Projektarbeit-Quick-Check

Mario Occhinegro
HKA University of Applied Sciences

Inhaltsverzeichnis

Zusammenfassung

Sowohl in Haskell als auch in Java ist QuickCheck ein ausgereiftes Werkzeug zur Verbesserung von Code-Qualität. Beide besitzen eine sehr gute Dokumentation und sind syntaktisch leicht verständlich. Haskell schreibt sich jedoch sehr viel leichter von der Hand und ist kompakter.

1 Einleitung

In dieser Projektarbeit werden wir uns intensiv mit dem Testwerkzeug QuickCheck befassen und dessen Verwendung in den Programmiersprachen Haskell und Java untersuchen. QuickCheck ist ein automatisches Testwerkzeug, das dazu verwendet wird, den Code auf Fehler und unerwartete Ausgaben zu überprüfen. QuickCheck generiert zufällige Eingaben für die zu testenden Funktionen und überprüft, ob das Ergebnis den erwarteten Ergebnissen entspricht. Wir werden die Unterschiede zwischen der Implementierung von QuickCheck in Haskell [1] und QuickCheck in Java [2] untersuchen und die Vor- und Nachteile der jeweiligen Implementation diskutieren. Darüber hinaus werden wir einige praktische Beispiele durchgehen, um zu zeigen, wie QuickCheck in der Praxis verwendet werden kann. Eine Github Repository mit dem Source-Code zum ausführen findet sich hier: `https://github.com/yellow-tshirt/projektarbeit-quick-check`

2 Allgemeines zu QuickCheck

QuickCheck ist eine Bibliothek zum Testen von Software, die ursprünglich von Koen Claessen und John Hughes entwickelt wurde. Es ist ein Werkzeug für automatisiertes Testen, das auf dem Konzept der eigenschaftsbasierten Testen basiert. Statt wie beim traditionellen Beispiel-basierten Testen eine feste Anzahl an Testfällen zu schreiben, werden beim eigenschaftsbasierten Testen allgemeine Eigenschaften der zu testenden Funktionen definiert. QuickCheck verwendet dann diese Eigenschaften, um eine große Anzahl zufälliger Testfälle zu generieren und auszuführen.

QuickCheck ist in Haskell, einer funktionalen Programmiersprache, geschrieben. Es gibt jedoch auch Implementierungen von QuickCheck in anderen Programmiersprachen wie Python, Java und C#.

2.1 Generatoren und Properties

2.2 Finden von Bugs

QuickCheck findet Bugs, indem es automatisch eine große Anzahl zufälliger Eingaben generiert und diese an die zu testende Funktion weitergibt. Die generierten Eingaben können dabei unterschiedlich komplex sein und verschiedene Randfälle abdecken. Die zu testende Funktion muss dazu mit QuickCheck kompatibel sein und bestimmte Eigenschaften definieren, die von QuickCheck getestet werden sollen. Diese Eigenschaften werden als Funktionen geschrieben und geben vor, welche Eigenschaften die Funktion erfüllen sollte. Zum Beispiel könnte eine Eigenschaftsfunktion für eine Funktion zur Addition von Zahlen aussagen, dass das Ergebnis immer gleich der Summe der beiden addierten Zahlen sein sollte. QuickCheck nutzt dann die generierten Eingaben und die Eigenschaften, um

die Funktion zu testen. Wenn eine Eingabe gefunden wird, die die Eigenschaften nicht erfüllt, gibt QuickCheck einen Fehler aus, der auf den Bug in der Funktion hinweist. Es ist wichtig zu beachten, dass QuickCheck nicht garantiert, alle Fehler in der Funktion zu finden, sondern eher eine Ergänzung zu anderen Testmethoden darstellt. Es ist jedoch ein sehr effektives Werkzeug, um schnell und automatisch eine große Anzahl von Testfällen zu generieren und auszuführen und somit potenzielle Fehler in der Funktion aufzudecken.

Im weiteren wird für beide Implementationen eine bewusst fehlerhafte Implementation der Funktion **reverseList** mit QuickCheck getestet, um den damit verbundenen Bug zu entdecken.

3 QuickCheck in Haskell

Erläuterung am Beispiel einer selbst implementierten Liste, die Integer speichert mit den Funktionen:

- addToListB: Fügt einen neuen Integer x an den Anfang der Liste.
- addToListE: Fügt einen neuen Integer x an das Ende der Liste.
- reverList: Kehrt die Reihenfolge der Liste um.
- listSize: Berechnet die Länge der Liste.
- popFirst: Entfernt das erste Elemente aus der Liste.
- sumList: Berechnet die Summe aller Zahlen in der Liste.

