Kapitel 6
Erst mal testen

Sicher haben Sie schon vom Paradigma der *testgetriebenen Entwicklung* gehört (Test Driven Development, kurz: TDD). Aber wann und warum ist es eigentlich sinnvoll, zuerst den Test zu schreiben und dann den Code? Ist es überhaupt notwendig, den *gesamten* Code zu testen? Ist das nicht viel zu viel Aufwand? Sind bestimmte Codeteile überhaupt testbar? Wie testet man eine Funktion, die von externen Parametern oder gar Zufallswerten abhängt (wie z. B. dem Wetter)?

Es ist klar zu unterscheiden zwischen *Modultests* auf der untersten Ebene und *Integrationstests*, die den Gesamtzusammenhang betreffen. Sie benötigen letztlich beides, aber ohne Modultests brauchen Sie streng genommen mit den Integrationstests gar nicht erst anzufangen: Alles Mögliche könnte schiefgehen und Sie könnten unter Umständen die Ursache nicht leicht lokalisieren.

Eines steht fest: Mit ordentlichen Tests finden Sie Fehler, die Sie sonst übersehen. Außerdem dauern manuelle Tests viel länger als automatisch ablaufende. Wie Sie Ihren Code möglichst effizient testen können und wie sie gegebenenfalls schon vorhandene Tests verbessern können, erklärt Ihnen dieses Kapitel.

6.1 Gute und schlechte Unit-Tests

Unit-Tests testen jeweils genau ein Modul.

Ein Modultest oder Unit-Test soll einzelne Komponenten einer Software auf ihre korrekte Funktion hin überprüfen.

Eine »einzelne Komponente« ist zumeist eine Funktion einer Klasse – und diese Anforderung verrät Ihnen schon, dass Unit-Tests nur sinnvoll einsetzbar sind, wenn Ihre Software stark modularisiert ist. Klassen sollten möglichst unabhängig voneinander funktionieren, sonst betrifft Modultest A plötzlich auch Modul B.

Keinesfalls sollten Ihre Tests vorhandene Daten verändern, etwa in einer Datenbank, oder auf die Existenz solcher Daten angewiesen sein. Bestimmte knifflige Fälle zu diesem Thema finden Sie im weiteren Verlauf dieses Kapitels. Ganz gewiss sollten Unit-Tests nicht zweckentfremdet werden, um etwa bequem eine Datenbank zu initialisieren oder eine Applikation zu starten (Sie ahnen sicher, dass ich das nicht ohne Grund erwähne …).

Ferner sollten Unit-Tests automatisch ausführbar sein, also ohne geschultes Personal. Die einzige Anforderung an das Personal (oder eine installierte Automatik) ist es, den Ausführen-Knopf zu drücken. Nach einer gewissen Wartezeit sind entweder alle Tests erfolgreich durchgelaufen oder auch nicht. Weder muss das Personal Voreinstellungen vornehmen noch Zeilen ein- oder auskommentieren, um verschiedene Testfälle zu berücksichtigen (auch dies erwähne ich nicht ohne Grund).

Wenn ich im nächsten Abschnitt mit einem extrem einfachen Beispiel beginne, schütteln Sie bitte nicht den Kopf: Ich habe genug Projekte kennengelernt, in dem selbst einfachste Testfälle fehlten oder falsch aufgebaut waren. Ein genauer Blick kann daher nicht schaden.

6.1.1 Einfache Unit-Tests

Grundsätzlich sind Unit-Tests, egal ob in Java, C# oder einer anderen Sprache, simple Klassen mit zumeist mehreren parameterlosen Funktionen. Üblicherweise ist eine Unit-Test-Klasse dafür zuständig, eine Klasse Ihrer Software zu testen. Bei Java sind die Package-Namen von Klasse und Testklasse identisch, nur liegen Letztere in einem separaten Sourcecode-Verzeichnis. Die übliche Verzeichnisstruktur sieht wie folgt aus:

project

src/main/java

src/test/java

Auf diese Weise sind Produktiv- und Testcode völlig voneinander getrennt. Ihre Applikation enthält überhaupt keinen Testcode, wenn Sie sie veröffentlichen. Wozu auch? Für den Test sind Sie zuständig und nicht die Endanwenderinnen und -anwender (ein häufiges und bedauerliches Missverständnis).

Jede Funktion in der Testklasse testet üblicherweise eine Funktion der zu testenden Klasse.

Zeit für ein Beispiel!

Sie haben ein Fußballspiel programmiert. Darin gibt es unter anderem eine Model-Klasse, die ein einzelnes Spiel darstellt:

```
public class Match {
  private int goals1, goals2;
  private Team team1, team2, winner, loser;
  private boolean played;
  public Match(Team team1, Team team2) {
    this.team1 = team1;
    this.team2 = team2;
    goals1 = goals2 = 0;
    played = false;
  }
  ...
}
```

Listing 6.1 Die Model-Klasse »Match« verfügt über einen Konstruktor für zu spielende Paarungen. Ferner verfügt sie über getter und setter, die hier der Übersicht halber weggelassen wurden.

Um sicherzugehen, dass der Konstruktor korrekt funktioniert, können Sie einen Unit-Test schreiben.

Wenn Sie Eclipse verwenden, nimmt Ihnen ein Wizard die Arbeit ab. Wählen Sie im Menü NEW… – JUNIT TEST CASE. Wenn Ihre `Match`-Klasse gerade im Editor geöffnet ist, schlägt Ihnen der Wizard vor, einen Test namens `MatchTest` zu erzeugen (siehe Abbildung 6.1). Achten Sie darauf, dass das richtige Source-Verzeichnis eingestellt ist.

Wenn Sie im ersten Schritt des Wizards auf FINISH drücken, erzeugt er nur eine leere Klasse. Drücken Sie stattdessen NEXT, können Sie auswählen, welche Funktionen Sie testen möchten (siehe Abbildung 6.2). Für dieses Beispiel wähle ich den Konstruktor.

Abbildung 6.1 Der Eclipse Wizard erzeugt den JUnit Test Case auf Knopfdruck.

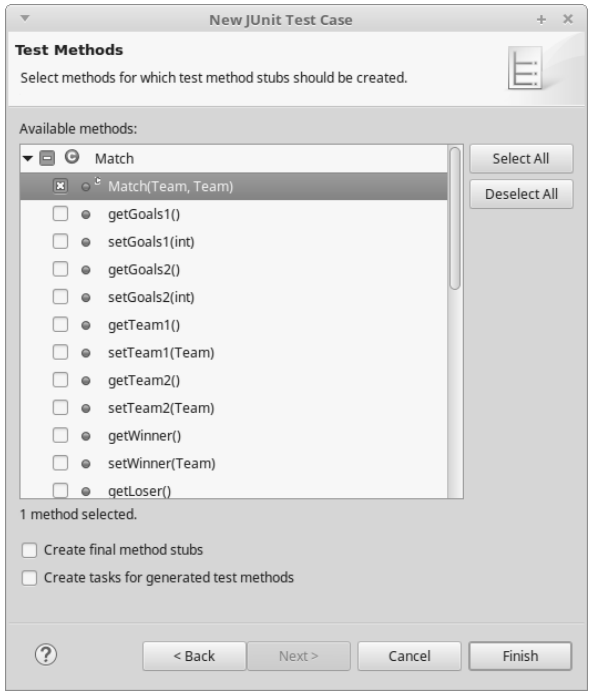

Abbildung 6.2 Der Wizard erzeugt auf Wunsch Funktionsrümpfe für die zu testenden Funktionen.

Das Resultat ist ein fehlschlagender Unit-Test:

```
package de.uwepost.java.unittestdemo;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
public class MatchTest {
  @Before
  public void setUp() throws Exception {
  }
  @Test
  public void testMatch() {
    fail("Not yet implemented");
  }
}
```

Listing 6.2 Der automatisch erzeugte Unit-Test schlägt natürlich fehl. Fatal wäre es, wenn er durchlaufen würde. Manche Programmiererinnen und Programmierer wären dann nämlich an dieser Stelle fertig: Alle Tests sind »grün«!

Grundsätzlich arbeitet JUnit mit Annotations. So erhält die eigentliche Testfunktion für den `Match`-Konstruktor den Vermerk `@Test`, um ihn ausführbar zu machen. Des Weiteren gibt es eine Methode `setUp()`, die mit `@Before` annotiert ist. Diese Methode wird vor jedem einzelnen Test aufgerufen. Sie können dort Daten initialisieren, die für mehrere Tests erforderlich sind. Tests dürfen diese Daten verändern, denn `setUp()` (bzw. jede mit `@Before` annotierte Methode) wird vor jedem einzelnen Test erneut aufgerufen. Verwenden Sie also keinesfalls einen Konstruktor der Testklasse, um Testdaten vorzubereiten!

