3 Unit Tests mit JUnit

Eine Art von Tests, die wir zeitnah zur Programmierung erstellen und nach jeder Programmmodifikation ausführen wollen, sind Unit Tests. Im Unit Test werden kleinere Programmteile in Isolation von anderen Programmteilen getestet. Die Granularität der unabhängig getesteten Einheit kann dabei von einzelnen Methoden über Klassen bis hin zu Komponenten reichen. Unit Tests sind in den überwiegenden Fällen *White-Box-Tests*. Das bedeutet, der Test kennt die Implementierungsdetails seines Prüflings und macht von diesem Wissen Gebrauch.

JUnit 4 wird in Kapitel 3.16 besprochen

Basis für die Unit Tests in diesem Buch stellt JUnit in Version 3.8.1 dar. JUnit ist ein kleines mächtiges Java-Framework zum Schreiben und Ausführen automatisierter Unit Tests. Die Software ist frei und im Kern von Kent Beck und Erich Gamma geschrieben. Die Tests werden direkt in Java kodiert, sind selbstüberprüfend und damit wiederholbar. Testfälle können mit JUnit sehr einfach organisiert und über eine Bedienungsoberfläche ausgeführt werden. JUnit hat sich seit 1998 schnell zum *De-facto-Standard für Entwicklertests* gemausert und ist inzwischen in praktisch alle modernen Java-Entwicklungsumgebungen integriert.

3.1 Download und Installation

JUnit ist *Open-Source-Software* unter *IBM Common Public License*. Aktuelle Version und Release-Kandidaten finden Sie auf *SourceForge*:

```
http://sourceforge.net/projects/junit/
```

Entsprechende Frameworks sind auch für alle anderen heute gängigen Programmiersprachen frei erhältlich:

```
http://www.xprogramming.com/software.htm
```

Wahrscheinlich bringt Ihre Entwicklungsumgebung schon JUnit mit. Wenn Ihr Compiler sich nicht darüber beschwert, dass er die JUnit-Klassen im Klassenpfad nicht finden kann, bleibt Ihnen die Installation also erspart. Den Test dafür werden wir in zwei Minuten antreten.

junit3.8.1.zip

Die vollständige JUnit-Distribution besteht gegenwärtig aus einem ZIP-Archiv, in dem neben dem Test-Framework (in `junit.jar`) auch seine Quellen (in `src.jar`), seine Tests, einige Beispiele, die Javadoc-Dokumentation, die FAQs, ein Kochbuch und zwei Artikel beiliegen.

Machen Sie sich mit der JUnit-Distribution vertraut, lesen Sie die beigefügten Artikel und spielen Sie die Beispiele durch.

Zur Installation entpacken Sie bitte das ZIP-Archiv und übernehmen `junit.jar` in Ihren `CLASSPATH`. Fertig!

Sollte Sie JUnit jemals vor scheinbar unlösbare Probleme stellen, versuchen Sie Ihr Glück in der *JUnit Yahoo! Group*:

```
http://groups.yahoo.com/group/junit/
```

Wenn Sie in die Welt der zahlreichen JUnit-Erweiterungen eintauchen wollen, werden Sie auf der *JUnit-Website* fündig:

```
http://www.junit.org
```

3.2 Ein erstes Beispiel

Wir wollen eine Klasse `Euro` ins Leben testen, die Geldbeträge akurater repräsentieren kann als der primitive Java-Typ `double`, den wir im vorherigen Kapitel verwendet haben. Anhand dieses kleinen Beispiels können Sie den Aufbau eines Testfalls kennen lernen und Ihre ersten kleinen Erfolge mit JUnit feiern. In der JUnit-Dokumentation finden Sie ein vergleichbares Beispiel.

Eine Warnung jedoch: Weder JUnits `Money`-Klasse noch die hier entwickelte Klasse genügt den Anforderungen der Finanzwirtschaft. Wenn es auf hohe Präzision ankommt, benutzen Sie bitte `BigDecimal`. Für unseren kleinen DVD-Verleih tut es jedoch auch weniger.

Wertsemantik

Unsere Klasse `Euro` stellt Wertobjekte für geldliche Beträge dar. Das heißt, das Objekt wird eindeutig durch seinen Wert beschrieben. Im Unterschied zu einem Referenzobjekt können wir ein Wertobjekt nicht verändern. Bei einer Operation auf einem Wertobjekt erhalten wir immer ein *neues* Objekt als Ergebnis der Operation zurück. Bekanntestes Beispiel dieser Wertsemantik in Java ist wahrscheinlich die Klasse `String`.

Entwurf der Klassenschnittstelle

Einen Test für eine Klasse zu schreiben bietet immer auch eine gute Gelegenheit dazu, über ihre öffentliche Schnittstelle nachzudenken. Was also erwarten wir von unserer Klasse? Nun, zunächst möchten wir sicherlich ein `Euro`-Objekt instanziieren können, indem wir dem Konstruktor einen Geldbetrag mitgeben. Wenn wir ein `Euro`-Objekt zu einem anderen hinzuaddieren, möchten wir, dass unsere Klasse mit einem neuen `Euro`-Objekt antwortet, das die Summe der beiden Beträge enthält. Eurobeträge unterliegen dabei einer besonderen Auflösung in ihrer numerischen Repräsentation. Zum Beispiel erwarten wir, dass auf den Cent genau gerundet wird und dass 100 Cents einen Euro ergeben. Aber fangen wir mit der einfachen Datenhaltung an. Hier sehen Sie den ersten Test:

```
import junit.framework.TestCase;

public class EuroTest extends TestCase {
  public void testAmount() {
    Euro two = new Euro(2.00);
    assertTrue(2.00 == two.getAmount());
  }

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

JUnit ist wirklich einfach zu verwenden und es wird nicht schwerer.

3.3 Anatomie eines Testfalls

TestCase

Sie erkennen, dass wir unsere Tests *getrennt* von der Klasse `Euro` in einer Klasse namens `EuroTest` definieren. Um unsere Testklasse in JUnit einzubinden, leiten wir sie von dessen Framework-Basisklasse `TestCase` ab.

Unser erster Testfall verbirgt sich hinter der Methode `testAmount`. Das Framework erkennt und behandelt diese Methode automatisch als Testfall, weil ihre Signatur der Konvention für Testfallmethoden folgt: `public void test...()`

In diesem Testfall erzeugen wir uns zunächst ein Objekt mit dem Wert »zwei Euro«. Der eigentliche Test erfolgt mit dem Aufruf der `assertTrue`-Methode, die unsere Testklasse aus ihrer Oberklasse erbt. Die `assertTrue`-Anweisung formuliert eine Annahme oder Forderung, die für den Code gilt und die JUnit automatisch für uns verifiziert.

Die `assertTrue`-Methode dient dazu, eine Bedingung zu testen. Als Argument akzeptiert sie einen booleschen Wert bzw. einen Ausdruck, der einen solchen liefert. Der Test ist erfolgreich, wenn die Bedingung erfüllt ist, das heißt der Ausdruck zu `true` ausgewertet werden konnte. Ist der Ausdruck dagegen `false`, protokolliert JUnit einen Fehlschlag. In unserem Beispiel testen wir, ob das Objekt `two` als Ergebnis der `getAmount`-Abfrage die erwarteten »zwei Euro« liefert.

Ältere JUnit-Tests

Wenn Ihnen Testcode begegnet, der mit der JUnit-Version 3.7 und früher geschrieben wurde, werden Sie sehen, dass jede Testklasse einen Konstruktor für den Namen des auszuführenden Testfalls benötigte. Ab JUnit 3.8 können Sie sich diesen Konstruktor sparen:

```
public class EuroTest...
  public EuroTest(String name) {
    super(name);
  }
}
```

Durch Aufruf der `main`-Methode können wir unseren ersten JUnit-Test ausführen. Der `junit.swingui.TestRunner` stellt dafür eine grafische Oberfläche auf Basis von Java Swing dar. In Entwicklungsumgebungen wie Eclipse [ec] und IntelliJ IDEA [int] lassen sich Tests auch direkt ausführen, womit wir uns die `main`-Methode sparen können. Beide IDEs bringen eigene JUnit-Oberflächen mit und verfügen über eine hervorragende JUnit-Integration. Wenn Sie zwei Seiten weiterblättern, erfahren Sie, was Eclipse im Detail zu bieten hat.

Direkte Testausführung in Eclipse und IntelliJ IDEA

Im Moment lässt sich unsere Testklasse noch nicht kompilieren. Der Compiler bemängelt, keine Klasse mit Namen `Euro` zu kennen.

3.4 Test-First

Unser Ziel ist es, ein Programm inkrementell in kleinen Schritten zu entwickeln. In diesem Kapitel werden Sie sehen, wie wir immer zuerst einen Test schreiben, bevor wir die Klasse weiterentwickeln, die diesen Test erfüllt. Natürlich können Sie mit JUnit aber auch Tests für schon bestehenden Code schreiben. Allerdings sind Klassen häufig schlecht testbar, wenn sie nicht von vornherein unter diesem Aspekt entworfen wurden.

Test-Last

Möglicherweise werden Sie bei der Lektüre dieses Kapitels denken, dass die Programmierschritte ja unglaublich winzig sind, um dann im nächsten Kapitel zu lernen, wie viel winziger unsere Schritte noch sein können. Für den Anfang wollen wir uns jedoch erst mal allein auf das Testen mit JUnit konzentrieren. Lassen Sie uns direkt den Test erfüllen.

Was müssen wir dafür tun?

```
public class Euro {
  private final double amount;

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

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

Wenn Sie Ihren Code kompilieren und `EuroTest` ausführen, sollte ein Fenster mit JUnits grafischer Oberfläche und darin der grün strahlende Testbalken erscheinen. Erfolg! Keinen Lärm zu machen, solange die Welt in Ordnung ist, gehört zur Philosophie von JUnit. Der Test läuft:

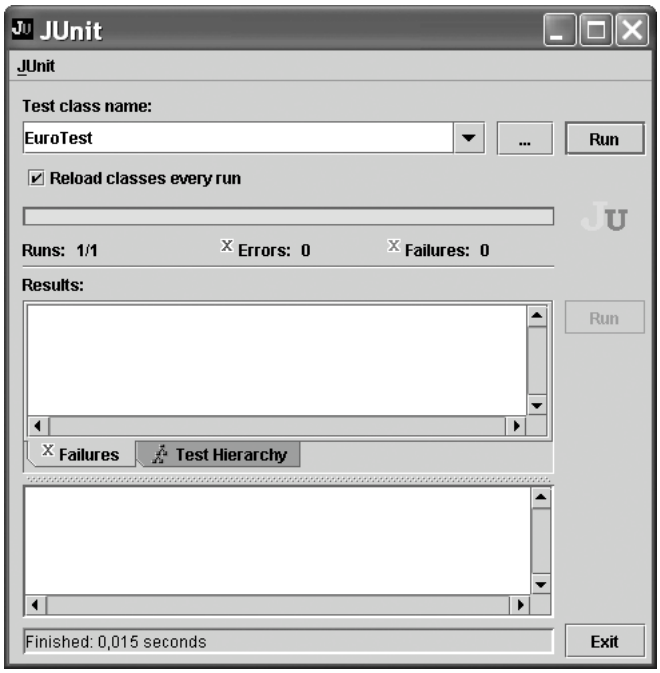

Abb. 3–1
Der legendäre grüne JUnit-Testbalken

Ein erfolgreicher Testlauf wird im Buch so am Seitenrand dargestellt:

JUnit: *OK*

Klicken Sie ruhig noch einmal auf den »Run«-Knopf. Dieser Knopf macht süchtig: Sie werden sich später ganz sicher mal dabei ertappen, die Tests zwei- oder dreimal nacheinander auszuführen, nur wegen des zusätzlichen Vertrauens, wenn hunderte von Tests ablaufen und der Fortschrittbalken dabei langsam mit Grün voll läuft.

Achtung, Suchtgefahr!

3.5 JUnit in Eclipse

»Run As ... JUnit Test«-Kommando

In Eclipse (und ebenso IDEA) hat JUnit eine hervorragende Integration gefunden. Selektieren Sie einfach Ihre Testklasse im *Package Explorer*, öffnen Sie das »Run«-Menü, wählen Sie »Run As« und »JUnit Test«. Alternativ können Sie »Run As« auch über das Kontextmenü Ihrer Klasse im Package Explorer sowie über ein Tastaturkürzel erreichen. In jedem Fall sollte sich der JUnit-View zeigen:

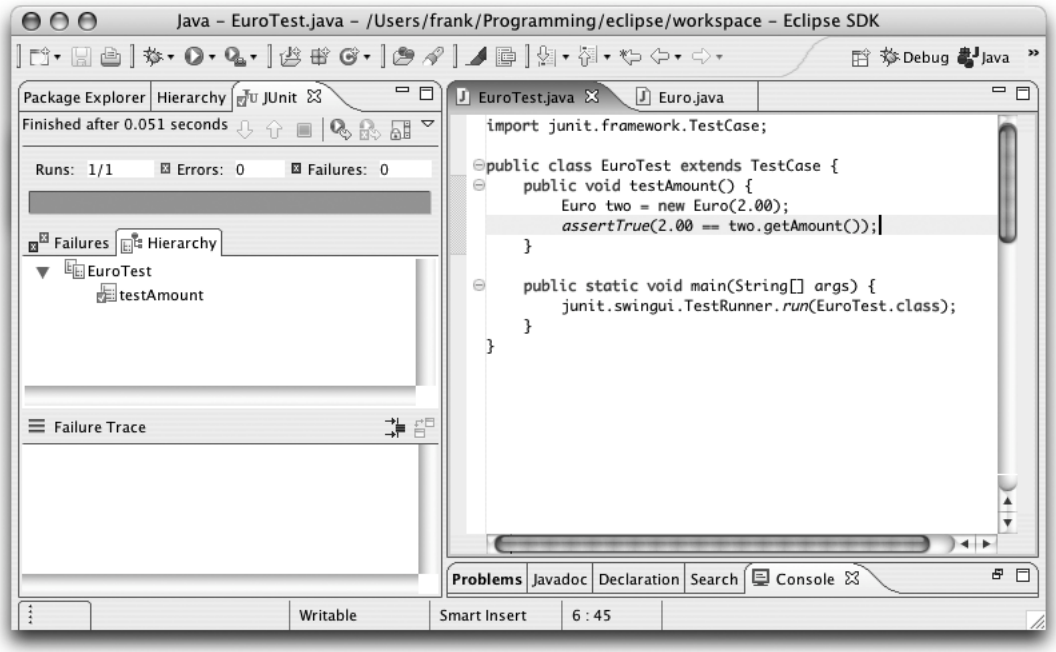

Abb. 3–2 *JUnit-View in Eclipse*

Für jeden weiteren Testlauf können Sie im View auf den »Run«-Knopf klicken oder auf Ihre Run-Konfiguration (das »Play«-Symbol neben dem Debugger) zurückgreifen. Sie müssen also nicht jedes Mal das »Run As«-Kommando neu bemühen.

Im Kontextmenü des Views können Sie Eclipse über die Auswahl des Menüpunktes »Activate on Error/Failure only« auch so einrichten, dass sich der View nur noch öffnet, sobald Tests fehlschlagen – aber wer will schon auf den grünen Balken verzichten?

JUnit-Klassen generieren lassen

Außerdem lohnt sich ein Blick in den »File > New«-Menüpunkt, wo Sie sich auf Knopfdruck JUnit-Klassen generieren lassen können (unter »Other...« und dann im Wizard unter »Java > JUnit« versteckt).