3.1 Funktionsimplementierung

```
addToListB :: Int -> MyList -> MyList
addToListB x xs = Cons x xs

addToListE :: Int -> MyList -> MyList
addToListE x Empty = Cons x Empty
addToListE x (Cons y ys) = Cons y (addToListE x ys)

reverseList :: MyList -> MyList
reverseList Empty = Empty
reverseList (Cons x xs) = addToListE x (reverseList xs)

listSize :: MyList -> Int
listSize Empty = 0
listSize (Cons x xs) = 1 + listSize xs

popFirst :: MyList -> MyList
popFirst Empty = Empty
popFirst (Cons x xs) = xs

sumMyList :: MyList -> Int
sumMyList Empty = 0
sumMyList (Cons x xs) = x + (sumMyList xs)
```

3.2 Generieren von Testdaten

Das Generieren von Testdaten erfolgt über eine Instanz der Klasse Arbitrary. Arbitrary ist eine Klasse in der QuickCheck-Bibliothek, die es ermöglicht, zufällige Werte für einen gegebenen Typ zu generieren. Hier ein Beispiel für die Klasse MyList:

```
data MyList = Empty | Cons Int MyList deriving (Eq)
instance Show MyList where
  show Empty = ""
  show (Cons x xs) = show x ++ " " ++ show xs

fromList :: [Int] -> MyList
fromList [] = Empty
fromList (x:xs) = Cons x (fromList xs)

instance Arbitrary MyList where
  arbitrary = do
    xs <- listOf arbitrary
    return (fromList xs)
```

Die Funktion fromList dient hier als Hilfsfunktion, Listen von Integern in Listen von MyList umzuwandeln. Eine MyList ist entweder leer oder besteht aus dem ersten Element x mit dem Rest der Liste. Die Show Methode dient lediglich dem Zweck unsere Testdaten genauer ansehen zu können und ggf. zu debuggen.

3.3 Definieren von Properties

In QuickCheck sind Properties Aussagen, die für bestimmte Eigenschaften einer Funktion oder eines Datentyps gelten sollten. Properties werden als Funktionen definiert, die einen oder mehrere Eingabewerte akzeptieren und einen booleschen Wert zurückgeben. Wenn der boolesche Wert True ist, bedeutet dies, dass die Eigenschaft für die gegebene Eingabe erfüllt ist, andernfalls ist sie nicht erfüllt.
Hier ein paar mögliche Property für die oben genannte Funktion reverseList:

```
prop_reverseList :: MyList -> Bool
prop_reverseList xs = reverseList (reverseList xs) == xs
```

Eine Eigenschaft, die für die **reverseList** Funktion gelten muss, ist dass, wenn sie zweimal hintereinander auf eine Liste angewendet wird, diese wieder im Originalzustand sein muss.

3.3.1 Bedingungen auf generierten Testdaten

Für manche Properties möchte man nur eine Teilmenge aller möglichen Testdaten betrachten. Nehmen wir als Beispiel hierfür die Methode **popFirst**. Bei dieser ist es wichtig, dass die Liste nicht leer ist. Dafür gibt es zwei unterschiedliche Ansätze

3.3.2 Ansatz 1 Einschränken der Werte in der Property selber

Zuerst muss gesagt werden, dass bei diesem Ansatz der Rückgabewert kein Bool ist, sondern eine Property. In die Property selber kodiert, ist es möglich, einen boolschen Ausdruck von der eigentliche Property zu stellen.

Falls diese erfüllt ist, läuft der Test ganz normal durch. Falls sie jedoch zu false evaluiert, wird der Test verworfen.
Hier ein Beispiel für **popList**:

```
prop_popFirst :: MyList -> Property
prop_popFirst xs = listSize xs > 0
    ==> (listSize (popFirst xs) == (listSize xs) - 1)
```

3.3.3 Ansatz 2 Einschränken der Werte mit suchThat

Der zweite Ansatz setzt dort an, wo Testdaten von Arbitrary generiert werden. Dies geschieht beim eigentlichen Verwenden der Property mit QuickCheck. Beispiel:

```
quickCheck (forAll (arbitrary
    `suchThat` (\xs -> listSize xs > 0))prop_popFirstSuchThat)
```

3.3.4 Unterschiede und generative Ansätze

Beide funktionieren wie ein Filter, der sich den generierten Wert anschaut und mit der Bedingung prüft, ob dieser zulässig ist oder nicht.
Ein sehr wichtiger Unterschied zwischen den beiden Methoden ist, dass es bei Ansatz 1 eine Obergrenze für die Anzahl von Versuchen gibt. Falls kein Wert, der die Bedingung erfüllt, gefunden wird, gibt der Test auf und markiert diesen ebenfalls als nicht bestanden. Im zweiten Fall wird unendlich lange versucht, einen passenden Wert zu finden.
Man bemerke, dass sehr spezifische Filter, die eine bestimmte Teilmenge abbilden, sollen eventuell nicht gefunden werden oder sehr lange brauchen, um gefunden zu werden.
Falls dies der Fall ist ,lohnt es sich, einen neuen Generator zu schreiben der gezielt Werte mit der gewünschten Bedingung produziert.

