigt zur Ausführung auch hier 200ms.

Da drei `Runnables` nun parallel ausgeführt werden, kann zugesichert werden, dass die Gesamtdauer der Ausführung nicht größer ist 400ms plus einem kleinem `Delta` (50 ms)

```

@Test
public void testWithTreeThreadsAndQueueWithOneElement() {
    final Stopwatch watch = new Stopwatch();
    final Executor pool =
        new Executor(3, new FixedQueue<Executor.Task>(1));
    final AtomicInteger count = new AtomicInteger();
    final int n = 6;
    final int sleepTime = 200;
    watch.start();
    for (int i = 0; i < n; i++) {
        pool.put(() -> {
            Threading.sleep(sleepTime);
            count.incrementAndGet();
        });
    }
    pool.shutdown();
    watch.stop();
    Assert.assertEquals(n, count.get());
    Assert.assertTrue(
        watch.elapsedMilliseconds() < sleepTime * 6 / 3 + 50);
}

```

Im dritten Test schließlich werden dem `Executor` Objekte des Typs `Supplier<Integer>` übergeben. Dabei wird von der aufgerufenen `put`-Methode jeweils ein `Supplier<Integer>`-Objekt zurückgeliefert. Nach dem `shutdown` werden in einer Schleife alle `Future`-Resultate abgeholt und mit den erwarteten Resultaten verglichen.

```

@Test
public void testFuture() {
    final Stopwatch watch = new Stopwatch();
    final Executor pool =
        new Executor(3, new FixedQueue<Executor.Task>(1));
    final int n = 6;
    final int sleepTime = 200;

    final int[] results = new int[n];
    for (int i = 0; i < n; i++) {
        results[i] = i + 10;
    }

    final List<Supplier<Integer>> funcs = new ArrayList<>();
    funcs.add(() -> {
        Threading.sleep(sleepTime);
        return results[0];
    });
    funcs.add(() -> {
        Threading.sleep(sleepTime);
        return results[1];
    });
    funcs.add(() -> {
        Threading.sleep(sleepTime);
        return results[2];
    });
    funcs.add(() -> {
        Threading.sleep(sleepTime);
    });
}

```

```
        return results[3];
    });
    funcs.add(() -> {
        Threading.sleep(sleepTime);
        return results[4];
    });
    funcs.add(() -> {
        Threading.sleep(sleepTime);
        return results[5];
    });

    final List<Future<Integer>> futures = new ArrayList<>();

    watch.start();
    for (int i = 0; i < n; i++) {
        futures.add(pool.put(funcs.get(i)));
    }
    pool.shutdown();
    watch.stop();
    Assert.assertTrue(
        watch.elapsedMilliseconds() < sleepTime * 6 / 3 + 50);

    for (int i = 0; i < n; i++) {
        Assert.assertEquals(
            results[i], (int) futures.get(i).get());
    }
}
```


11.4 Nodes

11.4.1 Node

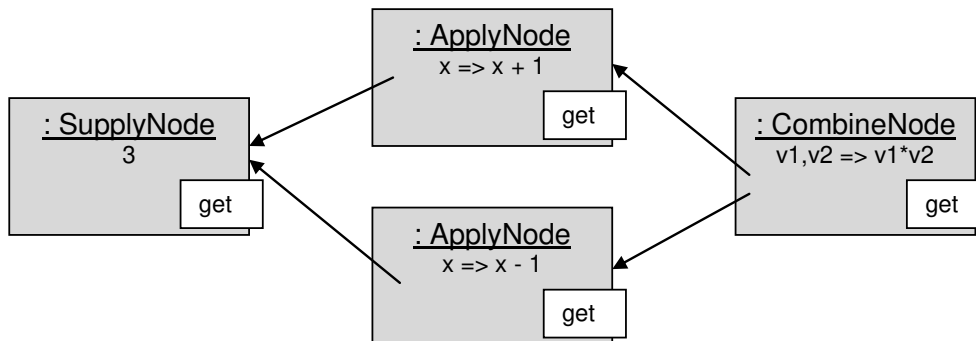
In einer komplexen Berechnung gibt ein Knoten (ein `SupplyNode`) den Input vor. Dieser Input kann von weiteren Berechnungen transformiert werden – von Berechnungen, die möglicherweise parallel ausgeführt werden können. Solche Berechnungen können in `ApplyNodes` ausgeführt werden. Die Ergebnisse parallel ausgeführter Berechnungen müssen natürlich wieder zusammengefasst werden können – diese Aufgabe erledigen `CombineNodes`.

