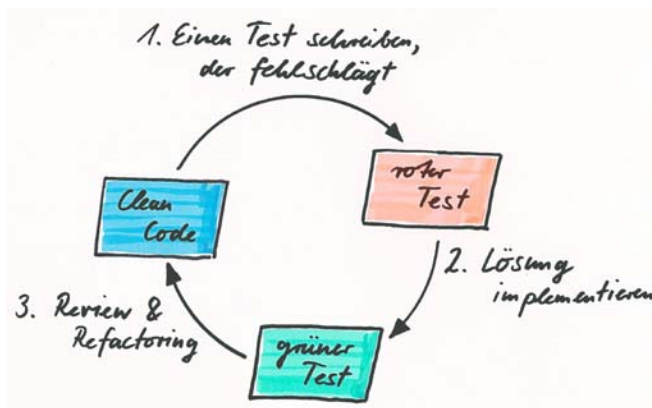
4 Testgetriebene Entwicklung

Der Autor von JUnit, Kent Beck, ist nicht nur als Autor von JUnit bekannt, sondern auch als Erfinder des *Extreme Programming*. Und das ist kein Zufall: Eine der Praktiken des Extreme Programming ist nämlich das sogenannte *Test-First Programming*, bei dem immer erst ein fehlschlagender Test geschrieben wird, bevor der eigentliche Produktionscode geändert wird.

Allerdings hat sich die Bezeichnung *Test-First Programming* nicht durchgesetzt. Heutzutage ist diese Vorgehensweise unter dem Namen *testgetriebene Entwicklung* bzw. *Test Driven Development* (oder kurz *TDD*) bekannt. Um zu verstehen, wozu JUnit ursprünglich gedacht war (nämlich um einfach Unit-Tests schreiben zu können), sollten Sie den TDD-Zyklus kennen, der in diesem Kapitel beschrieben wird.

4.1 Einmal rundherum

Um den TDD-Zyklus zu veranschaulichen, möchte ich diesen anhand eines kleinen Beispiels mit Ihnen durchexerzieren. Stellen Sie sich vor, Sie entwickeln ein Computerspiel. Natürlich darf dabei eine Highscore-Liste nicht fehlen. Und natürlich entwickeln Sie testgetrieben, schließlich wollen Sie qualitativ hochwertigen Code schreiben und potenzielle Spieler nicht durch ein Spiel mit Bugs verprellen. Der TDD-Zyklus sieht so aus:



Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

Der TDD-Zyklus beginnt immer damit, dass Sie als Erstes einen Test schreiben, der fehlschlägt. Und »immer« bedeutet in diesem Fall auch wirklich immer, denn die erste Regel beim testgetriebenen Entwickeln lautet:

Sie dürfen keinen Produktionscode schreiben oder ändern, ohne einen fehlschlagenden Test zu haben.

Und zwar egal, ob Sie ein neues Feature entwickeln oder einen Bug fixen. Wenn Sie sich nicht an diese Regel halten, entwickeln Sie auch nicht testgetrieben. Wenn Sie sich jedoch an diese Regel halten, also immer erst einen Test erstellen, bevor Sie ein neues Feature entwickeln oder bevor Sie einen Bug in Ihrer Software fixen, werden sich mit der Zeit eine Menge Tests anhäufen. Diese Tests sind Ihr Sicherheitsnetz, das es Ihnen ermöglicht, umfangreiche *Refactorings*[1] in Ihrer Codebasis vorzunehmen, um beispielsweise das Design zu verbessern.

Zugegeben, die Tests sind sozusagen nur »die Fäden« des Sicherheitsnetzes. Sie müssen das Sicherheitsnetz auch aufspannen, indem Sie einen Continuous-Integration-Server aufsetzen, der alle Tests bei jedem Commit in die Versionskontrolle ausführt. Und darüber hinaus müssen Sie in den Köpfen aller Entwickler in Ihrem Team verankern, dass sich um fehlschlagende Builds sofort gekümmert werden muss, anstatt diese zu ignorieren.

Außerdem erhöht eine funktionierende Testsuite das Vertrauen in eine Applikation ungemein: Wenn alle Tests grün sind, wissen Sie, dass Ihre Applikation immer noch genau das tut, was Sie von ihr erwarten. Sie brauchen also keine Angst davor haben, selbst in großen Softwareprojekten, die Sie nicht gänzlich überblicken, irgendetwas zu ändern. Sollten Sie dabei tatsächlich aus Versehen etwas kaputtmachen, sollte wenigstens ein Test rot werden.

4.2 Einen roten Test schreiben

Der erste Schritt beim Schreiben eines neuen Tests besteht darin, sich zu überlegen, in welcher Testklasse der Test platziert und wie die Testmethode benannt werden soll. (Tipps hierfür finden Sie in den ersten beiden Abschnitten von Kap. 7).

Falls Sie sich für eine neue Testklasse entscheiden, legen Sie diese einfach an. Hierbei kann Ihnen übrigens Ihre IDE etwas Arbeit abnehmen, wenn Sie ein Template für Testklassen hinterlegt haben, das dafür sorgt, dass schon alle statischen Imports für die Assertion- sowie `Matcher`-Factory-Methoden vorhanden sind (siehe S. 312).

1. Refactoring – das Verändern von Quellcode, ohne dabei das Programmverhalten zu verändern, um zum Beispiel die Verständlichkeit, Wartbarkeit oder Erweiterbarkeit zu verbessern.

Enthält Ihr Projekt jedoch bereits eine passende Testklasse, so öffnen Sie sie und überprüfen Sie, ob alle in der Testklasse enthaltenen `@Rule`-Member und `@Before`-Methoden für die neue Testmethode, die Sie schreiben wollen, tatsächlich Sinn machen.

Ist dies nicht der Fall, so können Sie faul sein und die neue Testmethode trotzdem zu der Testklasse hinzufügen. Seien Sie sich aber bewusst, dass dadurch Ihr Test (und damit Ihre gesamte Testsuite) unnötig länger laufen würde. Noch schlimmer ist jedoch, dass dies Ihren Test auch unnötig schwer verständlich machen würde, denn wenn Sie oder jemand anderes später Ihre Testmethode liest, wird derjenige darüber stolpern, dass es Setup-Methoden und Regeln gibt, die für Ihren Test gar nicht nötig sind. Seien Sie also lieber nicht faul, und legen Sie in einem solchen Fall eine neue Testklasse mit einem passenden Namen an. Oder noch besser: Ändern Sie die Testklasse so, dass die Setup-Methoden nicht mehr implizit bei jedem Test aufgerufen werden, sondern explizit am Anfang jeder Testmethode (siehe Abschnitt 7.3).

Wenn Sie sich für eine Testklasse entschieden und die Signatur der neuen Testmethode geschrieben haben, heißt es, die neue Testmethode mit Inhalt zu füllen. Ich persönlich beginne oft damit, zunächst den Test in natürlicher Sprache als Kommentar in die Testmethode zu schreiben, um meine Gedanken zu ordnen. Für das Beispiel mit der Highscore-Liste für ein Computerspiel könnte die erste Version der Testklasse `HighScoreListTest` so aussehen:

```
public class HighScoreListTest {

    @Test
    public void test_update() {
        // Given the following high score list:
        //   1. Michael ... 12345
        //   2. Paul ....... 9876
        // When Daniel played a game with score 10000
        //   and the high score list is updated
        // Then he high score list should be:
        //   1. Michael ... 12345
        //   2. Daniel .... 10000
        //   3. Paul ....... 9876
    }
}
```

Wie Sie sehen, teilt sich der Test in drei Teile auf, die jeweils mit *Given*, *When* und *Then* anfangen. Diese Aufteilung habe nicht ich mir ausgedacht, sondern Dan North, der Erfinder des sogenannten *Behavior Driven Development* (kurz: *BDD*).[2] Bei dieser Vorgehensweise hat jeder Test (bzw. jedes Szenario, wie ein Test im Vokabular von *BDD* heißt) immer die Struktur Given … When … Then …, die sehr hilfreich ist, um gut verständliche Tests zu schreiben. Genau die gleiche Drei-

2. *http://dannorth.net/introducing-bdd/*

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

teilung beschreibt auch der AAA-Stil (siehe Abschnitt 7.5) für Tests, wobei AAA für *Arrange, Act, Assert* steht.