3.6 Das JUnit-Framework von innen

Mit unserem ersten grünen Balken im Rücken wenden wir uns nun dem Aufbau von JUnit zu. Ich empfehle Ihnen, sich den Code von JUnit einmal anzusehen. Wenn Sie so sind wie ich, dann lernen Sie ein Framework am besten, indem Sie dessen Code studieren. JUnit ist ein gutes Beispiel für ein kleines fokussiertes Framework mit einer hohen Dichte säuberlich verwendeter Entwurfsmuster. JUnit ist vor allem deshalb ein gutes Beispiel, weil es inklusive seines eigenen Testcodes kommt.

Eines guten Tages sollten Sie sich die Zeit nehmen und den `junit.framework`-Code studieren und gegebenenfalls um Ihre eigenen Anforderungen erweitern.

Der Framework-Kern besteht aus wenigen Klassen und circa tausend Zeilen Code. Ich werde im Folgenden die zentralen Klassen vorstellen, mit denen Sie am häufigsten in Berührung kommen werden. Damit Sie sehen, wie die Puzzlestücke zum Ganzen zusammengelegt werden, wenden wir uns hin und wieder unserer `Euro`-Klasse zu.

3.7 »Assert«

Wie testen wir mit JUnit?

JUnit erlaubt uns, Werte und Bedingungen zu testen, die jeweils erfüllt sein müssen, damit der Test erfolgreich durchläuft. Die Klasse `Assert` definiert dazu eine Familie spezialisierter `assert...`-Methoden, die unsere Testklassen aus JUnit erben und mit deren Hilfe wir in unseren Testfällen eine Reihe unterschiedlicher zu testender Behauptungen über unseren Code aufstellen können:

assert...-Methoden

- `assertTrue(boolean condition)` ist die allgemeinste Zusicherung. Sie verifiziert, ob eine logische Bedingung wahr ist:

  ```
  assertTrue(jungleBook.isChildrensMovie());
  ```

- `assertFalse(boolean condition)` verifiziert, ob eine Bedingung falsch ist. Diese Zusicherung ist erst seit JUnit 3.8 dabei:

  ```
  assertFalse(store.hasMovie("Buffalo 66"));
  ```

- `assertEquals(Object expected, Object actual)` verifiziert die *Gleichheit* zweier Objekte. Der Vergleich erfolgt in JUnit über die `equals`-Methode.

  ```
  assertEquals(new Euro(2.00), rental.getCharge());
  ```

Der Vorteil dieser und der noch folgenden `assertEquals`-Varianten gegenüber dem Check mit `assertTrue` liegt darin, dass JUnit uns nützliche Zusatzinformationen anbieten kann, wenn der Test tatsächlich fehlschlägt. JUnit benutzt in diesem Fall die `toString`-Repräsentation Ihres Objekts, um den erwarteten Wert darzustellen.

- `assertEquals(String expected, String actual)` verifiziert, ob zwei Zeichenketten gleich sind. Der Vergleich der `Strings` erfolgt wie gewohnt über die `equals`-Methode. Sind die `Strings` ungleich, wird nur das betreffende Segment dargestellt. Die Teile, die sich nicht unterscheiden, werden durch »`...`« ersetzt.

  ```
  assertEquals("Pulp Fiction", movie.getTitle());
  ```

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

  ```
  assertEquals(40, xpProgrammer.workingHours());
  ```

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

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

- `assertNull(Object object)` sichert zu, dass eine Referenz `null` ist:

  ```
  assertNull(Prices.getPrice("unknown"));
  ```

- `assertNotNull(Object object)` sichert zu, dass eine Objektreferenz nicht `null` ist:

  ```
  assertNotNull(store.getMovie(1));
  ```

- `assertSame(Object expected, Object actual)` sichert die *Identität* zweier Objekte zu:

  ```
  assertSame(movie, store.getMovie(1));
  ```

- `assertNotSame(Object expected, Object actual)` sichert zu, dass zwei Objekte nicht identisch sind:

  ```
  assertNotSame(Price.REGULAR, Price.NEWRELEASE);
  ```

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

3.8 »AssertionFailedError«

Was passiert, wenn ein Test fehlschlägt?

Fehlschläge

Die im Test durch `assert...`-Anweisungen kodierten Behauptungen werden von der Klasse `Assert` automatisch verifiziert. Im Fehlerfall bricht JUnit den laufenden Testfall sofort ab und wirft den Fehler `AssertionFailedError` mit entsprechendem Protokoll aus.

Sehen wir uns dazu ein Beispiel an. Um den Test des centweisen Rundens nachzuholen, könnten wir folgenden Testfall schreiben:

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

Der Test läuft natürlich nicht, weil unsere Klasse noch kein Konzept für das Runden hat. Der Fortschrittbalken verfärbt sich rot:

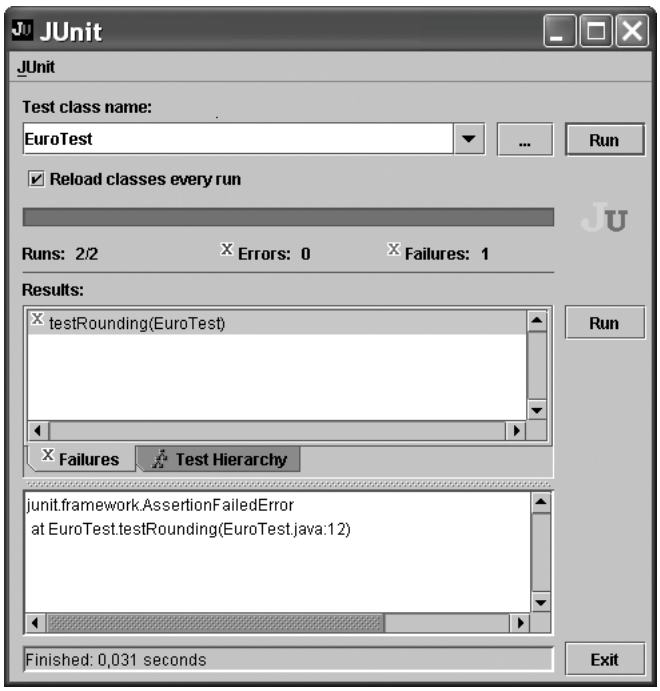

Abb. 3–3
Ein fehlschlagender Test ist ein wertvoller Test, da er uns mit wertvollen Informationen versorgt

Die JUnit-Oberfläche klärt uns sofort über den fehlschlagenden Test auf und protokolliert ihn zur Rückmeldung im unteren Textfenster.

»Der Testfall testRounding schlägt fehl«, sagt uns JUnit. »Werfen Sie einen Blick auf Zeile 12 der Klasse EuroTest.«

JUnit: *Failure*

```
junit.framework.AssertionFailedError
at EuroTest.testRounding(EuroTest.java:12)
```

Wenn Ihnen diese Fehlermeldung nicht ausreichen sollte, bietet Ihnen JUnit für alle assert...-Methoden an, optional einen Erklärungstext einzutippen, der im Fehlerfall mitprotokolliert wird. Ich rate Ihnen, diese Variante zur Dokumentation langer Testfälle auszunutzen:

```
assertTrue("amount not rounded", 2.00 == rounded.getAmount());
```

Der Unterschied liegt im ausdrucksstärkeren Begleittext, den JUnit im Fehlerfall zusätzlich ausgibt, statt nur den Ort des Fehlschlags:

JUnit: *Failure*

```
junit.framework.AssertionFailedError: amount not rounded
at EuroTest.testRounding(EuroTest.java:12)
```

Viel praktischer ist hier allerdings die assertEquals-Variante:

```
assertEquals("amount not rounded",
             2.00, rounded.getAmount(), 0.001);
```

Diese Zusicherung vergleicht erwartete und tatsächliche Werte:

JUnit: *Failure*

```
junit.framework.AssertionFailedError: amount not rounded
  expected:<2.0> but was:<1.995>
at EuroTest.testRounding(EuroTest.java:12)
```

Zum Vergleich der intern verwendeten Fließkommarepräsentation tolerieren wir in diesem Beispiel ein Delta von 0.001. Das bedeutet, dass die verglichenen Werte sich nicht um einen Betrag unterscheiden dürfen, der größer ist als ein Tausendstel Euro.