Angenommen, folgende "Formel" soll ausgeführt werden – auf den Input-Wert 3:

$$f(x) = (x + 1) * (x - 1)$$

Die Schritte $x + 1$ und $x - 1$ können parallel ausgeführt werden. Wenn die Ergebnisse dieser beiden Berechnungen vorliegen ($v1$, $v2$), dann können diese beiden Ergebnisse wie folgt zusammengefasst werden: $v1 * v2$.

Es werden vier Nodes benötigt: einen `SupplyNode` (der den Input zur Verfügung stellt), zwei `ApplyNodes` und einen `CombineNode`. Alle Nodes kennen ihren Vorgänger (resp. ihre Vorgänger (Plural)):



Die `get`-Operationen der beiden `ApplyNodes` werden parallel ausgeführt werden.

Hier die Implementierung der Berechnung:

```

final Node<Integer> f1 = Node.supply(() -> 3);
final Node<Integer> f2 = f1.apply(x -> x + 1);
final Node<Integer> f3 = f1.apply(x -> x - 1);
final Node<Integer> f4 = f2.combine(f3, (v1, v2) -> v1 * v2);
int result = f4.get();
System.out.println(result); // -> 8
  
```

`Node.supply` erzeugt einen `SupplyNode<Integer>`. Die `apply`-Methode erzeugt jeweils einen `ApplyNode<Integer, Integer>` und die `combine`-Methode einen `CombineNode<Integer, Integer, Integer>`. Auf den `CombineNode` wird dann `get` aufgerufen. Der `Combi-`

neNode startet zwei Threads, in denen die `get`-Methoden der beiden Vorgänger (der beiden `ApplyNodes`) ausgeführt werden. Diese holen sich den Input vom `SupplierNode` (via `get`). Die `get`-Methoden führen die Berechnung jeweils nur ein einziges Mal aus und liefern beim wiederholten Aufruf das beim ersten Aufruf berechnete Resultat zurück.

Hier die `Node`-Klasse:

```
package util;

import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.function.Supplier;

import util.Executor.Task;

public abstract class Node<T> {

    private static Executor executor =
        new Executor(4, new FixedQueue<Task>(10));

    public abstract T get();

    protected boolean ready;
    protected T value;

    private static class SupplierNode<T1> extends Node<T1> {
        final Supplier<T1> supplier;

        SupplierNode(final Supplier<T1> supplier) {
            this.supplier = supplier;
        }

        @Override
        public T1 get() {
            synchronized (this) {
                if (!this.ready) {
                    this.value = this.supplier.get();
                    this.ready = true;
                }
            }
            return this.value;
        }
    }

    private static class ApplyNode<T1, R> extends Node<R> {
        final Function<T1, R> function;
        final Node<T1> previous;

        ApplyNode(final Node<T1> previous,
            final Function<T1, R> function) {
            this.previous = previous;
            this.function = function;
        }

        @Override
        public R get() {
            synchronized (this) {
```

```

        if (!this.ready) {
            this.value =
                this.function.apply(this.previous.get());
            this.ready = true;
        }
    }
    return this.value;
}

private static class CombineNode<T1, T2, R> extends Node<R> {
    final BiFunction<T1, T2, R> function;
    final Node<T1> previous1;
    final Node<T2> previous2;

    CombineNode(final Node<T1> previous1,
                final Node<T2> previous2,
                final BiFunction<T1, T2, R> function) {
        this.previous1 = previous1;
        this.previous2 = previous2;
        this.function = function;
    }

    @Override
    public R get() {
        synchronized (this) {
            if (!this.ready) {
                final Future<T1> f1 = executor.put(
                    () -> this.previous1.get());
                final Future<T2> f2 = executor.put(
                    () -> this.previous2.get());
                this.value =
                    this.function.apply(f1.get(), f2.get());
                this.ready = true;
            }
        }
        return this.value;
    }
}

public static <R> Node<R> supply(final Supplier<R> supplier) {
    return new SupplierNode<R>(supplier);
}

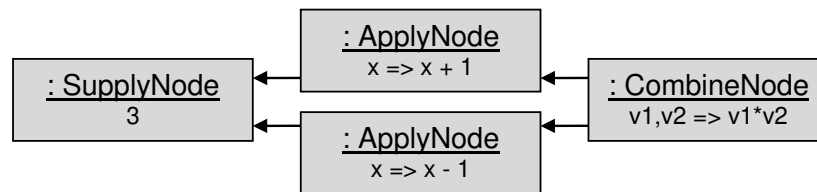
public <R> Node<R> apply(final Function<T, R> function) {
    return new ApplyNode<T, R>(this, function);
}

public <R, O> Node<R> combine(final Node<O> other,
    final BiFunction<T, O, R> function) {
    return new CombineNode<T, O, R>(this, other, function);
}
}

