Property-Based Testing mit Java

Entwicklertag Karlsruhe
4. Juni 2019

@johanneslink

johanneslink.net

Beispiel-basierte Tests

Ein *Beispiel* zeigt, dass unser Code
bei ganz konkreten Eingaben
ein ganz konkretes Ergebnis liefert.

```
@Example
void reverseList() {
   List<Integer> aList = Arrays.asList(1, 2, 3);
   Collections.reverse(aList);
   assertThat(aList).containsExactly(3, 2, 1);
}
```

Funktioniert *reverse()* nur für die getesteten Beispiele?

Wie repräsentativ sind unsere Tests?

Wie viele Beispiele benötigen wir um ausreichendes Vertrauen zu schaffen?

```
@Example void emptyList() {
    List<Integer> aList = Collections.emptyList();
    assertThat(Collections.reverse(aList)).isEmpty();
}

@Example void oneElement() {
    List<Integer> aList = Collections.singletonList(1);
    assertThat(Collections.reverse(aList)).containsExactly(1);
}

@Example void manyElements() {
    List<Integer> aList = asList(1, 2, 3, 4, 5, 6);
    assertThat(Collections.reverse(aList)).containsExactly(6, 5, 4, 3, 2, 1);
}

@Example void duplicateElements() {
    List<Integer> aList = asList(1, 2, 2, 4, 6, 6);
    assertThat(Collections.reverse(aList)).containsExactly(6, 6, 4, 2, 2, 1);
}
```

Properties

Eine *Property* zeigt, dass unser Code
für eine Klasse von Eingaben (Vorbedingung)
bestimmte allgemeine Eigenschaften (Invarianten)
erfüllt.

```
Collections.reverse(List aList):
    // Vorbedingungen?
    // Nachbedingungen und Invarianten?
```

```
Collections.reverse(List aList):
    // Vorbedingungen?
    // Nachbedingungen und Invarianten?
```

Vorbedinungen

- Beliebige Liste
- Nicht null

Invarianten

- Länge der Liste bleibt gleich
- Alle Elemente bleiben erhalten
- Nach reverse ist das erste Elemente das letzte
- 2 x reverse erzeugt wieder das Original

Eine Property als Java Code

```
boolean theSizeRemainsTheSame(List<Integer> original) {
   List<Integer> reversed = reverse(original);
   return original.size() == reversed.size();
}


private <T> List<T> reverse(List<T> original) {
   List<T> clone = new ArrayList<>(original);
   Collections.reverse(clone);
   return clone;
}
```

Jqwik

```
@Property
boolean theSizeRemainsTheSame(@ForAll List<Integer> original) {
   List<Integer> reversed = reverse(original);
   return original.size() == reversed.size();
}
```

```java
@Property
void allElementsStay(@ForAll List<Integer> original) {
    List<Integer> reversed = reverse(original);
    Assertions.assertThat(original).allMatch(
        element -> reversed.contains(element)
    );
}
```

```java
@Property
boolean reverseMakesFirstElementLast(@ForAll List<Integer> original) {
    Assume.that(original.size() > 2);
    Integer lastReversed = reverse(original).get(original.size() - 1);
    return original.get(0).equals(lastReversed);
}
```

```java
@Property
boolean reverseTwiceIsOriginal(@ForAll List<Integer> original) {
    return reverse(reverse(original)).equals(original);
}
```

```haskell
prop_reversed :: [Int] -> Bool
prop_reversed xs =
    reverse (reverse xs) == xs
```

Haskell!

Demo

- Reverse List
- Length of String
- Absolute value of Integer
- Sum of two integers
- Einbindung in Gradle und IntelliJ

Was ist jqwik?

https://jqwik.net

- Eine Test-Engine für die JUnit5–Plattform
- Ein Generator für Testfälle mit
  - zufälligen und typischen Eingabewerten
  - manchmal sogar die vollständige Menge aller möglichen Eingabekombinationen
- Aktuelle Version: 1.1.5

Was ist jqwik nicht?