3.4 Finden von Bugs (mit Haskell QuickCheck)

Angenommen wir betrachten eine nun fehlerhafte Implementation von reverseList der oben erwähnten Datenstruktur MyList.

```
reverseListEasyBug :: MyList -> MyList
reverseListEasyBug Empty = Empty
reverseListEasyBug (Cons x (Cons y Empty)) = Cons x Empty
reverseListEasyBug (Cons x xs) = addToListE x (reverseListEasyBug xs)
```

Diese produziert für den Fall mir zwei Elementen ein fehlerhaftes Ergebnis.
Die passende Property dafür sieht so aus.

```
prop_reverseListEasyBug :: MyList -> Bool
prop_reverseListEasyBug xs = reverseListEasyBug (reverseListEasyBug xs) == xs
```

Nach dem Ausführen des Test mit der Zeile:

```
quickCheck prop_reverseListEasyBug
```

Erreichen wir das erwartete Resultat

```
=======QuickCheck======
*** Failed! Falsified (after 4 tests):
-1 -3
```

Der Test war Fehlerhaft, da die Funktion die von uns vorausgesetzte Eigenschaft nicht gewährleisten kann. In unseren 100 Versuchen, war eine Liste, die 2 oder mehr Elemente hatte. In diesem Konkreten Fall die Liste [-1,-3]

Dies heißt aber nicht, dass QuickCheck jeden Bug so einfach findet. Betrachten wir den Fall der folgenden Implementation.

```
reverseLisBug [] = []
reverseListBug
  | length xs >= 1000 = []
  | otherwise = reverseListBug (tail xs) ++ [head xs]
```

Diese produziert zwar für Listen, die weniger als 1000 Elemente haben, das gewünschte Ergebnis. Für einen Großteil aller möglichen Listen ist jedoch die leere Liste die Ausgabe.
Hierzu noch eine entsprechende Property.

```
prop_reverseListBug :: MyList -> Bool
prop_reverseListBug xs = reverseListBug (reverseListBug xs) == xs
```

Obwohl die Funktion offensichtlich falsch ist erhalten wir nach dem Ausführen das Ergebnis OK:

```
quickCheck prop_reverseListBug
```

Ergebnis:

```
=======QuickCheck======
+++ OK, passed 100 tests.
```

Da nur ein Bruchteil aller möglichen Eingaben getestet wird bleibt der Bug unentdeckt. Aber selbst wenn wir die Anzahl der Versuche mit folgendem Code erhöhen bleibt der Fehler unentdeckt.

```
quickCheck (withMaxSuccess 100000 prop_reverseListBug)
```

Ergebnis:

```
=======QuickCheck======
+++ OK, passed 100000 tests.
```

Anscheinend werden sehr selten Liste mit einer Länge von mehr als 1000 produziert.
Selbst Listen mit mehr als 100 Elementen nach oftmaligem testen so gut wie gar nicht vor. Dies zeigt, dass die Art wie Werte generiert werden sehr wichtig ist. Eine breite Streuung und Abdeckung der Randfälle ist sehr wichtig. Wenn man gewisse Filter benutzt kann man das erzeugen von Listen mit mehr als 1000 Elementen erzwingen und der Fehler wird gefunden, jedoch ist dies ohne das Vorwissen über die Art des Bugs nicht sehr hilfreich.
Es macht aber auch Sinn keine allzu großen Listen zu testen, da dadurch die Geschwindigkeit der Tests zu stark beeinflusst wird.
Dies wird später am Beispiel mit Java noch ersichtlich.

3.5 Ablauf eines Haskell Quickcheck-Tests

Ein Haskell QuickCheck wird mithilfe der quickCheck Funktion und einer Property eingeleitet:

```
quickCheck  example_property
```

Innerhalb der quickCheck Methode wird die Property nun in eine Schleife gewrappt. Innerhalb dieser passiert folgendes:

1. Es werden Werte via Arbitrary für einen beliebigen Typ t generiert.
2. Die Property wird mit dem generierten Wert aufgerufen
3. - Falls die Propery True zurückgib beginnt ein neuer Schleifendurchlauf
   - Falls die Property False zurückgibt wird auf der Konsole ausgegeben, dass der Test nicht bestanden wurde und welche Werte die Property falsifiziert haben
4. Falls es keinen Fehler gab, wird eine Ausgabe gemacht, dass der Test erfolgreich war.

Im Fehlerfall wird neben dem ausgeben des Fehlers auch versucht den eigentlichen Fehler zu minimieren (via shrink). Die zusätlichen Informationen werden bei der Ausgabe berücksichtigt.

4 QuickCheck in Java

um quickcheck in java zu verwenden, muss man zunächst die junit-quickcheck-bibliothek in sein projekt einbinden. danach kann man eine testklasse erstellen und diese mit der annotation **@runwith** und dem runner **junit-quickcheck** versehen, um zu signalisieren, dass quickcheck für den testlauf verwendet werden soll. Als nächstes definiert man eine oder mehrere Eigenschaften, die man für eine Funktion oder Methode überprüfen möchte. Eine Eigenschaft ist eine Methode, die mit der Annotation **@Property** versehen wird und einen oder mehrere Parameter enthält. Diese Parameter werden von QuickCheck mit zufälligen Werten gefüllt. Die Methode sollte dann eine Aussage darüber treffen, welche Eigenschaften für diese Eingabewerte gelten sollten. Wenn eine Eigenschaft für mindestens einen Testfall nicht erfüllt ist, gibt QuickCheck eine Fehlermeldung aus.
Erläuterung am Beispiel einer selbst implementierten Liste, die Integer speichert mit den Funktionen:

- add: Fügt einen Wert ans Ende der Liste an
- insert: Fügt einen Wert an den Anfang der Liste
- reverse: Dreht die Elemente in der Liste um.
- size: Gibt die Länge der Liste aus.
- sum: Gibt aus, was die Summe aller Listenelemente ist.