```

11.4.2 NodeTest

Im ersten Test wird folgender Graph konstruiert:

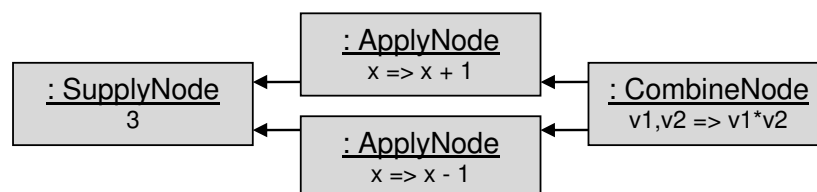


Der Aufruf der `get`-Methode auf den `CombineNode` muss folgenden Wert ergeben:

$$(3-1) * (3 + 1) = 8$$

```
@Test
public void testCorrectResult() {
    final Node<Integer> f1 = Node.supply(() -> 3);
    final Node<Integer> f2 = f1.apply(x -> x + 1);
    final Node<Integer> f3 = f1.apply(x -> x - 1);
    final Node<Integer> f4 = f2.combine(f3, (v1, v2) -> v1 * v2);
    Assert.assertEquals(8, (int) f4.get());
}
```

Auch im ersten Test wird derselbe Graph konstruiert:



Allerdings wird nun mitgezählt, wie häufig die jeweilige Berechnung aufgerufen wird. Jede der vier Berechnungen muss exakt einmal ausgeführt werden.

Die Berechnung des `CombineNodes` muss – da dessen `get`-Methode im Haupt-Thread aufgerufen wird – ebenfalls in diesem Thread ausgeführt werden. Die Berechnung der beiden `ApplyNodes` muss in jeweils einem neuen Thread ausgeführt werden. Und die Berechnung des `SupplyNodes` muss in einem dieser neuen Threads ausgeführt werden (welcher der beiden dies sein wird, ist unbestimmt).

