DV- Anwendungen in der Wirtschaft

Projekt: ICEE-Filecollector
QS: WhiteBox Tests

Auftraggeber: Dr. Nikolai Bauer
Projektteam: DVA Gruppe 3
Verfasser: Nazar Kulyk - 258360030100

Einleitung

Meine Rolle im Projekt war von einer Seiter die Verwaltung von Source Coden und Berreitstellung deren Online zu verfügung. Aber die Hauptaugabe war die Qualitätsicherung von Entwickler-team rausgegebenen Klassen.

Erste Aufgabe würde mit Hilfe von SVN (Subversion) gelöst. Subversion ist eine Open-Source-Software zur Versionsverwaltung. Da es viele Schwächen des in Entwicklerkreisen sehr beliebten Programms CVS behebt, wird Subversion oft als dessen Nachfolger bezeichnet, obwohl es sich um ein eigenständiges Projekt handelt. Es ist jedoch absichtlich von der Bedienung sehr ähnlich gehalten.

- Es würden bestimmte Vorteile bei Verwendung von SVN in diesem Projekt ersichtlich:
- In Subversion können Dateien (und Verzeichnisse) auch umbenannt und verschoben werden.
- Subversion bietet einen verbesserten Umgang mit Binärdaten.
- Commits sind in Subversion atomar, das heißt eine Änderung wird entweder komplett oder gar nicht ins Repository gespeichert.
- Zusätzlich zu einem eigenen Server, dem Zugriff via Secure Shell und der Speicherung im lokalen Dateisystem existiert auch ein Modul für den Apache 2 Webserver, mit dem die Daten auch mit der HTTP-Erweiterung WebDAV übertragen und über eine einfache eingebaute Oberfläche mit einem gewöhnlichen Webbrowser angesehen werden können.

Die Qualität Sicherung von Klassen(White Box Tests) ist mit JUNIT Framework abgedeckt. JUNIT ist besonders für automatisierte Unit-Tests einzelner Units (meist Klassen oder Methoden) geeignet.

Nach implämntierung von Tests habe ich verschiedene logische und auch technische Fehler gefunden. Die dann von Entwiklern in nächsten Versionen verbessert worden. Mit jedem automatischem Ablauf von Test Suits könnte man feststellen ob neue Version die alten Fehler verbessert und auch ob es neue bringt.

SVN und JUNIT Tests haben sich ausreichen für die vor mir stehenden Aufgaben erwiesen. Ich kann einen Einsatz von diesen Tools nur noch weiter empfällen.

White-Box-Test

Der Begriff **White-Box-Test** bezeichnet eine Methode des Software-Tests, bei der die Tests mit Kenntnissen über die innere Funktionsweise des zu testenden Systems entwickelt werden. Im Gegensatz zum Black-Box-Test ist für diesen Test also ein Blick in den Quellcode gestattet, d.h. es wird am Code geprüft.

Ein Beispiel für einen White-Box-Test ist ablaufbezogenes Testen, bei welchem der Ablaufgraph im Vordergrund steht. Ziel des Tests ist es, sicherzustellen, dass Testfälle im Bezug auf die Überdeckung des Quellcodes gewisse **Hinlänglichkeitskriterien** erfüllen. Gängig sind dabei u.a. folgende Maße:

- **Anweisungsüberdeckung**: Ausführung aller Anweisungen
- **Kantenüberdeckung**: Durchlaufen aller möglichen Kanten von Verzweigungen des Kontrollflusses
- **Bedingungsüberdeckung** (mehrere Varianten): Bewertung der Bedingungen
- **Pfadüberdeckung** (mehrere Varianten): Betrachtung der Pfade durch ein Modul

Die Zahl der benötigten Testfälle nimmt in der obigen Aufzählung von oben nach unten stark zu, so dass man sich bei der Testdurchführung oft auf einfacher zu erfüllende Maße einigt. Kantenüberdeckung wird im Allgemeinen als minimales Testkriterium angesehen. Je nach Art und Struktur der zu testenden Software können andere Maße für ein System als Ganzes oder für Module sinnvoll sein.

