Kapitel 4

Was testen: Der richtige BICEP

Es kann ganz schön schwer sein, sich eine Methode oder eine Klasse anzusehen und sich alle Situationen vorzustellen, in denen sie fehlerhaft reagiert. Kurz gesagt: alle Fehler im Voraus zu ahnen. Mit ausreichend Erfahrung bekommen Sie ein Gefühl für die Dinge, die wahrscheinlich schief gehen, und können sich dann zuerst darauf konzentrieren. Aber ohne ausreichende Erfahrung ist es schwer und frustrierend, nach allen möglichen Fehlersituationen zu suchen. Anwender sind ziemlich geschickt darin, unsere Fehler zu finden. Das ist sowohl peinlich als auch schlecht für unsere Karriere. Was wir brauchen, ist eine gute Gedankenstütze für die Dinge, die beim Testen wichtig sind.

Lassen Sie uns sechs besondere Bereiche betrachten, die unsere Testfähigkeiten mit dem „Richtigen BICEP“ trainieren werden.

- **Right** – Sind die Ergebnisse *richtig*?
- **B**oundary condition – Sind die *Grenzfälle korrekt*?
- **I**nverse operation – Können Sie die *umgekehrte Operation* prüfen?
- **C**ross-check – Können Sie eine *Gegenprobe* mit den Ergebnissen machen?
- **E**rror condition – Können Sie *Fehlersituationen* provozieren?
- **P**erformance – Liegt die *Performance* im Bereich des Erwarteten?

4.1 Sind die Ergebnisse richtig?

Right *BICEP*

Das Erste und Offensichtlichste, was es zu testen gibt, ist, ob die erwarteten Ergebnisse richtig sind – die Ergebnisse überprüfen.

Ein einfache Überprüfung von Daten haben wir schon gesehen: Die Tests in Kapitel 2 überprüfen, ob eine Methode die größte Zahl in einem Array findet.

Das sind normalerweise die einfachen Tests. Viele davon sind schon in den Anforderungen enthalten. Wenn nicht, müssen Sie jemanden danach fragen. Folgende Schlüsselfrage müssen Sie beantworten können:

Woher weiß ich, ob sich der Quelltext korrekt verhalten hat?

Wenn Sie diese Frage nicht zufriedenstellend beantworten können, dann wäre es Zeitverschwendung, den Test oder den eigentlichen Quelltext dazu zu programmieren. Sie werden sagen: „Halt, das klingt nicht sehr agil! Was ist, wenn die Anforderungen vage oder unvollständig sind? Heißt das, dass man erst Quelltext schreiben darf, wenn die Anforderungen fest stehen?“

Überhaupt nicht. Wenn die Anforderungen wirklich noch unbekannt oder unvollständig sind, können Sie sich fürs Erste welche ausdenken. Es kann sein, dass sie aus Anwendersicht nicht korrekt sind, aber *Sie* wissen nun, was die Software Ihrer Meinung nach tun sollte.

Natürlich müssen Sie sich dann um Rückmeldung von Ihren Anwendern kümmern, um Ihre Annahmen zu korrigieren. Die Definition von „korrekt“ kann sich während der Lebenszeit der fraglichen Software ändern, aber Sie sollten zu jedem Zeitpunkt nachweisen können, dass die Software genau das tut, was Sie erwarten.

```
import junit.framework.*;
import java.io.*;
import java.util.ArrayList;
import java.util.StringTokenizer;
```

```
public class TestLargestDataFile extends TestCase {

  public TestLargestDataFile(String name) {
    super(name);
  }

  /* Führe alle Tests in testdata.txt aus (beinhaltet
   * nicht den Ausnahmefall). Wir bekommen einen Fehler,
   * wenn eine Dateioperation fehlschlägt.
   */
  public void testFromFile() throws Exception {
    String line;
    BufferedReader rdr = new BufferedReader(
                           new FileReader(
                             "testdata.txt"));

    while ((line = rdr.readLine()) != null) {
      if (line.startsWith( "#")) { // Kommentare ignorieren
        continue;
      }
      StringTokenizer st = new StringTokenizer(line);
      if (!st.hasMoreTokens()) {
        continue; // Leerzeile
      }

      // Lies den Erwartungswert
      String val = st.nextToken();
      int expected = Integer.valueOf(val).intValue();
      // Parameter hinzufügen
      ArrayList argument_list = new ArrayList();
      while (st.hasMoreTokens()) {
        argument_list.add(Integer.valueOf(st.nextToken()));
      }
      // Objekt-Liste in ein Array übertragen
      int[] arguments = new int[argument_list.size()];
      for (int i=0; i < argument_list.size(); i++) {
        arguments[i] =
          ((Integer)argument_list.get(i)).intValue();
      }
      // Und assert-Anweisung ausführen
      assertEquals(expected,
      Largest.largest(arguments));
    }
  }
}
```