Die Berechnung beider `ApplyNodes` muss als Input den (vom `Supply-Node` gelieferten) Wert 3 bekommen. Einer der beiden Inputs der `CombineNode`-Berechnung muss den Wert 2 bekommen – der jeweils andere Input muss den Wert 4 bekommen.

```
@Test
public void testCorrectFlow() {
    final AtomicInteger c1 = new AtomicInteger();
    final AtomicInteger c2 = new AtomicInteger();
    final AtomicInteger c3 = new AtomicInteger();
    final AtomicInteger c4 = new AtomicInteger();

    final AtomicReference<Thread> t1 = new AtomicReference<>();
    final AtomicReference<Thread> t2 = new AtomicReference<>();
    final AtomicReference<Thread> t3 = new AtomicReference<>();
    final AtomicReference<Thread> t4 = new AtomicReference<>();

    final Node<Integer> f1 = Node.supply(() -> {
        c1.incrementAndGet();
        t1.set(Thread.currentThread());
        return 3;
    });

    final Node<Integer> f2 = f1.apply(x -> {
        Assert.assertEquals(3, (int) x);
        c2.incrementAndGet();
        t2.set(Thread.currentThread());
        return x + 1;
    });

    final Node<Integer> f3 = f1.apply(x -> {
        Assert.assertEquals(3, (int) x);
        c3.incrementAndGet();
        t3.set(Thread.currentThread());
        return x - 1;
    });

    final Node<Integer> f4 = f2.combine(f3, (v1, v2) -> {
        Assert.assertTrue(v1 == 2 && v2 == 4 || v1 == 4 && v2 ==
2);

        t4.set(Thread.currentThread());
        c4.incrementAndGet();
        return v1 * v2;
    });

    Assert.assertEquals(8, (int) f4.get());

    Assert.assertEquals(1, c1.get());
    Assert.assertEquals(1, c2.get());
    Assert.assertEquals(1, c3.get());
    Assert.assertEquals(1, c4.get());

    Assert.assertTrue(t1.get() == t2.get() || t1.get() ==
t3.get());
    Assert.assertNotSame(Thread.currentThread(), t2.get());
    Assert.assertNotSame(Thread.currentThread(), t3.get());
    Assert.assertNotSame(t2, t3.get());
    Assert.assertSame(Thread.currentThread(), t4.get());
}
```

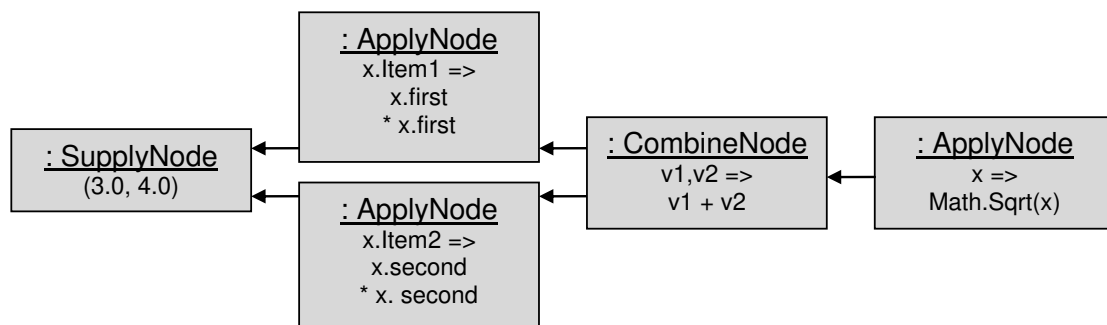
Im dritten Test wird die Hypotenuse zweier Katheten berechnet:

```
c(a, b) = Math.Sqrt(a * a + b * b)
```

Die beiden Quadrat-Berechnungen können parallel ausgeführt werden.

Der (vom `SupplyNode` gelieferte) Input ist ein `Pair (3.0, 4.0)`:

```
class Pair <T1, T2> {  
  
    public final T1 first;  
    public final T2 second;  
  
    public Pair(final T1 first, final T2 second) {  
        this.first = first;  
        this.second = second;  
    }  
}
```



In den parallel ausgeführten Berechnungen simulieren wir "harte Arbeit": `Threading.sleep(500)`. Würden die Berechnungen sequentiell ausgeführt, würde die gesamte Berechnung länger als 1000ms dauern. Da sie aber parallel ausgeführt werden, kann zugesichert werden, dass die Gesamtberechnung weniger als 600ms benötigt. Und es wird natürlich das korrekte Ergebnis der Berechnung zugesichert: 5.0.

```
@Test
public void testPythagoras() {
    final Stopwatch watch = new Stopwatch();
    watch.start();

    final Node<Tuple<Double, Double>> f1
        = Node.supply(() -> new Tuple<Double, Double>(3.0, 4.0));
    final Node<Double> f2 = f1.apply(x -> {
        Threading.sleep(500);
        return x.first * x.first;
    });
    final Node<Double> f3 = f1.apply(x -> {
        Threading.sleep(500);
        return x.second * x.second;
    });
    final Node<Double> f4 = f2.combine(f3, (v1, v2) -> v1 + v2);
    final Node<Double> f5 = f4.apply(v -> Math.sqrt(v));

    Assert.assertEquals(5.0, f5.get(), 0);

    watch.stop();
    Assert.assertTrue(watch.elapsedMilliseconds() < 600);

    watch.start();
    Assert.assertEquals(5.0, f5.get(), 0);
    watch.stop();
    Assert.assertTrue(watch.elapsedMilliseconds() < 20);
}
```