Selbst wenn ein Softwaresystem im Bezug auf ein Hinlänglichkeitskriterium erfolgreich getestet wurde, schließt das nicht aus, dass es Fehler enthält. Dies liegt in der Natur des White-Box-Tests begründet und kann eine der folgenden Ursachen haben:

- Der White-Box-Test leitet Testfälle nicht aus der Spezifikation des Programms her, sondern aus dem Programm selbst. Getestet werden kann nur die Korrektheit eines Systems, nicht, ob es eine geforderte Semantik erfüllt.
- Auch wenn alle Programmpfade getestet worden sind, bedeutet dies nicht, dass ein Programm fehlerfrei arbeitet. Der Fall, dass im Graphen des Kontrollflusses Kanten fehlen, wird nicht erkannt.

Zusammenfassend kann man sagen, dass White-Box-Tests alleine als Testmethodik nicht ausreichen. Eine sinnvolle Testreihe sollte White-Box-Tests und Black-Box-Tests kombinieren.

Will man ein System auch in seinen Teilsystemen testen, benötigt man dazu Kenntnisse über die innere Funktionsweise des zu testenden Systems. White-Box-Tests eignen sich besonders gut, um in Erscheinung getretene Fehler zu lokalisieren, d.h. die fehlerverursachende Komponente zu identifizieren, und als Regressionstest ein Wiederauftreten des Fehlers bereits an der Komponente zu vermeiden.

Weil die Entwickler der Tests Kenntnisse über die innere Funktionsweise des zu testenden Systems besitzen müssen, werden White-Box-Tests von dem selben Team, häufig sogar von denselben Entwicklern entwickelt wie die zu testenden Komponenten. Spezielle Testabteilungen werden für White-Box-Tests in der Regel nicht eingesetzt, da der Nutzen speziell für diese Aufgabe abgestellter Tester meist durch den Aufwand der Einarbeitung in das System eliminiert wird.

Automatisierte Unit Tests in Java

JUnit ist ein kleines, mächtiges Java-Framework zum Schreiben und Ausführen automatischer Unit Tests. Da die Tests direkt in Java programmiert werden, ist das Testen mit JUnit so einfach wie das Kompilieren. Die Testfälle sind selbstüberprüfend und damit wiederholbar.

Unit Testing ist der Test von Programmeinheiten in Isolation von anderen im Zusammenhang eines Programms benötigten, mitwirkenden Programmeinheiten. Die Größe der unabhängig getesteten Einheit kann dabei von einzelnen Methoden über Klassen bis hin zu Komponenten reichen.

Das JUnit-Framework

JUnit ist ein gutes Beispiel für ein kleines, fokussiertes Framework mit einer hohen Dichte sauber verwendeter Entwurfsmuster. JUnit ist vor allem deshalb ein gutes Beispiel, weil es inklusive seines eigenen Testcode kommt.
Wir betrachten jetzt JUnit-Klassen, mit denen wir am meisten in Berührung kommen werden. Dazu wenden wir am besten uns eine Beispiel `Euro` Klasse zu.

```
public class Euro {
  private double amount;

  public Euro(double amount) {
    this.amount = amount;
  }

  public double getAmount() {
    return this.amount;
  }
}
```

"Assert"

Wie testen wir mit JUnit?

JUnit erlaubt uns, Werte und Bedingungen zu testen, die jeweils erfüllt sein müssen, damit der Test okay ist. Die Klasse `Assert` definiert dazu eine Menge von `assert` Methoden, die unsere Testklassen aus JUnit erben und mit denen wir in unseren Testfällen eine Reihe unterschiedlicher Behauptungen über den zu testenden Code aufstellen können:

`assertTrue(boolean condition)` verifiziert, ob eine Bedingung wahr ist.