Abbildung 4.1: TestLargestDataFile: Die Test-Spezifikation aus einer Datei lesen

Testdaten in Dateien

Für Tests mit großen Datenmengen sollten Sie darüber nachdenken, Eingabewerte und/oder Ergebnisse in separaten Dateien zu speichern, die dann vom Unit-Test eingelesen werden. Das muss nicht kompliziert sein – Sie brauchen nicht mal XML.[1] In Abbildung 4.1 auf Seite 45 sehen Sie eine Version von `TestLargest`, die alle Daten aus einer Datei liest.

Die Testdatei hat ein sehr einfaches Format. Jede Zeile enthält eine Menge von Zahlen. Die erste ist das erwartete Ergebnis, die restlichen sind die Werte für das zu übergebende Array. Ein „#"-Zeichen markiert Kommentare, so dass man sinnvolle Beschreibungen ergänzen kann.

Die Testdatei kann dann so aussehen:

testdata.txt

```
#
# Einfache Tests:
#
9 7 8 9
9 9 8 7
9 9 8 9
#
# Tests mit negativen Zahlen:
#
-7 -7 -8 -9
-7 -8 -7 -8
-7 -9 -7 -8
#
# Gemischt:
#
7 -9 -7 -8 7 6 4
9 -1 0 9 -7 4
#
# Grenzfälle:
#
1 1
0 0
2147483647 2147483647
-2147483648 -2147483648
```

Für die Hand voll Tests in diesem Beispiel lohnt sich der Aufwand sicher noch nicht. Aber nehmen wir an, es wäre eine komplexere Anwendung mit einigen Dutzend oder gar Hunderten von Testfällen dieser Art. Dann ist der Ansatz mit Testdateien ziemlich überzeugend.

---

[1] Das ist natürlich ein Scherz. XML ist heutzutage bei allen Projekten obligatorisch, nicht wahr?

Denken Sie dran, dass auch die Testdaten selbst – egal ob in einer Datei oder im Quelltext der Tests – fehlerhaft sein können. In der Tat zeigt die Erfahrung sogar, dass die Testdaten *mit höherer Wahrscheinlichkeit* fehlerhaft sind als der getestete Quelltext. Das gilt insbesondere, wenn die Werte von Hand berechnet sind oder von einer Software stammen, die wir gerade ersetzen wollen (weil durch neue Funktionalitäten natürlich auch gewollt andere Ergebnisse entstehen können). Wenn die Testdaten sagen, dass der Quelltext falsch ist, sollten Sie doppelt und dreifach prüfen, ob die Testdaten korrekt sind, bevor Sie den Quelltext verändern.

Es gibt noch etwas anderes, über das Sie nachdenken sollten. Der gezeigte Quelltext prüft keine Ausnahmen. Wie könnte man das implementieren?

Wählen Sie die einfachste Möglichkeit, um zu beweisen, dass die Methode korrekt funktioniert.

4.2 Grenzfälle

Right ***B*** *ICEP*

In dem vorangegangenen Beispiel zur „größten Zahl“ haben wir verschiedene Grenzfälle entdeckt: Die größte Zahl liegt am Ende des Arrays, das Array enthält negative Zahlen, das Array ist leer und so weiter.

Grenzfälle zu identifizieren, gehört zu den wertvollsten Aspekten bei der Arbeit mit Unit-Tests, denn Fehler sind im Allgemeinen genau dort zu finden – an den Rändern. An die folgenden Grenzfälle sollten Sie denken:

- Völlig falsche oder unerlaubte Werte, wie zum Beispiel „`!*W:X\&Gi/w~>g/h#WQ@`“ als Dateinamen;
- ungültig formatierte Daten, wie eine E-Mail-Adresse ohne TopLevel Domäne (`fred@foobar.`);
- leere oder fehlende Werte (wie 0, 0.0, `""` oder `null`);
- Werte, die weit außerhalb des Erwartungsbereiches liegen, wie beispielsweise ein Lebensalter von 10.000 Jahren;
- Duplikate in Listen, in denen keine Duplikate vorkommen sollten;

- sortierte Listen, die unsortiert sind, und umgekehrt. (Übergeben Sie mal eine vorsortierte Liste an einen Sortieralgorithmus, oder sogar eine umgekehrt sortierte Liste.)
- Ereignisse, die nicht in der erwarteten Reihefolge auftreten, wenn beispielsweise ein Dokument gedruckt werden soll, bevor man sich am System angemeldet hat.