4.1 Funktionsimplementierung

```
public List add(int num) {
    Node newNode = new Node(num);
    if (this.head == null) {
        this.head = newNode;
    } else {
        Node current = this.head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
    return this;
}

public String displayString() {
    Node current = this.head;
    String s = "";
    while (current != null) {
        s += current.value + ", ";
        current = current.next;
    }
    return s;
}

public int size() {
    if (this.head == null) {
        return 0;
    }
    Node current = this.head;
```

```
    int c = 0;
    while (current != null) {
        c++;
        current = current.next;
    }
    return c;
}

public int pop() throws Exception{
    if(this.head == null){
        throw new Exception("can't pop from empty List");
    }else{
        int toRet = this.head.value;
        this.head = this.head.next;
        return toRet;
    }
}

public List insert(int val){
    Node newNode = new Node(val);
    if(this.head == null){
        this.head = newNode;
        return this;
    }else{
        Node oldHead = this.head;
        this.head = newNode;
        this.head.next = oldHead;
        return this;
    }
}
public List reverse(){
    List newList = new List();
    while(this.head != null){
        try {
            newList.insert(this.pop());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    this.head = newList.head;
    return this;
}

public int sum(){
    if (this.head == null) {
        return 0;
    }
    Node current = this.head;
    int sum = 0;
    while (current != null) {
        sum+= current.value;
        current = current.next;
    }
```

```
        return sum;
    }
```

4.2 Generieren von Testdaten

Testdaten werden in Java entweder über Generatoren oder den Konstruktor generiert. Beispiel anhand des List-Generators:

```
public class ListGenerator extends Generator<MyList> {
    public ListGenerator(){
        super(MyList.class);
    }

    @Override
    public MyList generate(
            SourceOfRandomness r,
            GenerationStatus status
    ){
        int listLength = r.nextInt(0,1000);
        MyList genMyList =new MyList();
        for (int i = 0; i < listLength; i++) {
            genMyList.add(r.nextInt());
        }
        return genMyList;
    }
}
```

Hier sind die zwei wichtigsten Bestandteile das Erben von der Klasse Generator und die Methode generate. Durch das Vererben erhalten wir Zugriff auf **SourceOfRandomness r** und **GenerationStatus status**. Die Methode Generate ist die, die letztendlich für das eigentliche generieren Verantwortlich ist. Da aber bereits Generatoren für alle primitiven Typen existieren, könnte man auch die für den Konstruktor notwendigen Typen generieren lassen und das Objekt so erstellen.

4.3 Definieren von Properties

Definieren und Ausführen von Properties passiert in Java im gleichen Schritt. Hier ein Beispiel:

```
@Property(trials = 1000)
public void popAssume(@From(ListGenerator.class) MyList myList) throws Exception {
    assumeTrue(myList.size() > 0);
    int sizeBeforePop = myList.size();
    myList.pop();
    int sizeAfterPop = myList.size();
    assertThat(sizeBeforePop-1,equalTo(sizeAfterPop));
}
```

Die Anatomie eines QuickCheck-Tests in Java beginnt mit der **@Property** Annotation.
Hier kann man auch die Anzahl der Versuche erhöhen. Dies lohnt sich vor allem dann, wenn die Bedingung, die man an seine generierten Testdaten stellt, eventuell schwer zu erreichen ist. Der Default-Wert liegt wie bei QuickCheck in Haskell bei 100 Versuchen, die Property zu falsifizieren.
In der Signatur der Methode kann man mit **@From(YourGenerator.class)** einen Generator angeben, den man zum Erzeugen des Objekts benutzen

möchte. So könnte man auch für generative Ansätze unterschiedliche Generatoren für unterschiedliche Properties benutzen.
Anstatt **suchThat** benutzt man in Java **assumeTrue** um seine Bedingung an die Testdaten zu stellen. Eine Möglichkeit, unendlich lange nach passenden Daten zu suchen, existiert hier nicht. Sobald die **Trials** aus der Annotation ausgeschöpft, sind wird auch dieser Test als fehlgeschlagen markiert. Danach kann man wie bei normalen Unit-Tests mittels einer Assertion-Bibliothek wie **Hamcrest** prüfen, ob gewisse Annahmen gelten, und so die Property so zu true oder false evaluieren lassen. Ein sehr nützliches Feature an dieser Stelle ist, dass man durch Annotation der Parameter nützliche Filter wie **@Where** benutzen oder sogar generativ Zahlen mittels **@InRange** in einem Intervall generieren lassen kann.

```
@Property()
public void testAdditionUsingInRange(@InRange(minInt = 0) int number) {...}
@Property()
public void testAdditionUsingSatisfies(@When(satisfies = "#_ >= 0") int number) {...}
```

4.4 Finden von Bugs (mit Java QuickCheck)

Wir betrachten die gleichen fehlerhaften reverseFunktionen wie bei Haskell QuickCheck. In Java sehen diese in unserer Implementierung so aus:

```
public MyList reverseEasyBug(){
    if(this.size() == 2){
        MyList toRet = new MyList();
        toRet.add(this.head.value);
        return toRet;
    }else{
        return this.reverse();
    }
}
public MyList reverseBug(){
    if(this.size() > 1000){
        return new MyList();
    }else{
        return this.reverse();
    }
}
```