Wenn Ihnen klar ist, wie Ihr nächster Test aussehen soll, schreiben Sie als Nächstes den Testcode. Dabei sollten Sie – falls Sie zuvor Kommentare geschrieben haben – versuchen, diese Kommentare eins zu eins in Code zu überführen. Falls Sie von Anfang an eine gute Vorstellung davon haben, wie der Test aussehen soll, können Sie sich den Zwischenschritt mit den Kommentaren natürlich auch sparen und gleich den Testcode hinschreiben. Wie auch immer, die zweite Version der Beispielmethode `test_update()` könnte so aussehen:

```
@Test
public void test_update() {
    HighScoreList highScoreList = createHighScoreList(
        "Michael", 12345,
        "Paul", 9876
    );
    Game game = createGameWithPlayerAndScore("Daniel", 10000);

    highScoreList.update(game);

    assertThat(highScoreList, is(
        "Michael", 12345,
        "Daniel", 10000,
        "Paul", 9876
    ));
}
```

Diese Methode kompiliert erst einmal nicht. Es gibt weder die Klasse `HighScoreList` noch existieren die Hilfsmethoden `createHighScoreList` und `createGameWithPlayerAndScore`. Aber das ist auch so gewollt, denn TDD ist nicht nur eine Vorgehensweise für das Schreiben von Tests, sondern in erster Linie eine Designtechnik. Wenn Sie den Testcode vor dem Produktionscode schreiben, entwerfen Sie gleichzeitig die Architektur für Ihr Programm. Die hier gezeigte Methode `test_update()` legt zum Beispiel fest, dass es eine Klasse `HighScoreList` mit der Methode `update` geben wird, die als Parameter ein `Game`-Objekt erwartet.

Der Vorteil von testgetriebener Entwicklung ist, dass Sie die Schnittstellen Ihrer neuen Klassen gleich so entwerfen, wie sie später einmal benutzt werden sollen. Ihr Test ist sozusagen der erste Client für Ihre API und außerdem auch eine »lebendige« Dokumentation. Lebendig deshalb, weil Sie Ihre Tests bei Veränderungen an Ihrer Applikation immer grün halten werden, da Ihr Continuous-Integration-Server Sie benachrichtigt, falls irgendein Test fehlschlägt. Und in einem solchen Fall werden Sie entweder Ihren Produktionscode oder den Testcode so anpassen, dass Sie wieder einen grünen Build haben. Diese nötigen Anpassungen werden bei statischer Dokumentation (in separaten Dateien, in einem Wiki oder auch in Kommentaren im Quelltext) leider oft vergessen.

Wichtig beim Schreiben eines Tests ist:

Schreiben Sie eine Testmethode immer gleich vom Anfang bis zum Ende fertig, und zwar genau so, wie Sie sie gerne lesen würden, ohne fehlende Klassen oder Methoden bereits anzulegen oder gar schon zu implementieren.

Es ist völlig in Ordnung, wenn Sie zunächst Code schreiben, der gar nicht kompiliert. Schließlich befinden Sie sich gerade in der Designphase für Ihren Produktionscode, aber auch für Ihren Testcode: In der eben gezeigten Version von `test_update()` habe ich beispielsweise die Assertion so geschrieben, wie ich sie gerne lesen würde, obwohl es noch gar keine Factory-Methode für einen `Matcher` gibt, der eine beliebige Anzahl von Strings und Integer-Zahlen akzeptiert und diese mit einer `HighScoreList` vergleicht.

Wichtig beim Schreiben eines Tests ist außerdem, dass Sie unwesentliche Details aus der Testmethode heraushalten, damit diese kurz und übersichtlich bleibt. Merken Sie sich und beherzigen Sie deshalb auch noch folgende Regel:

Wenn etwas nicht wichtig für einen Test ist,
ist es wichtig, dass es auch nicht im Test steht.

Genau deswegen habe ich in der Testmethode `test_update()` den Aufruf der noch nicht existierenden Hilfsmethode `createGameWithPlayerAndScore` eingeführt, weil die Details, wie genau ein `Game`-Objekt für einen bestimmten Spieler mit einer bestimmten Punktzahl angelegt wird, nicht wichtig für das Verständnis des Tests sind.

Wenn Sie Ihre neue Testmethode fertig geschrieben haben, versuchen Sie – bei der sich jetzt anschließenden Implementierung – den Code in der Testmethode nicht mehr zu ändern. Alle zusätzlichen Variablen und Methodenaufrufe machen die Testmethode nur schwerer verständlich. Lagern Sie allen zusätzlichen Code, der eventuell nötig ist, um Ihre Testmethode laufen zu lassen, deshalb immer in Hilfsmethoden aus.

Angenommen, Sie haben Ihre Testmethode mit Inhalt gefüllt. Als Nächstes werden Sie versuchen, diese auszuführen, was typischerweise erst einmal nicht klappt. Damit haben Sie auch schon die erste Etappe im TDD-Zyklus erreicht: Sie haben einen Test, der fehlschlägt. Dabei zählt auch ein Test, der noch nicht kompiliert werden kann, als ein Test, der fehlschlägt. In den nächsten Minuten (während Sie versuchen, Ihren neuen Test grün zu machen) werden Sie die Testmethode noch viele Male ausführen, typischerweise im Abstand von wenigen Sekunden. Das geht in jeder modernen IDE mit einem Mausklick oder mit einer bestimmten Tastenkombination.[3]

3. [Strg]+[F11] in Eclipse (wenn Sie im Preferences-Dialog als Launch-Operation »Always launch the previously launched application« eingestellt haben) bzw. [Shift]+[F10] in IntelliJ IDEA. In Netbeans müssen Sie leider immer erst zum Test wechseln, bevor Sie ihn mit [Strg]+[F6] ausführen können.

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

4.3 Den roten Test grün machen

Sollten Sie, wie in unserem Beispiel, Testcode geschrieben haben, der nicht kompiliert, so muss der Test zuerst einmal lauffähig gemacht werden. Hierzu legen Sie alle dafür benötigten Klassen und Methoden an. Auch dabei hilft Ihnen Ihre Java-IDE, indem sie Ihnen sogenannte Quickfixes anbietet, wenn Sie den Cursor dort in Ihrer Testmethode positionieren, wo ein Kompilierfehler angezeigt wird, und anschließend entweder auf das kleine Glühlampen-Symbol klicken oder die für Ihre IDE passende Tastenkombination drücken.[4] Typische Quickfixes sind das automatische Anlegen neuer Klassen, das Importieren bestehender Klassen sowie das Anlegen neuer Methoden, wobei die IDE anhand des nicht kompilierenden Aufrufs automatisch erkennt, welche Signatur eine neue Methode haben muss. So können Sie sich viel Tipparbeit sparen.

Geraten Sie dabei aber nicht in Versuchung, irgendeine der Klassen oder Methoden, die Sie neu anlegen, schon zu implementieren. Konzentrieren Sie sich zunächst auf das Ziel, Ihr Projekt wieder kompilieren zu können. Was Sie als Nächstes tatsächlich implementieren sollten, wird Ihnen gleich Ihre IDE sagen, sobald Ihr Projekt wieder kompiliert und Sie Ihre neue Testmethode tatsächlich ausführen können. Ich habe deshalb meine IDE so konfiguriert, dass jede neue erzeugte Methode automatisch immer folgende Zeile enthält:

```
throw new UnsupportedOperationException("Not implemented yet.");
```

Um unser Beispiel kompilieren zu können, habe ich zum einen die Klasse `HighScoreList` im Ordner für Produktionscode angelegt:

```
public class HighScoreList {

    public void update(Game game) {
        throw new UnsupportedOperationException("Not implemented yet.");
    }
}
```

Und zum anderen habe ich die folgenden Hilfsmethoden zur Klasse `HighScoreListTest` hinzugefügt:

```
private HighScoreList createHighScoreList(Object... namesAndScores) {
    throw new UnsupportedOperationException("Not implemented yet.");
}

private Game createGameWithPlayerAndScore(String name, int score) {
    throw new UnsupportedOperationException("Not implemented yet.");
}

private Matcher<HighScoreList> is(Object... namesAndScores) {
    throw new UnsupportedOperationException("Not implemented yet.");
}
```

4. [Strg]+[1] in Eclipse, [Alt]+[Enter] in IntelliJ IDEA und Netbeans.

Sobald Sie alle benötigen Klassen und Methoden angelegt haben, sodass Ihr Projekt wieder erfolgreich kompiliert werden kann, führen Sie Ihren Test aus, um zu sehen, ob und warum er fehlschlägt. Die erste Exception, die beim Ausführen von `test_update()` jetzt auftritt, ist natürlich eine `UnsupportedOperationException`, und zwar diejenige, die von der Hilfsmethode `createHighScoreList` geworfen wird.

Um diese Hilfsmethode zu implementieren, muss auch Produktionscode angefasst werden, nämlich die Klasse `HighScoreList`. Im Übrigen, falls Sie beim Implementieren einer Methode merken, dass Sie relativ komplizierten Code schreiben, können Sie jederzeit Ihren aktuellen TDD-Zyklus pausieren und einen neuen TDD-Zyklus starten. Ein guter Indikator dafür ist, wenn Sie länger als 5 Minuten Code schreiben, ohne einen Test auszuführen. Der ein oder andere mag beispielsweise für die Implementierung der Hilfsmethode `createHighScoreList` zunächst einen Test für das Erzeugen und Füllen einer `HighScoreList` schreiben und diesen grün machen, bevor der TDD-Zyklus für `test_update()` fortgesetzt wird.