Die Testmethode schlägt automatisch fehl, solange Sie sie nicht ersetzen. Um sich davon zu überzeugen, können Sie in Eclipse mit der rechten Maustaste auf eine Testmethode klicken und Run as... JUnit Test wählen. Führen Sie den Rechtsklick auf dem Klassennamen aus, werden alle Methoden in der Klasse durchlaufen, allerdings nur bis zum ersten Fehlschlag. Das JUnit-Fenster zeigt Ihnen die Resultate und ermöglicht es per Klick, einen Test erneut auszuführen – alle oder alle zuvor fehlgeschlagenen (siehe Abbildung 6.3).

Es ist an der Zeit, den tatsächlichen Test zu schreiben. Um den `Match`-Konstruktor aufzurufen, benötigen Sie `Team`-Objekte, die Sie in diesem einfachen Beispiel einfach so erzeugen können. In der realen Anwendung kämen sie vermutlich aus einer Datenbank und müssten gewisse Eigenschaften erfüllen (z. B. eine eindeutige ID besitzen). Verwenden Sie auf kei-

nen Fall Objekte aus einer solchen Datenbank! Wie Sie reale Objekte imitieren können, erkläre ich ausführlich in Abschnitt 6.4.

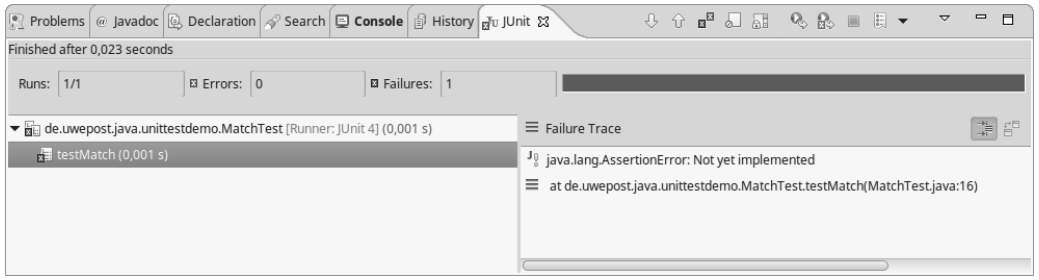

Abbildung 6.3 Das JUnit-Fenster quittiert Ihnen nicht nur alle Resultate, sondern misst auch die Laufzeit.

Da Sie möglicherweise für spätere Tests ebenfalls `Team`-Objekte benötigen, erstellen Sie sie ruhig in `setUp()`:

```
private Team team1,team2;
@Before
public void setUp() throws Exception {
  team1 = new Team();
  team2 = new Team();
}
```

Listing 6.3 Erzeugen Sie für Tests erforderliche Objekte keinesfalls im Konstruktor der Testklasse, sondern in »setUp()«.

Beachten Sie, dass Sie in dieser Funktion nicht den Konstruktor der Klasse `Team` testen, dafür ist ein anderer Test zuständig.

Jetzt können Sie den Test schreiben:

```
@Test
public void testMatch() {
  Match match = new Match(team1,team2);
  assertEquals(team1, match.getTeam1());
  assertEquals(team2, match.getTeam2());
}
```

Listing 6.4 Dieser Test des »Match«-Konstruktors ist noch nicht fertig.

Es gehört möglicherweise etwas Fantasie dazu, den Sinn in einem solch einfachen Test zu sehen. Aber ich bin sicher, Ihnen sind schon Tippfehler untergekommen, die ein so simpler Test gefunden hätte.

Gar nicht so selten passieren Copy-and-paste-Fehler.

Halten Sie sich die beiden hier getesteten Zeilen der Klasse `Match` noch mal vor Augen:

```
this.team1 = team1;
this.team2 = team2;
```

Fragen Sie sich jetzt einmal selbst: Würden Sie die erste Zeile mit Copy-and-paste kopieren, um die zweite zu erzeugen? Wahrscheinlich. Wir Programmiererinnen und Programmierer sind tippfauler als ein Goldfisch, nicht wahr?

In dem Moment, in dem Sie Strg+V drücken, piept Ihr Telefon. Breaking News: Ihr Lieblingsverein hat den Trainer gefeuert! Das muss sofort in der Teeküche diskutiert werden. Sie machen aber noch schnell aus der einen 1 eine 2 ...

```
this.team1 = team1;
this.team1 = team2;
```

Als Sie an Ihren Platz zurückkehren, schreiben Sie die nächste Funktion.

Wie gut, dass es einen Unit-Test gibt, nicht wahr?

Was die anderen Initialisierungen des Konstruktors angeht, können Sie sich beruhigt zurücklehnen:

```
goals1 = goals2 = 0;
played = false;
```

Tatsächlich sind nicht initialisierte `ints` in Java immer `0` und `booleans` immer `false`, sodass diese Zeilen an sich unnötig sind und entsprechende `asserts` im Unit-Test ebenso. In anderen Sprachen, etwa in C, ist das aber unter Umständen anders. Belassen wir es aber für den Moment dabei und testen lieber eine Methode, die echte Geschäftslogik implementiert.

6.1.2 Whitebox-Tests

Lassen Sie uns nun einen etwas schwierigeren Test für unser Fußballspiel schreiben.

Nach jedem Spiel muss festgestellt werden, wer der Gewinner ist. Da es sich hierbei um eine überraschend komplexe Logik handelt (z. B. wenn eine Europapokal-Auswärtsregel angewendet werden muss), gehört diese keinesfalls in die Model-Klasse `Match`, sondern in eine Dienstklasse `MatchResultAnalyzer`:

```
public class MatchResultAnalyzer {
  public void determineWinner(Match match) {
```

```
    if(match.isPlayed())
      match.setWinner( match.getGoals1()>match.getGoals2() ?
        match.getTeam1() : match.getTeam2() );
  }
}
```

Listing 6.5 Die Funktion »determineWinner()« in »MatchResultAnalyzer« ist dafür zuständig, den Gewinner eines Spiels zu ermitteln und ins »Match«-Objekt zu schreiben.

Wenn Sie in der obigen Implementierung einen Fehler gefunden haben, behalten Sie diesen zunächst im Hinterkopf. Schreiben Sie jetzt erst einmal den Test:

```
public class MatchResultAnalyzerTest {
  private MatchResultAnalyzer analyzer;
  @Before
  public void setUp() throws Exception {
    analyzer = new MatchResultAnalyzer();
  }
  @Test
  public void testDetermineWinner() {
    Team team1 = new Team();
    Team team2 = new Team();
    Match match = new Match(team1,team2);

    match.setFinalResult(2,1);
    analyzer.determineWinner(match);
    assertEquals(team1, match.getWinner());
  }
}
```

Listing 6.6 Dieser Test für »determineWinner()« funktioniert wirklich prima. Besser gesagt: Er zeigt keinen Fehler.

Die Methode `match.setFinalResult()` ist eine Abkürzung für den Aufruf der beiden setter für `goals1` und `goals2` und setzt gleichzeitig `played` auf `true`:

```
public void setFinalResult(int goals1,int goals2) {
  this.goals1=goals1;
  this.goals2=goals2;
  played=true;
}
```

Listing 6.7 Dieser Mehrfach-setter verkürzt das Setzen des Endergebnisses auf einen Aufruf – und sollte in einem separaten Test selbst getestet werden.

Beachten Sie bei diesem Test zunächst drei Dinge:

- Die zu testende Klasse wird in `setUp()` instanziiert.
- Der Konstruktor `Match()` wird hier nicht erneut getestet.
- Auch dieser Test erzeugt Eingabewerte, ruft die zu testende Funktion auf und prüft das Ergebnis.

Auch Tests zu programmieren, erfordert volle Konzentration.

Vor allem aber erkennen Sie an diesem Beispiel, dass Sie gut daran tun, Tests nicht mal eben so nebenbei kurz vor Feierabend (oder am Montagmorgen als Allererstes) zu schreiben: Genau wie die eigentliche Funktion sollte auch der Test mit angeschaltetem Gehirn geschrieben werden. Sie müssen jede Reaktion der zu testenden Funktion auf jede mögliche Eingabe prüfen, selbst wenn Sie glauben, dass sie ganz bestimmt nicht mit unsinnigen Parametern aufgerufen wird.

Ein »grüner« Test ist nur sinnvoll, wenn die gesamte Geschäftslogik überprüft wird. Wie Sie diese sogenannte *Testabdeckung* messen können, erkläre ich im übernächsten Abschnitt.

Also, was passiert eigentlich, wenn das Ergebnis überhaupt nicht gesetzt wurde?

```
Match match = new Match(team1,team2);
analyzer.determineWinner(match);
assertNull(match.getWinner());
```

Ohne Ergebnis sollte der Gewinner nicht feststehen, also `null` sein.