Die Properties, zum testen der Funktionen in Java sieht so aus.

```
@RunWith(JUnitQuickcheck.class)
public class MyListQuickCheckTest {
    @Property
    public void doubleReverseIsOriginalEasyBug(@From(ListGenerator.class) MyList myList){
        MyList reversedTwice = myList.reverseEasyBug().reverseEasyBug();
        assertThat(reversedTwice, equalTo(myList));
    }
    @Property
    public void doubleReverseIsOriginalBug(@From(ListGenerator.class) MyList myList){
        MyList reversedTwice = myList.reverseEasyBug().reverseEasyBug();
        assertThat(reversedTwice, equalTo(myList));
    }
}
```

Führen wir diese nun mittels eines Kommandos aus, erhalten wir ein überraschendes Ergebnis. Keiner unserer absichtlich platzierten Bugs wurde erkannt.
Input:

```
mvn clean test -Dtest=MyListQuickCheckTest
```

Output:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.example.MyListQuickCheckTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.845 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Keiner unserer eingebauten Bugs wurde erkannt. Hierbei muss man anmerken, dass unsere Implementation aus Effizienzgründen hier nicht rekursiv mit unveränderbaren Objekten, sondern durch While-Schleifen realisiert ist. Deswegen in der Implementation der Fehler nur bei Listen, die genau aus zwei Elementen bestehen. Falls alle Listen gleich wahrscheinlich sind, ist der 2. Fall sogar wahrscheinlicher. Manchmal wird der Fehler auch erkannt. Der Zweite Fall ist sogar unmöglich, basierend auf unserem jetzigen Generator, da dieser mit der Zeile alle Listen mit einer Länge über 999 gar nicht erst erzeugt werden.

```
int listLength = r.nextInt(0,1000);
```

Aber selbst wenn man das Intervall größer machen würde, reicht dies bei 100 Testfällen wie bei Haskell QuickCheck immer noch nicht aus um den Fehler konsistent zu finden.

An dieser Stelle ist zu bemerken, dass das generieren von Komplexen Typen in Java zwar möglich ist, sich die Implementation von Features wie Shrinking oder sich an neue Werte herantasten ist schwierig. Die von Java QuickCheck bereits zur verfügung gestellten Generatoren für integer haben damit kein Problem. Wenn man die Liste z.B. aus einem int Array erstellt, das den bereits implementierten Generator benutzt wird der erste Fehler sehr konsistent gefunden.

```
@Property
public void doubleReverseIsOriginalEasyBugQCGen(int[] array){
        MyList myList = ListGenerator.generateFromIntArray(array);
        MyList reversedTwice = myList.reverseEasyBug().reverseEasyBug();
        assertThat(reversedTwice, equalTo(myList));
}
```

Output:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.example.MyListQuickCheckTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.74 sec <<< FAILURE!
doubleReverseIsOriginalEasyBugQCGen(org.example.MyListQuickCheckTest)
Time elapsed: 0.169 sec  <<< FAILURE!
java.lang.AssertionError: Property doubleReverseIsOriginalEasyBugQCGen
```

```
falsified via shrinking:
Expected: <0, 1, >
     but: was <0, >
Shrunken args: [[0, 0]]
Original failure message: [
Expected: <0, 1, >
```

Der Grund warum der Bug vom QuickCheck implementierten Generator schneller gefunden wird und **shrinking** bereitstellt ist, dass der Generator für diese Werte ein bisschen mehr kann als unserer aus dem naiven Ansatz mit komplett zufälligen Werten.

4.4.1 Shrinking in einem eigenem Generator

Shrinking in einem selbstgeschriebenen Generator zu implementieren ist sehr einfach. Es geschieht über das Überschreiben der doShrink Methode. Diese akzeptiert einen Wert, der zum Nicht-Erfüllen der Property führt und versucht diesen dekremental kleiner zu machen. Dies versucht er dekremental, bis er einen kleineren Wert gefunden hat, der den Fehler ebenfalls produziert. Hier ein Beispiel für die doShrink Methode in der Generatoren-Klasse:

```
@Override
public List<MyList> doShrink(SourceOfRandomness r, MyList larger){
    if(emptyMyList.equals(larger)){
        return Collections.EMPTY_LIST;
    }else{
        MyList adjustedValuesList = larger.getCopy();
        for (int i = 0; i < larger.size(); i++) {
            int currentNumber = adjustedValuesList.getAtIndex(i);
            adjustedValuesList.setAtIndex(i, currentNumber / 3);
        }
        return Stream.of(
                adjustedValuesList,
                larger.getCopy().popAmount(larger.size()/2)
        ).distinct().collect(Collectors.toList());
    }
}
```

Der Rückgabewert ist eine Liste von Werten, die kleiner sind als das Attribut larger. Die Konstante Origin beschreibt hierbei den einfachsten Typ, dem wir uns Schritt für Schritt zu nähern versuchen.
Diese erreichen wir, indem wir entweder die Zahlen innerhalb der Liste näher zu 0 bringen oder Elemente aus der Liste entfernen.
Falls unsere Eingabe schon die leere Liste ist, können wir nicht kleiner werden und geben eine leere Liste zurück. Wir machen die Liste kleiner, indem wir einmal versuchen, den Fehler mit Werten zu reproduzieren, die nur $\frac{1}{3}$ der Werte aus larger sind, und versuchen, den Fehler mit den gleichen Werten zu reproduzieren, wobei wir aber die Hälfte der Werte weglassen. Theoretisch könnte man hier noch beliebig granular vorgehen und bspw. noch um diese Fälle erweitern.

- entferne 1 Element
- entferne 1/4 der Elemente

Durch das Implementieren dieser Methode erreichen wir den folgenden Output beim Testen unserer Methode.