Meine Implementierung von `createHighScoreList` sieht so aus:

```
private HighScoreList createHighScoreList(Object... namesAndScores) {
    List<HighScoreList.Entry> entries = new ArrayList<HighScoreList.Entry>();
    for (int i = 0; i < namesAndScores.length; i += 2) {
        HighScoreList.Entry entry = new HighScoreList.Entry();
        entry.playerName = (String) namesAndScores[i];
        entry.score = (Integer) namesAndScores[i + 1];
        entries.add(entry);
    }
    return new HighScoreList(entries);
}
```

Eventuell missfällt Ihnen, dass die Methode ein `Object`-Array als Parameter hat und deshalb die beiden Casts `(String)` und `(Integer)` nötig sind. Ja, auch ich betrachte Casts im Produktionscode als *Code-Smell*[5] und bin der Meinung, dass man Casts generell vermeiden sollte, damit Typfehler (Type Errors) schon beim Schreiben von Code bzw. beim Kompilieren aufgedeckt werden und nicht erst zur Laufzeit durch das Auftreten einer `ClassCastException`.

Aber Testcode ist nicht gleich Produktionscode. Wenn Sie sich bewusst machen, dass dies nur eine Hilfsmethode ist, die ausschließlich von Ihren Tests benutzt wird, finde ich den Vorteil der guten Lesbarkeit des Codes, der diese Hilfsmethode aufruft, größer als den Nachteil, dass ein Typfehler erst zur Ausführungszeit entdeckt werden würde. Schließlich wird der Testcode beim testgetriebenen Entwickeln ja immer unmittelbar nach dem Kompilieren ausgeführt, sowohl auf dem eigenen Rechner als auch später auf dem Continuous-Integra-

5. Code Smell – auf Deutsch: übelriechender Code, eine Metapher für Code, der zwar funktioniert, aber gewisse Qualitätsmängel aufweist. Beispiele dafür sind sehr langer oder sehr komplizierter Code, nichtssagende Bezeichner oder der Klassiker: duplizierter Code (siehe auch *http://de.wikipedia.org/wiki/Smell_(Programmierung)*).

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

tion-Server. Ein Laufzeitfehler im Testcode wird also sehr wohl bereits beim Bauen der Software entdeckt – im Gegensatz zu einem Laufzeitfehler im Produktionscode, der sich durchaus lange verstecken kann.

Die Klasse `HighScoreList` sieht jetzt so aus:

```
public class HighScoreList {

    static class Entry {
        String playerName;
        int score;
    }

    final List<Entry> entries;

    @TestOnly
    HighScoreList(List<Entry> entries) {
        this.entries = entries;
    }

    public void update(Game game) {
        throw new UnsupportedOperationException("Not implemented yet.");
    }
}
```

Wie Sie sehen, habe ich eine innere Klasse `Entry` eingeführt, die den Namen eines Spielers sowie dessen erreichte Punktzahl enthält. Außerdem hat die Klasse `HighScoreList` jetzt die Membervariable `entries` sowie einen Konstruktor, der eine Liste von `Entry`-Objekten entgegennimmt.

Beachten Sie die Sichtbarkeit aller neu hinzugekommenen Elemente. Sowohl auf die innere Klasse `Entry` als auch auf den neuen Konstruktor kann nur im aktuellen Package zugegriffen werden. Das ist ganz typisch für Code, der testgetrieben entwickelt wurde: Eine der vielen Regeln der Java-Entwicklung besagt ja, dass die Sichtbarkeit von neuen Membern zunächst so klein wie möglich gehalten werden soll, damit man später möglichst viele Freiheiten beim Refactoring hat. Die kleinste Sichtbarkeitsstufe ist `private`. Aber auf `private` Member könnte die Testklasse `HighScoreListTest` nicht zugreifen, weshalb ich die nächstkleinere Sichtbarkeitsstufe gewählt habe, nämlich *package private*.

Auffällig ist außerdem auch die Annotation `@TestOnly` am Konstruktor. Diese wird von IntelliJ IDEA zur Verfügung gestellt (diejenige IDE, die ich zum Entwickeln des Beispiels benutzt habe). Sie drückt aus, dass die annotierte Methode bzw. der annotierte Konstruktor ausschließlich von Testcode aufgerufen werden sollte und nicht von Produktionscode. Diese Annotation hat nicht nur dokumentierenden Charakter; IntelliJ IDEA überprüft auch automatisch, ob irgendwelcher Produktionscode eine mit `@TestOnly` annotierte Methode aufruft. Ist dies der Fall, so erscheint sofort ein Warnhinweis.

Die Guava-Bibliothek[6] stellt übrigens mit `@VisibleForTesting` eine ähnliche Annotation zur Verfügung, die aber lediglich dokumentierenden Charakter hat. Methoden, die mit dieser Annotation versehen sind, sollten ebenfalls nicht von anderen Klassen im Produktionscode benutzt werden, da die Sichtbarkeit solcher Methoden nur für Testzwecke erweitert wurde.

Im Beispiel habe ich den Konstruktor, der eine Liste von `Entry`-Objekten akzeptiert, deshalb mit `@TestOnly` annotiert, da ich im Moment noch nicht sehe, dass dieser Konstruktor sinnvoll für den Produktionscode sein könnte, schließlich ist `Entry` ja ein Implementierungsdetail der Klasse `HighScoreList`, das nicht nach außen sichtbar sein sollte.

Diese Entscheidung führt zunächst dazu, dass im Produktionscode keine Instanz der Klasse `HighScoreList` angelegt werden kann, da der einzige existierende Konstruktor nur vom Testcode aufgerufen werden soll. Dieses Problem könnte schnell gelöst werden, indem zu der Klasse ein Default-Konstruktor ohne Parameter hinzufügt wird, der ein leeres `HighScoreList`-Objekt erzeugt. Aber dies habe ich mit Absicht nicht getan, denn eine weitere wichtige Regel beim testgetriebenen Entwickeln besagt:

Schreiben Sie nur so viel Produktionscode, wie nötig ist, um Ihren roten Test grün zu machen.

Würde ich bereits jetzt einen weiteren Konstruktor hinzufügen, hätte ich ungetesteten Produktionscode geschrieben. Das kann man machen, aber ich versuche dies stets zu vermeiden. Zum einen habe ich größeres Vertrauen in getesteten Code, und zum anderen möchte ich die Menge an Produktionscode immer so klein wie möglich halten. Denn je weniger Zeilen Produktionscode ein Projekt hat, desto besser kann es verstanden und gewartet werden, und desto kleiner ist die Wahrscheinlichkeit für Fehler, wenn man den Statistiken glaubt, dass im Mittel 2 bis 3 Programmierfehler pro 1000 Zeilen Code zu finden sind.[7]

Nachdem ich die erste Hilfsmethode `createHighScoreList` implementiert und die dazu nötigen Änderungen im Produktionscode vorgenommen habe, führe ich den `HighScoreListTest` erneut aus, um zu sehen, wie weit er jetzt kommt. Diesmal läuft er schon etwas weiter und steigt erst beim Aufruf von `createGameWithPlayerAndScore` wieder aus.

Jetzt wird Ihnen vielleicht auch klar, warum es »testgetriebene« Entwicklung heißt: Ich führe den Test immer wieder aus und lasse mich beim Programmieren von den dabei auftretenden Fehlern treiben. Jede Exception zeigt mir, was ich als Nächstes programmieren muss – in diesem Fall die Hilfsmethode `createGameWithPlayerAndScore`, die ich wie folgt umgesetzt habe:

6. *http://guava-libraries.googlecode.com*
7. *http://de.wikipedia.org/wiki/Programmfehler*

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

```
private Game createGameWithPlayerAndScore(String name, int score) {
    Player player = mock(Player.class);
    when(player.getName()).thenReturn(name);
    Game game = mock(Game.class);
    when(game.getPlayer()).thenReturn(player);
    when(game.getScore()).thenReturn(score);
    return game;
}
```

Diese Hilfsmethode erzeugt kein echtes `Game`-Objekt, sondern ein sogenanntes *Mock-Objekt* (siehe Kap. 6). Hierfür habe ich die Bibliothek Mockito[8] verwendet, die durch ihre gut lesbare API besticht.

Das mit `mock(Game.class)` erzeugte Mock-Objekt verfügt zwar über alle Methoden der Klasse `Game`, aber keine dieser Methoden macht irgendetwas. Und die Methoden, die einen Rückgabewert haben, geben (falls sie aufgerufen werden) je nach Typ des Rückgabewerts entweder `null` oder `0` zurück.

Da die Methode `HighScoreList.update(...)` sowohl `game.getPlayer().getName()` als auch `game.getScore()` aufruft, habe ich – wie im obigen Listing zu sehen ist – das zurückgegebene `Game`-Mock-Objekt so konfiguriert, dass die Aufrufe dieser Methoden den gewünschten Spielernamen bzw. die gewünschte Punktzahl zurückliefern.