Wird `f5.get` das zweite Mal aufgerufen, werden – da die Ergebnisse im Graph bereits gespeichert sind – keine neuen Berechnungen ausgeführt. Die Zeit, die dieser zweite `get`-Aufruf benötigt, ist dann kleiner als 20ms.

12

Anhang

12.1	Reflection	12-4
12.2	Dynamic Proxy	12-7

12 Anhang

Dieses Kapitel enthält eine kleine Einführung in Reflection und Dynamic-Proxies – die grundlegenden Techniken, auf denen JUnit und Mock-Werkzeuge beruhen.

12.1 Reflection

Reflection ermöglicht es, zur Laufzeit Klassen zu inspizieren und Objekte von Klassen zu "generisch" zu erzeugen und "generisch" zu manipulieren.

Jede Klasse wird zur Laufzeit repräsentiert durch ein von der VM erzeugtes `Class`-Objekt. Solche `Class`-Objekte gibt's nicht nur für Klassen, sondern auch für Interfaces, Enums, Annotations und für primitive Typen – und sogar für den Typ `void`.

Auf jedes `Object` ist die in der Klasse `Object` definierte Methode `getClass` aufrufbar. Sie liefert das `Class`-Objekt derjenigen Klasse zurück, von welcher das Objekt eine Instanz ist.

Ein `Class`-Objekt kann auch über die Notation `<Typname>.class` angesprochen werden (über das sog. Klassenliteral).

Ein `Class`-Objekt kann schließlich auch ermittelt werden mittels der statischen `Class`-Methode `forName`, welcher der Name der Klasse in Form eines Strings übergeben wird.

Mittels der `Class`-Methode `getMethod` kann ein `Method`-Objekt ermittelt werden. Ein `Method`-Objekt repräsentiert eine Methode der entsprechenden Klasse (mittels `getMethods` können alle `Method`-Objekte einer Klasse ermittelt werden).

Mittels eines `Class`-Objekts kann via `newInstance` ein Objekt der Klasse erzeugt werden, die von dem `Class`-Objekt beschrieben ist. Ein `Class`-Objekt kann also als Factory fungieren.

Mittels der `Method`-Methode `invoke` kann genau diejenige Methode auf ein Objekt aufgerufen werden, die von dem `Method`-Objekt beschrieben ist.

Reflection ermöglicht also u.a. zweierlei: Ohne den Namen der Klasse fest im Programm zu verdrahten, kann ein Objekt der Klasse erzeugt werden (via `Class.newInstance`). Und ohne die Namen der Methoden fest zu verdrahten, können diese aufgerufen werden (via `Method.invoke`).

Die folgende Demo-Anwendung benutzt die Beispiel-Klassen `MathService` und `Calculator`:

```
package appl;

public class MathService {

    public int gcd(final int x, final int y) {
        // ...
    }

    public int lcm(final int x, final int y) {
        // ...
    }
}
```

```
package appl;

public class Calculator {
    private int value;
    public void add(final int v) {
        this.value += v;
    }
    public void subtract(final int v) {
        this.value -= v;
    }
    public int getValue() {
        return this.value;
    }
}
```