Klappt auch! Alles wunderbar. Oder?

Nein, keineswegs. Es genügt fast nie, nur das »positive« Verhalten einer Methode zu testen. Sie sollten nicht nur prüfen, was sie tut, sondern auch, was sie *nicht* tut. Achten Sie darauf, dass Sie bei veränderlichen Objekten wie `Match` immer ein frisches Objekt erzeugen:

Testen Sie mal ein Unentschieden:

```
match = new Match(team1,team2);
match.setFinalResult(1,1);
analyzer.determineWinner(match);
assertEquals(null, match.getWinner());
```

Listing 6.8 Testen Sie jeden denkbaren Fall, auch ein Unentschieden.

Natürlich schlägt dieser Test fehl: Den Bug in der Methode `determineWinner()` haben Sie ja sicher vorhin schon gefunden. Ein Unentschieden wird dort nämlich überhaupt nicht berücksichtigt.

Beim Versuch, den Bug zu beheben, stellen Sie hoffentlich fest, dass so ein `?:`-Operator verdammt unübersichtlich ist, spätestens wenn man zwei davon verschachtelt und mit einer `if`-Konstruktion zwar mehr Zeilen produziert, aber der dunklen Seite der Macht mal gehörig eins auswischt:

```
public void determineWinner(Match match) {
  if(match.isPlayed()) {
    if(match.getGoals1() > match.getGoals2())
        match.setWinner(match.getTeam1());
    else if(match.getGoals2() > match.getGoals1())
        match.setWinner(match.getTeam2());
  }
}
```

Listing 6.9 Eine fehlerfreie Implementierung von »determineWinner()«.

Das sieht doch schon viel besser aus. Es gibt allerdings noch einen Fall zu berücksichtigen.

Im Gegensatz zu anderen Sprachen (wie z. B. Kotlin) ist Java anfällig für `null`-Werte.

Ergänzen Sie Ihren Test wie folgt:

```
analyzer.determineWinner(null);
```

Diese Zeile wirft eine `NullPointerException`, weil die zu testende Methode ungeprüft auf den getter des übergebenen Objekts zugreift.

Bitte kommen Sie jetzt nicht auf die Idee, diese Exception zu fangen und in dem Fall den Test fehlschlagen zu lassen:

```
try {
  analyzer.determineWinner(null);
} catch(NullPointerException npe) {
  Assert.fail();
}
```

Listing 6.10 So bitte nicht!

Der Aufruf lässt den Test ohnehin fehlschlagen, es erscheint der zugehörige Stacktrace – das `Assert.fail()`, das den Fehlschlag eigens provoziert, ist (insbesondere ohne Nachrichttext als Parameter) völlig überflüssig.

Falls eine Funktion eine Unchecked Exception werfen könnte, fangen Sie sie trotzdem nicht, sondern ergänzen Sie `throws Exception` an der Testfunktion. Wenn `determineWinner()` beispielsweise eine von Ihnen definierte `MatchUnplayedException` wirft, solange das Spiel noch gar nicht gespielt wurde, fordert Java, dass Sie sie behandeln oder weiter nach oben durchreichen, also tun Sie genau das:


```
public void testDetermineWinner() throws Exception { ...
```

Wenn wirklich eine solche Ausnahme auftritt, wird der Test fehlschlagen und den zugehörigen Stacktrace anzeigen.

Ausnahmen erfordern keine Testfälle.

Beachten Sie, dass Exceptions immer Ausnahmebehandlungen sind und keine Geschäftslogik implementieren. Deshalb müssen Sie auch normalerweise nicht testen, ob eine Funktion wirklich eine bestimmte Exception wirft. Unit-Tests testen Geschäftslogik. Ausnahmen hingegen enthalten im Stacktrace ihre Ursache. Wenn sie nicht verschluckt oder ignoriert werden, ist es meist leicht, die Ursache zu erkennen und zu beseitigen.

Im vorliegenden Fall sollte entweder die Methode `determineResult()` nichts tun, wenn der Parameter `null` ist (ähnlich wie die `StringUtils` in Apache Commons), oder Sie sollten diesen Fall überhaupt nicht testen, weil er eben nicht zur Geschäftslogik gehört, sondern nur bei einem krassen Programmierfehler vorkommen kann.

6.1.3 Ping-Pong

Unit-Tests sind ein perfektes Beispiel für die Notwendigkeit guter Teamarbeit.

Um eine Funktion korrekt zu implementieren, müssen Sie die geforderte Geschäftslogik begriffen haben. Dann können Sie den zugehörigen Test auf Basis des gleichen Wissens schreiben.

Was aber, wenn Sie bei der Definition der Geschäftslogik eine Winzigkeit falsch verstanden haben? Dann schreiben Sie auch den Test falsch, d. h., er zeigt »grün«, obwohl die Logik nicht so implementiert ist wie eigentlich gewünscht.

Um bei dem Beispiel mit dem Fußballspiel zu bleiben: Sicher gibt es neben der Methode `setFinalResult()` auch eine, um ein Zwischenergebnis zu speichern:

```
public void setResult(int goals1,int goals2) {
  this.goals1=goals1;
  this.goals2=goals2;
}
```

Listing 6.11 Die Methode »setResult()« speichert ein Zwischenergebnis, ohne das Spiel zu beenden.

Die Methode `setResult()` setzt also nicht `Match.played` auf `true`.

Würden Sie jetzt den `MatchResultAnalyzer` und seinen Test entsprechend implementieren, ohne dass Ihnen der Unterschied klar ist, könnte das Ergebnis wie folgt aussehen:

```
match = new Match(team1,team2);
match.setResult(2,1);
analyzer.determineWinner(match);
assertEquals(team1, match.getWinner());
```

Listing 6.12 Dieser Test verwendet irrtümlich die falsche »set«-Methode …

Dementsprechend würden Sie auch `determineWinner()` ohne das `if()` schreiben:

```
public void determineWinner(Match match) {
  if(match.getGoals1()>match.getGoals2())
    match.setWinner(match.getTeam1());
  else if(match.getGoals2()>match.getGoals1())
    match.setWinner(match.getTeam2());
}
```

Listing 6.13 … aber mit der so implementierten Methode ist der Test »grün«.

Ihr Test würde durchlaufen, Sie wären fertig. Und hätten dennoch einen Fehler eingebaut, über dessen Auswirkungen wir jetzt nicht spekulieren müssen.

Außerdem kann Ihnen auch in einem Testfall ein simpler Tippfehler unterlaufen, der einen Test versehentlich durchlaufen lässt.

Der entscheidende Punkt ist: Tests und Implementierung sollten am besten von zwei verschiedenen Entwicklerinnen oder Entwicklern geschrieben werden. Zumindest aber erfordern Tests ebenso ein Vieraugenprinzip wie die Implementierung: Der »grünste« Test hilft nichts, wenn er nicht die geforderte Logik testet, sondern die tatsächlich implementierte.

Sie sehen: Unit-Tests zu schreiben macht vielleicht nicht unbedingt den größten Spaß, aber es ist keinesfalls eine langweilige Arbeit, bei der Sie den Kopf nicht benötigen. Vielmehr erfordert es Ihre ganze Aufmerksamkeit, gute Tests zu schreiben, auf die sich das Team hinterher auch verlassen kann.

6.1.4 Testabdeckung

Es ist kein Geheimnis, dass Unit-Tests oft etwas vernachlässigt werden. Wenn Sie in ein Projekt involviert sind, auf das dies zutrifft, können Sie

versuchen, nach und nach die Anzahl der Tests zu erhöhen. Allerdings ist die pure Anzahl von Tests natürlich kein besonders geeignetes Maß – das ist vielmehr die *Testabdeckung*, also der prozentuale Anteil aller Codezeilen, die bei Unit-Tests durchlaufen werden.

Im Beispiel mit dem `MatchResultAnalyzer` wäre beispielsweise die Testabdeckung um eine Zeile geringer, wenn Sie auf den Test eines Auswärtssieges verzichteten:

```
match = new Match(team1,team2);
match.setFinalResult(0,2);
analyzer.determineWinner(match);
assertEquals(team2, match.getWinner());
```

Listing 6.14 Vergessen Sie nicht den Testfall »Auswärtssieg«.

Um die Testabdeckung zu messen, ist es erforderlich, den Code vor dem Ablauf des Unit-Tests zu instrumentieren, d. h. mit zusätzlichem Code zu versehen, der für jede durchlaufene Zeile zählt. Natürlich gibt es Tools dafür, die Ihnen diese Arbeit abnehmen. Beispiele habe ich in folgendem Kasten in einer kleinen Liste zusammengestellt (ohne Anspruch auf Vollständigkeit).