Beispiele:

```
assertTrue(theJungleBook.isChildrensMovie());
assertTrue(40 == xpProgrammer.workingHours() * days);
```

`assertEquals(Object expected, Object actual)` verifiziert, ob zwei Objekte gleich sind. Der Vergleich der Objekte erfolgt in JUnit über die `equals` Methode.

Beispiele:

```
assertEquals("foobar", "foo" + "bar");
assertEquals(new Euro(2), Movie.getCharge(1));
```

Der Vorteil dieser und der folgenden `assertEquals` Varianten gegenüber dem Test mit `assertTrue` liegt darin, daß JUnit Ihnen nützliche zusätzliche Informationen bieten kann, wenn der Test tatsächlich fehlschlägt. JUnit benutzt in diesem Fall die `toString` Repräsentation Ihres Objekts, um den erwarteten Wert auszugeben.

`assertEquals(int expected, int actual)` verifiziert, ob zwei ganze Zahlen gleich sind. Der Vergleich erfolgt für die primitiven Java-Typen über den == Operator.
Beispiele:

```
assertEquals(9, customer.getFrequentRenterPoints());
assertEquals(40, xpProgrammer.workingHours() * days);
```

`assertEquals(double expected, double actual, double delta)` verifiziert, ob zwei Fließkommazahlen gleich sind. Da Fließkommazahlen nicht mit unendlicher Genauigkeit verglichen werden können, wird zusätzlich eine Toleranz erwartet.
Beispiele:

```
assertEquals(3.1415, Math.pi(), 1e-4);
```

`assertNull(Object object)` verifiziert, ob eine Objektreferenz `null` ist.
Beispiele:

```
assertNull(hashMap.get(key));
```

`assertNotNull(Object object)` verifiziert, ob eine Objektreferenz nicht `null` ist.
Beispiele:

```
assertNotNull(httpRequest.getParameter("action"));
```

`assertSame(Object expected, Object actual)` verifiziert, ob zwei Referenzen auf das gleiche Objekt verweisen.
Beispiele:

```
assertSame(bar, hashMap.put("foo", bar).get("foo"));
```

Die `assertEquals` Methode ist neben den oben aufgeführten Argumenttypen auch für die primitiven Datentypen `float`, `long`, `boolean`, `byte`, `char` und `short` überladen. Der Phantasie beim Testen sind also keine Grenzen gesetzt.

"AssertionFailedError"

Was passiert, wenn ein Test fehlschlägt?

Die im Testcode durch `assert` Anweisungen kodierten Behauptungen werden von der Klasse `Assert` automatisch verifiziert. Im Fehlerfall bricht JUnit den laufenden Testfall sofort mit dem Fehler `AssertionFailedError` ab.

Um den Test des centweisen Rundens nachzuholen, könnten wir zum Beispiel folgenden Testfall schreiben:

```
public class EuroTest...
```

```
  public void testRounding() {
    Euro roundedTwo = new Euro(1.995);
    assertTrue(2.00 == roundedTwo.getAmount());
  }
}
```

Der Test läuft natürlich nicht, weil unsere Klasse noch kein Konzept für das Runden hat. Der Fortschrittbalken von dem GUI Part des JUNIT Frameworks Test Resultat Ausgabe verfärbt sich rot(zeigt auf fehlerhafte Ausführung von Test Case).

Testen von Exceptions

Wie testen wir, ob eine Exception folgerichtig geworfen wird?