Hier die Demo-Anwendung (welche in der Form eines JUnit-Tests daherkommt – die "Tests" haben hier natürlich nur die Bedeutung von "Demos"):

```
package appl;

import java.lang.reflect.Method;

import org.junit.Assert;
import org.junit.Test;

public class Demo {

    @Test
    public void demoClass() throws Exception {
        final MathService service = new MathService();
        final Class<?> cls1 = service.getClass();
        final Class<?> cls2 = MathService.class;
        final Class<?> cls3 =
            Class.forName("appl.MathService");
        Assert.assertSame(cls1, cls2);
        Assert.assertSame(cls2, cls3);
    }

    @Test
    public void demoCreation() throws Exception {
        final Class<?> cls =
```

```

        Class.forName("appl.MathService");
        final Object obj = cls.newInstance();
        Assert.assertTrue(obj instanceof MathService);
    }

    @Test
    public void demoMethod() throws Exception {
        final Class<?> cls =
            Class.forName("appl.MathService");
        final Method m1 =
            cls.getMethod("gcd", int.class, int.class);
        final Method m2 =
            cls.getMethod("lcm", int.class, int.class);
        Assert.assertEquals("gcd", m1.getName());
        Assert.assertEquals("lcm", m2.getName());
        final Class<?> returnType = m1.getReturnType();
        Assert.assertSame(int.class, returnType);
        final Class<?>[] parameterTypes =
            m1.getParameterTypes();
        Assert.assertEquals(2, parameterTypes.length);
        Assert.assertSame(int.class, parameterTypes[0]);
        Assert.assertSame(int.class, parameterTypes[1]);
    }

    @Test
    public void demoInvoke() throws Exception {
        final Class<?> cls =
            Class.forName("appl.MathService");
        final Object obj = cls.newInstance();
        final Method m1 =
            cls.getMethod("gcd", int.class, int.class);
        final Object result = m1.invoke(obj, 40, 30);
        Assert.assertTrue(result instanceof Integer);
        final int res = (Integer)result;
        Assert.assertEquals(10, res);
    }

    @Test
    public void demoInvokeCalculator() throws Exception {
        final Class<?> cls =
            Class.forName("appl.Calculator");

        final Method m1 =
            cls.getMethod("add", int.class);
        final Method m2 =
            cls.getMethod("subtract", int.class);
        final Method m3 =
            cls.getMethod("getValue");

        final Object obj = cls.newInstance();

        m1.invoke(obj, 44);
        m2.invoke(obj, 2);
        final int result = (Integer)m3.invoke(obj);

        Assert.assertEquals(42, result);
    }
}

```

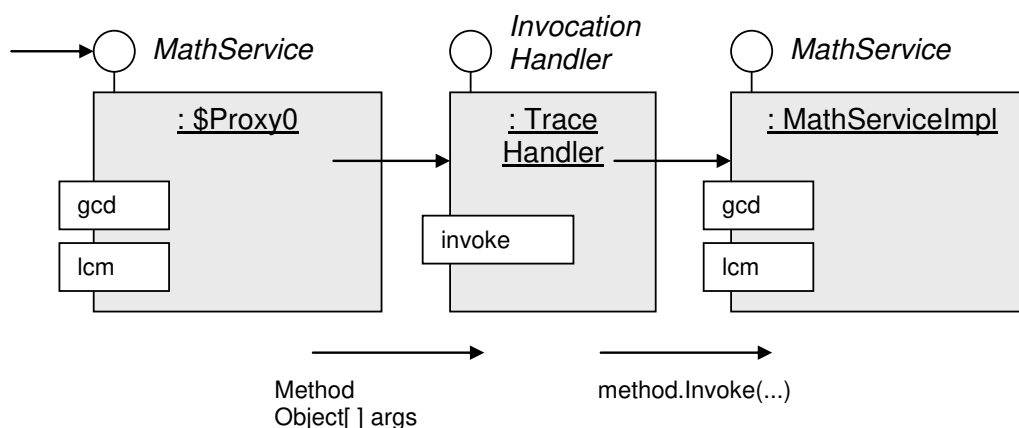
12.2 Dynamic Proxy

Im Folgenden wird das Konzept des Dynamic-Proxies erläutert. Dieses Konzept wird in fast allen Mocking-Werkzeugen verwendet.

Dynamic-Proxies sind Objekte von Proxy-Klassen, welche erst zur Laufzeit generiert werden (erst zur Laufzeit wird also der Bytecode solcher Klassen erzeugt). Ein Dynamic-Proxy implementiert stets ein Interface. Die Methoden eines solchen Proxies delegieren an eine zentrale Methode eines `InvocationHandlers`. Dieser Handler kann dann z.B. an ein Ziel-Objekt delegieren, dessen Klasse dasselbe Interface wie die generierte Proxy-Klasse implementiert.