```
Expected: <length: 2 [0, 0]>
 but: was <length: 1 [0]>
Shrunken args: [length: 2 [0, 0]]
Original failure message: [
Expected: <length: 2 [-1927522581, 901617223]>
 but: was <length: 1 [-1927522581]>]
```

Obwohl der originale gefundene Fehlerfall bei der Liste **[-1927522581, 901617223]** auftrat, haben wir durch das dekrementale Herantasten herausgefunden, dass dieser bereits bei der Liste **[0,0]** auftritt.

4.4.2 Generieren von Werten abhängig von State

Der Generator von Default-Werten tastet sich langsam mehr und mehr an größere Werte heran. Dies wird mittels des **state** Parameters in der generate Methode erreicht. Status enthält z.B. das sogenannte **size** Attribut, das zufällig auswählt wie Komplex der zu produzierende Testinput sein soll. In diesem speziellen Beispiel benutze ich auch noch die Anzahl der Versuche, die der Testcase schon durchlaufen hat.
Eine mögliche Verwendung dieser könnte so aussehen:

```
//improved Generator
@Override
public MyList generate(
        SourceOfRandomness r,
        GenerationStatus status
){
    int size = status.size();
    int upperLimit;
    int listLength;
    if(size < 10){
        upperLimit = 10;
    } else if (size > 75) {
        upperLimit = size*status.attempts();
    }else{
        upperLimit = 1000;
    }
    listLength = r.nextInt(0,upperLimit);
    MyList genMyList =new MyList();
    for (int i = 0; i < listLength; i++) {
        genMyList.add(r.nextInt());
    }
    return genMyList;
}
```

Size ist ein variabler Wert, der zwischen 1 und 100 liegt (erhöhen der Trials für einer Property erhöht auch den maximalen Wert). Je kleiner der Wert, desto weniger komplex soll das zu generierende Value sein. Hier unterschieden wir zwischen den Fällen.

- kleiner als 10
- größer 75
- Wert dazwischen

Die obere Grenze für die Listen-Länge wird hier entsprechend angepasst. Sie wächst in diesem Generator mit der Anzahl der Versuche, was dabei hilft, eventuell auch Fehlschläge abzudecken, die erst bei noch längeren

Listen entstehen. Falls wir aber sehr wenige Versuche eingestellt haben, bewegen wir uns eher in kleineren Dimensionen.
Mit diesem Test werden beide unserer eingebauten Bugs gefunden, da wir sowohl gezielt die kleineren abdecken, als auch größere mit Längen über 1000. Hier den Output für den nicht bestandenen zweiten Bug.

```
Expected: <length: 1193 [0, 0, 0, ...]>
     but: was <length: 0 []>
Shrunken args: [length: 1193 [0, 0, 0, ...]
Original failure message: [
Expected: <length: 1193 [-624066378, -1813999667, 437046618, ...]>
     but: was <length: 0 []>]
```

Der Generator hat erfolgreich einen Wert über 1000 generiert und diesen über Weiteres auf 1 minimiert, bei dem alle Elemente 0 sind.
Die Länge der kleiner gemachten Inputs blieb jedoch gleich, da die Liste immer halbiert wird und der Fehler beim ersten Halbieren somit nicht mehr auftreten würde.
Jetzt besitzen wir einen Generator, der bei mehr Versuchen sogar Fehler finden kann, die bei beliebig kleinen oder beliebig großen Eingaben auftreten könnten.
Um das Generieren von komplexen Typen zu gewährleisten, sollte man an dieser Stelle noch die Untergrenze der zu generierenden Listenlänge anpassen.

4.4.3 Evaluation

Abschließend kann man sagen, dass QuickCheck ein sehr nützliches Werkzeug ist, um Bugs zu finden. Jedoch ist es wie oben gezeigt keine absolute Wissenschaft. Meist bekommt man für einen mäßig implementierten Generator auch nur mäßige Ergebnisse. Und oft sind 100 Versuche nicht genug, um gut versteckte Bugs zu finden. Des Weiteren ist es wichtig, kleine Werte, mittlere Werte und sehr große Werte zu generieren, um zu versuchen, möglichst viele Teilmengen abzudecken, die einen Bug produzieren könnten. Das Generieren von zu großen Eingaben führt eventuell zu Performance-Problemen, auch wenn man diese gerne getestet hätte. QuickCheck ist gedacht, selbst geschriebene Unit-Tests zu ergänzen, und das macht es sehr gut. Über die Spanne eines Projekts werden die Tests auch deutlich mehr als einmal ausgeführt, was dazu führt, dass, wenn der Test in vielen Durchläufen lokal oder in einer Deployment-Pipeline oft nicht fehlschlägt, kann man sich über die Zeit immer sicherer sein, dass es keinen Input gibt, der die Property falsifiziert.

4.5 Ablauf eines Java Quickcheck-Tests

In Java initiiert man einen QuickCheck Test, indem man (am Beispiel von Maven) folgenden Befehl in einem Maven-Projekt ausführt.