- Es ist kein vollständig randomisiertes Testwerkzeug, das man ohne Nachdenken auf sein Programm loslässt.
- Properties werden nicht bewiesen, sondern widerlegt (aka falsifiziert)

```java
@Property
void squareOfRootIsOriginalValue(@ForAll double aNumber) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
java.lang.AssertionError:
  Expecting:
    <NaN>
  to be close to:
    <-1.0>
  by less than 1% but difference was NaN%.
  (a difference of exactly 1% being considered valid)
```

Beschränkung generierter Werte

Häufig gilt eine Property nur für eine
beschränkte Untermenge eines Typs

```java
@Property
void squareOfRootIsOriginalValue(
    @ForAll @DoubleRange(min=0) double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

```java
@Property
void squareOfRootIsOriginalValue(
    @ForAll @Positive double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

```java
@Property
void squareOfRootIsOriginalValue(
    @ForAll("positiveDoubles") double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}

@Provide
Arbitrary<Double> positiveDoubles() {
    return Arbitraries.doubles().between(0, Double.MAX_VALUE);
}
```

```
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

Arbitrary:
"Monaden-ähnliche" Factory von Generatoren
für zufällige Werte

```
public interface Arbitrary<T> {
    RandomGenerator<T> generator(int genSize);

    default Arbitrary<T> filter(final Predicate<T> filterPredicate) {…}
    default <U> Arbitrary<U> map(final Function<T, U> mapper) {…}
    …
}
```

```
public interface RandomGenerator<T> {
    Shrinkable<T> next(Random random);
}
```

```java
@Property
void squareOfRootIsOriginalValue(@ForAll double aNumber) {
    Assume.that(aNumber > 0);

    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
tries = 1000,
checks = 489,
seed = -1808546598028468149
```

```
static <E> List<E> brokenReverse(List<E> aList) {
    if (aList.size() < 4) {
        aList = new ArrayList<>(aList);
        reverse(aList);
    }
    return aList;
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
  Property [reverseShouldSwapFirstAndLast] falsified with sample
     [[0, -2147483648, 2147483647, -7997, 7997, -3223, -6474, 1915, -7151,
     3102, 4362, 714, 3053, 1919, -445, 7498, -2424, 3016, -5127, -7401,
     -7946, -3801, -305]]
```

```java
static <E> List<E> brokenReverse(List<E> aList) {
    if (aList.size() < 4) {
        aList = new ArrayList<>(aList);
        reverse(aList);
    }
    return aList;
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
  Property [reverseShouldSwapFirstAndLast] falsified with sample
     [[0, 0, 0, -1]]
```

The Importance of Being Shrunk

- "Schrumpfen" einer falsifizierten Property: Finde das einfachste Eingabe-Beispiel, das immer noch fehlschlägt.
- Manchmal gibt es das einfachste Beispiel nicht, oder die Suche danach würde sehr lange dauern.
- Benutze Heuristiken um Werte zu schrumpfen, z.B.
  - Versuche Zahlen-Werte näher bei Null
  - Verkleinere Listen, Mengen, Arrays

Type-based vs Integrated Shrinking

- Type-Based Shrinking: Nur der Typ von Werten dient als Constraint für die Schrumpfversuche
  - Problem: Schrumpfen kann zu Ergebnissen führen, die eigentlich bei der Generierung ausgeschlossen wurden
- Integrated Shrinking: Alle Schritte und Bedingungen der Generierung werden beim Schrumpfen berücksichtig
- jqwik implementiert integriertes Schrumpfen

Werte generieren

Arbitraries sind der Anfang von allem…

```
Arbitraries
  .strings()
  .integers()
  .floats()
  …
  .of(…) // values, enums
  .frequency(…) // add weights
  .constant(…)
  .oneOf(…)
  …
```

Generierte Werte verändern

- Manchmal möchte man generierte Werte filtern
- Manchmal möchte man generierte Werte abbilden
- Manchmal möchte man generierte Werte miteinander kombinieren

Werte filtern

```
@Property
boolean evenNumbersAreEven(@ForAll("evenUpTo10000") int anInt) {
    return anInt % 2 == 0;
}

@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 10000)
                .filter(i -> i % 2 == 0);
}
```

Werte abbilden

```
@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 5000)
                .map(i -> i * 2);
}
```

```
@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 10000)
                .filter(i -> i % 2 == 0);
}
```

Werte kombinieren