Mittlerweile wird es aber Zeit, den Test endlich einmal zu erfüllen. Was müssen wir also schreiben? Voilà!

JUnit: *OK*

```
public class Euro {
  private final long cents;

  public Euro(double euro) {
    cents = Math.round(euro * 100.0);
  }

  public double getAmount() {
    return cents / 100.0;
  }
}
```

3.9 »TestCase«

Wie gruppieren wir unsere Testfälle um eine gemeinsame Menge von Testobjekten?

Ein Testfall sieht in der Regel so aus, dass zunächst 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 Objekte getestet. Betrachten wir auch dazu ein Beispiel:

```
public class EuroTest...
  public void testAdding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", 4.00, sum.getAmount(), 0.001);
    assertEquals("two", 2.00, two.getAmount(), 0.001);
  }
}
```

In diesem Testfall erzeugen wir uns ein `Euro`-Objekt und addieren es mit sich selbst. Das Resultat dieser Addition soll ein `Euro`-Objekt sein, dessen Wert »vier Euro« beträgt. Unser ursprüngliches »zwei Euro«-Objekt soll durch die Addition nicht verändert werden. Um diesen Test zu erfüllen, verpassen wir unserer Klasse einen *privaten* Konstruktor:

JUnit: *OK*

```
public class Euro...
  private Euro(long cents) {
    this.cents = cents;
  }

  public Euro add(Euro other) {
    return new Euro(this.cents + other.cents);
  }
}
```

Der JUnit-Balken ist zurück auf Grün – doch was würde passieren, wenn wir hier ein `Euro`-Objekt mit negativem Betrag definiert hätten? Wäre damit noch die Intention des Addierens ausgedrückt oder sollte es dann besser `subtract` heißen? Verschieben wir die Antwort auf diese Frage für ein paar Momente. Damit die Idee aber nicht verloren geht, notieren wir uns »Wie geht `add()` mit negativen Beträgen um?«

Memo:
– add(): negative Beträge?

Halten Sie Ideen sofort fest, verlieren Sie sie nicht wieder.

Wir sind dabei stehen geblieben, eine geeignete Test-Fixture herauszuziehen. Werfen wir dazu noch einen Blick auf unsere drei Testfälle:

```
public class EuroTest...
  public void testAmount() {
    Euro two = new Euro(2.00);
    assertTrue(2.00 == two.getAmount());
  }

  public void testRounding() {
    Euro rounded = new Euro(1.995);
    assertEquals("amount not rounded",
                 2.00, rounded.getAmount(), 0.001);
  }

  public void testAdding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", 4.00, sum.getAmount(), 0.001);
    assertEquals("two", 2.00, two.getAmount(), 0.001);
  }
}
```

Da sich der Code zum Aufbau einer bestimmten Testumgebung häufig wiederholt, sollten Sie alle Testfälle, die gegen die gleiche Menge von Objekten laufen, unter dem Dach einer Testklasse zusammenfassen. Im ersten und dritten Testfall verwenden wir beispielsweise jeweils ein »zwei Euro«-Objekt. Wir könnten damit anfangen, dieses Objekt in die Test-Fixture zu übernehmen. Damit würden wir die duplizierten Testobjekte in einer gemeinsamen Testumgebung zusammenfassen und könnten so bestehende Testfälle vereinfachen. Der zweite Testfall hat keine Verwendung für unsere »zwei Euro«. Wir notieren uns also »`EuroTest: testRounding()` refaktorisieren« und kommen später auf die Notiz zurück.

Memo:
– add(): negative Beträge?
– EuroTest: testRounding() refaktorisieren

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 gutes 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 auch durchaus vorkommen, dass zu einer Klasse mehrere korrespondierende Testfallklassen existieren, von denen jede ihre individuelle Fixture besitzt.

Testklasse per Fixture

Organisieren Sie Testklassen um eine gemeinsame Fixture von Testobjekten, nicht um die getestete Klasse.

Damit fehlerhafte Testfälle nicht andere Testfälle beeinflussen können, läuft jeder Testfall im Kontext seiner eigenen Fixture. Eine Test-Fixture erhalten Sie auf folgende Weise:

- Erklären Sie die Objekte Ihrer Testumgebung zu Instanzvariablen.
- Initialisieren Sie diese Variablen in der `setUp`-Methode, um so eine definierte Testumgebung zu schaffen. *setUp()*
- Geben Sie wertvolle Testressourcen wie zum Beispiel Datenbank- oder Netzwerkverbindungen in der `tearDown`-Methode wieder frei. *tearDown()*

JUnit behandelt Ihre Testklasse dann wie folgt:

- `setUp` wird gerufen, bevor ein Testfall ausgeführt wird.
- `tearDown` wird gerufen, nachdem ein Testfall ausgeführt wurde.

Hier sehen Sie unsere Testklasse mit extrahierter Test-Fixture:

***JUnit:** OK*

```
public class EuroTest...
  private Euro two;

  protected void setUp() {
    two = new Euro(2.00);
  }

  public void testAmount() {
    Euro two = new Euro(2.00);
    assertTrue(2.00 == two.getAmount());
  }

  public void testRounding() {
    Euro rounded = new Euro(1.995);
    assertEquals("amount not rounded",
                 2.00, rounded.getAmount(), 0.001);
  }

  public void testAdding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", 4.00, sum.getAmount(), 0.001);
    assertEquals("two", 2.00, two.getAmount(), 0.001);
  }
}
```

Bitte beachten Sie, dass Sie Fixture-Variablen im `setUp` initialisieren, nicht etwa im Deklarationsteil oder im Konstruktor der Testfallklasse. Hätten wir in diesem Beispiel wertvolle Testressourcen freizugeben, würden wir symmetrisch zum `setUp` auch `tearDown` definieren.

3.10 Lebenszyklus eines Testfalls

Wie führt JUnit die Tests unserer Klasse aus? Im Prinzip unterscheidet JUnit zwischen zwei Phasen:

1. **Testfallerzeugung:** Das Framework durchsucht Ihre Testklasse nach `test...`-Methoden und erzeugt für sie jeweils ein eigenes Objekt der Klasse.
2. **Testlauf:** JUnit führt die gesammelten Testfälle voneinander isoliert aus. Die Reihenfolge, in der Testfälle vom Framework ausgeführt werden, ist dabei prinzipiell undefiniert.

Damit zwischen einzelnen Testfällen keinerlei Seiteneffekte entstehen, erzeugt JUnit für jeden Testfall ein neues Exemplar Ihrer Testklasse. Das ist überaus wichtig zu verstehen. Ins Bild gesetzt, sieht es so aus:

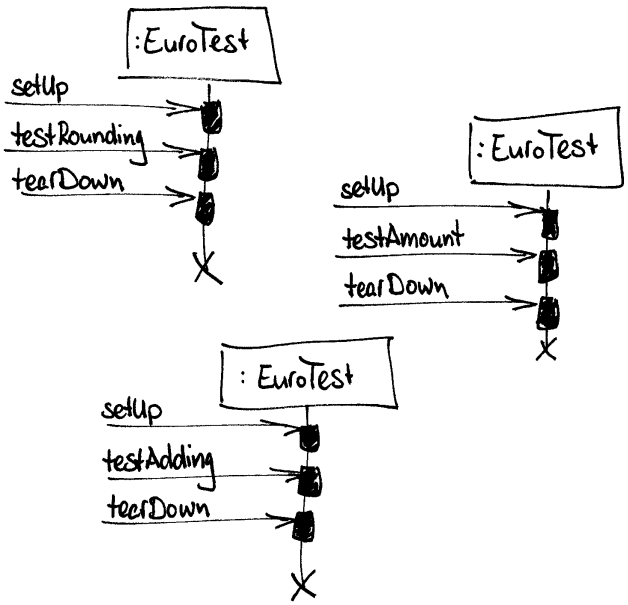

Abb. 3–4
Für jede Testfallmethode existiert ein Testfallobjekt

Testlaufsequenz

Ihre Testfallklasse resultiert in so vielen Testfallobjekten, wie sie Testfallmethoden enthält. Auf diesen Objekten erfolgt dann der Testlauf:

1. Vor der Ausführung eines Testfalls wird jeweils erst die `setUp`-Methode ausgeführt, sofern diese redefiniert wurde.
2. Anschließend wird die den auszuführenden Testfall betreffende `test...`-Methode gerufen.
3. Abschließend wird noch die `tearDown`-Methode ausgeführt, falls diese in der Testklasse redefiniert wurde.