Eine einfache Gedankenstütze für diese möglichen Grenzfälle ist das Akronym CORRECT. Überlegen Sie für jede der folgenden Bedingungen: Ist sie bei der zu testenden Methode anwendbar, und was würde passieren, wenn die Bedingung nicht eingehalten wird.

- **C**onformance / Konformität – Entspricht der Wert dem erwarteten Format?
- **O**rdering / Reihenfolge – Sind die Werte geordnet oder ungeordnet (je nachdem, was benötigt wird)?
- **R**ange / Wertebereich – Ist der Wert innerhalb vernünftiger Minimal- und Maximalwerte?
- **R**eference / Beziehungen – Bezieht sich der Quelltext auf irgendetwas Externes, worüber er keine unmittelbare Kontrolle hat?
- **E**xistence / Existenz – Existiert der Wert (nicht `null`, ungleich 0, in einer Liste vorhanden etc.)?
- **C**ardinality / Kardinalität – Sind genug Werte vorhanden, weder zu viel noch zu wenig?
- **T**ime / Zeit, absolut und relativ – Passiert alles in der erwarteten Reihenfolge? Zur richtigen Zeit? Und pünktlich?

Im folgenden Kapitel werden wir alle diese Arten von Grenzfällen ausführlicher betrachten.

4.3 Die umgekehrte Operation prüfen

*Right B**I**CEP*

Manche Methoden lassen sich testen, indem man die logisch umgekehrte Operation anwendet. Beispielsweise kann man eine Methode zur Berechnung der Quadratwurzel prüfen, indem man das Ergebnis quadriert und mit der Eingabe vergleicht (die Differenz sollte hinreichend klein sein):

```
public void testSquareRootUsingInverse() {
  double x = mySquareRoot(4.0);
  assertEquals(4.0, x * x, 0.0001);
}
```

Das Einfügen in eine Datenbank zum Beispiel können Sie testen, indem Sie anschließend nach den Werten suchen, und so weiter.

Seien Sie vorsichtig, wenn Sie sowohl die Methode als auch die umgekehrte Operation selbst programmiert haben, weil sich Implementierungsfehler an beiden Stellen eventuell gegenseitig aufheben und so nicht erkennbar sind. Nutzen Sie, wo immer es möglich ist, eine fremde Implementierung für den Test. In dem Beispiel mit der Quadratwurzel haben wir den eingebauten Multiplikationsoperator verwendet. Für die Datenbanksuche nehmen wir die Suchfunktion der Herstellers.

4.4 Die Gegenprobe machen

Right BI C EP

Vielleicht können Sie die Gegenprobe auf die Ergebnisse mit anderen Mitteln machen. Im Allgemeinen gibt es mehrere Wege, etwas zu berechnen. Sie wählen einen bestimmten Algorithmus für Ihre Anwendung aus, weil er besonders schnell ist oder andere wünschenswerte Eigenschaften hat. Für die Gegenprobe können Sie einen alternativen Algorithmus implementieren. Diese Strategie ist insbesondere dann sinnvoll, wenn es einen bekannten und bewährten Weg gibt, das Problem zu lösen, der aber aus irgend einem Grund für Ihre Anwendung ungeeignet ist.

Wir können diese in unseren Augen irgendwie schlechtere Variante nutzen, um zu prüfen, ob unsere super-tolle Lösung die gleichen Ergebnisse liefert:[2]

```
public void testSquareRootUsingStd() {
  double number = 3880900.0;
  double root1 = mySquareRoot(number);
  double root2 = Math.sqrt(number);
  assertEquals(root2, root1, 0.0001);
}
```

[2] Manche Tabellenkalkulationen (wie zum Beispiel Microsoft Excel® und andere) benutzen ähnliche Strategien, um zu überprüfen, ob die gewählten Verfahren für die Lösung eines Problem geeignet sind und ob verschiedene, mögliche Verfahren zu den gleichen Ergebnissen kommen.

Eine andere Herangehensweise für Gegenproben ist, sich verschiedene Teile der Daten anzusehen und zu prüfen, ob sie untereinander konsistent sind. Angenommen, wir haben eine Bibliotheksdatenbank. In diesem System sollte die Anzahl von Exemplaren eines Buches immer konsistent sein. Das heißt, die Zahl der ausgeliehenen Exemplare und die Zahl der Exemplare in den Regalen ergibt immer die Gesamtzahl der Exemplare eines Buches. Das sind verschiedene Teile der Daten, die eventuell sogar von verschiedenen Klassen verwaltet werden, aber sie müssen immer konsistent sein und können daher als Gegenprobe verwendet werden.