Die Verwendung von Mock-Objekten ist typisch für Unit-Tests, die ja nur eine bestimmte Unit testen sollen. In unserem Beispiel ist die getestete Unit die `update`-Methode der Klasse `HighScoreList`. Um diese zu testen, ist es nicht notwendig, tatsächlich ein `Game`-Objekt oder ein `Player`-Objekt zu erzeugen (die wiederum vielleicht noch andere Objekte für ihre Erzeugung benötigen).

Die Verwendung von Mock-Objekten bietet neben der Tatsache, dass Ihre Tests sehr schnell laufen, noch einen weiteren Vorteil: Ihre Tests hängen nur von wenigen Klassen des Produktionscodes ab. So ist die Testmethode `test_update()` unabhängig von den genauen Implementierungsdetails der Klassen `Game` und `Player`. Diese Klassen können also fast beliebig geändert werden, ohne dass dies die Testmethode beeinflussen würde. Einzige Ausnahme sind die Signaturen der Methoden `Game.getPlayer()`, `Game.getScore()` sowie `Player.getName()`. Aber falls sich eine dieser Methoden ändert, muss ja sowieso nicht nur der Test, sondern auch die Implementierung der `update`-Methode in der Klasse `HighScoreList` geändert werden.

Versuchen auch Sie, wenn Sie Unit-Tests schreiben, möglichst nur das Stückchen Code zu durchlaufen, das Sie tatsächlich testen wollen. Sollte die zu testende Klasse Abhängigkeiten zu anderen Klassen haben, dann versuchen Sie zuerst, falls möglich, einfach `null` im Konstruktor oder als Methodenparameter zu übergeben. Vielleicht wird die Abhängigkeit ja von dem zu testenden Produktionscode gar nicht benutzt. Falls doch, verwenden Sie Mock-Objekte. So können Sie

8. *http://www.mockito.org/*

zum einen nicht nur den *Happy Path* testen, sondern auch Fehlersituationen nachstellen (siehe Kap. 9).

Zum anderen führt der Einsatz von Mock-Objekten meist auch zu einer guten Architektur, da Abhängigkeiten explizit in Form von Konstruktorparametern bzw. Methodenparametern ausgedrückt werden. Damit folgen Sie dem Prinzip der *Dependency Inversion*, einem der Grundprinzipien für gute objektorientierte Programmierung.[9]

Nach der Implementierung der zweiten Hilfsmethode `createGameWithPlayerAndScore(...)` führe ich `test_update()` erneut aus. Hier ist die Testmethode noch einmal, damit Sie nicht zurückblättern müssen:

```
@Test
public void test_update() {
    HighScoreList highScoreList = createHighScoreList(
        "Michael", 12345,
        "Paul", 9876
    );
    Game game = createGameWithPlayerAndScore("Daniel", 10000);

    highScoreList.update(game);

    assertThat(highScoreList, is(
        "Michael", 12345,
        "Daniel", 10000,
        "Paul", 9876
    ));
}
```

Da jetzt alle Hilfsmethoden für das Setup des Testszenarios implementiert sind, kommt der Testlauf diesmal bis zum Aufruf von `highScoreList.update(game)` – also bis zu der Methode, für die der Test eigentlich geschrieben wurde –, bevor er wieder mit einer Exception aussteigt, weil diese Methode immer noch wie folgt aussieht:

```
public void update(Game game) {
    throw new UnsupportedOperationException("Not implemented yet.");
}
```

Jetzt denken Sie vielleicht, da der Test bis zur `update`-Methode gekommen ist, dass diese als Nächstes implementiert werden sollte. Aber um weiter durch den Test zu kommen, reicht zunächst folgende »Implementierung« völlig aus:

```
public void update(Game game) {
    // TODO: implement me
}
```

9. *http://de.wikipedia.org/wiki/Prinzipien_Objektorientierten_Designs*

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

Das ist natürlich noch nicht die endgültige Lösung, und wir werden später darauf zurückkommen, aber für den Moment ist der TODO-Kommentar völlig ausreichend, um weiter durch den Test zu kommen, denn ich versuche, jeden Test immer erst einmal so weit fertigzustellen, dass er mit einem aussagekräftigen `AssertionError` fehlschlägt statt mit einer `Exception`. In diesem Fall bedeutet das, dass ich den `Matcher` für die Assertion in `test_update()` noch vorher schreiben möchte, damit ich sehe, dass er ordnungsgemäß funktioniert und darüber hinaus eine aussagekräftige Fehlermeldung produziert. Und dafür muss der Test erst einmal rot bleiben.

Nachdem ich die `throw`-Anweisung in der `update`-Methode durch den TODO-Kommentar ersetzt habe, führe ich den Test erneut aus. Jetzt sehe ich, dass er bis zu der Assertion kommt, wo er allerdings wieder mit einer Exception aussteigt, da ich die Factory-Methode für den Hamcrest-Matcher noch nicht implementiert habe, was ich deshalb als Nächstes mache. Hier das Ergebnis:

```
private Matcher<HighScoreList> is(Object... namesAndScores) {
    final HighScoreList expected = createHighScoreList(namesAndScores);
    return new CustomTypeSafeMatcher<HighScoreList>(expected.toString()) {
        @Override
        protected boolean matchesSafely(HighScoreList actual) {
            int n = expected.entries.size();
            if (actual.entries.size() != n) {
                return false;
            }
            for (int i = 0; i < n; ++i) {
                HighScoreList.Entry expectedEntry = actual.entries.get(i);
                HighScoreList.Entry actualEntry = actual.entries.get(i);
                if (!reflectionEquals(expectedEntry, actualEntry)) {
                    return false;
                }
            }
            return true;
        }
    };
}
```

Um die eigentliche `HighScoreList` mit der erwarteten zu vergleichen, erzeuge ich zunächst aus den übergebenen Namen und Punktzahlen die Instanz `expected`. Hierfür kann ich die Hilfsmethode `createHighScoreList` wiederverwenden, die ich bereits für den Arrange-Teil des Beispieltests geschrieben habe.

Für den zurückgegebenen `Matcher` benutze ich eine anonyme innere Klasse, die von `CustomTypeSafeMatcher` abgeleitet ist – eine der Basisklassen, die Hamcrest für die Implementierung eigener `Matcher` zur Verfügung stellt (siehe auch Abschnitt 5.3).

In der `for`-Schleife, in der die `Entry`-Objekte verglichen werden, benutze ich außerdem die `reflectionEquals`-Methode der Klasse `EqualsBuilder` aus der Bibliothek commons-lang[10] des Apache-Commons-Projekts. Diese nützliche Methode

verwende ich in vielen Tests, die ich schreibe. Sie vergleicht die Klassen sowie alle Felder zweier Objekte (auch private, ausgenommen sind lediglich `transient`-Felder).

Alternativ hätte man an der `Entry`-Klasse auch die `equals`-Methode überschreiben können, aber von dieser Vorgehensweise rate ich generell ab: Zum einen sollten Sie Testcode möglichst aus dem Produktionscode heraushalten. Und Sie hätten in diesem Fall die `equals`-Methode ausschließlich für Testzwecke zum Produktionscode hinzugefügt. Zum anderen signalisiert eine überschriebene `equals`-Methode im Produktionscode, dass Sie sich über Objektgleichheit Gedanken gemacht haben, was ein ganz eigenes Problemfeld ist. Und schließlich steht die Frage im Raum, wie verhalten sich Ihre Tests, wenn die Implementierung der `equals`-Methode geändert wird? Wenn Sie also nicht gerade die `equals`-Methode selbst testen oder es mit unveränderbaren Wertobjekten (*immutable value objects*[11]) zu tun haben, sollten Sie `reflectionEquals` bevorzugen, um Ihre Tests unabhängig von der Implementierung der `equals`-Methode zu halten.

Nachdem nun die letzte Hilfsmethode in der Testklasse implementiert ist, schlägt der Test nicht mehr mit einer Exception fehl, sondern mit folgendem `AssertionError`:

```
java.lang.AssertionError:
Expected: HighScoreList@7d3eea2f
     but: was <HighScoreList@22ea86e3>
```

Das ist allerdings noch keine sehr aussagekräftige Fehlermeldung, weshalb ich in der Klasse `HighScoreList` die `toString`-Methode noch wie folgt überschreibe:

```
@Override
public String toString() {
    final StrBuilder sb = new StrBuilder();
    sb.appendln(getClass().getSimpleName());
    for (Entry entry : entries) {
        String name = entry.playerName;
        String score = Integer.toString(entry.score);
        sb.append('\t')
            .append(name)
            .append(' ')
            .appendPadding(30 - name.length() - score.length(), '.')
            .append(' ')
            .appendln(score);
    }
    return sb.toString();
}
```

10. *http://commons.apache.org/lang/*
11. *http://c2.com/cgi/wiki?ValueObject*

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

Sie mögen jetzt vielleicht einwenden, dass ich Ihnen gerade erklärt habe, dass man Testcode aus dem Produktionscode heraushalten soll, und ich gerade genau das Gegenteil gemacht habe. Aber wie heißt es so schön: keine Regel ohne Ausnahme. Und die `toString`-Methode ist genau so eine Ausnahme zu dieser Regel, da die `toString`-Methode sowieso nur zu Debug-Zwecken benutzt werden sollte und Testen schließlich auch eine Art von Debugging ist.