Es sollte klar geworden sein, dass Ihre `TestCase`-Klasse wirklich eine *Sammlung* von Testfällen darstellt, denn erst die instanziierten Objekte dieser Klasse repräsentieren jeweils einen Testfall. Für JUnit sind diese Testfälle vollkommen unabhängig voneinander: Sie besitzen keinen inhärenten Zustand, bauen sich selbst ihre Testumgebung auf, räumen hinter sich selbst auf und können in beliebiger Reihenfolge laufen.

Zum effektiven Testen müssen Testfälle isoliert voneinander ausführbar sein. Treffen Sie deshalb keine Annahmen über ihre Reihenfolge im Testlauf. Führen Sie voneinander abhängige Tests stattdessen gemeinsam in einem Testfall aus.

Damit sind wir an einer Stelle angekommen, an der es sich lohnt, JUnit im größeren Bild zu betrachten:

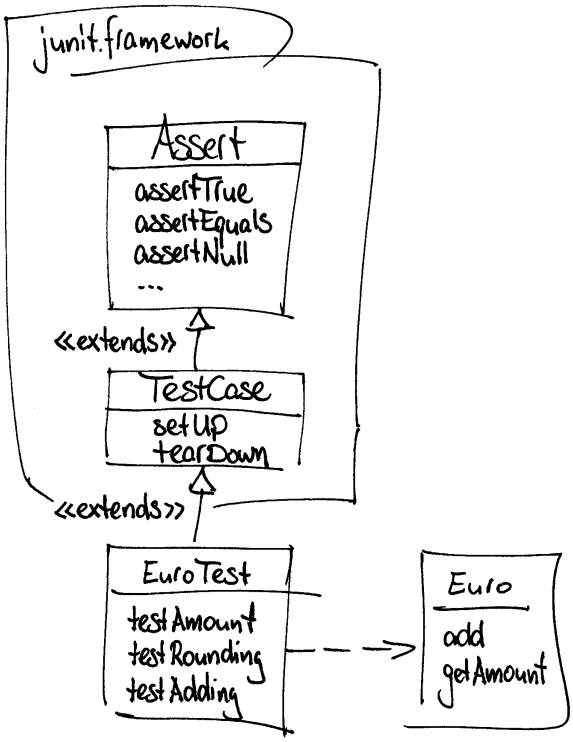

Abb. 3–5
Vererbungshierarchie von Assert und TestCase

- `Assert` testet Werte und Bedingungen.
- `TestCase` isoliert eine Reihe von Testfällen um eine gemeinsame Testumgebung.
- `EuroTest` testet das Verhalten unserer entwickelten `Euro`-Klasse.

3.11 »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 wiederholt durchführen können. Wichtig ist schließlich, die Unit Tests möglichst häufig auszuführen, idealerweise nach jedem Kompilieren. Wenn wir dann immer nur winzig kleine Schritte nehmen, werden wir uns nie länger als wenige Minuten mit Programmieren aufhalten, ohne grünes Licht zum Weiterprogrammieren einzuholen. Eher selten wollen wir dabei nur einzelne Testfälle ausführen. Meistens wollen wir alle gesammelten Unit Tests in einem Testlauf zusammen ausführen, um so ungewollten Seiteneffekten möglichst frühzeitig zu begegnen.

Führen Sie nach jedem erfolgreichen Kompiliervorgang alle gesammelten Unit Tests aus.

suite()

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

AllTests

Wollen wir beispielsweise alle bisher geschriebenen Testfälle in einer Testsuite organisieren, muss unsere Testsuiteklasse die folgende Gestalt haben:

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {
  public static Test suite() {
    TestSuite suite = new TestSuite();

    suite.addTestSuite(CustomerTest.class);
    suite.addTestSuite(EuroTest.class);
    suite.addTestSuite(MovieTest.class);

    return suite;
  }
}
```

Um eine Testsuite zu erhalten, definieren wir ein `TestSuite`-Exemplar und fügen diesem mithilfe der `addTestSuite`-Methode die gewünschten Testfallklassen hinzu. Für jede Testfallklasse wird dabei implizit eine Testsuite definiert, in der alle Testfallmethoden eingebunden werden, die in der betreffenden Klasse definiert sind.

Test

Sie werden vielleicht über den Rückgabetyp der `suite`-Methode verwundert sein: Warum `Test` und nicht `TestSuite`? Hier begegnen wir erstmalig dem `Test`-Interface, das sowohl von der `TestCase`- als auch von der `TestSuite`-Klasse implementiert wird. Dieses Entwurfsmuster ist als *Composite* [ga95] bekannt geworden. Das Muster erlaubt uns, beliebig viele `TestCase`- und `TestSuite`-Objekte zu einer umfassenden Testsuite-Hierarchie zu kombinieren:

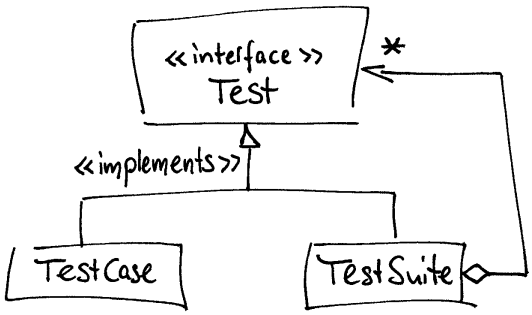

Abb. 3–6
Teil-und-Ganzes-Hierarchie von TestCase und TestSuite

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

Momentan liegen alle Klassen einfach im Default-Package. In der Regel verteilen wir unsere Klassen natürlich über verschiedene Pakete und müssen das Package beim Bündeln der Testsuite mit spezifizieren.

In den meisten Fällen ist es ohnehin praktisch, pro Package eine Testsuiteklasse zu definieren. Nach diesem Muster können Sie dann auch Hierarchien von Hierarchien von Testsuiten bilden:

```
suite.addTest(other.AllTests.suite());
```

Unsere `main`-Methode verschieben wir von der `EuroTest`-Testfallklasse in die `AllTests`-Testsuiteklasse, da sie hier viel besser aufgehoben ist:

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

3.12 »TestRunner«

Wie automatisieren wir den Test?

In einigen Entwicklungsumgebungen ist die JUnit-Integration so eng, dass Tests direkt ausgeführt werden können. Bietet Ihre IDE das nicht, können Sie zur Ausführung der Tests zwischen diesen drei Alternativen wählen:

- `junit.swingui.TestRunner` mit grafischer Swing-Oberfläche
- `junit.awtui.TestRunner` auf einfacher Basis des Java AWT
- `junit.textui.TestRunner` als Batch-Job auf der Textkonsole

Beide grafischen Varianten bieten den bekannten Fortschrittbalken und sind sich sehr ähnlich. Zur Ausführung können Sie Testfallklassen und Testsuiteklassen mit vollklassifiziertem Namen eingeben und unter Swing auch aus dem Klassenpfad oder Ihrer Historie auswählen.

ClassLoader-Problem

Die GUI-Oberflächen haben gegenüber der textuellen Darstellung den Vorteil, dass sie modifizierte Klassen automatisch neu nachladen. In unerfreulichen Fällen kollidiert JUnits eigener Klassenlader jedoch mit anderen und Sie müssen das Laden über die Checkbox »reload classes every run« unterbinden oder sich in der JUnit-Doku anlesen, was über die `excluded.properties`-Konfiguration geschrieben steht.

Im mittleren Fenster werden die fehlschlagenden Tests aufgelistet, die Sie nach Mausklick im unteren Fenster mit Fehlermeldung und Stacktrace unter die Lupe nehmen können. Unter Swing können Sie im mittleren Fenster zusätzlich den Testbaum öffnen und dann Testfälle und Testsuiten daraus auch einzeln ausführen.

Die rein textuelle Variante bietet eine Alternative zur grafischen Oberfläche, die besonders für den Testlauf zum Integrationszeitpunkt oder für einen nächtlichen Build geeignet ist:

```
public class AllTests...
  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

Die Benutzungsoberfläche sieht entsprechend sparsam aus:

```
........
Time: 0,06

OK (8 tests)
```

Für jeden erfolgreichen Test tickt ein weiterer Punkt auf die Konsole. Fehlschläge dagegen hinterlassen weniger freundlich ihren Stacktrace.

