Property Based Testing – eine Einführung

Nicolai Mainiero, sidion

Testen ist schwer. Sämtliche Pfade durch einen Code-Abschnitt aufzuzählen, ist aufwändig und die Anzahl der Parameter lässt die möglichen Kombinationen für Eingaben in die Höhe schnellen. Dieser Artikel stellt ein sehr hilfreiches Werkzeug vor, das bestehende Testmethoden ergänzt und den erwähnten Schwierigkeiten begegnet. Property Based Testing ist eine Methode, um zu prüfen, ob eine Funktion für alle Eingaben korrekt ist. Dies wird erreicht, indem sie mit zufällig generierten Daten ausgeführt wird.

Testen ist nach wie vor die am weitesten verbreitete Methode, um Softwarequalität sicherzustellen, und mit dreißig bis vierzig Prozent der Gesamtkosten *[1]* ein nicht zu vernachlässigender Kostenfaktor in der Entwicklung. Man kann die Kosten reduzieren, indem man die Tests zu einem Großteil automatisiert. Die Testpyramide von Mike Cohn *[2]* liefert ein Modell dafür, wie die Tests über die verschiedenen Schichten einer Anwendung verteilt sein sollten. Der größte Anteil entfällt hierbei auf die sogenannten „Unit-Tests", die sich jeweils um einen sehr kleinen und lokalen Bereich kümmern und nur diesen testen.

Als verbreiteter Prozess für das Entwickeln von Unit-Tests hat sich das Test Driven Development und später darauf aufbauend das Behavior Driven Development entwickelt. Testgetriebene Entwicklung nach Kent Beck *[3]* gliedert sich in folgende drei Phasen:

1. Red: Schreibe einen Test, der ein neues zu programmierendes Verhalten (Funktionalität) prüfen soll. Fange mit dem einfachsten Beispiel an. Existiert die Funktion bereits, kann dies auch ein gemeldeter Fehler oder eine neu zu implementierende Funktionalität sein. Dieser Test wird vom vorhandenen Programmcode erst einmal nicht erfüllt und muss dementsprechend fehlschlagen.
2. Green: Ändere den Programmcode mit möglichst wenig Aufwand und ergänze ihn, bis er nach dem anschließend angestoßenen Testdurchlauf alle Tests besteht.
3. Clean up (Refactoring): Entferne Wiederholungen (Code-Duplizierung), abstrahiere, wo nötig, und richte ihn nach den verbindlichen Code-Konventionen aus. In dieser Phase darf kein neues Verhalten eingeführt werden, das nicht durch Tests schon abgedeckt ist. Nach jeder Änderung werden die Tests ausgeführt. Schlagen sie fehl, darf die offenbar fehlerhafte Änderung nicht übernommen werden. Ziel des Aufräumens ist es, den Code verständlich und leicht nachvollziehbar zu machen.

Diese drei Schritte werden so lange wiederholt, bis der Code die gewünschte Funktionalität liefert, es keine bekannten Fehler mehr gibt und dem Entwickler keine weiteren sinnvollen Tests mehr einfallen. Dieses eigentlich sehr gut strukturierte Vorgehen birgt jedoch Risiken. Wenn die Funktionalität nicht ausreichend beschrieben ist oder wenn in dem Refactoring-Schritt unsauber gearbeitet wird, kann es passieren, dass die Codequalität trotz Tests sinkt.

Folgendes einfaches Beispiel soll diese Problematik demonstrieren.

In den *Listings 1* bis *4* sind Tests und die dazu passenden Implementierungen für eine Addierfunktion aufgelistet. *Listing 2* zeigt die einfachste mögliche Implementierung, um den Test aus *Listing 1* zu bestehen. In *Listing 3* wird dann ein weiterer Testfall hinzugefügt. *Listing 4* zeigt wiederum den Code, der nötig ist, um auch diesen Testfall zu bestehen. In diesem Beispiel wurde auf das Refactoring verzichtet, um aufzuzeigen, dass es entscheidend ist, nicht nur die Anforderungen im Test und in der Implementierung stur zu erfüllen, sondern die Anforderungen auch zu verstehen, selbst wenn sie nur exemplarisch formuliert worden sind. Property Based Testing zwingt einen dazu, sich mit den Anforderungen an eine Funktion genauer auseinanderzusetzen.