```
public class Person {
    public Person(String firstName, String lastName) {…}
    public String fullName() {return firstName + " " + lastName;}
}
```

```
@Provide
Arbitrary<Person> validPerson() {
    Arbitrary<Character> initialChar = Arbitraries.chars().between('A', 'Z');
    Arbitrary<String> firstName = Arbitraries.strings()… ;
    Arbitrary<String> lastName = Arbitraries.strings()… ;
    return Combinators.combine(initialChar, firstName, lastName)
            .as((initial, first, last) -> new Person(initial + first, last));
}
```

Exhaustive Value Generation

```
@Property(generation = GenerationMode.EXHAUSTIVE)
void allChessSquares(
      @ForAll @CharRange(from = 'a', to = 'h') char column,
      @ForAll @CharRange(from = '1', to = '8')char row
) {
   String square = column + "" + row;
   System.out.println(square);
}
```

Demo

- `ExhaustiveGenerationExamples`

Patterns of PBT

- Obvious Property
- Fuzzying
- Inverse functions
- Idempotent functions
- Commutativity
- Black-box testing
- Induction
- Test oracle
- Invariant properties
- Stateful Testing

Obvious Property

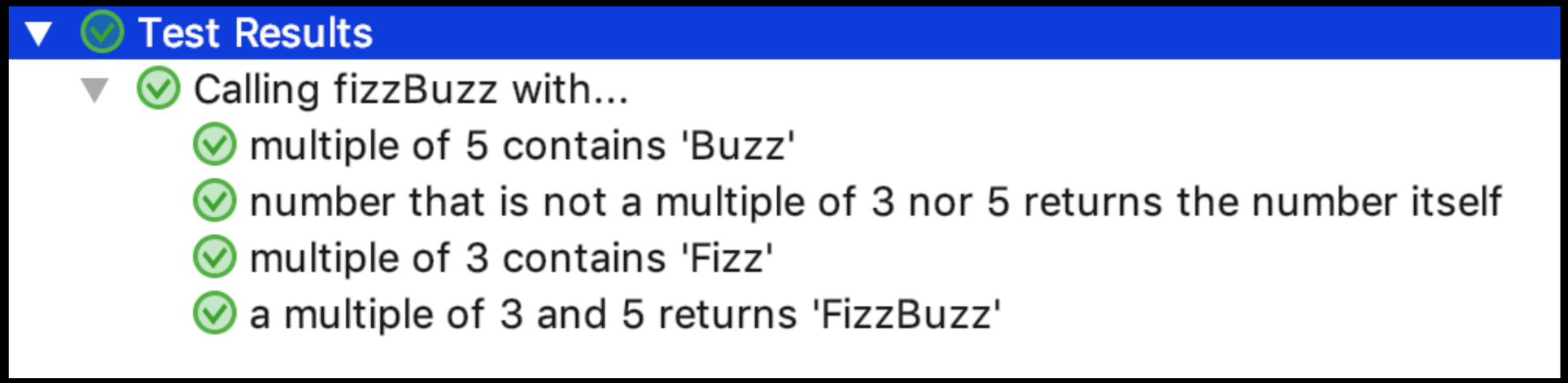
- Manchmal besteht die Spezifikation (zumindest teilweise) aus Properties
- Beispiel: Typische Business-Rule
  - *"Für alle Kunden*
    mit einem jährlichen Geschäftsvolumen > X €
    gilt ein zusätzlicher Rabatt von Y %,
    wenn die Rechnungssumme Z € übersteigt"

Fizz Buzz

- Zähle aufwärts von 1 bis 100, aber
  - Vielfache von 3 werden "Fizz" gezählt
  - Vielfache von 5 werden "Buzz" gezählt
  - Vielfache von 3 und 5 werden "FizzBuzz" gezählt

```java
@Property
@Label("multiple of 3 contains 'Fizz'")
boolean multiple3ContainsFizz(@ForAll("multipleOf3") int anInt) {
    return fizzBuzz(anInt).contains("Fizz");
}