3.13 Zwei Methoden, die das Testen vereinfachen

Abschließend wollen wir auf eine der beiden Fragen zurückkommen, die wir im Zuge unserer Programmierepisode offen gelassen haben, um den angefangenen Schritt zu Ende zu führen.

Erinnern Sie sich noch an unseren Notizzetteleintrag »`EuroTest: testRounding()` refaktorisieren«? Wir hatten die Absicht, uns erneut den Testfall `testRounding` vorzuknöpfen, da er keinerlei Verwendung für die »zwei Euro« aus der Test-Fixture hatte. Ist das nun wirklich ein Anzeichen dafür, dass wir eine neue Testfallklasse abspalten sollten? Nein, wir können noch guten Gebrauch von den »zwei Euro« machen, wenn wir den Testfall umschreiben:

Memo:
– add(): negative Beträge?
– ~~EuroTest: testRounding() refaktorisieren~~

```
public class EuroTest...
  public void testRounding() {
    Euro rounded = new Euro(1.995);
    assertEquals("amount not rounded", two, rounded);
  }
}
```

Für ein richtiges Wertobjekt gehört es sich ohnehin, dass wir zwei Objekte mittels der `equals`-Methode auf Gleichheit testen können. Lassen Sie uns diese Intention deshalb auch in unserem restlichen Code ausdrücken:

```
public class EuroTest...
  public void testAdding() {
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Aber warten Sie! Unsere Tests fangen an, sich lautstark zu beschweren:

JUnit: *Failure*

```
junit.framework.AssertionFailedError:
  sum expected:<Euro@8b456289> but was:<Euro@8b496289>
at EuroTest.testAdding(EuroTest.java:23)

junit.framework.AssertionFailedError: amount not rounded
  expected:<Euro@18d18634> but was:<Euro@19318634>
at EuroTest.testRounding(EuroTest.java:18)
```

Was geht hier vor? Die Ausgabe der Objektreferenzen, die Java hier standardmäßig als Hashcode ausgibt, den Java wiederum als Resultat der `toString`-Methode liefert, hilft uns nicht weiter.

toString()

Überschreiben wir also einfach das Standardverhalten mit einer ausdrucksstärkeren `String`-Repräsentation:

```
public class Euro...
  public String toString() {
    return "EUR " + getAmount();
  }
}
```

Ungetesteter Code? Tja, diese Methode ist so einfach, dass ein Test daran meines Erachtens verschwendet wäre. Die Methode `getAmount` hat ja bereits einen Test und mehr kann hier im schlimmsten Fall nicht schief gehen.

Schreiben Sie nur Tests für solchen Code, der unter Umständen fehlschlagen könnte.

Testen wir jetzt, erhalten wir folgende Rückmeldung:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
  sum expected:<EUR 4.0> but was:<EUR 4.0>
at EuroTest.testAdding(EuroTest.java:23)

junit.framework.AssertionFailedError: amount not rounded
  expected:<EUR 2.0> but was:<EUR 2.0>
at EuroTest.testRounding(EuroTest.java:18)
```

Hmm, das wird ja immer seltsamer! Wissen Sie, was jetzt los ist?

Die `assertEquals`-Anweisung in den Tests greift intern auf die `equals`-Methode unseres Objekts zurück und die wiederum vergleicht zwei Objekte standardgemäß anhand ihrer Referenzen.

Damit unser `Euro`-Objekt als wirkliches Wertobjekt und nicht etwa als Referenzobjekt behandelt wird, müssen wir die `equals`-Methode überschreiben. Doch dafür schreiben wir zunächst wieder einen Test, denn hier will der *Equals-Vertrag* sichergestellt werden:

```
public class EuroTest...
  public void testEquality() {
    assertEquals(two, two);
    assertEquals(two, new Euro(2.00));
    assertEquals(new Euro(2.00), two);

    assertFalse(two.equals(new Euro(7.00)));
    assertFalse(two.equals(null));
    assertFalse(two.equals(new Object()));
  }
}
```

Die Implementierung der `equals`-Methode sieht so aus:

equals()

***JUnit:** OK*

```
public class Euro...
  public boolean equals(Object o) {
    if (o == null || !o.getClass().equals(this.getClass())) {
      return false;
    }

    Euro other = (Euro) o;
    return this.cents == other.cents;
  }
}
```

Unsere Tests laufen wieder.

Noch einmal nachdenken: Der *Hashcode-Vertrag* erwartet von uns, dass wir mit `equals` auch die `hashCode`-Methode redefinieren:

```
public class Euro...
  public int hashCode() {
    return (int) cents;
  }
}
```

Wenn unser Objekt niemals mit `HashMap` oder `HashSet` verwendet wird, könnten wir hier sogar eine `UnsupportedOperationException` werfen, doch der Hashcode ist in diesem einfachen Fall schneller getippt.

Wir schreiben hier zusätzliche Methoden, nur um unsere Klassen zu testen. Das ist interessanterweise nicht mal schlimm, wenn es unsere Klassen dahin lenkt, dass sie dadurch auch einfacher testbar sind.

Spendieren Sie Ihren Klassen zusätzliche öffentliche Methoden, wenn sie dadurch leichter getestet werden können.

Durch die Verwendung von `assertEquals` wandert hier lediglich ein Teil der Verantwortlichkeiten aus dem Testcode in den funktionalen Code. Das bedeutet, der Anwendungscode nimmt Komplexität auf, damit der Testcode Komplexität abgeben kann.

Wir haben jetzt schon neun Tests(!). Sicherlich können wir immer mehr Tests schreiben. Nach oben gibt es keine Grenzen. Wichtig ist nur, im Auge zu behalten, welche Tests ihren Aufwand wirklich wert sind und welche nicht.

Beobachten Sie Ihren Return-on-Investment. Schreiben Sie nur Tests, die wertvoll sind und aus denen Sie auch etwas lernen.

3.14 Testen von Exceptions

Wie testen wir, ob eine Exception wie erwartet geworfen wird?

Memo: ~~*– add(): negative Beträge?*~~

Ein anderer offener Punkt unserer Randnotizen war, »Wie geht `add()` mit negativen Beträgen um?« Sicherlich ist es schon gar nicht sinnvoll, negative Beträge im Konstruktor zu erlauben. Wir sollten ihm deshalb eine *Guard Clause* [be$_{96}$] spendieren und entsprechend eine Exception werfen:

JUnit: *OK*

```
public class EuroTest...
  public void testNegativeAmount() {
    try {
      final double NEGATIVE_AMOUNT = -2.00;
      new Euro(NEGATIVE_AMOUNT);
      fail("amount must not be negative");
    } catch (IllegalArgumentException expected) {
    }
  }
}
```

Das Muster von JUnit, um erwartete Exceptions und Fehler zu testen, ist denkbar einfach, nachdem es einmal klar geworden ist:

Wir versuchen, ein `Euro`-Exemplar mit negativem Wert zu bilden. Was wir im Fall negativer Eurobeträge erwarten würden, wäre eine `IllegalArgumentException`. Wenn der Konstruktor diese Exception also auslöst, kommt der `catch`-Block zum Tragen. Der Variablenname `expected` drückt schon aus, dass wir eine `IllegalArgumentException` erwarten. Wir fangen diese Exception und alles ist gut.