QuickCheck

Die Idee von Property Based Testing geht auf die Veröffentlichung einer Bibliothek für Haskell zurück. Koen Claessen und John Hughes *[4]* stellten die folgenden Anforderungen an ein Werkzeug zum randomisierten Testen von Haskell-Programmen: Durch eine formale Spezifikation soll es automatisch feststellen können, ob ein Test erfolgreich ist oder nicht, und selbstständig Testdaten erzeugen können. Zu deren Erstellen soll es dem Tester eine Möglichkeit bieten zu bestimmen, wie Testdaten generiert werden. Außerdem soll es leichtgewichtig sein. Das Ergebnis war QuickCheck, das all diese Anforderungen erfüllen konnte. Dieses Werkzeug hat dann den Begriff „Property Based Testing" geprägt.

Property Based Testing

Im Gegensatz zum klassischen Unit-Testing werden hier keine konkreten Zusicherungen wie zum Beispiel in *Listing 1* gemacht, sondern es werden Eigenschaften der Methode gesucht, die für alle Eingabe-Parameter gelten. Diese Eigenschaften werden dann mit vielen automatisch generierten, randomisierten Daten überprüft, um die Implementierung zu testen.

Für die Erzeugung dieser Daten gibt es sogenannte „Generatoren". Sie kennen die Domäne und wissen, wie Eingabe-Parameter zu erzeugen sind. Damit das Fehlschlagen eines Tests nicht mit komplexen Eingaben nachvollzogen werden muss, existiert das sogenannte „Shrinking". Es ist ein wesentliches und zugleich das spannendste Feature von Property Based Testing. Damit wird eine Eingabe so lange vereinfacht, bis eine kleinstmögliche Eingabe gefunden ist, die den Test fehlschlagen lässt. Dadurch ist es meist sehr einfach, den Fehler in der Implementierung nachzuvollziehen und zu korrigieren.

JUnit-Quickcheck

Es gibt von QuickCheck mehr als fünfzig Implementierungen in mehr als vierzig Sprachen, unter anderem auch Clojure, Scala oder Java. Allein für Java existieren mindestens fünf Implementierungen, um Property Based Testing einzusetzen. Die nachfolgenden Beispiele sind alle mit JUnit-Quickcheck *[5]* realisiert. Sie lassen sich aber einfach auf andere Implementierungen übertragen.

Listing 5 zeigt, wie man mit der Bibliothek einen Property Based Test schreibt. Wichtig sind die Annotationen „@RunWith(JUnitQuickcheck.class)" und „@Property", mit denen JUnit angewiesen wird, einen speziellen Test-Runner zu verwenden. Dieser Test-Runner versteht neben den üblichen JUnit-Annotationen auch „@Property". Eine damit annotierte Methode wird mit Eingaben aufgerufen, die durch einen passenden Generator automatisch entstanden sind. In diesem Fall werden also zufällige Strings erzeugt und überprüft, ob die „length()"-Methode erwartungsgemäß funktioniert, also die Summe der Längen zweier Strings gleich der Länge des verketteten Strings ist. Standardmäßig wird die Testmethode hundertmal aufgerufen und die Zusicherung geprüft. Dies erfolgt wie in klassischen JUnit-Tests auch mit „assertEquals()".

In diesem Beispiel werden Basis-Typen als Parameter benutzt.

```
public class AdderTest {
  @Test
  public void testApp() {
     assertEquals(4, Adder.sum(1, 3));
  }
}
```

Listing 1

```
public class Adder {
  public static int sum(int a, int b) {
     return 4;
  }
}
```

Listing 2

```
public class AdderTest {
  @Test
  public void testApp() {
     assertEquals(4, Adder.sum(1, 3));
     assertEquals(7, Adder.sum(4, 3));
  }
}
```

Listing 3

```
public class Adder {
public static int sum(int a, int b) {
  switch (a){
     case 4:
        return 7;
     }
     return 4;
  }
}
```

Listing 4

```
import com.pholser.junit.quickcheck.Property;
import com.pholser.junit.quickcheck.runner.JUnitQuick-
check;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

@RunWith(JUnitQuickcheck.class)
public class StringProperties {
  @Property
  public void concatenationLength(String s1, String
s2) {
  assertEquals(s1.length() + s2.length(), (s1 + s2).
length());
     }
}
```

Listing 5

JUnit-Quickcheck kann damit von Haus aus umgehen und weiß, wie diese erzeugt werden müssen. *Listing 6* zeigt, wie man für andere Typen einen Generator implementiert, der auch Shrinking unterstützt. Dies ist nötig, um für den zu testenden Code geeignete Domänen-Objekte erzeugen zu können. Es müssen lediglich zwei Methoden überschrieben werden, wie im Beispiel zu sehen.