Tools zum Messen der Testabdeckung

EclEmma für Java und Eclipse (Open Source): *https://www.eclemma.org*

Cobertura für Java (Open Source): *https://github.com/cobertura/cobertura*

Clover für Java und Groovy (Open Source): *https://www.atlassian.com/de/software/clover*

gcov für C/C++ (Teil der GNU Compiler Collection, kurz: GCC)

OpenCover für .NET 2 und höher – außer MONO (Open Source): *https://github.com/sawilde/opencover*

JetBrains dotCover für .NET: *https://www.jetbrains.com/dotcover*

Testabdeckung in Java messen mit EclEmma

Um beispielsweise *EclEmma* auszuprobieren, können Sie einfach in Eclipse das Plug-in vom Marketplace installieren.

Starten Sie danach einen Unit-Test mit Run • Coverage As • JUnit Test, um EclEmma dazu zu bringen, die Abdeckung zu messen.

Die Tests werden ausgeführt, und Sie erhalten als Resultat ein Baumdiagramm, in dem Sie genau ablesen können, welche Codeteile durchlaufen wurden (siehe Abbildung 6.4).

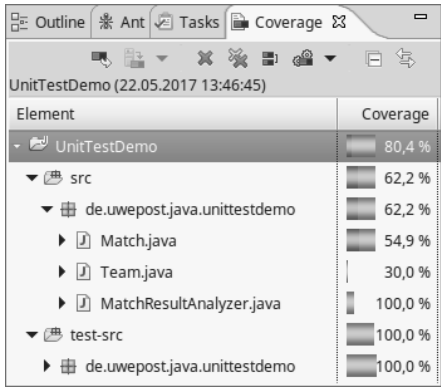

Abbildung 6.4 EclEmma zeigt Ihnen für jede Codedatei die Testabdeckung.

Natürlich ist die Testabdeckung nicht sonderlich hoch, denn in der Klasse `Match` gibt es eine ganze Menge getter und setter, die nicht verwendet werden.

Dies führt sofort zu der kniffligen Frage nach der Aussagekraft eines Messwertes: Was bedeutet eine hohe Abdeckung von Code, der von der IDE generiert wurde? Die getter und setter schreiben ja normalerweise nicht Sie, sondern Eclipse. Wenn Ihre Model-Klassen tatsächlich keinerlei Logik enthalten, können Sie sie von der Berechnung der Testabdeckung explizit ausnehmen.

Eine andere Strategie ist es, getter und setter erst zu erzeugen, wenn sie benötigt werden, und zunächst nur die als private deklarierten Attribute hinzuschreiben. Das verringert gleichzeitig die Menge nutzlosen Codes.

Oder Sie betrachten nicht den Abdeckungswert des Gesamtprojekts, sondern jenen eines Packages, das die tatsächliche Geschäftslogik enthält.

Letztlich ist das eine Frage, die Sie im Team diskutieren sollten, damit auch jeder unter dem Wert für die Testabdeckung das Gleiche versteht.

Ebenso ist es eine Geschmacksfrage, welcher Wert denn als »hoch genug« angesehen werden darf. In einer Klausur an der Uni zählen meist 80 % der erreichbaren Punktzahl als »gut«. In den meisten Projekten, die ich zur Qualitätsanalyse vorgelegt bekomme, beträgt die Abdeckung 10 % oder weniger. Das mag als Anhaltspunkt dienen.

Wichtiger ist jedoch, dass die Testabdeckung während der Weiterentwicklung eines Projekts nicht sinkt. Sie sollten sie also im Auge behalten und am besten für jedes Release dokumentieren. Ähnlich gehen Sie vor, wenn Sie sich im Team vornehmen, die Testabdeckung nach und nach zu verbessern. Anreize sind hier sicher nicht verkehrt: Für jeweils 5 % Verbesserung gibt's eine Runde Pizza fürs ganze Team!

6.2 Testbar und nicht so gut testbar

Wenn Sie im Moment eine Testabdeckung unterhalb der Zimmertemperatur haben, sitzen Sie jetzt sicher mit hochgekrempelten Ärmeln vor Ihrem Code und versuchen, die Lage zu verbessern. Gut so! Allerdings dürfen Sie sich nicht wundern, wenn Sie früher oder später verzweifelt aufschreien: »Wie soll ich *den* Code denn testen?«

Sobald Sie an diesem Punkt angekommen sind, setzen Sie die Lektüre beim nächsten Abschnitt fort.

6.2.1 Getrieben von Tests

Bereits am Anfang des Kapitels habe ich darauf hingewiesen, dass es eine gute Idee ist, die Tests zunächst zu schreiben und danach erst den eigentlichen Code. Dann müssten Sie sich jetzt nicht fragen, wie Sie Ihren Code testen sollen – denn die Tests wären ja schon da.

Natürlich sollen Sie nicht zuerst alle Tests für alle Module Ihrer Software schreiben. Vielmehr gehen Sie Modul für Modul vor:

- Sie schreiben einen Test, der eine leere Funktion daraufhin prüft, ob sie das gewünschte Resultat für einen einfachen Testfall liefert. Dieser Test schlägt natürlich zunächst fehl.
- Sie schreiben den Programmcode so, dass der Test durchläuft.
- Anschließend räumen Sie Ihren Code noch auf, optimieren ihn, achten auf Einhaltung von Konventionen und stellen sicher, dass der Test weiterhin durchläuft. Wenn die fragliche Funktion noch weitere Fälle abdecken soll, fangen Sie wieder von vorn an.

Test Driven Development

Dieses Paradigma des *Test Driven Developments* geht auf den Entwickler Kent Beck zurück, der zu den Erfindern des Extreme Programming gehört.

Zu den Vorteilen der testgetriebenen Entwicklung gehört nicht nur die von vornherein höhere Testabdeckung, sondern auch die zumeist bessere Strukturierung bei der Implementierung. Dass diese Art der Entwicklung eine höhere Codequalität hervorbringt, ist ein Erfahrungswert – vor allem aber ist es so ausgeschlossen, nicht oder schwer testbaren Code zu produzieren.

Wie sieht gut testbarer Code also aus?

6.2.2 Gut testbar

Wie im Beispiel mit dem Fußballspiel gezeigt, wollen Sie eine Funktion möglichst losgelöst vom Rest der Software testen können. Dazu muss sie

möglichst wenige Abhängigkeiten von anderen Funktionen oder Klassen enthalten. Diese zu testen ist nämlich wiederum Aufgabe anderer Tests. Sie wollen, um den `MatchResultAnalyzer` zu testen, nichts mit Klassen wie `StadiumManager` oder `RefereeBehaviour` zu tun haben.

Falls es externe Abhängigkeiten gibt, sollte es leicht möglich sein, sie durch einfache Ersatzobjekte zu ersetzen, die ein bestimmtes Verhalten simulieren (mehr dazu in Abschnitt 6.3 und 6.4).

Werfen Sie noch einmal einen Blick auf die Funktion `determineWinner()`:

```
public class MatchResultAnalyzer {
public void determineWinner(Match match) {
  if(match.isPlayed()) {
    if(match.getGoals1() > match.getGoals2())
      match.setWinner(match.getTeam1());
    else if(match.getGoals2() > match.getGoals1())
      match.setWinner(match.getTeam2());
    }
  }
}
```

Listing 6.15 Gut testbar ist eine Funktion mit hoher Isolation.

Diese Funktion besitzt keinerlei externe Abhängigkeiten. Alles, was sie braucht, wird ihr im Parameter `match` übergeben. Sie ist *stateless*, modifiziert also keine Attribute ihrer Klasse.

Die Anzahl der möglichen Laufwege durch die Funktion ist gering (vier, um genau zu sein).

Bildhaft gesprochen: Ein 15-Teile-Puzzle können Sie leichter auf Vollständigkeit prüfen als eines mit 2.000 Teilen, das ein frustrierter Sohnemann ohne Verpackung in seinen Spieleschrank geballert hat, dessen Türen sich schon lange nicht mehr gewaltfrei schließen lassen.

Natürlich ist die Anforderung nicht immer so übersichtlich wie hier. Glücklicherweise erreichen Sie aber fast immer eine gute Testbarkeit, wenn Sie alles vermeiden, was sie erschwert.

6.2.3 Nicht so gut testbar

Externe Abhängigkeiten sind der wichtigste Kandidat auf der Liste der Hindernisse auf dem Weg zu guten Tests. Am schlimmsten sind Abhängigkeiten, die Sie beim Ablauf eines Unit-Tests nicht ohne Weiteres ersetzen können, z. B. Singletons.

So ein Singleton führt Operationen aus, wenn die Klasse geladen wird, also bevor Ihr JUnit-Test überhaupt aktiv wird.