@Provide
Arbitrary<Integer> multipleOf3() {
    return Arbitraries.integers().between(1, 33).map(i -> i * 3);
}
```



Fuzzing:
The Code Should not Explode

- Generiere alle denkbaren Arten von Inputs und teste, dass der Basis-Kontrakt einer Funktion immer erfüllt wird, z.B.:
  - keine Exceptions,
  - keine Nulls als Rückgabe
  - Rückgabe im erlaubten Wertebereich
  - Laufzeit unter einer bestimmten Grenze
- Besonders wertvoll bei Integrierten Tests

Inverse Functions

- Funktion + inverse Funktion
  ergibt die ursprüngliche Eingabe
  - Encode / Decode

```java
class InverseFunctions {

    @Property
    void encodeAndDecodeAreInverse(
            @ForAll @StringLength(min = 1, max = 20) String toEncode,
            @ForAll("charset") String charset
    ) throws UnsupportedEncodingException {
        String encoded = URLEncoder.encode(toEncode, charset);
        assertThat(URLDecoder.decode(encoded, charset)).isEqualTo(toEncode);
    }

    @Provide
    Arbitrary<String> charset() {
        Set<String> charsets = Charset.availableCharsets().keySet();
        return Arbitraries.of(charsets.toArray(new String[charsets.size()]));
    }
}
```

```
sample = ["€", "Big5"]
original-sample = ["鯒阧퇼ꞵ̡찝蚨ڬ⁑냍ᄋ؟۴虋", "x-IBM1098"]

  org.opentest4j.AssertionFailedError:
  Expecting:
   <"?">
  to be equal to:
   <"€">
  but was not.
```

Idempotent Functions

- Mehrfache Anwendung einer Funktion verändert nichts
  - Mehrfache Sortierung einer Liste
  - Duplikate aus Liste entfernen

Invariant Properties

Manche Dinge ändern sich nie...

- Die Größe einer Liste nach dem Mapping
- Der Inhalt einer Liste nach dem Sortieren

Commutativity:
Different paths, same destination

- Erst Sortieren, dann Filtern
  == Erst Filtern, dann Sortieren

Test Oracle:
Mit alternativer Implementierung verifizieren

- Einfach, aber nicht-performant
- Parallel versus Single-Threaded
- Selbst-gemacht versus kommerziell
- Alt (vor dem Refactoring) versus Neu

Black-box Testing

Hard to compute, easy to verify

- Primzahlermittlung
- Pfad durch ein Labyrinth

Induction:
Solving a smaller problem first

- Eine Liste ist sortiert, wenn
  - Das erste Element kleiner als das zweite ist
  - Alles nach dem ersten Element auch sortiert ist

```java
@Property
boolean sortingAListWorks(@ForAll List<Integer> unsorted) {
   return isSorted(sort(unsorted));
}

private boolean isSorted(List<Integer> sorted) {
   if (sorted.size() <= 1) return true;
   return sorted.get(0) <= sorted.get(1)
           && isSorted(sorted.subList(1, sorted.size()));
}
```

Stateful Testing

Bei einem zustandsbehafteten Objekt...

- Welche Aktionen sind möglich?
- Wie wird der Zustand verändert?
- Welche Invarianten gelten immer?

Lass den Computer viele zufällig gewählte Aktionen ausprobieren...

Lessons Learned

- Beispiel-basierte Tests…
  - sind oft bessere Treiber für das funktionale Verhalten
  - helfen oft besser beim Verstehen des Codes
- Interaktion mit externen Welt machen Properties langsam
- Randomisierte Tests werden leichter nicht-deterministisch
- Investiere in domänen-spezifische Generatoren!

Alternative PBT-Tools für Java

- JUnit-Quickcheck:
  Enge Integration mit JUnit 4
- QuickTheories: Arbeitet mit beliebigen Test-Libraries zusammen
- Vavr: Die funktionale Java-Bibliothek hat auch ein eigenes PBT-Modul

jqwik auf Github:

http://github.com/jlink/jqwik

Mitstreiter gesucht!

Code:

http://github.com/jlink/?

Slides:

http://johanneslink.net/downloads/PropertyTesting-ETKA-2019.pdf

Blog:

http://blog.johanneslink.net/2018/03/24/property-based-testing-in-java-introduction/

andrena
OBJECTS

Bitte geben Sie uns jetzt Ihr Feedback!

Property-based Testing in Java

Johannes Link