Die `invoke`-Methode des `InvocationHandlers` kann vor und nach der Delegation an das eigentliche Zielobjekt noch weitere Aktionen ausführen – im einfachsten Fall z.B. den Aufruf der Methode loggen – und nach Rückkehr der Zielmethode das von dieser gelieferte Resultat loggen. Somit enthält die Interceptor-Methode i.d.R. einen "Dreisatz": Pre-Invoke, Invoke und Post-Invoke.

Hier ein beispielhaftes Objektdiagramm:



Rechts im Bild erkennt man ein `MathServiceImpl`-Objekt – den "richtigen" Service, der ausschließlich die eigentliche Fachlogik implementiert. Die Klasse `MathServiceImpl` implementiert das Interface `MathService`.

Links im Bild erkennt man ein Proxy-Objekt, welches dasselbe Interface implementiert wie der richtige `MathServiceImpl`. Es existiert nun ein weiteres (technisches) Interface namens

`java.lang.reflect.InvocationHandler`, welches eine `invoke`-Methode spezifiziert. Diese `invoke`-Methode verlangt als Parameter u.a. ein `Method`-Objekt und einen `Object`-Array.

Dann können alle Methoden der Proxy-Klasse einfach wie folgt implementiert werden:

- Berechne dasjenige `Method`-Objekt, welches die aktuelle Methode beschreibt (die Methode des Interfaces! – hier von `MathService`).
- Verpacke die individuellen Aufruf-Parameter in einem `Object`-Array.
- Rufe die `invoke`-Methode auf den an das Proxy-Objekt angeschlossenen `InvocationHandlers` auf.
- Gebe das Resultat dieses Aufrufs als Returnwert zurück.

Da alle Methoden der Proxy-Klasse nach exakt demselben Schema implementiert sind, kann diese Klasse automatisch generiert werden – der Generator muss nur das `Class`-Objekt des zu implementierenden Interfaces kennen (hier: `MathService.class`).

Angenommen nun weiterhin, der `InvocationHandler` besitze eine Referenz auf das eigentliche Ziel-Objekt (hier: auf das "richtige" `MathServiceImpl`-Objekt) – eine Referenz vom allgemeinsten Typ `Object`. Dann kann das an `invoke` übergebene `Method`-Objekt genutzt werden, um die diesem `Method`-Objekt entsprechende Methode auf das Ziel-Objekt aufzurufen. Natürlich können dann vor der Delegation Trace-Ausgaben erfolgen ("Pre-Invoke"). Auch nach Rückkehr der Ziel-Methode können Traces geschrieben werden ("Post-Invoke").

Im Hauptprogramm schließlich wird ein via `Proxy.newProxyInstance` eine Klasse generiert, welche das Interface `MathService` implementiert. Diese Klasse wird dann instanziiert werden, wobei der erzeugten Instanz der `TraceHandler` übergeben wird (welcher seinerseits das eigentliche Ziel-Objekt kennt).

```
package appl;

public interface MathService {
    public abstract int gcd(final int x, final int y);
    public abstract int lcm(final int x, final int y);
}
```

```
package appl;

public interface Calculator {
    public abstract void add(final int v);
    public abstract void subtract(final int v);
    public abstract int getValue();
}
```

```
package appl;

public class MathServiceImpl implements MathService {

    public int gcd(final int x, final int y) { ... }
    public int lcm(final int x, final int y) { ... }
}
```

```
package appl;

public class CalculatorImpl implements Calculator {
    private int value;
    @Override
    public void add(final int v) {
        this.value += v;
    }
    @Override
    public void subtract(final int v) {
        this.value -= v;
    }
    @Override
    public int getValue() {
        return this.value;
    }
}
```

```
package util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Arrays;

public class TraceHandler implements InvocationHandler {

    private final Object target;

    public TraceHandler(final Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(final Object proxy, final Method method,
final Object[] args) throws Throwable {
        System.out.println(">> " + methodName(method) +
            Arrays.toString(args));
        try {
            final Object result;
            if (this.target instanceof InvocationHandler)
                result = ((InvocationHandler) this.target)
                    .invoke(proxy, method, args);
            else
                result = method.invoke(this.target, args);
            if (method.getReturnType() == void.class)
                System.out.println("<< " + methodName(method));
            else
                System.out.println("<< " + methodName(method)
                    + " => " + result);
            return result;
        }
        catch (final InvocationTargetException e) {
            System.out.println("<< " + methodName(method)
                + " => " + e.getTargetException());
            throw e.getTargetException();
        }
    }
}
```



```

        catch(final Throwable e) {
            System.out.println("<< " + methodName(method)
                + " => " + e);
            throw e;
        }
    }

    private static String methodName(final Method method) {
        return method.getClass().getSimpleName() + "." +
method.getName();
    }
}