Hier ein abschreckendes Beispiel:

```
public class GodSingleton {
  private static GodSingleton instance = new GodSingleton();
  public static GodSingleton getInstance() {
    return instance;
  }

  private GodSingleton() {
    loadPropertiesFromFile();
    loadStuffFromDatabase();
  }
...
}
```

Listing 6.16 Singletons, die externe Abhängigkeiten im Konstruktor enthalten, sind schwer testbar.

Was hier im Konstruktor geschieht, entzieht sich der Kontrolle Ihres Unit-Tests. Womöglich versucht die Klasse, eine Verbindung zu einer Datenbank aufzubauen, die von Ihrem Rechner aus überhaupt nicht zugänglich ist. Oder sie versucht erfolglos, eine nur auf dem Produktivserver existierende Properties-Datei zu laden.

Natürlich sind solche Probleme nicht auf den Konstruktor beschränkt.

Werfen Sie einen Blick auf die folgende Funktion, die möglicherweise in Ihrem Fußballspiel vorkommt:

```
public void generateStatistics(Match match) {
  MatchResultAnalyzer analyzer = new MatchResultAnalyzer();
  generateStatisticsInternal(analyzer, match);
  ...
}
```

Listing 6.17 Eine Funktion sollte kein Geschäftslogik-Objekt erzeugen.

Hier liegt eine externe Abhängigkeit in Form eines Konstruktor-Aufrufs vor. Die Klasse `MatchResultAnalyzer` ist allerdings Gegenstand eines ganz anderen Tests, die Isolation ist nicht so hoch wie möglich. Letztlich ist die Methode `generateStatisticsInternal()` jene, die getestet werden sollte – aber sie ist höchstwahrscheinlich als `private` deklariert, obwohl sie eigentlich den Einsprungpunkt darstellen sollte.

Erforderliche Geschäftslogik-Objekte sollten als Konstruktor-Parameter übergeben werden: So geschieht das Erzeugen von Objekten unabhängig von der Programmlogik. Keinesfalls sollten »auf Verdacht« Objektverweise eingebaut werden, obwohl davon später tatsächlich nichts benötigt wird. Besonders gefährlich sind Verwaltungsklassen, die Referenzen zu zig anderen Klassen halten (häufig `Context` benannt). Denn jede Abhängigkeit bringt automatisch ihre eigenen Abhängigkeiten mit – wie der sprichwörtliche Rattenschwanz.

Auch die Instanziierung von Datenobjekten sollte Fabrikmethoden (und -klassen) überlassen bleiben und nicht innerhalb von Geschäftslogik-Klassen geschehen. Das ist eine Frage der Zuständigkeit. Eine Trennung erlaubt es, die für Tests nötigen Objekte nach Bedarf aufzubauen, bevor sie der zu testenden Funktion übergeben werden.

Private Methoden nicht testen

Oft werde ich gefragt, wie man eigentlich `private`-Methoden testet. Es ist durchaus möglich, über Java Reflection die gewünschten Zugriffsrechte einzufordern:

```
Method method = myClass.getDeclaredMethod(methodName,
argClasses);
method.setAccessible(true);
method.invoke(targetObject, argObjects);
```

Listing 6.18 Auch »private«-Methoden können Sie via Java Reflection aufrufen und testen. Tun Sie's aber nicht!

Allerdings gibt es hierbei ein Problem. Der Entwickler der zu testenden Klasse hatte einen guten Grund, die Methode als `private` zu deklarieren. Sie gehört nicht zu der öffentlichen API, die die Klasse anbietet. *Die »private«-Methode geht Sie nichts an.*

Anders ausgedrückt: Behandeln Sie die zu testende Klasse wie eine Blackbox, die eine bestimmte Funktionalität mit `public`-Methoden zur Verfügung stellt. Was intern geschieht, ist aus Ihrer Sicht irrelevant. Soweit es die Tests betrifft, wird jede `private`-Methode letztlich von irgendeiner `public`-Methode verwendet und damit von einem Ihrer Tests.

Wenn nicht, haben Sie entweder einen Testfall vergessen, oder die `private`-Methode ist überflüssig, weil sie überhaupt nicht verwendet wird.

Wenn das der Fall ist, fragen Sie den zuständigen Entwickler oder die zuständige Entwicklerin, ob die Methode nicht entfernt werden sollte, um die relative Testabdeckung zu erhöhen. Unbenutzte `private`-Methoden sollte es schließlich eigentlich gar nicht geben. Wozu auch?

Wenn Sie in einer Klasse sehr viele `private`-Methoden finden, die Sie gerne einzeln testen möchten, dann ist das meist ein Zeichen dafür, dass die Klasse zu groß ist. Extrahieren Sie logisch zusammengehörige Funktionen in eine andere Klasse, wo sie `public` sind und damit testbar.

Analog gilt das Gesagte für `protected`-Methoden: Entweder, es gibt eine abgeleitete Klasse, die diese verwendet, oder sie müssen entfernt werden.

Sie sehen: Zum Teil können Sie schwer testbaren Code umbauen, aber es gibt eine Kategorie, an der Sie sich die Zähne ausbeißen: untestbarer Code.

6.2.4 Unmöglich testbar

Code ist insbesondere dann untestbar, wenn eine externe Abhängigkeit unkontrollierbar oder nicht deterministisch ist. Am schlimmsten sind diese Fälle, wenn die fragliche Abhängigkeit unsichtbar ist – dann ist der Test »grün«, bis sich die externe Bedingung aus irgendeinem Grund ändert. Und dann knallt's.

Einen solchen Fall gab es im November 2016 bei Googles Chromium-Browser. Nicht ohne Folgen. Von einem Tag auf den anderen funktionierten diverse Webseiten nicht mehr. Der Browser bemängelte plötzlich bestimmte Symantec-Zertifikate von SSL-Servern. Seine wenig hilfreiche Meldung sehen Sie in Abbildung 6.5.

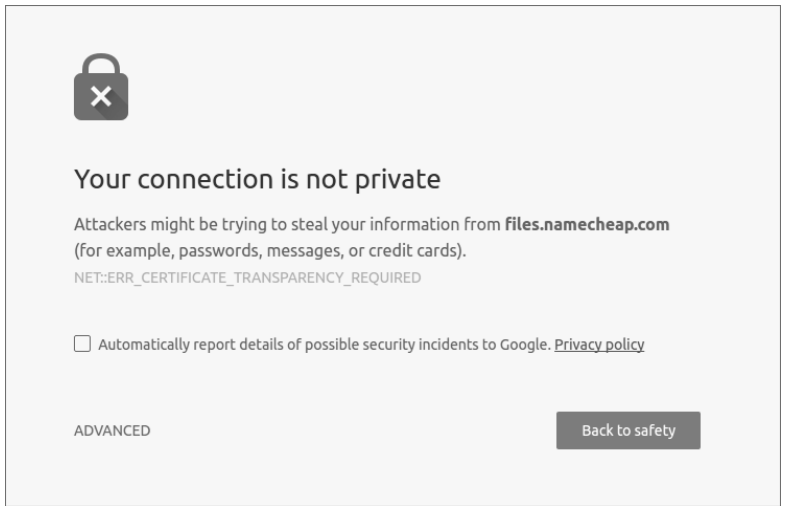

Abbildung 6.5 Chromium-Bug-Nummer 664177

Beim Besuch einer betroffenen Webseite erhielten die Nutzerinnen und Nutzer den Warnhinweis, dass die Verbindung nicht vertrauenswürdig

sei. Diese Warnung konnte man wegklicken, mit dem Resultat, dass das bekannte SSL-Schloss-Symbol nicht mehr grün, sondern rot erschien.

Damit hätte man noch leben können. Aber der meiste HTTP-Verkehr findet heutzutage im Hintergrund statt: mit Ajax-Aufrufen durch JavaScript an irgendwelche Webservices. Auch denen fehlte die Vertrauenswürdigkeit, folglich schlugen sie fehl, ohne dass die Nutzerinnen und Nutzer etwas daran ändern konnte. Populäre Webseiten wie *amazon.com* oder *flickr.com* funktionierten nur eingeschränkt.

Was war passiert?

Glücklicherweise zeigt ein Blick in den Code sofort die Ursache. Im C++-Code fanden Sie in der Datei *ct_policy_enforcer.cc* dies:

```
// Returns true if the current build is recent enough to
// ensure that
// built-in security information (e. g. CT Logs) is fresh
// enough.
// TODO(eranm): Move to base or net/base
bool IsBuildTimely() {
  const base::Time build_time = base::GetBuildTime();
  // We consider built-in information to be timely for 10 weeks.
  return (base::Time::Now() - build_time).InDays() < 70
  // 10 weeks
}
```

Diese Funktion wird verwendet, um festzustellen, ob das Programm vor mehr als zehn Wochen gebaut wurde (`base::GetBuildTime()`). Ist es älter, dann unterstellt es, dass die fraglichen Zertifikate nicht mehr als gültig betrachtet werden können, weil dann möglicherweise aktualisierte Listen mit zurückgezogenen Zertifikaten nicht mehr aktuell sind. Diese Funktion gibt also `false` zurück, wenn die Anwendung vor mehr als zehn Wochen gebaut wurde, was letztlich dazu führt, dass den Zertifikaten nicht vertraut wird und die Verbindung als unsicher bezeichnet wird.