Wird die Exception dagegen nicht geworfen, läuft der Test zu weit, die `fail`-Anweisung wird ausgeführt und JUnit protokolliert unsere Fehlermeldung in einem `AssertionFailedError`-Objekt.

fail()

Die `fail`-Methode wird von JUnits `Assert`-Klasse realisiert und wird verwendet, um einen Testfall durch Fehlschlag abzubrechen.

Um obigen Testfall zu erfüllen, fügen wir eine Guard Clause im Konstruktor ein:

```
public class Euro...
  public Euro(double euro) {
    if (euro < 0)
      throw new IllegalArgumentException("negative amount");

    cents = Math.round(euro * 100.0);
  }
}
```

3.15 Unerwartete Exceptions

Was passiert mit Exceptions, die uns im Test nicht interessieren?

JUnit unterscheidet zwischen einem *Failure* und einem *Error*:

- ***Failure* kennzeichnet eine fehlschlagende Zusicherung**, die wir im Test sicherstellen wollen.
- ***Error* kennzeichnet ein unerwartetes Problem** wie zum Beispiel eine `NullPointerException`, die wir auch nicht behandeln wollen.

Wenn *Unchecked Exceptions* wie `RuntimeException`, `Error` und ihre Untertypen in eine `setUp`-, `test...`- oder `tearDown`-Methode gelangen, werden sie von JUnit gefangen und protokolliert.

Checked Exceptions müssen Sie in Ihrem Testcode nicht fangen, sondern können sie direkt an das Framework weiterreichen, indem Sie die betreffende Methode um eine `throws Exception`-Klausel erweitern.

Lassen Sie unerwartete Exceptions einfach von JUnit fangen, denn auch diese stellen Fehler dar.

Das Handling von Exceptions geschieht durch JUnit dann wie folgt:

- Tritt im `setUp` ein Fehler auf, wird sowohl der Testfall als auch `tearDown` nicht mehr ausgeführt.
- Tritt in der `test...`-Methode ein Fehler auf, wird trotzdem noch `tearDown` ausgeführt.

Das bedeutet, dass die Testumgebung, die Sie im `setUp` aufgebaut haben, in `tearDown` selbst dann noch ordnungsgemäß abgerissen wird, wenn der Test schief geht. Testressourcen werden somit auf jeden Fall wieder freigegeben, es sei denn, `setUp` schlägt schon fehl.

Woran erkennt man, dass etwas testgetrieben entwickelt wurde?

von Johannes Link, andrena objects ag

Daran, dass es uns auch leicht fällt, darauf aufbauende Software in kleinen isolierten Units zu testen. Negativbeispiele gibt es ausreichend. Beinahe alle komplexeren Java-APIs widersetzen sich dem Testen mehr oder weniger hartnäckig. Mit unseren selbst geschriebenen Bibliotheken und Frameworks haben wir das Problem praktisch nie. Wenn doch, dann zeigt sich bei näherem Hinsehen, dass es sich um eine Stelle handelt, die selbst nur sehr oberflächlich getestet wurde (und damit nicht wirklich test-first entwickelt wurde).

3.16 JUnit 4

Kurz vor Drucklegung dieses Buches steht JUnit in Version 4 knapp vor der Freigabe. Das neue Release soll die Einstiegshürde zum Testen weiter senken und dieses Versprechen wird mit Sicherheit gehalten. Das einfachste Test-Framework der Welt ist noch einfacher geworden.

... für Java 1.5

Da sich dessen Neuerungen jedoch komplett auf Java 1.5 beziehen und viele Projekte noch eine kleine Weile mit Java 1.4 auskommen müssen, gehe ich auf das neue JUnit nur in diesem Abschnitt ein und verwende ansonsten die Version 3.8.1. Wenn Sie später auf JUnit 4 umsatteln, können Sie Ihre alten JUnit-Tests wie gewohnt weiter benutzen.

Mit dem Versionssprung haben Kent Beck und Erich Gamma nach mittlerweile sieben Jahren zum ersten Mal die Architektur ihres Test-Frameworks grundlegend geändert und alle Story-Karten auf die mit

Annotationen

Java 1.5 eingeführten *Annotationen* gesetzt. Annotationen sind ein neues Ausdrucksmittel der Metaprogrammierung. Per Annotation können Sie Code mit frei wählbaren Anmerkungen versehen und auf die so markierten Codeelemente über den Reflection-Mechanismus später wieder zugreifen. In JUnit 4 wird dieses Sprachkonstrukt nun dazu verwendet, jede x-beliebige Methode jeder x-beliebigen Klasse als ausführbaren Testfall kennzeichnen zu können. Hier ist ein Test der neuen Schule:

```
~~import junit.framework.TestCase;~~

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

public class EuroTest ~~extends TestCase~~ {
  @Test public void ~~test~~adding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Sie erkennen, dass die Namenskonvention `public void test...()` wie auch das Ableiten der Klasse `TestCase` der Vergangenheit angehören. Sie kleben künftig einfach eine `@Test`-Annotation an Ihre Testfälle und können Ihre Methoden nennen, wie es Ihnen gerade gefällt. Betrachten wir aber die Neuigkeiten doch einmal mit dem Vergrößerungsglas – behalten Sie dabei jedoch im Hinterkopf, dass Unterschiede zwischen dieser Vorschau und dem tatsächlichen JUnit 4-Release möglich sind.

org.junit-Package

Mit JUnit 4 kommt ein neuer Namensraum: Im Package `org.junit` steckt der neue annotationsbasierte Code. Das `junit.framework`-Paket bleibt so weit bestehen und hat lediglich kleine Änderungen erfahren, um die Aufwärtskompatibilität herzustellen. Für Tests, die der neuen Schule folgen, benötigen wir von dem alten Zeugs jedoch nichts mehr:

```
~~import junit.framework.TestCase;~~
```

Stattdessen importieren wir jetzt die `@Test`-Annotation und Methoden der `Assert`-Klasse aus dem neuen JUnit-Package:

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

Statischer Import der assert...-Methoden

Falls Sie mit den neuen Sprachkonstrukten noch nicht vertraut sind: Ab Java 1.5 können Sie mithilfe von *statischen Imports* die statischen Methoden einer anderen Klasse einfach in den Namensraum Ihrer eigenen Klasse einblenden. Mit der Zeile:

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

... könnten wir beispielsweise die `assertEquals`-Methode importieren, als wäre sie eine `static`-Methode unserer Klasse. Der oben verwendete *-Joker holt einfach alle statischen Methoden auf einen Schwung.

Als Nächstes ist augenfällig, dass unsere Klasse nicht mehr von der Klasse `TestCase` abgeleitet ist. Ab sofort können Sie Ihre Tests nämlich in jede beliebige Klasse stecken. Die einzige Bedingung ist: Ihre Klasse muss über einen öffentlichen Defaultkonstruktor instanziierbar sein:

```
public class EuroTest ~~extends TestCase~~ {
```

@Test-Annotation

Welche Methoden als Testfälle auszuführen sind, markieren wir jetzt mithilfe der `@Test`-Annotation. Den Klammeraffen nicht vergessen! Welche Namen Sie Ihren Methoden geben, ist egal. Das `test...`-Präfix können Sie als Zeichen alter Tradition oder aus guter Konvention beibehalten oder es auch lassen. Einzige Bedingung: Ihre Methode muss öffentlich sein, darf keine Parameter und keinen Rückgabewert haben:

```
  @Test public void ~~test~~adding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Das ist, auf einer Seite zusammengefasst, was sich grob geändert hat.