Zum Beispiel - "Wie geht add mit negativen Beträgen um?" Nun, sicherlich ist es nicht sinnvoll, negative `double` Werte in den Konstruktor der Klasse geben zu dürfen. Wir sollten deshalb die Vorbedingung des Konstruktors spezifizieren. Normalerweise würde ich nicht empfehlen, Tests für Vorbedingungen zu schreiben, doch möchte ich an dieser Stelle eine Ausnahme machen, weil Sie so mal sehen, wie Sie mit JUnit Exceptions testen können.

```
public class EuroTest...
  public void testNegativeAmount() {
    try {
      final double NEGATIVE_AMOUNT = -2.00;
      new Euro(NEGATIVE_AMOUNT);
      fail("Should have raised an IllegalArgumentException");
    } catch (IllegalArgumentException expected) {
    }
  }
}
```

Das JUnit-Muster, um Exceptions zu testen, ist denkbar einfach, nachdem es einmal klar geworden ist. Wir probieren, ein `Euro` Exemplar mit negativem Wert zu bilden. Was wir im Fall einer solchen Verletzung der Vorbedingung erwarten würden, wäre eine `IllegalArgumentException` zu werfen. Wenn der Konstruktor also diese Ausnahme auslöst, kommt der `catch` Block zum Tragen. Der Variablenname der Exception drückt es schon aus, wir erwarten eine `IllegalArgumentException`. Wir fangen die erwartete Exception und alles ist gut. Wenn der Konstruktor die Ausnahme dagegen nicht auslöst, wird die `fail` Anweisung ausgeführt und der spezifizierte Fehlertext von JUnit in einem `AssertionFailedError` Fehlerobjekt protokolliert. Der `fail` Methode sind wir bisher noch nicht begegnet. Sie wird von der `Assert` Klasse realisiert. Sie können sie immer dann verwenden, wenn Sie einen Testfall mit einem Fehler abbrechen möchten.

"TestCase"

Wie gruppieren wir Testfälle um eine gemeinsame Menge von Testobjekten?

Ein Testfall sieht in der Regel so aus, daß eine bestimmte Konfiguration von Objekten aufgebaut wird, gegen die der Test läuft. Diese Menge von Testobjekten wird auch als Test-Fixture bezeichnet. Pro Testfallmethode wird meist nur eine bestimmte Operation und oft sogar nur eine bestimmte Situation im Verhalten der Fixture getestet.

Allgemein gilt, daß alle Testfälle einer Testklasse von der gemeinsamen Fixture Gebrauch machen

sollten. Hat eine Testfallmethode keine Verwendung für die Fixture-Objekte, so ist dies meist ein guter Indiz dafür, daß die Methode auf eine andere Testklasse verschoben werden will. Generell sollten Testklassen um die Fixture organisiert werden, nicht um die getestete Klasse. Somit kann es durchaus vorkommen, daß zu einer Klasse mehrere korrespondierende Testklassen existieren, von denen jede ihre individuelle Test-Fixture besitzt.