Um neue Werte zu erzeugen, wird zunächst „generate()" überschrieben und ein neuer Punkt mit einer zufälligen X-Y-Koordinate erzeugt. Das API stellt dafür einen Zufallsgenerator als Parameter zur Verfügung, der es einem ermöglicht, den Test bei Bedarf immer wieder mit denselben zufällig erzeugten Objekten aufzurufen. Dies kann beim Debuggen von Tests hilfreich sein.

Als Zweites sollte die Methode „doShrink()" implementiert sein, damit klar ist, wie vereinfachte Objekte aus einem bestehenden erzeugt werden können. Je nach Domäne bedeutet dieses Vereinfachen etwas anderes. Im Beispiel sind vereinfachte Punkte nur näher am Ursprung des Koordinatensystems. Man sieht auch, dass nicht nur ein kleinerer Punkt erzeugt wird, sondern vier neue Punkte. Je nach Domänen-Objekt können beliebig viele vereinfachte Objekte zurückgegeben werden. Jetzt lassen sich die Tests einfach mit JUnit-Quickcheck erstellen.

Auf den ersten Blick erscheint es schwierig, geeignete Eigenschaften für eine Methode zu finden, um Property Based Testing anwenden zu können. Aber es gibt eine ganze Reihe von Mustern, die einem helfen, solche Eigenschaften zum Testen zu formulieren.

Unterschiedliche Wege, dasselbe Ziel

Dem Muster „Unterschiedliche Wege, dasselbe Ziel" sind Eigenschaften zuzuordnen, deren Reihenfolge bei der Ausführung keine Rolle spielt. Man kennt dieses Muster auch unter dem Namen „Kommutativität", zum Beispiel bei der Addition. Hier spielt es keine Rolle, ob zuerst „+ 1" und danach „+ 2" gerechnet wird oder zuerst „+ 2" und danach erst „+ 1". Das Ergebnis ist in beiden Fällen dasselbe.

Abbildung 1 zeigt ein konkretes Beispiel für dieses Muster. In diesem Fall wird geprüft, ob die Liste korrekt sortiert wurde, indem wir die beiden Wege zum Ergebnis vergleichen. Das vorherige Anhängen von „Integer.MIN_VALUE" an die Liste und das darauffolgende Sortieren muss dasselbe Resultat zurückliefern, als wenn die Liste als Erstes sortiert und danach erst „Integer.MIN_VALUE" vorne an die Liste gesetzt wird.

Hin und zurück

Dieses Muster eignet sich für alle Operationen, für die eine Umkehrung existiert. Der bekannteste Vertreter dieser Art von Operationen ist die Codierung/Decodierung von Daten. Das ist jedoch nicht das einzige Paar, das geeignet ist. Zum Beispiel können auch folgende Paare überprüft werden: Addition/Subtraktion, Schreiben/Lesen oder auch Getter/Setter. Außerdem gibt es noch Operations-Paare, bei denen es sich zwar nicht um die Umkehrung der Operation handelt, die aber dennoch die Anforderungen dieses Musters erfüllen. Beispiele sind: Einfügen/Enthält oder Erzeugen/Existiert.

In *Abbildung 2* ist noch ein Sonderfall für dieses Muster dargestellt: Manche Operationen sind ihre eigene Umkehrfunktion – in diesem Fall die Methode zum Umkehren einer Liste. Eine wiederholte Anwendung der Methode führt wieder zur ursprünglichen Eingabe.

```
import java.awt.Point;

public class Points extends Generator<Point> {
  private static final Point ORIGIN = new Point();

  public Points() {
     super(Point.class);
  }

  @Override
  public Point generate(SourceOfRandomness random,
GenerationStatus status) {
     return new Point(random.nextInt(), random.nex-
tInt());
  }

  @Override
  public List<Point> doShrink(SourceOfRandomness ran-
dom, Point larger) {
     if (ORIGIN.equals(larger)) {
        return Collections.emptyList();
     }

     List<Point> shrinks = new ArrayList<>();
     shrinks.add(new Point(0, larger.y));
     shrinks.add(new Point(larger.x, 0));
     shrinks.add(new Point(larger.x / 2, larger.y));
     shrinks.add(new Point(larger.x, larger.y / 2));
     return shrinks;
  }
}
```

Listing 6

Manche Dinge ändern sich nie

Im nächsten Muster geht es um invariante Eigenschaften von Transformationen. Typische Vertreter sind beispielsweise die Größe einer Liste, die sich nicht ändert, wenn man jedes einzelne Element transformiert, oder der Inhalt einer Liste nach dem Sortieren. *Abbildung 3* zeigt dieses Muster anhand einer Liste, in der jedes einzelne Element transformiert wird. Es ist erkennbar, dass sich die Anzahl der Elemente nicht verändert hat.