JUnit 4 führt sechs unterschiedliche Annotationen ein:

- `@Test` kennzeichnet Methoden als *ausführbare* Testfälle.
- `@Before` und `@After` markieren Setup- bzw. Teardown-Aufgaben, die für *jeden* Testfall wiederholt werden sollen.
- `@BeforeClass` und `@AfterClass` markieren Setup- bzw. Teardown-Aufgaben, die nur *einmal* pro Testklasse ausgeführt werden sollen.
- `@Ignore` kennzeichnet temporär *nicht auszuführende* Testfälle.

@Before und @After

`setUp`- und `tearDown`-Methoden werden wie Testfälle via Annotation gekennzeichnet:

- `@Before`-Methoden werden vor jedem Testfall ausgeführt,
- `@After`-Methoden nach jedem Testfall.

Auch diese Methoden können beliebige Namen tragen, müssen nun aber *öffentlich* zugänglich sein, parameterlos und ohne Rückgabewert. Der Fixture-Aufbau erfolgt auf dem gewohnten Weg:

```
import org.junit.Before;

public class EuroTest {
  private Euro two;

  @Before public void setUp() {
    two = new Euro(2.00);
  }

  @Test public void amount() {
    assertEquals(2.00, two.getAmount(), 0.001);
  }

  @Test public void adding() {
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Neu ist, dass auch mehrere `@Before`- und `@After`-Methoden pro Klasse laufen können. Eine bestimmte Ausführungsreihenfolge wird dabei jedoch nicht zugesagt. Vererbte und nicht überschriebene Methoden werden in symmetrischer Weise geschachtelt gerufen:

- `@Before`-Methoden der Oberklasse vor denen der Unterklasse,
- `@After`-Methoden der Unterklasse vor denen der Oberklasse.

@BeforeClass und @AfterClass

Für kostspieligere Test-Setups, die nicht für jeden einzelnen Testfall neu aufgebaut und danach gleich wieder abgerissen werden können, existieren zwei weitere Annotationen:

- `@BeforeClass` läuft für jede Testklasse nur ein einziges Mal und noch vor allen `@Before`-Methoden,
- `@AfterClass` entsprechend für jede Testklasse nur einmal und zwar nach allen `@After`-Methoden.

Mehrere `@BeforeClass`- und `@AfterClass`-Annotationen pro Klasse sind zulässig. Die so markierten Methoden müssen aber *statisch* sein:

```
import org.junit.BeforeClass;
import org.junit.AfterClass;

public class EuroTest...
  @BeforeClass public static void veryExpensiveSetup() { ... }
  @AfterClass public static void releaseAllResources() { ... }
}
```

Die `@BeforeClass`-Methoden einer Oberklasse würden entsprechend noch vorher ausgeführt, ihre `@AfterClass`-Methoden anschließend.

Erwartete Exceptions

Zum Testen von Exceptions können Sie der `@Test`-Annotation über ihren optionalen Parameter `expected` mitteilen, dass die Ausführung Ihres Testfalls zu einer gezielten Exception führen soll:

```
public class EuroTest...
  @Test(expected = IllegalArgumentException.class)
  public void negativeAmount() {
    final double NEGATIVE_AMOUNT = -2.00;
    new Euro(NEGATIVE_AMOUNT); // should throw the exception
  }
}
```

Wird keine Exception geworfen oder eine Exception anderen Typs, schlägt dieser Testfall eben fehl.

Wenn Sie das Exception-Objekt in dem Test noch weiter unter die Lupe nehmen wollen, sollten Sie den altbekannten Weg über den `try/catch`-Block nehmen. Das Gleiche gilt, wenn Sie zusichern wollen, dass sich der Prüfling nach einer Exception noch in einem konsistenten Zustand befindet. Ansonsten ist diese Annotation sehr elegant.

Timeouts

Ein für Performanztests interessanter optionaler Parameter ist `timeout`. Geben Sie Ihrem Testfall mit auf die Reise, in welcher Zeitspanne von Millisekunden er laufen sollte. Überschreitet er sein Zeitlimit, wird er zwecks Fehlschlags *abgebrochen*:

```
public class EuroTest...
  @Test(timeout = 100)
  public void performanceTest() { ... }
}
```

@Ignore

Wenn Sie einen Test *kurzzeitig* außer Gefecht setzen wollen, können Sie das folgendermaßen tun:

```
import org.junit.Ignore;

public class EuroTest...
  @Ignore("not today")
  @Test(timeout = 100)
  public void performanceTest() { ... }
}
```

Der `@Ignore`-Kommentar darf, sollte aber niemals fehlen! Der Test wird dann im Testlauf unter Protokollierung dieses Textes übergangen. Sorgen Sie jedoch dafür, dass ignorierte Tests schnellstens auf Grün kommen und im besten Fall gar nicht erst eingecheckt werden können!

Neue Tests mit altem Runner ausführen

Damit die existierenden und zum Teil in die Entwicklungsumgebungen direkt integrierten `TestRunner` unsere neuen Tests ausführen können, müssen wir einen kleinen Kunstgriff vornehmen:

```
import junit.framework.JUnit4TestAdapter;

public class EuroTest...
  public static junit.framework.Test suite() {
    return new JUnit4TestAdapter(EuroTest.class);
  }
}
```

Etwas unschön muss uns die gute alte `suite`-Methode als Adapterfunktion herhalten, zumindest bis unsere Werkzeuge die Brücke zur neuen annotationsbasierten Form geschlagen haben.

Alte Tests mit neuem Runner ausführen

JUnit 4 liefert nur noch einen textuellen Runner mit. Die grafische Integration wird konsequent den Werkzeugherstellern überlassen. Ebenso wandert die Organisation von Testfällen zu den Entwicklungswerkzeugen über. Der neue Runner `JUnitCore` akzeptiert tatsächlich nur noch ein flaches Array von Testklassen:

```
import org.junit.runner.JUnitCore;

public class AllTests {
  public static void main(String[] args) {
    JUnitCore.run(CustomerTest.class,
                  EuroTest.class,
                  MovieTest.class);
  }
}
```

Testfallerzeugung und Testlauf

Erwähnenswert ist noch, dass JUnit 4 nicht mehr in zwei Phasen läuft: Es werden also nicht erst alle Testfallobjekte auf Halde erzeugt und dann ausgeführt. Die Erzeugung erfolgt *just-in-time* zum Testlauf.

Schlüsselwort assert

Wenn Sie mögen, können Sie in Ihren Tests jetzt auch das Schlüsselwort `assert` aus Java 1.4 verwenden. Sie müssen nur daran denken, die Zusicherungen zur Laufzeit auch mit der Option `-ea` zu aktivieren. JUnit ist dazu intern zur Verwendung von `AssertionError` gewechselt, was auch dazu geführt hat, dass nicht mehr zwischen möglichen und unerwarteten Fehlschlägen, *Failures* und *Errors*, unterschieden wird. Fehler sind Fehler sind Fehler!

Subtile Unterschiede existieren zwischen altem und neuem `Assert`: `junit.framework.Assert` vs. `org.junit.Assert`. Unter Java 1.5 sollten Sie nur noch die neue Klasse einsetzen. Da Arrays untereinander nun auch über `equals` vergleichbar sind, werden Sie sogar mit einer neuen `assert`-Methode beschenkt:

```
assertEquals(Object[] expected, Object[] actual)
```

JUnit 3.8.1 spuckt bei `String`-Vergleichen mit `assertEquals` manchmal nahezu unbrauchbare Fehlermeldungen aus. Dieser Bug ist mit dem neuen Release behoben.

Umstieg auf JUnit 4

Der Umstieg zur neuen Schule ist leicht. Tests der alten Schule müssen nicht zwangsläufig auf Annotationen umgestellt werden, sie können friedlich weiterexistieren. Wichtig ist lediglich, dass Sie die alte und neue Form nicht in einer Testklasse mischen können.

Das reicht erst einmal. Genug Rüstzeug gesammelt – tauchen wir nun tiefer in die Testgetriebene Entwicklung ein ...