Lebenszyklus eines Testfalls

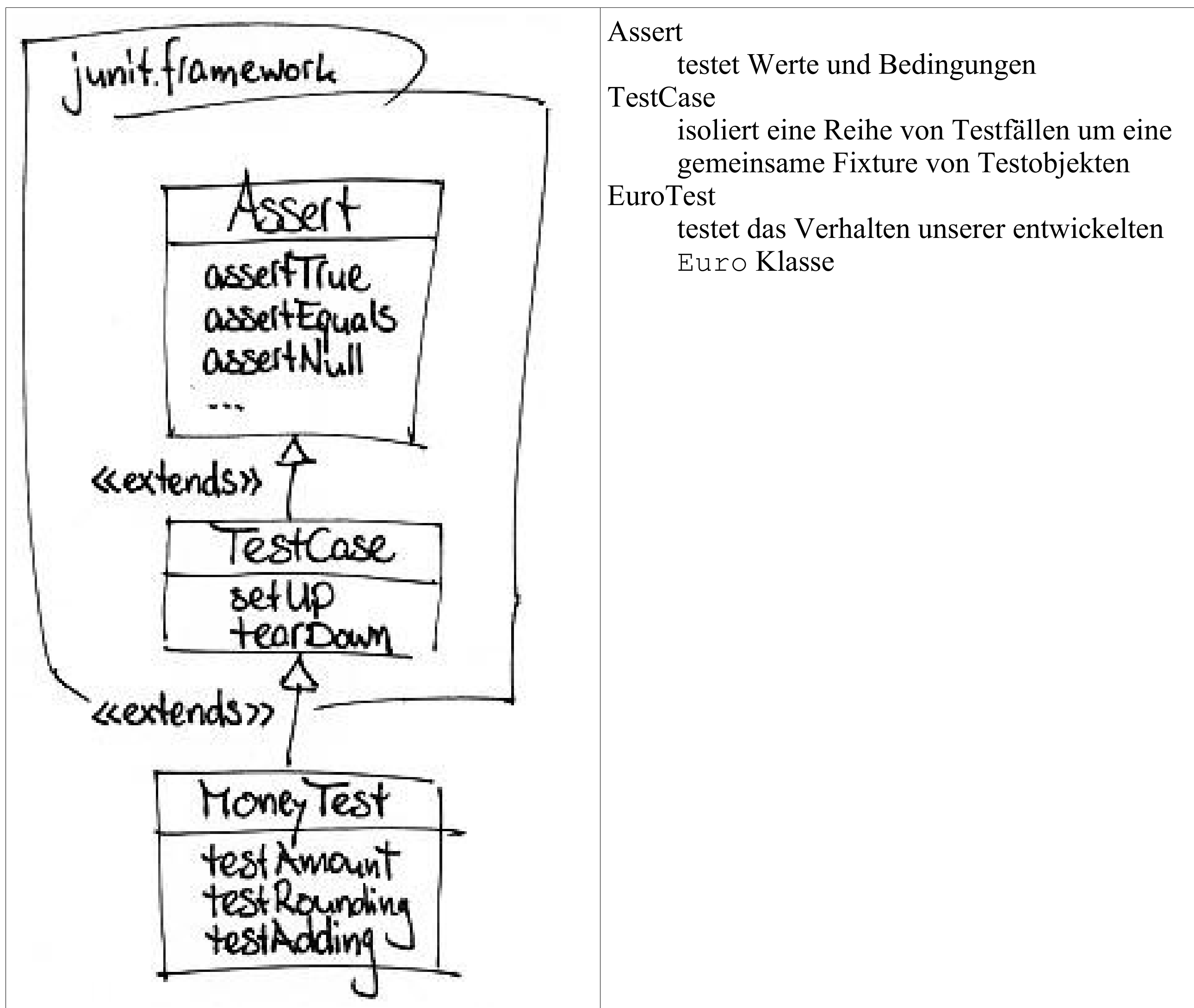
Was passiert, wenn JUnit die Tests dieser Klasse ausführt?

1. Das Test-Framework durchsucht die Testklasse mit Hilfe des Reflection API nach öffentlichen Methoden, die mit `test` beginnen und weder Parameter noch Rückgabewert besitzen.
2. JUnit sammelt diese Testfallmethoden in einer Testsuite und führt sie voneinander isoliert aus. Die Reihenfolge, in der Testfallmethoden vom Framework gerufen werden, ist dabei prinzipiell undefiniert.
3. Damit zwischen einzelnen Testläufen keine Seiteneffekte entstehen, erzeugt JUnit für jeden Testfall ein neues Exemplar der Testklasse und damit eine frische Test-Fixture.
4. Der Lebenszyklus dieses Exemplars ist so gestaltet, daß vor der Ausführung eines Testfalls jeweils die `setUp` Methode aufgerufen wird, sofern diese in der Unterklasse redefiniert wurde.
5. Anschliessend wird eine der `test...` Methoden ausgeführt.
6. Nach der Ausführung des Testfalls ruft das Framework die `tearDown` Methode, falls diese redefiniert wurde, und überläßt das `EuroTest` Objekt dann der Speicherbereinigung.
7. Dieser Zyklus wird vereinfacht erklärt ab Schritt 3 solange wiederholt, bis alle Testfälle jeweils einmal ausgeführt wurden.