Je öfter man sie anwendet, desto weniger machen sie aus

Diese Klasse von Eigenschaften zeichnen sich durch Idempotenz aus, eine mehrfache Ausführung der Operation liefert also das gleiche Ergebnis zurück wie eine einzige. *Abbildung 4* zeigt die „distinct"-Operation auf einer Liste. Es ist zu sehen, dass sich nach der ersten Anwendung das Ergebnis auch bei Wiederholung nicht mehr ändert. Dieses Prinzip lässt sich auf weitere Operationen wie Datenbank-Updates oder Nachrichtenverarbeitung ausweiten.

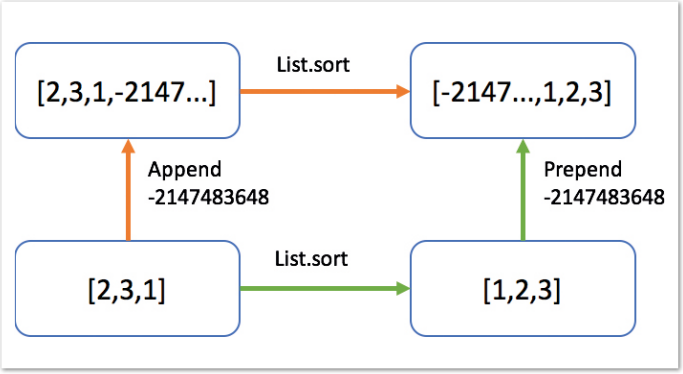

Abbildung 1: Ein Beispiel für „Unterschiedliche Wege, dasselbe Ziel"

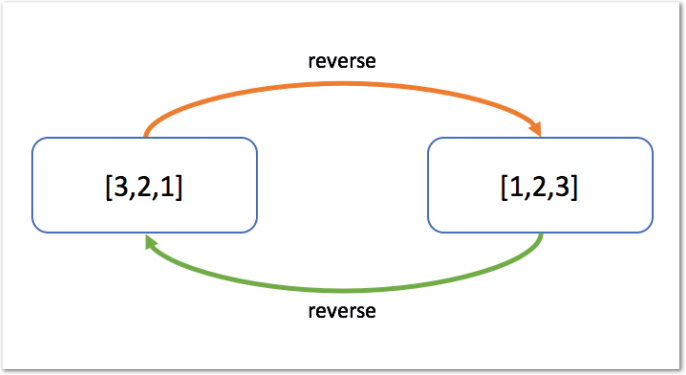

Abbildung 2: Ein Beispiel für „Hin und zurück"

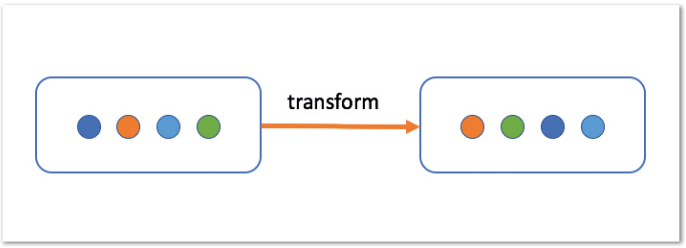

Abbildung 3: Ein Beispiel für „Manche Dinge ändern sich nie"

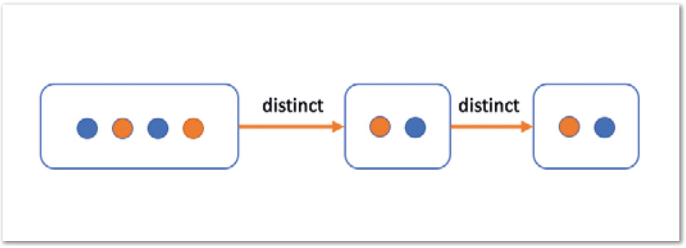

Abbildung 4: Ein Beispiel für „Je öfter man sie anwendet, desto weniger machen sie aus"