Die hierbei verwendete Klasse `StrBuilder` stammt ebenfalls aus der commons-lang-Bibliothek und bietet einige Komfortmethoden, wie `appendln` und `appendPadding`, die die bekannte Klasse `StringBuilder` nicht hat.

Wie auch immer, mit der obigen `toString`-Methode schlägt `test_update()` jetzt mit folgendem `AssertionError` fehl:

```
java.lang.AssertionError:
Expected: HighScoreList
    Michael .................. 12345
    Daniel ................... 10000
    Paul ...................... 9876

  but: was <HighScoreList
    Michael .................. 12345
    Paul ...................... 9876
>
```

Das ist eine aussagekräftige Fehlermeldung, und jetzt folgt als letzter Schritt auf dem Weg hin zu einem grünen Test die Implementierung der Methode `update` in der Klasse `HighScoreList`. Eine Lösung, die den Beispieltest grün machen würde, ist folgende:

```
public void update(Game game) {
    final Entry newEntry = new Entry();
    newEntry.playerName = game.getPlayer().getName();
    newEntry.score = game.getScore();
    this.entries.add(1, newEntry);
}
```

Das ist zwar schön einfach, aber natürlich keine akzeptable Lösung, da sie nur den getesteten Fall berücksichtigt und nicht allgemeingültig ist. Auch wenn eine wichtige Regel bei TDD lautet, dass man nur so viel Produktionscode schreiben soll, um den roten Test grün zu machen, sollten Sie diese Regel nicht wortwörtlich interpretieren. Worum es bei dieser Regel geht, ist die Extreme-Programming-Praxis: *Do The Simplest Thing That Could Possibly Work*.[12] Die Lösung soll zwar möglichst einfach sein, aber immer auch allgemeingültig (und nicht nur einen bestimmten Testfall grün machen). Sie wollen sich doch nicht selbst betrügen, oder?

12. *http://c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork*

Eine echte Lösung ist zum Beispiel diese hier:

```
public void update(Game game) {
    final Entry newEntry = new Entry();
    newEntry.playerName = game.getPlayer().getName();
    newEntry.score = game.getScore();
    int i = 0;
    while (i < entries.size()) {
        if (entries.get(i).score < newEntry.score) {
            break;
        }
        ++i;
    }
    entries.add(i, newEntry);
}
```

Oft stolpert man beim Implementieren über die sogenannten *Edge Cases* (Randfälle). Als ich die obige Implementierung geschrieben habe, sind mir sofort folgende aufgefallen: Was ist, wenn die `HighScoreList` leer ist? Und was ist, wenn es bereits einen Eintrag mit der gleichen Punktzahl gibt? Auch wenn dies im TDD-Zyklus nicht explizit vorgesehen ist, macht es an dieser Stelle Sinn, zusätzliche Tests für die Randfälle zu schreiben, wie beispielsweise diesen hier:

```
@Test
public void test_update_with_empty_list() {
    HighScoreList highScoreList = createHighScoreList();
    Game game = createGameWithPlayerAndScore("Daniel", 10000);

    highScoreList.update(game);

    assertThat(highScoreList, is("Daniel", 10000));
}
```

Diesen Test habe ich in unter einer Minute programmiert, indem ich die Methode `test_update()` kopiert, umbenannt und angepasst habe. Alle Hilfsmethoden waren schon da. Und der Test war auch sofort grün. Wenig Aufwand, großer Nutzen: Ich habe jetzt mehr Vertrauen in die Richtigkeit meiner Implementierung.

Idealerweise erstellen Sie für jeden möglichen Ablaufpfad durch Ihre gerade implementierte Methode einen Test, insbesondere auch für die Pfade, in denen Exceptions gefangen und behandelt werden (siehe Kap. 9). Hierbei ist die Messung und Visualisierung der sogenannten *Code Coverage* (im Deutschen *Testabdeckung* genannt) hilfreich, mit der solche Codestellen aufgedeckt werden können, die noch von keinem Test durchlaufen werden. Wie Sie die Testabdeckung mit Ant, Maven oder Ihrer IDE ermitteln können, zeige ich Ihnen in den letzten vier Kapiteln dieses Buches.

Gelegentlich kann es vorkommen, dass Sie einen Test schreiben und in Ihrer IDE ausführen, in der Erwartung, dass er rot ist, aber Ihre IDE präsentiert Ihnen einen grünen Balken. Das kann zweierlei Gründe haben: Entweder ist Ihnen beim

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

Schreiben des Tests ein Fehler unterlaufen – schauen Sie sich deshalb in einem solchen Fall immer nochmals Ihre Assertions genau an –, oder Ihre Applikation kann bereits das, wofür Sie gerade einen Test geschrieben haben.

Auch wenn Letzteres eher selten der Fall ist, vergessen Sie nicht, jeden fertiggestellten Test zuerst auszuführen, um sicherzustellen, dass er tatsächlich fehlschlägt, bevor Sie Hand an Ihren Produktionscode legen, um ihn grün zu machen.

Noch einmal zurück zu unserem Beispiel: Für den Fall, dass bereits ein Eintrag mit gleicher Punktzahl existiert, ist zu entscheiden, wo genau der neue Eintrag eingeordnet werden soll: über oder unter dem bereits existierenden Eintrag?

Falls Sie agil entwickeln, können Sie in solchen Fällen den *Product Owner* oder den Kunden sofort fragen. Steht Ihnen diese Möglichkeit nicht offen und gehen die Anforderungen auf den von Ihnen entdeckten Spezialfall nicht ein, so können Sie entweder auf die Klärung der aufgetauchten Frage warten, oder aber Sie treffen selbst eine Entscheidung. In diesem Fall sollten Sie Ihre Entscheidung aber auch kommunizieren (zum Beispiel durch einen entsprechenden Kommentar an der Anforderung). So oder so sollten Sie für alle Spezialfälle einen Test schreiben.

4.4 Codereview und Refactoring

Angenommen, Sie haben einen roten Test geschrieben, diesen grün gemacht und sogar zusätzlich noch Tests für die Randfälle geschrieben, die ebenfalls alle grün sind. Die meisten Programmierer denken jetzt: »Super, ich bin fertig. Ab damit in die Versionskontrolle, und auf zur nächsten Aufgabe.«

Dabei ist der TDD-Zyklus an dieser Stelle aber noch gar nicht zu Ende. Nachdem ein Test grün gemacht wurde und bevor der nächste rote Test geschrieben wird, kommt nämlich noch das *Refactoring*. Funktionierender Code ist gerade in größeren Softwareprojekten nicht ausreichend. Wichtig ist auch, dass der Code gut lesbar und somit gut wartbar ist. Und genau, um dies zu erreichen, dient der besagte dritte Schritt im TDD-Zyklus.

Leider wird er in der Praxis viel zu oft vergessen, und das ist auch gar nicht so unverständlich, denn in erster Linie betrachten die meisten Programmierer es als ihren Job, neue Anforderungen zu implementieren. Aus dieser Sicht ist die Arbeit tatsächlich abgeschlossen, wenn alle Tests grün sind. Dies geht mit dem Erfolgserlebnis einher, den Test, den man initial geschrieben hat und von dem man sich vielleicht mehrere Stunden hat treiben lassen, endlich grün zu sehen. Das ist so wie ein Zieleinlauf und erscheint somit wie das Ende der aktuellen Aufgabe.

Aber das »Grün-Werden« eines Tests sollte eigentlich nur als Etappensieg betrachtet werden. Jeder Programmierer sollte sich nicht nur für die äußere Qualität der von ihm programmierten Software verantwortlich fühlen, sondern auch für die innere Qualität. Dazu gehört das Einhalten von Coding-Konventionen, Architekturvorgaben und Programmierprinzipien, wie beispielsweise DRY (*Don't Repeat Yourself*).[13] Hierfür möchte ich Ihnen das Buch »Clean Code« von

Robert C. Martin [Martin 2008] wärmstens ans Herz legen, das ausführlich beschreibt, was guten Code ausmacht.

Aber auch wenn Sie sich darüber im Klaren sind, dass guter Code wichtig ist, ist es immer noch eine große Herausforderung, nach jedem grünen Test innezuhalten und die innere Qualität des gerade von Ihnen geschriebenen Codes zu bewerten und eventuell zu verbessern. Schließlich haben Sie den Code ja gerade erst – so gut, wie Sie es eben können – geschrieben.

Hierbei hilft es, wenn Sie sich den Fokuswechsel ganz klar vor Augen führen: Ihr Fokus auf der vorherigen Etappe war, den Code zum Laufen zu bringen, sodass er korrekt funktioniert. Ihr Fokus jetzt sollte die Frage sein: Ist der neu geschriebene Code auch gut verständlich?