4.5 Fehlersituationen provozieren

*Right BIC **E** P*

In der realen Welt passieren Fehler. Festplatten laufen voll, Netzwerkverbindungen brechen ab, E-Mails verschwinden und Programme stürzen ab. Um zu testen, ob Ihr Quelltext in der Lage ist, mit diesen realen Problemen umzugehen, müssen Sie Fehlersituationen provozieren.

Testen mit ungültigen Parametern und Ähnlichem ist einfach, aber um besondere Netzwerkfehler zu simulieren – ohne an Kabeln zu ziehen –, braucht man schon besondere Techniken. In Kapitel 6 auf Seite 73 werden wir eine Möglichkeit vorstellen, dies mit Hilfe so genannter Mock-Objekte zu tun.

Aber bevor wir dorthin kommen, sollten Sie überlegen, welche anderen Fehlerarten oder Umgebungsbedingungen Sie für Ihre Methode noch abtesten sollten. Stellen Sie ein kurze Liste auf, bevor Sie weiterlesen.

***STOP:* Denken Sie einen Moment nach, bevor Sie weiterlesen.**

Hier sind die Dinge, die uns in den Sinn kamen:

- Kein Hauptspeicher mehr verfügbar,
- Kein Festplattenplatz mehr verfügbar,
- Uhrzeitdifferenzen und -umstellungen,
- Netzwerkverfügbarkeit und Netzwerkfehler,

- Hohe Systemauslastung,
- Eingeschränkte Farbpalette,
- Sehr hohe oder sehr niedrige Grafikauflösungen.

4.6 Performance-Eigenschaften

Right BICE ***P***

Performance-Tests können sich als besonders nützlich erweisen. Es geht nicht um die absolute Geschwindigkeit, sondern vielmehr um Trends bei wachsenden Eingabemengen, komplexeren Problemen und so weiter.

Unser Ziel sind Regressions-Tests, mit denen wir die Performance-Eigenschaften messen können. Nur zu häufig veröffentlichen wir eine Version der Software, die zufrieden stellend arbeitet, aber in der nächsten Version läuft alles auf einmal furchtbar langsam. Wir wissen weder, wieso das so ist, noch was geändert wurde, auch nicht, wer es wann getan hat. Aber die Anwender schreien Zeter und Mordio.

Um diese unangenehme Situation zu vermeiden, sollten Sie sich einige grobe Tests ausdenken, die sicherstellen, dass die Performance-Kurve nicht unerwartet ausschlägt. Nehmen wir beispielsweise an, wir hätten einen Filter geschrieben, um unerwünschte Webseiten zu blockieren (weil wir jede Menge Schwierigkeiten bekommen würden, wenn jemand mit unserem neuen Produkt schmutzige Bildchen ansehen könnte).

Der Quelltext funktioniert mit einigen Dutzend Webseiten tadellos, aber wird er das auch noch mit 10.000? Oder 100.000? Lassen Sie uns einen Test schreiben, um das herauszufinden.

```
public void testURLFilter() {
  Timer timer = new Timer();
  String naughty_url = "http://www.xxxxxxxxxx.com";

  // Zuerst wird gegen eine kleine Liste geprüft
  URLFilter filter = new URLFilter(small_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 1.0);
```

```
  // Anschliessend gegen eine grosse Liste prüfen
  URLFilter filter = new URLFilter(big_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 2.0);

  // Letztlich gegen eine riesige Liste prüfen
  URLFilter filter = new URLFilter(huge_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 3.0);
}
```

Das gibt uns einige Sicherheit, dass wir die Performance-Kriterien einhalten können. Dieser Test dauert 6–7 Sekunden, daher wollen wir ihn nicht jedes Mal mit ausführen. Solange wir ihn jede Nacht oder zumindest alle paar Tage ausführen, wird er uns bei problematischen Änderungen immer noch rechtzeitig alarmieren, bevor keine Zeit mehr bleibt, die Fehler zu beheben.

Sie sollten sich eventuell nach *TestDecorators* umsehen, die beim Timing von Tests, Erzeugen von Systemlast und Ähnlichem helfen, zum Beispiel das frei verfügbare JunitPerf.[3]

[3] http://www.clarkware.com/