Wenn wir die Sequenz mit einem Profiler aufzeichnen würden, in der JUnit unsere Testklasse benutzt, ergibt sich hier beispielsweise folgende Aufrufreihenfolge:

1. `new EuroTest("testAdding")`
2. `setUp()`
3. `testAdding()`
4. `tearDown()`
5. `new EuroTest("testAmount")`
6. `setUp()`
7. `testAmount()`
8. `tearDown()`
9. `new EuroTest("testRounding")`
10. `setUp()`
11. `testRounding()`
12. `tearDown()`

Wir sind jetzt an einer Stelle angelangt, an der es lohnen könnte, das grössere Bild zu betrachten. In UML läßt sich der Sachverhalt in etwa wie folgt darstellen:



Assert
testet Werte und Bedingungen

TestCase
isoliert eine Reihe von Testfällen um eine gemeinsame Fixture von Testobjekten

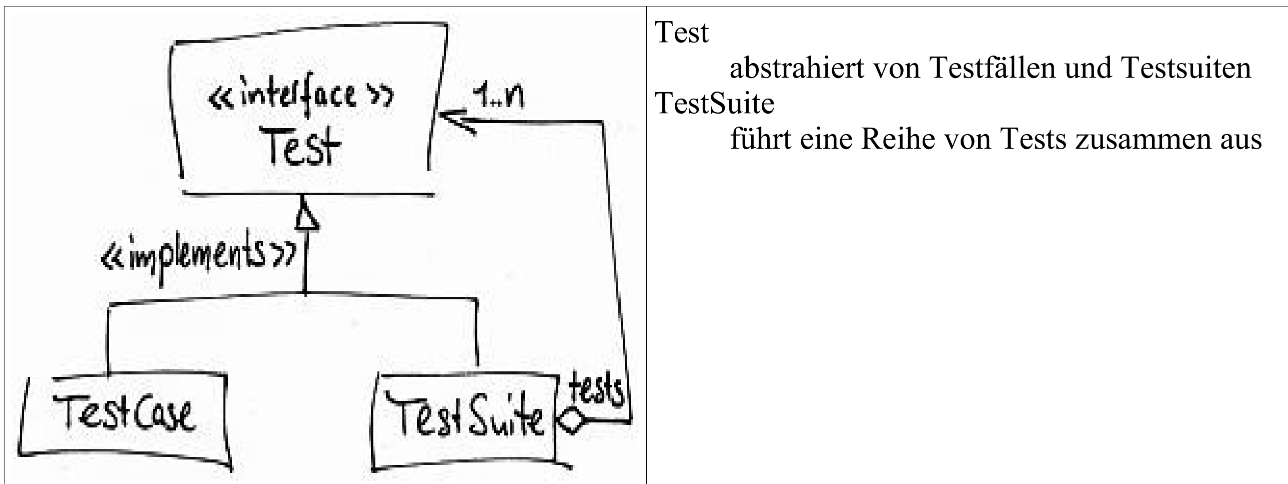
EuroTest
testet das Verhalten unserer entwickelten `Euro` Klasse

"TestSuite"

Wie führen wir eine Reihe von Tests zusammen aus?

Unser Ziel ist es, den gesamten Testprozess so weit zu automatisieren, daß wir den Test ohne manuellen Eingriff wiederholbar durchführen können. Wichtig ist schließlich, die Unit Tests möglichst häufig auszuführen, idealerweise nach jedem Kompilieren . Nur so erhalten wir unmittelbares Feedback darüber, wann unser Code zu funktionieren beginnt und wann er zu funktionieren aufhört. Wenn wir also immer nur winzig kleine Bissen vom Problemkuchen nehmen, werden wir uns nie länger als 1-3 Minuten mit Programmieren aufhalten, ohne grünes Licht zum Weiterprogrammieren einzuholen. Eher selten werden wir dabei nur einzelne Tests ausführen wollen. Meistens ist es schlau, alle gesammelten Unit Tests in einem Testlauf auszuführen, um ungewollten Seiteneffekten frühzeitig zu begegnen.