Mit anderen Worten: Sie müssen spätestens alle zehn Wochen eine frische Version von Chromium installieren! Nun mag das bei den heutigen Release-Zyklen vielleicht sogar für manche Nutzerinnen und Nutzer unproblematisch sein – aber es gibt beispielsweise Linux-Distributionen, die Software-Pakete erst ausführlich testen, bevor sie auf die Nutzer losgelassen werden, oder die nur kritische Sicherheits-Updates durchwinken.

Der fragliche Code führte aber dazu, dass Chromium nach exakt zehn Wochen nicht mehr richtig funktionierte und zwangsweise ein Update erforderte. Klingt harmlos?

Stellen Sie sich vor, Sie sind Fotograf, Journalist oder Politiker, befinden sich gerade auf einer Dienstreise in einer Gegend, in der sie froh sein können, wenn Sie überhaupt eine stabile Internetverbindung erhalten. Von einer Minute zur nächsten funktioniert die Webseite nicht mehr, auf die Sie beruflich angewiesen sind. Die Warnmeldung legt außerdem den Verdacht nahe, übertragene Daten seien nicht mehr sicher. Sie können nicht mehr arbeiten! Sie werden auf die Schnelle nicht einmal die Erklärung dafür googeln können, wenn Sie nicht zufällig technisch vorgebildet sind und die Beschreibungen in den offiziellen Bugtrackern finden und verstehen (zum Nachlesen: *https://bugs.chromium.org/p/chromium/issues/detail?id=664177*).

Selbst wenn Sie die Ursache begreifen – lösen können Sie das Problem trotzdem nicht ohne Weiteres, wenn Sie nicht auf Anhieb eine aktuelle Version herunterladen können. Im Fall von Ubuntu Linux dauerte es knapp drei Tage, bis über die offizielle Aktualisierungsverwaltung eine neue Version von Chromium zur Verfügung stand. (Ob die nun länger als zehn Wochen funktionierte, habe ich nicht überprüft.)

Time-Bomb-Bugs

Ein solches Verhalten nennt man eine *Time Bomb*, eine Bombe mit Zeitzünder: Code, der funktioniert, bis ein bestimmter Zeitpunkt erreicht ist. Der erste weltweit bekannt gewordene Fall dieser Art ist natürlich »y2k«, das *Jahr-2000-Problem*: Damals mussten Millionen und Abermillionen US-Dollar und D-Mark für die Aktualisierung teils völlig veralteter Software investiert werden, die Jahreszahlen nur zweistellig speicherte. Beim Millennium-Wechsel von »99« auf »00« musste mit Schwierigkeiten bis hin zum unbeabsichtigten Start von Atomraketen gerechnet werden.

Die Funktion hat als externe Abhängigkeit schlicht die aktuelle Uhrzeit, und diese entzieht sich der Kontrolle jedes Unit-Tests. Natürlich können Sie sicherheitshalber alle Tests einmal auf einer Maschine laufen lassen, deren Uhr ein paar Tage, Wochen oder Jahre vorgeht. Aber wer tut das schon?

Googles Chromium-Entwicklerinnen und -Entwickler jedenfalls nicht, das wissen wir jetzt.

Wie geht man mit solchen Fällen um?

Die Faustregel sollte lauten: Alles, was nicht maschinell testbar ist, muss manuell getestet werden und dazu besonders gut dokumentiert sein. Wenn man die Systemuhr umstellt, kann man das Verhalten ja reproduzieren. Dergleichen sollte aber nur in Ausnahmen nötig sein. Wenn nicht deterministische Abhängigkeiten unvermeidbar sind, sollten sie besonders gut durchdacht sein. Sprechen Sie im Team darüber.

Entwickler: »Du, unser Binary funktioniert nach zehn Wochen nicht mehr richtig, wenn wir diese Abfrage so drin lassen.«

Produktmanagerin: »Ach was, die Leute laden doch immer fleißig die Updates runter. Machst du doch auch, oder?«

Entwickler: »Ja, aber ich bin als Top-Entwickler einer Tech-Firma mit Glasfaser-Internet vielleicht nicht unbedingt repräsentativ.«

Produktmanagerin: »Jeder weiß doch, dass man bei Problemen erst mal ein Update installiert. Und das behebt es ja.«

Entwickler: »Aber was ist, wenn die Betroffenen lieber nicht das Update installieren, sondern einfach eine Konkurrenz-Software verwenden?«

Produktmanagerin: »Ähm. Das werden sie schon nicht tun. Da bin ich ganz zuversichtlich.«

Entwickler: »Und was, wenn unzufriedene Nutzer über Twitter, Facebook oder in der Tagesschau ihren Unmut kundtun, unterstützt von technischen Experten, die unseren Code lesen und sich darüber lustig machen? Wäre das nicht schlecht für den Ruf unseres Produkts?«

Produktmanagerin: »Das, äh, würde doch niemand tun, oder? Ich, äh ... warte ... lass mich noch mal drüber nachdenken ...«

Zum Abschluss fasst die Liste in folgendem Kasten Empfehlungen für gut testbaren Code zusammen.

So geht testbarer Code!

Empfehlungen für testbaren Code

- Datenobjekte und Logikklassen separieren
- Abhängigkeiten reduzieren
- Singletons vermeiden
- nicht von komplexen Klassen erben, da deren Funktionalität mit getestet werden müsste
- Konstruktoren sollten keine Logik enthalten.
- keine nicht deterministischen Abhängigkeiten

6.3 Umgekehrt wird ein Schuh draus

Weiter oben habe ich Ihnen empfohlen, auf Klassen zu verzichten, die viele Querverbindungen aufweisen. Aber wie sollen Sie dann sicherstellen, dass die vielen Abteilungen Ihrer Software miteinander kommunizieren können?

Sind überhaupt alle unnötigen Abhängigkeiten auflösbar?

Ja, sind sie. Sie müssen lediglich die Kontrolle über die nötigen Verbindungen an ein Framework übertragen.

6.3.1 Inversion of Control

Wenn die Klassen Ihrer Anwendung alle nötigen Funktionen implementieren, aber die Steuerung des Ablaufs von außerhalb kontrolliert wird, spricht man vom Paradigma *Inversion of Control (IoC)*. In diesem Abschnitt erkläre ich Ihnen die Vorteile und wie Sie sie nutzen können.

Ein Beispiel für IoC sind Servlets. Diese liegen untätig in einem Container (z. B. *Jetty*, *Tomcat* oder *JBoss*), bis ein Aufruf von einem Client kommt. Ihre Geschäftslogik-Klasse muss von der Klasse `HttpServlet` erben und Methoden wie `doGet()` oder `doPost()` implementieren, die der Container dann aufruft.

Ganz ähnlich ist der Ablauf, wenn Sie einen Message Broker wie *Apache ActiveMQ* verwenden: Immer wenn eine Nachricht an den von Ihnen implementierten Client eintrifft, tritt Ihr Code in Aktion.

In solchen Fällen ist Ihre Geschäftslogik fast immer stateless, alle verwendeten Daten sind nur aktuell, solange ein Request in Arbeit ist. Das eliminiert viele Querverbindungen. Es bleiben nur Abhängigkeiten in einer baumartigen Hierarchie, deren Stamm z. B. bei der `doGet()`-Methode beginnt. Zumindest im Idealfall.

Wie Sie im vorangegangenen Abschnitt gesehen haben, ist eine hohe Isolation eine Voraussetzung für gute Testbarkeit. Deshalb ist es nützlich, dieses Paradigma noch weiter zu treiben: Eine Software kann die Verwaltung *sämtlicher* Abhängigkeiten an ein Framework auslagern. Ein solches Framework stelle ich Ihnen im nächsten Abschnitt vor.

6.3.2 Dependency Injection mit Spring Boot

Eine konsequente Anwendung des Inversion-of-Control-Paradigmas erlaubt die *Dependency Injection (DI)*, ein Begriff, der auf Martin Fowler zurückgeht. Entscheidend dabei ist, dass ein Framework allein für alle Instanzen der Geschäftslogik-Klassen verantwortlich ist. Benötigt eine dieser Klassen eine Referenz zu einer anderen, stellt das Framework diese Verbindung zu der ihm bekannten Instanz her, indem es sie *injiziert*.