Diese Frage ist übrigens in den allermeisten Fällen wichtiger als die Frage: Ist der neu geschriebene Code auch effizient? Außer bei offensichtlich sehr ineffizienten Codepassagen ist eine gute Verständlichkeit fast immer wichtiger als verfrühte Performance-Optimierung. Die Performance Ihrer Software sollten Sie nur dann verbessern, wenn Sie tatsächlich Performance-Probleme haben. Und auch dann sollten Sie nur denjenigen Stellen optimieren, die Sie beim Profiling als Bottlenecks identifizieren, denn Performance-optimierter Code ist oft kompliziert und schwer verständlich und somit auch schwer wartbar.

Ich habe mir für den bewussten Fokuswechsel folgende Arbeitsweise angewöhnt: Sobald ich meinen initial roten Test endlich grün bekommen habe (also mein neuer Code funktioniert), öffne ich in meiner IDE den Commit-Dialog, um alle Änderungen in die Versionskontrolle zu übernehmen. Aber bevor ich den Commit-Kommentar schreibe und auf den *Commit*-Button drücke, gehe ich noch einmal alle geänderten Dateien einzeln durch und schaue mir jede geänderte Zeile noch einmal genau an. Hierbei ist der Diff-View meiner IDE sehr hilfreich, der auf der linken Seite die aktuelle Version in der Versionskontrolle zeigt und auf der anderen Seite die geänderte Version meiner lokalen Arbeitskopie. Und während ich mir alle meine Änderungen noch einmal ansehe, versuche ich mich in die Rolle eines Programmierers zu versetzen, der diesen Code nicht geschrieben hat, ihn aber später einmal lesen und verstehen muss. Sobald ich dabei über eine Stelle stolpere, an der ich vielleicht noch einen Kommentar ergänzen sollte oder an der eine Methode so lang geworden ist, dass ein Codeabschnitt besser in eine eigene Methode ausgelagert werden sollte, dann breche ich den Commit-Dialog ab und mache das.

Eine andere Variante ist, den Codereview durch eine zweite Person durchführen zu lassen. Wenn Sie ein Ticketsystem (wie beispielsweise JIRA[14] oder Redmine[15]) einsetzen, könnten Sie dafür den Workflow erweitern: Der zuständige Bearbeiter eines Tickets markiert dieses nicht mehr sofort als erledigt, sondern setzt es auf den neuen Status *Feedback* und weist es einer anderen Person zu.

13. *http://de.wikipedia.org/wiki/Prinzipien_Objektorientierten_Designs*
14. *http://www.atlassian.com/software/jira*
15. *http://www.redmine.org/*

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

Diese muss sich dann alle Commits zu dem Ticket noch einmal unter dem Aspekt der guten Verständlichkeit und der Einhaltung der Coding-Konventionen und Architekturregeln anschauen.

Falls Sie ein Scrum- oder Kanban-Board benutzen, können Sie für diesen neuen Zwischenschritt einfach eine zusätzliche Spalte einführen.

Wie auch immer Ihr Entwicklungsprozess aussieht, wichtig ist bei der Einführung von manuellen Codereviews durch eine zweite Person, dass – falls verbesserungswürdige Stellen im neuen Code gefunden werden – nicht der Reviewer das Refactoring selbst durchführt, sondern der ursprüngliche Autor des Codes. Dazu muss das Ergebnis des Reviews kommuniziert werden – entweder im direkten Gespräch, zum Beispiel in einer Pair-Programming-Session, oder durch einen Kommentar im Ticketsystem, wenn das Ticket auf *Reopened* gesetzt und dem initialen Bearbeiter wieder zugewiesen wird. Alternativ können Sie auch eine spezielle Codereview-Software (z. B. Crucible[16] oder Review Board[17]) einsetzen, die es Ihnen erlaubt, geänderte Codestellen zu kommentieren.

Egal wie Sie vorgehen: Nur wenn dem ursprünglichen Autor das Review-Ergebnis kommuniziert wird und dieser das Refactoring selbst durchführen muss, stellt sich ein Lerneffekt für diesen Programmierer ein. Nur so lernt er, welche Coding-Konventionen er verletzt hat oder an welchen Stellen er von der gewünschten Architektur abgewichen ist (die ihm vielleicht gar nicht bewusst war). Oder er lernt neue Hilfsklassen und -methoden kennen, die er anstelle eigener Implementierungen in Zukunft nutzen kann.

Dass dieser Knowledge-Transfer vom Reviewer zum Programmierer stattfindet, liegt in der Verantwortung des Reviewers. Aber auch hier hilft eine einfache Regel:

Ein Reviewer darf keinen Code ändern.

Allerdings sind manuelle Codereviews – ganz genauso wie manuelles Testen – teuer und fehleranfällig (in dem Sinne, dass leicht etwas übersehen wird). Und genauso wie automatisierte Tests eine gute Alternative zu manuellen Tests sind, sind automatisierte Codereviews eine gute Alternative zu manuellen Reviews. Hierüber könnte ich ein zweites Buch schreiben, aber Sie halten ja ein Buch über JUnit in der Hand, deshalb hier nur ganz kurz:

Ein bekanntes Open-Source-Tool zum Überprüfen, ob sich der Quellcode aller Java-Klassen an vorgegebene Coding-Konventionen hält, ist Checkstyle[18]. Wenn Sie dieses Tool ebenfalls einsetzen, benutzen Sie es hoffentlich nicht nur dazu, um die Regelverletzungen zu zählen. Wozu soll das gut sein? Lassen Sie den

16. *http://www.atlassian.com/software/crucible*
17. *http://www.reviewboard.org/*
18. *http://checkstyle.sourceforge.net/*

Build fehlschlagen, sobald eine Regelverletzung erkannt wird. Nur so kann das Einhalten von Coding-Konventionen erzwungen werden.

Und wenn Sie oder andere Entwickler in Ihrem Team davon genervt sind, dass dadurch dauernd der Build bricht, würde ich deshalb nicht den Build-Prozess infrage stellen, sondern Ihre Coding-Konventionen: Sind darin vielleicht Richtlinien enthalten, die immer wieder verletzt werden? Dann zögern Sie nicht, Ihre Coding-Konventionen Ihren Bedürfnissen anzupassen und unbeliebte Regeln zu ändern oder zu löschen. Falls jedoch alle Entwickler mit den vereinbarten Coding-Konventionen einverstanden sind, sollten sie auch lernen, sich daran zu halten, und zwar bevor Änderungen in die Versionskontrolle eingestellt werden.

Checkstyle kann aber noch einiges mehr, als nur das Einhalten von Coding-Konventionen zu überprüfen. Es kann zum Beispiel auch duplizierten Code erkennen, was ebenfalls zu den Features der Open-Source-Tools PMD[19] und ConQAT[20] gehört. Das Problem von Codeduplizierung ist (abgesehen von der reinen Zunahme der *Lines of Code* in Ihrer Codebasis), dass beim Kopieren von Quellcode immer auch alle darin enthaltenen Fehler kopiert werden. Wird später ein solcher Fehler entdeckt, aber nur an einer Stelle behoben, existiert er immer noch in allen sogenannten Codeklonen.

Auch hier empfehle ich Ihnen, wenn Sie ein Tool zum Erkennen von Codeduplizierungen einsetzen: Tolerieren Sie keine (oder zumindest keine neuen) Codeklone, sondern lassen Sie Ihren Build fehlschlagen, wenn eine Codeduplizierung entdeckt wird. Und falls tatsächlich einmal die Situation eintritt, dass die Codeduplizierung gerechtfertigt ist (oder wenn es sich um einen Fehlalarm handelt), stellen Sie nicht den Build-Prozess infrage, sondern pflegen Sie stattdessen eine Liste mit den erlaubten Ausnahmen.

Beim automatischen Finden von typischen Programmierfehlern leistet FindBugs[21] – ein Tool zur statischen Codeanalyse von Java-Klassen – gute Arbeit, weshalb ich Ihnen nur empfehlen kann, es ebenfalls einzusetzen.

Und schließlich gibt es auch noch Tools, um zu überprüfen, dass die von der Architektur vorgegebenen Abhängigkeiten von allen Klassen eingehalten werden, also dass zum Beispiel keine Klasse aus dem Package für den Datenbankzugriff Klassen benutzt, die sich im Package für die GUI befinden. Solche Architekturverletzungen lassen sich zum Beispiel mit den Open-Source-Tools Macker[22] oder auch mit ConQAT finden. Und es wird Sie an dieser Stelle wahrscheinlich nicht überraschen, wenn ich Ihnen empfehle, auch hier bei einer Regelverletzung den Build fehlschlagen zu lassen, denn niemand wird sich sonst jemals die Zeit nehmen, bestehende Regelverletzungen zu beheben.