Mit JUnit können wir beliebig viele Tests in einer Testsuite zusammenfassen und gemeinsam ausführen. Dazu verlangt JUnit von uns, daß wir in einer statischen `suite` Methode definieren, welche Tests zusammen ausgeführt werden sollen. Eine Suite von Tests wird dabei durch ein `TestSuite` Objekt definiert, dem wir beliebig viele Tests und selbst andere Testsuiten hinzufügen können. Auf welcher Klasse Sie diese `suite` Methode definieren ist nebensächlich. In den meisten Fällen werden Sie jedoch spezielle Klassen allein dafür definieren wollen, um solche Testsuiten zu repräsentieren.



Test
abstrahiert von Testfällen und Testsuiten
TestSuite
führt eine Reihe von Tests zusammen aus

Test Protokol

Testfall ReflectionInfo

Testspezifikation:	Test von Methoden: getName und getMetaClass
Test Beschreibung: Initialisierung	Set: ReflectionInfo("testname", "testmeta");
Test Beschreibung: Überprüfen	assertEquals("testname", getName()); assertEquals("testmeta", getMetaClass());
Testergebnis	grün

Testfall ConfigTest

Testspezifikation:	Test von Methoden: testGetInstance() testGetExtensions() testGetExtractorNames() testGetMetaExtraxtors() testGetSearchPaths() testInsertSearchPath() testRemoveSearchPath()
Test Beschreibung: Initialisierung	Set: config = Config.getInstance();

Test Beschreibung: Überprüfen	assertNotNull(config);
	assertEquals("mp3", config.getExtensions("MP3")[0]);
	assertEquals("doc", config.getExtensions("MS Office Files")[0]);
	assertNotNull(config.getMetaExtraxtors());
	assertEquals("C:\\test\\",config.getSearchPaths()[0]);

Testergebnis	grün

Testfall MetaInfoTest

Testspezifikation:	Test von Methoden:

	testMetaInfo() testGetPath() testSize() testContainsKey() testGetMetaData() testSetMetaData()
Test Beschreibung: Initialisierung	Set: metaInfo1 = new MetaInfo ("test_path");

Test Beschreibung: Überprüfen	assertNotNull(new MetaInfo("test"));
	assertEquals("test_path", metaInfo1.getPath());
	assertEquals("MetaInfo should have only one element",1,metaInfo1.size());
	assertTrue(metaInfo1.containsKey("voice"));
	assertEquals("test",metaInfo1.getMetaData("voice"));
	assertEquals("MetaInfo should have one more element now",size+1,metaInfo1.size());

Testergebnis	grün

Testfall FileSystemScannerTest

Testspezifikation:	Test von Methoden: testGetInstance() testScan() testScanException()
Test Beschreibung: Initialisierung	Set: fs = FileSystemScanner.getInstance();
Test Beschreibung: Überprüfen	assertNotNull(fs); assertNotNull(fs.scan("MP3"));
Testergebnis	grün

Testfall FileInfoTest

Testspezifikation:	Test von Methoden: testGetPath() testGetSize() testGetBytes()
Test Beschreibung: Initialisierung	Set: fileInfo = fs.scan("MP3");

Test Beschreibung: Überprüfen	assertNotNull(fileInfo[0].getPath());
	assertTrue(0 <= fileInfo[0].getSize());
	assertEquals(3,firstBytes.length);
	assertEquals("ID3",new String(firstBytes));
	assertTrue(new String(lastBytes).startsWith("TAG"));

Testergebnis	grün

Testfall MetaExtractorFactoryTest

Testspezifikation:	Test von Methoden: testGetInstance() testGetExtractor()

Test Beschreibung: Initialisierung	Set: factory = MetaExtractorFactory.getInstance();
Test Beschreibung: Überprüfen	assertNotNull(factory);
Testergebnis	grün

Fazit

JUnit ist eine sehr gute Auswahl für Abdeckung von WhiteBox Tests und hat sich sehr gut in dienen Projekt gezeigt.