```
mvn test
```

Dieser macht folgendes:

1. Alle Klassen in Source werden kompiliert und das Kompilat im target Ordner platziert
2. Wenn ein Test im Testorder liegt und das Suffix Test hat, wird dieser mithilfe des surefire Plugins ausgeführt.

Bis hierhin funktionieren normale Tests in Java gleich. Finden sich aber über einer Testklasse die Annotation **@RunWith(JUnitQuickcheck.class)** wird der QuickCheck-Testrunner statt des Standard-JUnit-Testrunners verwendet wird. Dieser Testrunner ist in der Lage die **@Property** Annotation richtig zu interpretieren und die Methode mit zufälligen Testdaten zu versorgen und diese mehrmals auszuführen.

Hier ein grober Ablauf des Vorgangs:

1. Der JUnitQuick-Testrunner wird durch die Annotation @RunWith(JUnitQuickcheck.class) angestoßen
2. Der JUnitQuickCheck-Testrunner geht durch alle Funktionen der Klasse und prüft ob sie die Annotation @Property besitzen
3. Er bestimmt mittels des Trials Parameter in der Annotation wie oft diese Funktion durchlaufen werden muss
4. Der JUnitQuickCheck-Testrunner kann durch die Methodensignatur einsehen welche Art von Parameter er für den Aufruf zu generieren hat. Es gibt hier zwei Fälle:
   - Primitiver Datentyp (ohne Generator Annotation): Hier liest der Testrunner den gefragten Wert aus der Datei aus und lässt sich dann von einem vorgeschriebenen Generator der Java QuickCheck-Blibliothek einen Wert erzeugen
   - Generator Annotation: In diesem Fall sucht sich der Testrunner die annotierte Generatorklasse und falls er diese findet, instaziiert er sie und lässt sich über die **generate-Methode** einen Wert generieren
5. Nachdem der Runner für alle nötigen Argumente einen entsprechenden Wert generiert hat wird die Funktion mit den Werten aufgerufen. Man könnte an dieser Stelle auch sagen, dass die Property Annotation als Wrapper fungiert.
6. Der nächste Schritt gleicht einem normalen Java-Testaufruf, bei dem über Assertion bestimmt wird ober gewisse Eigenschaften gegeben sind oder nicht.
   - Falls eine Assertion fehlschlägt markiert der Testrunner den Test als nicht bestanden und gibt aus, warum die Assertion fehlgeschlagen ist.
   - Falls die Funktion keine fehlschlagenden Assertions beinhaltet und die Anzahl der **Trials** noch nicht ausgenutzt wurde, genertiert der Testrunner neue Werte über die Generatoren und führt die Funktion erneut aus.
   - Falls alle Trials ohne fehlschlagenden Assertions durchlaufen wurden, wird der Test als bestanden markiert und wiederholt (falls vorhanden) die Schritte für die nächste mit **@Property** markierte Methode.

Um zu verstehen wie die Annotationen das oben beschriebene Verhalten ermöglicht, müssen wir man sich die Klasse **JUnitQuickCheck** genauer anschauen, da sie die Rolle des Testrunners übernimmt und in ihr die Annotationen entsprechen verarbeitet werden. Es ist nämlich genau die Klasse, die für das Generieren und Wiederholen zuständig ist. Der Testrunner ist ein eigenes Programm, das sich jeden einzelnen Test anschaut und die Funktionen, falls korrekt annotiert aufruft und vor allem mit den Testdaten befüllt. Die Annotation ist so gesetzt, dass diese bis

Run-Time erhalten bleibt und man mit dem Testrunner problemlos alle Werte auslesen kann. Tests die nicht mit **@Property** sonder mit **@Test** annotiert sind werden an die normale JUnitTestrunner-Funktionen weitergeleitet.

Zusammengefasst kann man also sagen, dass der JUnit Testrunner, bei einem QuickCheck-Test die Kontrolle, an den QuickCheck Testrunner weitergibt, dieser schaut sich dann alle Funktionen an. Falls diese mit @Property markiert sind, packt er sie in einen Schleife, so oft durchlaufen wird wie der Parameter Trials definiert ist. In einem Schleifendurchlauf kann der Testrunner sich die Methodensignatur der Funktionen anschauen und entsprechende Generatoren instaziieren, welche dann Werte für die Parameter aus der Methodensignatur bereitstellen. Dieser Wrapper ruft dann ruft letztendlich die Funktion auf und übergibt diese Werte. Die Funktion selber verläuft dann wie ein ganz normaler JUnit-Test.

5 Vergleich der beiden Implementationen