Und falls Sie firmen- oder projektspezifische Programmierregeln haben, die noch von keinem Tool unterstützt werden, können Sie entweder eines der genann-

19. *http://pmd.sourceforge.net/*
20. *https://www.conqat.org/*
21. *http://findbugs.sourceforge.net/*
22. *http://innig.net/macker/*

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

ten Tools um eigene Regeln erweitern, oder aber Sie schreiben einen JUnit-Test, der die Einhaltung Ihrer Architekturregeln automatisch überprüft (mehr dazu in Abschnitt 10.4).

Aber vergessen Sie nicht, wenn Sie anfangen, Ihren Codereview zu automatisieren, dass es nicht reicht, nur Statistiken über die Verletzungen aller Regeln zu produzieren. Regelverletzungen sollten den Build brechen, damit diese auch behoben werden.

Weiterhin sollte es für jeden Programmierer einfach sein, den automatisierten Codereview auf seinem lokalen Entwicklungsrechner auszuführen, und die Entwickler sollten dazu aufgefordert werden, dies auch tatsächlich immer dann zu tun, wenn sie die zweite Etappe im TDD-Zyklus beendet haben, also einen roten Test grün gemacht haben. Hierfür lege ich in von mir betreuten Projekten gerne eine Testsuite mit dem Namen `AllArchitectureTests` an, die neben projektspezifischen Architekturtests unter anderem auch Tests beinhaltet, die Checkstyle und FindBugs ausführen.

Falls manuelle Codereviews zu Ihrem Entwicklungsprozess gehören, hier noch ein Tipp: Verzichten Sie darauf, die Testsuite `AllArchitectureTests` zu einem Bestandteil Ihres normalen Builds zu machen und beschränken Sie sich darauf, diese bei jedem manuellen Codereview auf Ihrem Computer auszuführen. Und wenn Sie dann von FindBugs oder einem Ihrer eigenen Architekturtests auf eine Stelle im Produktionscode hingewiesen werden, die wahrscheinlich einen Fehler oder eine Regelverletzung aufweist, dann schauen Sie sich auch den Quellcode in der unmittelbaren Umgebung dieser Stelle genauer an – oft können Sie dort noch mehr verbesserungswürdigen Code finden.

Im Übrigen sollten sich Review und Refactoring nicht nur auf den Produktionscode beschränken, sondern auch den Testcode mit einbeziehen. Da auch der Testcode mit der Zeit immer umfangreicher wird und bei Änderungen an der Software meistens auch mit geändert werden muss, sollte er ebenfalls gut wartbar sein und deshalb zum Beispiel keinen duplizierten Code enthalten. Allerdings sollten Sie dabei – wenn Sie beispielsweise Testbasisklassen einführen, damit Testhilfsmethoden von mehreren Testklassen benutzt werden können – niemals die gute Verständlichkeit der Testmethoden aus den Augen verlieren.

Ich erstelle neue Testmethoden sehr oft durch Copy & Paste. Da sich viele Testfälle ähneln, ist natürlich auch der entsprechende Testcode ähnlich, und das ist auch in Ordnung so. Codeduplizierung sollte vor allen in den Testhilfsklassen und -hilfsmethoden bekämpft werden.

Lange Rede, kurzer Sinn: Merken Sie sich, dass der TDD-Zyklus drei Etappen hat und das dass »Grün-Machen« des initial geschriebenen, roten Tests nur das Ende der zweiten Etappe ist. Die letzte Etappe ist das Refactoring mit dem Ziel, möglichst gut verständlichen und sauberen Code in die Versionskontrolle einzustellen, der sich sowohl an die Prinzipien der objektorientierten Programmierung als auch an Ihre Coding-Konventionen und Architekturvorgaben hält.

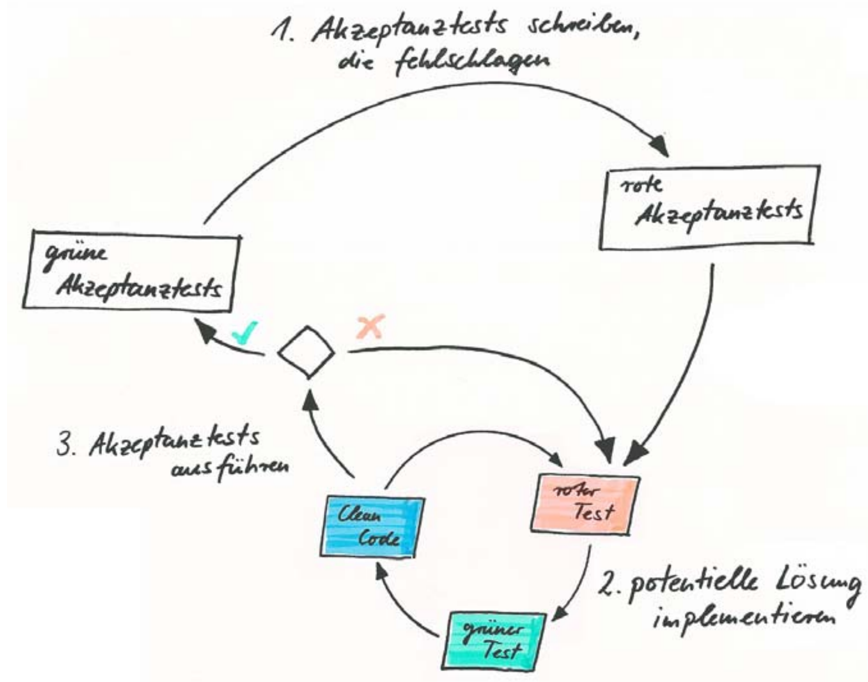
4.5 ATDD – der Kontext für TDD

Sie kennen jetzt den TDD-Zyklus. Und Sie wissen auch, an welcher Stelle dieser anfängt: Als Erstes schreibt man immer einen roten Test. Aber eine Frage bleibt offen: Wann ist man eigentlich fertig mit dem Programmieren?

Kent Beck beantwortet diese Frage so: »Ich bin fertig, wenn mir keine Tests mehr einfallen, die fehlschlagen könnten.«

Keine schlechte Antwort, aber so richtig zufrieden stellt sie mich trotzdem nicht: Was ist, wenn mir aus bloßer Ideenarmut keine Tests mehr einfallen, obwohl für eine gute Testabdeckung noch mehr nötig wären? Wie lange soll ich eigentlich darüber nachgrübeln, ob mir vielleicht noch Tests einfallen?

Ich möchte Ihnen deshalb eine alternative Antwort namens ATDD vorstellen. ATDD steht für *Acceptance Test Driven Development* und bettet den TDD-Zyklus in einen größeren Zyklus ein, der wie folgt aussieht:



In vielen agilen Softwareprojekten werden Anforderungen in Form von sogenannten *User Stories*[23] festgehalten. Für unser Beispiel mit der Highscore-Liste könnte die User Story beispielsweise so aussehen: *Als Spieler möchte ich nach einem Spiel sehen, an welcher Stelle ich in der Highscore-Liste gelandet bin, damit ich für ein neues Spiel motiviert werde.*

23. *http://de.wikipedia.org/wiki/User_Story*

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

Zu einer guten User Story gehören auch Akzeptanzkriterien, an denen fest gemacht wird, wann die User Story als fertig betrachtet wird. (Diese Kriterien sollten vorab mit dem Kunden geklärt werden.) Wenn Sie als Entwickler mit der Implementierung einer User Story anfangen und ATDD praktizieren, dann ist der erste Schritt immer, einen oder auch mehrere Akzeptanztests für die User Story zu schreiben. Diese sollten sich unmittelbar aus den Akzeptanzkriterien der User Story ableiten lassen.

Zum Schreiben der Akzeptanztests stehen Ihnen viele Möglichkeiten offen: ein simpler Texteditor, falls Sie zunächst nur einen Testplan für einen manuellen Test erstellen oder ein Framework wie JBehave[24] einsetzen. Oder Sie benutzen Ihren Webbrowser und ein Wiki, für die es ebenfalls spezielle Tools für automatisierte Akzeptanztests gibt, wie beispielsweise Fitnesse.[25] Oder Sie nehmen einfach Ihre IDE und eine Testbibliothek, die Sie schon kennen, nämlich JUnit.

Im letzteren Fall, auf den ich mich hier beschränken möchte, empfiehlt es sich (nachdem Sie sich für eine Testklasse und den Namen Ihrer Testmethode entschieden haben), den kompletten Testfall zunächst als Kommentar in die Testmethode zu schreiben, genau so wie bereits im vorhergehenden Abschnitt beschrieben.