Am Anfang, wo die Architektur Design von Komponenten und die Schnittstellen sich dauernd ändern, musste man öfters die Tests nach verbessern. Aber sobald keine großen Änderungen in Architektur von API selber vorkommen sind die Tests immer ein sehr gutes Mittel um die Lauffähigkeit vom Code festzustellen nach jeder Änderung in eine Methode selber. Als Resultat bekommt man eine hoch Qualitatives Sourcecode.

Sobald neue Klassen gibt, kann man die mit neuen Tests versorgen, die dann die Qualitätssicherung für ganzes Projekt abdecken.

Schwierigkeiten gab es auch wenn ein Testfall einen Fehler nicht selber im Code sondern in eine speziellem Verhalten bei den Schnittstellen findet. Der Tester kann kein Angriff mit Refactoring oder Code Änderung vornehmen, so wie wenn es in Methode selber wäre. In diesem Fall muss man sich in Kontakt mit System Designer und Entwicklern setzen.

Package Explorer | Hierarchy | JUnit

Finished after 0,918 seconds

Runs: 24/24 Errors: 0 Failures: 3

Failures | Hierarchy

- collector.testcases.ConfigTest
  - testGetInstance
  - testGetExtensions
  - testGetExtractorNames
  - testGetMetaExtraxtors
  - testGetSearchPaths
  - testInsertSearchPath
  - testRemoveSearchPath
- collector.testcases.MetaInfoTest
  - testMetaInfo
  - testGetPath
  - testSize
  - testContainsKey
  - testGetMetaData
  - testSetMetaData
- collector.testcases.FileSystemScannerTest
  - testGetInstance
  - testScan
  - testScanException
- collector.testcases.FileInfoTest
  - testGetPath
  - testGetSize

Failure Trace

```
junit.framework.ComparisonFailure: expected:<C:\test\> but was:</home/schamane1
at collector.testcases.ConfigTest.testGetSearchPaths(ConfigTest.java:53)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
```

CollectorUITestSuit.java | config.xml | ConfigTest.java

```java
    */
    public void testGetExtractorNames() {
        String src1 = "MP3";
        String src2 = "MS Office Files";
        String[] dest = config.getExtractorNames();
        assertEquals( src1, dest[0]);
        assertEquals(src2, dest[1]);
    }

    /*
     * Test method for 'collector.configuration.Config.getMetaExtraxtors()'
     */
    public void testGetMetaExtraxtors() {

        // Vielleicht nicht ganz Sinvoll,aber :
        assertNotNull(config.getMetaExtraxtors());
    }

    /*
     * Test method for 'collector.configuration.Config.getSearchPaths()'
     */
    public void testGetSearchPaths() {
        String src = "C:\\test\\";
        String dest[] = config.getSearchPaths();
        assertEquals(src, dest[0]);
    }

    /*
     * Test method for 'collector.configuration.Config.insertSearchPath()'
     */
    public void testInsertSearchPath() {
        String src = "c:\\test2\\";
        config.insertSearchPath(src);
        String dest[] = config.getSearchPaths();
        assertEquals(src, dest[1]);
    }

    /*
     * Test method for 'collector.configuration.Config.removeSearchPath()'
     */
    public void testRemoveSearchPath() {
        String src = "c:\\test2\\";
        config.removeSearchPath(src);
        String dest[] = config.getSearchPaths();
        assertEquals(1, dest.length);

    }
}
```

Problems | Javadoc | Declaration | Console

0 errors, 22 warnings, 0 infos

Description	Resource	In Folder	Location
The import java.awt.event.ItemEvent is	CollectorUI.ja\	dva/collector	line 8
The import java.awt.event.ItemListener	CollectorUI.ja\	dva/collector	line 9
The import java.awt.event.MouseListen	CollectorUI.ja\	dva/collector	line 12