Welche Instanzen das Framework erzeugt und wie es die Verbindungen vornimmt, ist Aufgabe einer Konfiguration. Nachdem sich solche zumeist

in externen Dateien abgelegte Informationen als wenig wartungsfreundlich erwiesen hatten, kamen immer vorwitzigere Frameworks auf den Markt, die letztlich beinahe eine Zero-Config-DI ermöglichen. In einem etwas ausführlicheren Beispiel stelle ich Ihnen an dieser Stelle *Spring Boot* vor, ein schlankes Framework, das es massiv erleichtert, Applikationsserver zu bauen.

Als Teilprojekt des epische Ausmaße umfassenden Spring Frameworks (*https://spring.io*) eignet sich Spring Boot zum schnellen Einstieg und gleichermaßen für komplexe Projekte. Am einfachsten lernen Sie Spring Boot kennen, indem Sie das Spring-Plug-in für Eclipse installieren. Wählen Sie dann File • New • Import Spring Getting Started Content, und suchen Sie dort nach Spring Boot.

Das so erzeugte Projekt besteht aus zwei äußerst überschaubaren Dateien. Zuallererst gibt es eine `Application`-Klasse:

```
@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Listing 6.19 Die Einstiegsklasse der Spring Boot-Anwendung enthält nur eine relevante Zeile. Den von der IDE erzeugten, darüber hinausgehenden Beispielcode habe ich weggelassen.

Die Klasse verfügt über die für Java übliche `main()`-Methode und tut nichts anderes, als die Kontrolle an die `SpringApplication` zu übergeben. Diese durchsucht daraufhin alle anderen Klassen Ihrer Anwendung nach bestimmten Annotations und bastelt daraus einen `ApplicationContext`, in dem Instanzen Ihrer Klassen gemeldet sind (die *Beans*) – plus eine Menge nützlicher anderer Klassen.

Die zweite im Beispiel generierte Klasse nennt sich `HelloController`:

```
@RestController
public class HelloController {
  @RequestMapping("/")
  public String index(@RequestParam String name) {
    return "Hello " + name;
  }
}
```

Listing 6.20 Der »HelloController« behandelt HTTP-Aufrufe.

Üblicherweise nennt man Klassen Controller, die zwecks einer gewünschten Operation die Kontrolle über Daten und User-Interface übernehmen. Genau das tut der `HelloController`.

REST mit Spring Boot

Die Annotation `@RestController` signalisiert Spring Boot, dass es sich um eine Klasse handelt, die entsprechend dem REST-Paradigma Requests entgegennimmt. Jeder REST-Controller hat zustandslos zu sein und verfügt über eine einheitliche Schnittstelle. Im hier vorliegenden einfachsten Fall sorgt die zweite Annotation, `@RequestMapping`, dafür, dass ein Request an die Adresse / von der Funktion `index()` behandelt wird. Diese Funktion nimmt einen Request-Parameter entgegen.

Wenn Sie die Anwendung starten (mit Run • Run As • Spring Boot App), fährt Spring Boot einen eingebetteten Tomcat Application Server hoch, der auf Port 8080 horcht. Sie können sich im Konsolenfenster von Ecplise die entsprechenden Log-Ausgaben anschauen.

Rufen Sie nun mit Ihrem Browser den Controller auf: *http://localhost:8080/?name=Elvis*

Der Tomcat von Spring Boot nimmt dies entgegen, und das Framework ruft die Funktion `index()` mit dem Parameter `name` und dem String »Elvis« auf. Der Rückgabewert, also »Hello Elvis«, erscheint im Browserfenster.

Übrigens können Sie den Port modifizieren, indem Sie ein System-Property ändern. Das können Sie beispielsweise im Code tun (am Anfang der `main()`-Methode):

```
System.getProperties().put( "server.port", 8088 );
```

Ferner versteht die Anwendung den Kommandozeilenparameter `--server.port`.

Bisher war noch keine Rede von Dependency Injection – aber dazu kommen wir sofort.

Angenommen, Sie möchten die Nutzerinnen und Nutzer auf verschiedene Weise begrüßen, etwa abhängig von der Tageszeit. Dafür soll eine andere Klasse zuständig sein, der `SalutationGenerator`:

```
@Service
public class SalutationGenerator {
  public String generateSalutation(LocalTime now) {
    if(isMorning(now))
      return "Moin";
    else
      return "Mahlzeit";
  }
```

```
  private boolean isMorning(LocalTime now) {
    ...
  }
}
```

Listing 6.21 Der »SalutationGenerator« erzeugt einen Gruß abhängig von der Tageszeit. Um eine nicht deterministische Abhängigkeit von der Systemuhr zu vermeiden, muss die Uhrzeit übergeben werden.

Diese Klasse erhält den Vermerk `@Service`. Spring Boot erzeugt dann eine einzelne Instanz (ein Singleton ohne Singleton-Nachteile) und speichert die Referenz in der internen Bean-Tabelle. Der vom Plug-in generierte Code loggt eine Liste aller Beans, dort können Sie auch Ihren `SalutationGenerator` sehen.

Jetzt können Sie im Controller eine Referenz auf den Generator einbauen und es Spring Boot überlassen, sie mit der tatsächlichen Instanz zu verknüpfen. Dazu gibt es die Annotation `@Autowired`:

```
@Autowired private SalutationGenerator salutationGenerator;
```

Genau hier geschieht die Dependency Injection: Das Framework verdrahtet automatisch den Verweis auf den `SalutationGenerator` mit der intern erzeugten Service-Instanz. Sie müssen weder einen Konstruktor von `SalutationGenerator` aufrufen noch sich von irgendwoher eine Referenz darauf beschaffen. Darum kümmert sich Spring.

In die Funktion `index()` schreiben Sie jetzt einfach:

```
return salutationGenerator.generateSalutation(LocalTime.now()) +
" " + name;
```

Auf diese Weise haben Sie mit minimalem Codeaufwand komplett entkoppelte Klassen, die Sie sehr leicht einzeln testen können.

Der Clou ist nämlich, dass Sie für Ihre Tests andere Verknüpfungen herstellen können (oder gar keine, wenn sie nicht benötigt werden). So eröffnet Ihnen Dependency Injection die nötigen Freiheiten beim Testen.

Übrigens: Wenn Sie statt eines einfachen Datentyps wie String ein Modell-Objekt zurückgeben, wird es automatisch in das bei REST übliche JSON-Format umgewandelt.

Das ist nur einer der vielen Tricks, die Spring Boot auf Lager hat. Ein anderer besteht darin, neben Java auch Groovy zu unterstützen. So passt, wie die Macher von Spring Boot betonen, eine komplette Webapplikation in

einen einzigen Tweet (*https://twitter.com/rob_winch/status/364871658483351552*).

Da Spring Boot auf die ganze Spring Framework-Infrastruktur zurückgreifen kann, erleichtert Ihnen diese Technologie eine Vielzahl von Standardaufgaben wie z. B. die Anbindung einer Datenbank (mit *Spring Data*) – ganz abgesehen von der guten Testbarkeit dank Dependency Injection.

6.4 Alles einzeln testen

Egal wie gut Sie die verschiedenen Klassen in Ihrem Projekt voneinander separieren: Letztlich gibt es natürlich immer Querverbindungen und Abhängigkeiten. Eine Dienstklasse, die mit bestimmten Datenobjekten hantiert, benötigt zwingend Zugriff auf eine Klasse, die dazu in der Lage ist, die Objekte aus einer Datenbank zu holen und wieder zu speichern.

Um die Dienstklasse einzeln testen zu können, müssen Sie den Datenzugriffs-Layer durch eine Testversion ersetzen.

Das ist die Stelle, an der Mocking-Frameworks ins Spiel kommen.

6.4.1 Unit-Tests mit JMockit

Es gibt eine ganze Reihe verschiedener Mocking-Frameworks für alle möglichen Sprachen. Die grundsätzliche Vorgehensweise ist dabei immer gleich. Daher beschränke ich mich hier auf ein Beispiel mit *JMockit* für Java (*https://jmockit.github.io/*). JMockit funktioniert prima mit JUnit und in Eclipse.

Um Ihren Unit-Tests die Mocking-Funktionalität zur Verfügung zu stellen, müssen Sie die JMockit-Bibliothek zum Build-Pfad des Projekts hinzufügen. Am einfachsten geht das, indem Sie das Release-Archiv von der Webseite herunterladen und die Datei *jmockit.jar* in ein neues Verzeichnis namens *lib* in Ihrem Projekt ablegen. Dann klicken Sie in Eclipse mit der rechten Maustaste auf diese Datei und wählen Build Path... • Add to Build Path.

Normalerweise genügt das, damit Eclipse bzw. Java die Bibliothek verwenden kann, aber im Fall von JMockit müssen Sie dafür sorgen, dass die Bibliothek in der Reihenfolge vor JUnit steht. Deshalb ist es eine gute Idee, erst zu diesem Zeitpunkt JUnit hinzuzufügen. Wählen Sie dazu im Kontextmenü Ihres Projekts Java Build Path, und fügen Sie auf der Registerkarte Libraries mit dem Button Add Library... JUnit 4 hinzu. Auf der Registerkarte Order and Export können Sie die Reihenfolge prüfen und nötigenfalls ändern.

Um Ihnen die Verwendung von JMockit zu demonstrieren, greife ich noch einmal zu dem bereits bekannten Beispiel mit dem Fußballspiel. Zusätzlich zum `MatchResultAnalyzer` und zu den nötigen Datenklassen führe ich einen `MatchService` ein sowie eine Datenzugriffsklasse `MatchDaoImpl` mit zugehörigem Interface `MatchDao`. In Abschnitt 5.3.5 habe ich Ihnen bereits erklärt, warum es eine gute Idee ist, Interface und Implementierung im Datenzugriffs-Layer zu trennen.

Abbildung 6.6 zeigt die Package-Hierarchie des gesamten Projekts in Eclipse.

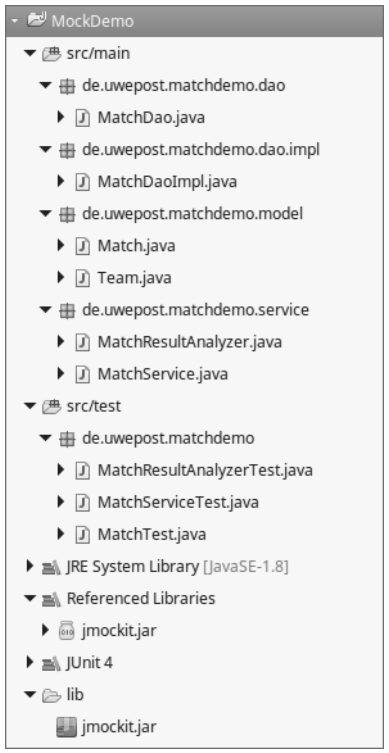

Abbildung 6.6 Das »MockDemo«-Projekt verfügt über zwei separate »src«-Verzeichnisse für Code und Tests sowie über passende Packages.

Die Aufgabe des `MatchService` soll es in diesem einfachen Beispiel sein, alle gespielten Spiele ohne Gewinner zu analysieren, d. h. mithilfe der Klasse `MatchResultAnalyzer` den Sieger zu ermitteln.

Dazu bietet `MatchDao` eine Funktion an, die alle gespielten Spiele ohne Gewinner aus der Datenbank holt. Wie diese Funktion genau aussieht, hängt von der verwendeten Datenbank ab, deshalb enthält `MatchDaoImpl` lediglich eine Dummy-Implementierung, die eine leere Liste zurückgibt. In diesem Kapitel geht es ja nur um die Tests!

Die zu testende Funktion sieht wie folgt aus:

```
public void analyzePlayedMatches() {
  List<Match> matches = matchDao.findPlayedMatchesWithoutWinner();
  for(Match m : matches) {
    analyzer.determineWinner(m);
  }
}
```

Listing 6.22 Die zu testende Funktion ermittelt den Gewinner für alle Spiele, in denen das noch nicht geschehen ist.

Der `MatchService`, also die zu testende Klasse, verfügt über einen Konstruktor, der Referenzen auf `MatchDao` und `MatchResultAnalyzer` erhält. Das sind die beiden externen Abhängigkeiten, die hier nicht zu testen sind, folglich müssen sie durch »Täuschobjekte«, also *Mocks*, ersetzt werden.

Im Fall von `MatchDao` könnten Sie ein solches Objekt leicht selbst schreiben, denn das ist ja glücklicherweise nur ein Interface, und dem `MatchService` ist es egal, wie die Implementierung ausschaut. Aber diese Arbeit überlassen Sie einfach dem Mock-Framework:

```
public class MatchServiceTest {
  MatchService matchService;
  @Mocked MatchDao matchDaoMock;
  @Mocked MatchResultAnalyzer matchResultAnalyzer;
  ...
}
```

Listing 6.23 Der Unit-Test für »MatchService« verwendet Mocks für die Abhängigkeiten.

Jetzt können Sie den eigentlichen Test schreiben. Beginnen Sie, indem Sie die nötigen Datenobjekte vorbereiten, in diesem Fall eine Liste mit einem `Match`-Objekt darin, das gespielt wurde, aber noch keinen Gewinner hat:

```
List<Match> matches = new ArrayList<Match>();
Match match = new Match(team1, team2);
match.setFinalResult(2, 3);
matches.add(match);
```

Listing 6.24 Die Liste der zu bearbeitenden Spiele enthält einen Eintrag.

Keine Datenobjekte mocken

Lassen Sie mich an dieser Stelle warnend den Zeigefinger heben: *Mocken Sie keine Datenobjekte!*

Datenobjekte müssen dumm sein, bar jeder Geschäftslogik. Wenn nicht, beginnen Sie am besten sofort mit dem Refactoring! Dumme Objekte, die lediglich Daten halten, können Sie einfach erzeugen, ohne sie mocken zu müssen. Falls Sie öfter Datenobjekte mit umfangreichen Vorgabewerten erzeugen müssen, schreiben Sie eine Fabrikmethode oder einen Builder, der nur im `test`-Zweig Ihres Codes existiert.

Den `MatchService` können Sie in der Funktion `setUp()` Ihres Tests instanzieren:

```
@Before
public void setUp() throws Exception {
  team1 = new Team();
  team2 = new Team();
  matchService =
    new MatchService(matchDaoMock, matchResultAnalyzer);
}
```

Listing 6.25 Service und Teams für Testzwecke generiert die »setUp()«-Methode ein Mal pro Testfall.

Der Clou bei Mocking-Frameworks wie JMockit besteht darin, die vorgetäuschte Funktion zur Laufzeit festzulegen. Genau das tun Sie bei JMockit, indem Sie anonyme Expectations-Klassen definieren. Die erste sorgt dafür, dass `matchDaoMock` die vorbereitete `matches`-Liste zurückgibt, wenn der `MatchService` die Funktion `findPlayedMatchesWithoutWinner()` aufruft. Der Testfall *erwartet*, dass die zu testende Funktion die genannte Funktion aufruft (Expectation), und gibt den genannten Wert zurück:

```
new Expectations() { {
  matchDaoMock.findPlayedMatchesWithoutWinner();
  result=matches;
} };
```

Die doppelten geschweiften Klammern sind kein Druckfehler! Diese ungewohnte Syntax erlaubt es, mehrere Expectations in einer Operation zu erstellen. Die Vorbereitungen sind abgeschlossen, jetzt können Sie die zu testende Methode aufrufen:

```
matchService.analyzePlayedMatches();
```


Der Test war erfolgreich, wenn diese Funktion den gemockten `matchResultAnalyzer` mit `match` als Parameter aufgerufen hat. Um das zu prüfen, verwenden Sie eine *Verification*:

```
new Verifications() {{
  matchResultAnalyzer.determineWinner(match);
} };
```

Damit ist der Test fertig und läuft durch.

Sie können probeweise einen Fehler in `analyzePlayedMatches()` einbauen, indem Sie beispielsweise den Aufruf des `analyzers` in der Schleife auskommentieren. Dann wird die gemockte Funktion überhaupt nicht aufgerufen, die Verification schlägt fehl und gibt als Fehler »Missing invocation...« aus.

Oder Sie übergeben statt des `Match`-Objekts einfach `null`, auch das passt nicht zur Verification.

Um sicherzustellen, dass die zu testende Funktion auch mit mehr als einem Eintrag in der Ausgangsliste funktioniert, können Sie einfach mehrere `Match`-Objekte in die Liste packen und innerhalb der Verification eine Zählung vornehmen:

```
new Verifications() { {
  matchResultAnalyzer.determineWinner((Match)any);
  times = matches.size();
} };
```

In diesem Beispiel akzeptiert die gemockte Funktion `determineWinner` mit `(Match)any` beliebige `Match`-Objekte als Parameter.

Wenn Ihre Codemodule hinreichend voneinander separiert sind, ist Mocking der Generalschlüssel zu schnellen, effizienten Tests. Sicher dürfen Sie nicht unterschätzen, dass der Einsatz eines Mocking-Frameworks eine mehr oder weniger steile Lernkurve mit sich bringt. Die Investition dürfte sich aber lohnen, probieren Sie's aus!

Mocking-Frameworks

JMockit (für Java, auch EE und Spring, *https://jmockit.github.io/*)

Mockito (für Java, *https://site.mockito.org/*)

Mockjax (Mocking-Plug-in für jQuery/JavaScript, *https://github.com/jakerella/jquery-mockjax*)

Moq (für C#, BSD-Lizenz, *https://github.com/moq/moq4*)