Sowohl in Haskell als auch in Java bietet QuickCheck eine automatisierte Methode, um Eigenschaften von Funktionen zu testen. An sich ist sich QuickCheck in beiden Sprachen sehr ähnlich. Im Detail gibt es jedoch Unterschiede. In Haskell werden zufällige Werte durch eine Instanz der Arbitrary-Typklasse erzeugt. Diese Instanz beschreibt wie zufällige Werte genertiert werden können. Quickcheck benutzt dann diese Instanzen, um die Werte zu generieren und die Funktionen mit diesen zu testen. Die Funktion wird dann in einer weiteren Funktion verwendet, die eine Property beschreibt, dessen Integrität mit den verschiedenen Werten getestet wird. In Java gibt es diese Typklassen-Instanzierung nicht. Die Generierung zufälliger Werte erfolgt über Generatoren, die Objekte von bestimmten Typs erzeugen. Die zu testende Funktion wird dann in einer Testklasse definiert, welche die Annotation **@Property** besitzt. Diese Methode wird dann von QuickCheck erkannt und aufgerufen. Hierbei verwendet QuickCheck die Generatoren, um die Werte, die in der Signatur stehen, zu befüllen und dann schließlich damit die Eigenschaft zu testen. Zuletzt gibt es noch den Unterschied, dass Haskell-Quickcheck, ohne zusätzlichen Aufwand, mehr Typen unterstützt als Java-Quickcheck. Beispielsweise bietet Haskell-QuickCheck nützliche Funktionen zum Generieren von Listen, Bäumen oder auch Graphen. In Java müssen diese mit einem Generator erstellt werden.

5.1 Gegenstellung verschiedener Aspekte

- Historie: QuickCheck wurde ursprünglich für Haskell entwickelt und ist seit langem ein fester Bestandteil der Haskell-Toolbox für Entwickler. Später wurde QuickCheck für Java portiert und ist seitdem zu einer populären Bibliothek für automatisierte Tests in Java geworden.
- Syntax: Die Syntax von QuickCheck ist sehr elegant und idiomatisch für Haskell, wohingegen Syntax von QuickCheck in Java zwar etwas weniger idiomatisch als in Haskell, aber immer noch einfach zu verwenden und gut dokumentiert.
- Leistung: Haskell QuickCheck ist sehr leistungsfähig und kann schnell und effizient eine große Anzahl von Testfällen generieren und ausführen. Java strahlt hier durch Parallelisierung, die die Leistung von QuickCheck steigern kann.
- Typsicherheit: Die Typsicherheit von Haskell ermöglicht es, typsichere Eigenschaften und generierte Eingabedaten zu schreiben, was die Fehlererkennung erleichtert. Java QuickCheck nutzt die Typsicherheit von Java, um typsichere Eigenschaften und generierte Eingabedaten zu schreiben, was auch die Fehlererkennung erleichtert.
- Funktionalität: Haskell QuickCheck unterstützt auch eine Vielzahl von nützlichen Funktionen wie Shrinking, um Testfälle zu minimieren, die den Fehler verursachen. Java QuickCheck unterstützt auch Shrinking, um Testfälle zu minimieren, die den Fehler verursachen.

5.2 Gegenstellung der Abläufe

In Haskell ist QuickChecks eine normale Funktion, die eine Property als Parameter übergeben bekommt. Diese Property ist wiederum auf einen

Arbitrary Typ t angewiesen. Der Typ t nun innerhalb der von der quickCheck Methode instaziiert, an die Property t übergeben und eine gewisse Anzahl oft aufgerufen. Die quickCheck Funktion fungiert hier also wie ein Wrapper, der die nötigen Werte generiert und letztendlich die Funktion aufruft. In Java ist es ähnlich, nur dass der Verantwortlichkeit des Wrappers nicht so transparent ist. Die Klasse JUnitQuickCheck ist der Testrunner, der sich um sammeln der @Property Funktionen, das Verpacken in eine Schleifen, Auslesen der Methodensignatur, Erzeugen von nötigen Generatoren basierend auf dieser, Aufrufen der generate Funktion zum erzeugen der Werte und letztendliches ausführen der Funktion kümmert. Java QuickCheck hat an dieser Stelle viel Overhead im Vergleich zu Haskell QuickCheck. Haskell QuickCheck ist sogar ergonomisch genug, dass man während des Schreibens von Funktionen diese nebenher in einer Zeile Code testen kann.

5.3 Fazit

Insgesamt sind beide Implementierungen sehr nützlich und effektiv, um die Qualität von Code zu verbessern. Welche Implementierung besser geeignet ist, hängt von den spezifischen Anforderungen des Projekts und den Präferenzen des Entwicklers ab. Wenn man mit einer funktionalen Sprache wie Haskell arbeitet, ist die Haskell QuickCheck-Implementierung wahrscheinlich die bessere Wahl. Wenn man hauptsächlich mit Java arbeiten, ist die Java QuickCheck-Implementierung möglicherweise die bessere Wahl.

Literatur

[1] Haskell quickcheck dokumentation. `https://hackage.haskell.org/package/QuickCheck`.

[2] Junit-quickcheck dokumentation. `https://pholser.github.io/junit-quickcheck/site/1.0/`.