```

```

package appl;

import java.lang.reflect.Proxy;

import util.TraceHandler;

public class Demo {

    public static void main(final String[] args) {

        final MathServiceImpl mathServiceImpl =
            new MathServiceImpl();
        final TraceHandler mathServiceHandler =
            new TraceHandler(mathServiceImpl);
        final MathService mathService =
            (MathService) Proxy.newProxyInstance(
                ClassLoader.getSystemClassLoader(),
                new Class<?>[] { MathService.class },
                mathServiceHandler);
        System.out.println(mathService.getClass().getName());

        final CalculatorImpl calculatorImpl =
            new CalculatorImpl();
        final TraceHandler calculatorHandler =
            new TraceHandler(calculatorImpl);
        final Calculator calculator =
            (Calculator) Proxy.newProxyInstance(
                ClassLoader.getSystemClassLoader(),
                new Class<?>[] { Calculator.class },
                calculatorHandler);
        System.out.println(calculator.getClass().getName());

        demo(mathService, calculator);
    }

    private static void demo(
        final MathService mathService,
        final Calculator calculator) {
        System.out.println(mathService.gcd(25, 30));
        System.out.println(mathService.lcm(20, 45));
        calculator.add(40);
        calculator.add(2);
        System.out.println(calculator.getValue());
    }
}

```

Die Ausgaben:

```
com.sun.proxy.$Proxy0  
com.sun.proxy.$Proxy1  
>> Method.gcd[25, 30]  
<< Method.gcd => 5  
5  
>> Method.lcm[20, 45]  
<< Method.lcm => 180  
180  
>> Method.add[40]  
<< Method.add  
>> Method.add[2]  
<< Method.add  
>> Method.getValuenull  
<< Method.getValue => 42  
42
```


13

Literatur

13 Literatur

Johannes Link: Softwaretests mit Junit (DPunkt 2005)

Das Buch enthält u.a. eine kleine Einführung in JUnit (ein Testwerkzeug für Java). Der weitaus größere Teil des Buches befasst sich aber mit den Konzepten der Testgetriebenen Softwareentwicklung. Das Buch ist also auf jeden Fall auch interessant für Java-Entwickler.

Johannes Link zitiert häufig Frank Westphal (s.u.)...

Frank Westphal: Testgetriebene Entwicklung mit JUnit und FIT (DPunkt 2006)

Das Buch ist eine brauchbare Ergänzung zum Buch von Joh. Link. Zusätzlich wird FIT vorgestellt (ein Framework für den Integrationstest). Allerdings ist das Buch reichlich geschwätzig...

Frank Westphal zitiert häufig Johannes Link (s.o.)...

Kent Beck: Test-Driven Development by Example (Addison Wesley 2005)

Beck ist der TDD-Papst ... Das Buch enthält hauptsächlich zwei ausführliche Beispiele: das Money-Beispiel (in Java) und ein xUnit-Beispiel (das wird in Python vorgestellt)

Martin Fowler: Refactoring (Addison-Wesley 2005)

Fowler schreibt nur vernünftige Bücher (sehr empfehlenswert auch: UML Distilled – für Leute, die UML einfach nur verwenden wollen, ohne zuvor ein achtsemestriges UML-Studium machen zu wollen – ohne also das offizielle Handbuch zu UML lesen wollen...)

Martin Fowler: <http://martinfowler.com/articles/mocksArentStubs.html>