Schwer zu beweisen, einfach zu überprüfen

Dies ist eine Variante des Musters „Hin und zurück". Es kommt häufig vor, dass es viel schwieriger ist, die Lösung für ein Problem zu finden, als die Lösung zu prüfen. Das kennt jeder, der schon mal ein Sudoku gemacht hat. Zu prüfen, ob ein Sudoku korrekt ausgefüllt ist, geht sehr einfach, indem man die Zeilen, Spalten und 3x3-Blöcke nacheinander überprüft. Ein Sudoku korrekt auszufüllen, ist jedoch ungleich schwieriger. Weitere Probleme, die zu diesem Muster passen, sind zum Beispiel die Primfaktor-Zerlegung oder das Zerlegen eines Strings in einzelne Token. *Abbildung 5* zeigt ein weiteres Beispiel: Sortieren. Eine Sortier-Funktion ist komplex zu implementieren – zu prüfen, ob korrekt sortiert wurde, jedoch relativ einfach. Es muss für jedes Paar geprüft werden, ob die Paare korrekt geordnet sind.

Das Test-Orakel

Es bietet sich an, wenn eine alternative Implementierung bereits vorhanden ist und gegen diese validiert werden soll. Das kann zum Beispiel der Fall sein, wenn eine Single-Thread-Lösung durch eine Multi-Threaded-Lösung, die vermutlich schwieriger korrekt zu implementieren ist, ersetzt werden soll. Ebenfalls interessant ist dieses Muster bei umfangreichen Modernisierungen oder Refactorings. Die ursprüngliche Implementierung kann dabei als Orakel für eine Neu-Implementierung dienen und damit alle, auch nicht dokumentierte Eigenschaften der Anforderungen abdecken.

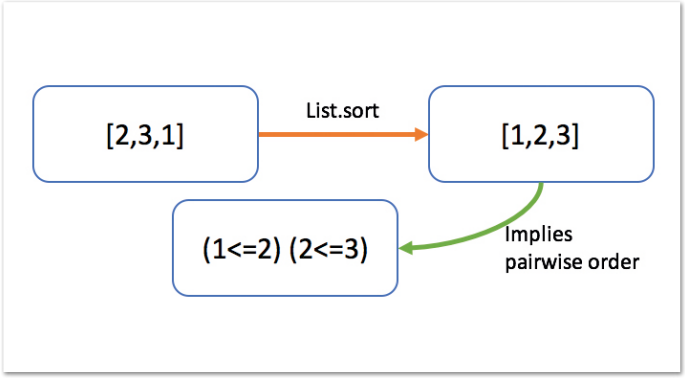

Abbildung 5: Ein Beispiel für „Schwer zu beweisen, einfach zu überprüfen"

Fazit

Property Based Testing ist eine nützliche Erweiterung des Werkzeugkastens, den ein Software-Entwickler zum Testen zur Verfügung hat. Es erfordert, dass die Anforderungen auf der Suche nach möglichen Test-Eigenschaften genauer und gründlicher analysiert und schlussendlich auch besser verstanden werden. Property Based Testing ist nicht dazu gedacht, das klassische Unit-Testing zu ersetzen, sondern es an geeigneter Stelle zu ergänzen. Es spricht nichts dagegen, beide Methoden zu kombinieren, sogar innerhalb eines Tests. Zu dem Trend, mehr und mehr Ansätze aus der funktionalen Programmierung in das Java-Ökosystem zu integrieren, passt Property Based Testing, das seinen Ursprung dort hat, in jedem Fall sehr gut.

Quellen

[1] M. Pol, T. Koomen und A. Spillner, Management und Optimierung des Testprozesses, dpunkt.verlag, 2002
[2] C. Mike, The Forgotten Layer of the Test Automation Pyramid: *https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid*
[3] K. Beck, Test Driven Development by Example, Addison-Wesley Verlag
[4] K. Claessen und J. Hughes, QuickCheck: a lightweight tool for random testing of Haskell programs, ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, 2000
[5] P. Holser, JUnit-quickcheck: *http://pholser.github.io/junit-quickcheck/site/0.7*

Nicolai Mainiero
nicolai.mainiero@sidion.de

Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Expert Software Developer bei der Firma sidion. Er entwickelt seit mehr als neun Jahren Geschäftsanwendungen in Java und PHP für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Aktorensysteme und reaktive Anwendungen.