Auch beim Schreiben eines Akzeptanztests in Java mithilfe von JUnit gelten alle Regeln wie für das Schreiben von guten Unit-Tests (siehe Kap. 7). Allerdings mit zwei Ausnahmen:

- Zum einen sind Akzeptanztests auf einem viel höheren Abstraktionsniveau angesiedelt als Unit-Tests. Während ein Unit-Test für Entwickler gedacht ist, sollte ein guter Akzeptanztest auch von einem Kunden verstanden werden können, der normalerweise keine allzu großen technischen Kenntnisse hat.
- Zum anderen sollten Sie beim Schreiben von Akzeptanztests auf den Einsatz von Mocking verzichten. Während Unit-Tests dafür gedacht sind, ein kleines Stück Code zu testen, wird mit einem Akzeptanztest eine komplette User Story getestet, wobei für gewöhnlich viele Klassen benutzt und viel Code ausgeführt wird. Ein Akzeptanztest ist also auch immer ein Szenariotest.

Sie können einen Akzeptanztest als sogenannten Whitebox-Test erstellen, also als einen Test, der die Klassen Ihres Produktionscodes direkt anspricht. Noch besser ist es jedoch, wenn Sie Ihre Akzeptanztests als Blackbox-Tests schreiben, also im Test Ihre Applikation nur über die Schnittstellen ansprechen, die später auch den Benutzern zur Verfügung stehen. Dann haben Sie nämlich nicht nur einen Akzeptanztest, sondern gleichzeitig auch einen Systemtest.

Wenn Sie beispielsweise eine Webapplikation entwickeln, sollten Ihre Akzeptanztests bzw. Systemtests einen Webserver hochfahren und einen Browser fernsteuern. Gerade beim Schreiben der ersten automatisierten Akzeptanztests bzw. Systemtests müssen Sie deshalb mit einem erheblichen Initialaufwand rechnen, da

24. *http://jbehave.org/*
25. *http://fitnesse.org/*

zunächst einmal die Testfixture programmiert werden muss, die Ihre Applikation startet und nach den Tests wieder beendet. Außerdem müssen Sie Hilfsmethoden schreiben, mit denen Sie Ihre Applikation von außen bedienen können. Nähere Informationen dazu finden Sie in Abschnitt 7.6, in dem das »Page Object Pattern« vorgestellt wird.

Aber auch, wenn im ATDD-Zyklus immer erst alle nötigen Akzeptanztests für die gerade umzusetzende User Story erstellt werden sollen, bevor die Entwicklung im bekannten TDD-Zyklus beginnt, heißt das nicht, dass alle Akzeptanztests bereits vollständig als Java-Code vorliegen müssen, bevor Sie mit der Implementierung beginnen können. Es reicht aus, wenn Sie für jeden Akzeptanztest lediglich ein Testmethode erstellen, in der Sie den Testfall in textueller Form in einem Kommentar beschreiben. So können Sie sehr schnell mit der eigentlichen Implementierung anfangen.

Ganz genauso wird es übrigens auch beim Einsatz von JBehave oder Fitnesse gemacht. Auch dort werden zunächst die Akzeptanztests in textueller Form erstellt. Die Implementierung des Codes, mit dem die textuellen Stories tatsächlich ausgeführt werden können, erfolgt später.

Und wenn Sie denken, dass Sie mit der Implementierung der Lösung fertig sind, können Sie den Akzeptanztest zunächst einmal manuell durchführen, bevor Sie mit der Automatisierung beginnen. Gerade am Anfang eines Projekts, wenn Sie die ersten Testhilfsklassen und -methoden für automatisierte Systemtests schreiben, ist es hilfreich, wenn die Applikation bereits so funktioniert, wie sie funktionieren soll.

Aber um auf die anfängliche Frage dieses Abschnitts zurückzukommen: Wann ist der TDD-Zyklus eigentlich zu Ende? Wahrscheinlich ahnen Sie es schon: Wenn alle Akzeptanztests grün sind. Denn da diese ja aus den Akzeptanzkriterien für die aktuelle User Story abgeleitet wurden, bedeutet das »Grün-Werden« aller Akzeptanztests, dass die User Story erfolgreich fertiggestellt wurde. Wenn Sie also in Zukunft denken, dass Sie fertig sind, sollten Sie einfach alle Akzeptanztests für die aktuelle User Story ausführen, um Gewissheit zu erhalten.

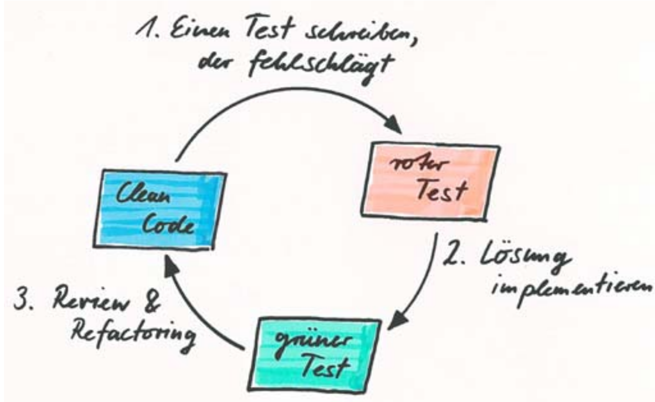
Für weiterführende Informationen zur testgetriebenen Entwicklung und ATDD möchten ich Ihnen an dieser Stelle noch das Buch »Growing Object-Oriented Software, Guided by Tests« von Steve Freeman und Nat Pryce [Freeman und Pryce 2009] empfehlen, in dem der (A)TDD-Zyklus sehr anschaulich anhand der Entwicklung einer größeren Beispielapplikation erklärt wird.

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4

4.6 Zusammenfassung

Hier noch einmal die wichtigsten Regeln für die testgetriebene Entwicklung:

- Der TDD-Zyklus sieht wie folgt aus:



- Die wichtigste Regel beim testgetriebenen Entwickeln besagt: Sie dürfen keinen Produktionscode schreiben oder ändern, ohne einen fehlschlagenden Test zu haben (egal ob Sie ein neues Feature entwickeln oder einen Bug fixen).
- Zuerst erstellen Sie immer einen roten Test. Dabei zählt auch ein Test, der nicht kompiliert, als roter Test. Das initiale Schreiben des Tests ist gleichzeitig die Designphase für die API Ihres Codes.
- Schreiben Sie eine Testmethode immer gleich vom Anfang bis zum Ende fertig, und zwar genau so, wie Sie sie gerne lesen würden, ohne fehlende Klassen oder Methoden anzulegen oder gar schon zu implementieren.
- Wenn etwas nicht wichtig für einen Test ist, ist es wichtig, dass es auch nicht im Test steht. Testmethoden sollten in erster Linie immer kurz und gut verständlich sein. Lagern Sie nebensächlichen Code in Hilfsmethoden aus.
- Wenn Ihr roter Test nicht kompiliert, konzentrieren Sie sich zunächst darauf, Ihr Projekt wieder kompilieren zu können. Legen Sie lediglich alle nötigen neuen Klassen und Methoden an. Hierbei ist es hilfreich, wenn Sie Ihre IDE so konfigurieren, dass neu angelegte Methoden automatisch immer folgende Implementierung bekommen:

```
throw new UnsupportedOperationException("Not implemented yet.");
```

- Erst wenn Ihr Projekt wieder erfolgreich kompiliert, beginnen Sie mit der Implementierung. Beschränken Sie sich dabei aber darauf, immer nur so viel Produktionscode zu schreiben, wie nötig ist, um Ihren roten Test grün zu machen. Betrügen Sie sich dabei aber nicht selbst: Die implementierte Lösung muss möglichst allgemeingültig sein.

- Wenn Sie längere Zeit programmieren, ohne den aktuellen roten Test auszuführen, sollten Sie den aktuellen TDD-Zyklus unterbrechen und einen neuen (kleineren) für das Problem beginnen, an dem Sie gerade sitzen. Typischerweise führen Sie den aktuellen roten Test beim testgetriebenen Entwickeln innerhalb weniger Minuten immer wieder aus, um zu sehen, was als Nächstes zu tun ist.
- Wenn Sie beim Schreiben von Produktionscode Randfälle (*Edge Cases*) entdecken, sollten Sie auch für diese Tests schreiben.
- Vergessen Sie nicht, dass das »Grün-Werden« eines (oder mehrerer) Tests nur ein Etappensieg ist – Sie wissen jetzt, dass Ihr Code funktioniert. Aber als Nächstes müssen Sie Ihren Code noch einmal einem Review unterziehen und überprüfen, ob der Code gut verständlich ist, sich an Ihre Coding-Konventionen und Architekturvorgaben sowie an die allgemeinen Prinzipien objektorientierter Programmierung hält. Falls nötig, führen Sie ein Refactoring durch.
- ATDD stellt einen Rahmen für TDD dar. Wenn Sie zunächst Akzeptanztests für eine neue User Story schreiben, wissen Sie, wann Sie mit der User Story fertig sind – nämlich dann, wenn alle Akzeptanztests grün sind. Hierbei sollten die Akzeptanztests keine Unit-Tests oder Integrationstests, sondern Systemtests sein, also die Applikation nur über die Schnittstellen ansprechen, die später auch den Benutzern zur Verfügung stehen.

Michael Tamm, JUnit-Profiwissen, dpunkt.verlag, ISBN 978-3-86490-020-4