

Property-Based Testing mit Java

JAX, Mainz
25. April 2018

@johanneslink

johanneslink.net

Softwaretherapeut

"In Deutschland ist die Bezeichnung Therapeut allein oder ergänzt mit bestimmten Begriffen gesetzlich nicht geschützt und daher **kein Hinweis auf** ein erfolgreich abgeschlossenes Studium oder auch nur **fachliche Kompetenz.**" Quelle: Wikipedia

Examples vs Properties

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?

Examples vs Properties

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

```
@Property
void reverseList() {
    // Vorbedingung?
    // Invariante?
}
```

@Property

```
void reverseList() {  
    // Vorbedingung?  
    // Invariante?  
}
```

Vorbedingungen

- ▶ Beliebige Liste
- ▶ Nicht null

Invariante

- ▶ 2 x reverse erzeugt wieder das Original

Haskell: Quick Check

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


Java: Jqwik

```
@Property
```

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

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

Was ist jqwik?

<http://jqwik.net>

- Eine **Test-Engine** für die JUnit5–Plattform
- Ein Generator für Testfälle mit
 - ▶ **zufälligen und typischen** Eingabewerten
 - ▶ und **Kombinationen von Eingabewerten**
- Aktuelle Version: **0.8.10**

Was ist jqwik **nicht**?

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

@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 kann / möchte man eine Property nur für einen Teil aller denkbaren Ausprägungen eines Typs formulieren

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

```
timestamp = 2017-10-20T17:23:53.351,
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

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

```
timestamp = 2017-10-20T17:23:53.351,
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

```
@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().filter(aDouble -> aDouble >= 0);
}
```

```
timestamp = 2017-10-20T17:23:53.351,
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```


@Property

```
void squareOfRootIsOriginalValue(@ForAll double aNumber) {  
    Assume.that(aNumber > 0);  
  
    double sqrt = Math.sqrt(aNumber);  
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));  
}
```

timestamp = 2017-10-20T17:34:27.857,

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]]**

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

@Property

```
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]]**

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

@Property

```
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 Shrunken

- "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 Schrumpfvversuche
 - ▶ Problem: Schrumpfen kann zu Ergebnissen führen, die eigentlich bei der Generierung ausgeschlossen wurden
- Integrated Shrinking: Alle Schritte und Bedingungen der Generierung werden beim Schrumpfen berücksichtigt
- **jqwik** implementiert **integriertes Schrumpfen**

```
@Property
boolean shrinkingCanBeComplicated(
    @ForAll("first") String first,
    @ForAll("second") String second
) {
    String aString = first + second;
    return aString.length() > 5 || aString.length() < 4;
}
```

```
@Provide
Arbitrary<String> first() {
    return Arbitraries.strings('a', 'z', 1, 10)
        .filter(string -> string.endsWith("h"));
}
```

```
@Provide
Arbitrary<String> second() {
    return Arbitraries.strings('0', '9', 0, 10)
        .filter(string -> string.length() >= 1);
}
```

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

```
public interface Shrinkable<T> {  
    Set<ShrinkResult<Shrinkable<T>>> shrinkNext(Predicate<T> falsifier);  
  
    T value();  
  
    int distance();  
}
```


Werte generieren

- Manchmal möchte man die Werte selbst generieren
- Manchmal möchte man Werte abbilden
- Manchmal möchte man generierte Werte miteinander kombinieren

Vorberechnete Werte zufällig auswählen

```
@Property
boolean primesCannotBeFactored(@ForAll("primes") int aPrime) {
    return factor(aPrime).equals(Collections.singletonList(aPrime));
}

@Provide
Arbitrary<Integer> primes() {
    return Arbitraries.of(2, 3, 5, 7, 11, 13, 17, 19);
}
```

Zufällige Werte generieren

```
@Property
boolean primesCannotBeFactored(@ForAll("primes") int aPrime) {
    return factor(aPrime).equals(Collections.singletonList(aPrime));
}

@Provide
Arbitrary<Integer> primes() {
    return Arbitraries.randomValue(random -> generatePrime(random));
}

private Integer generatePrime(Random random) {
    int candidate;
    do {
        candidate = random.nextInt(10000) + 2;
    } while (!isPrime(candidate));
    return candidate;
}
```

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;  
    }  
}
```

@Property

```
boolean anyValidPersonHasAFullName(@ForAll Person aPerson) {  
    return aPerson.fullName().length() > 0;  
}
```

@Provide

```
Arbitrary<Person> validPerson() {  
    Arbitrary<String> firstName = Generator.strings('a', 'z', 10);  
    Arbitrary<String> lastName = Generator.strings('a', 'z', 20);  
    return Combinators.combine(firstName, lastName)  
        .as((first, last) -> new Person(first, last));  
}
```

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 $> X \text{ €}$ gilt ein zusätzlicher Rabatt von $Y \%$, wenn die Rechnungssumme $Z \text{ €}$ übersteigt"*

Fizz Buzz

- Zähle aufwärts von 1 bis 100
- "Normale" Zahlen werden normal gezählt
- Vielfache von 3 werden "Fizz" gezählt
- Vielfache von 5 werden "Buzz" gezählt
- Vielfache von 3 und 5 werden "FizzBuzz" gezählt

@Property

```
boolean divisibleBy3ContainsFizz(@ForAll("divisibleBy3") int anInt) {  
    return fizzBuzz(anInt).contains("Fizz");  
}
```

@Provide

```
Arbitrary<Integer> divisibleBy3() {  
    return Arbitraries.integers().between(1, 33).map(i -> i * 3);  
}
```

@Property

```
boolean divisibleBy5ContainsBuzz(...) { ... }
```

@Property

```
boolean divisibleBy3and5ReturnFizzBuzz(...) { ... }
```

@Property

```
boolean indivisiblesReturnThemselves(...) { ... }
```

Fuzzying:

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

```

class InverseFunctions {
    @Property
    void encodeAndDecodeAreInverse(
        @ForAll @StringLength(min = 1, max = 20) String toEncode,
        @ForAll("charset") String charset
    ) throws UnsupportedOperationException {
        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()]));
    }
}

```

```

originalSample = ["u0xX-ZD9:t8", "x-MacDingbat"],
sample         = [":", "x-MacDingbat"]

```

```

java.lang.AssertionError:
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


```
class Commutativity {

    @Property
    void sortingAndFilteringAreCommutative(@ForAll("names") List<String> listOfNames) {
        List<String> filteredThenSorted = listOfNames.stream()
            .filter(name -> !name.contains("a"))
            .sorted()
            .collect(Collectors.toList());

        List<String> sortedThenFiltered = listOfNames.stream()
            .sorted()
            .filter(name -> !name.contains("a"))
            .collect(Collectors.toList());

        Assertions.assertThat(filteredThenSorted).isEqualTo(sortedThenFiltered);
    }

    @Provide
    Arbitrary<List<String>> names() {
        Arbitrary<String> name = Arbitraries.strings() //
            .withCharRange('a', 'z') //
            .ofMinLength(3).ofMaxLength(40);
        return name.list().ofMinSize(0).ofMaxSize(30);
    }
}
```

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

```
@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...

```
public class MyStringStack {  
    public void push(String element) {...}  
    public String pop() {...}  
    public void clear() {...}  
    public boolean isEmpty() {...}  
    public int size() {...}  
    public String top() {...}  
}
```

```
public interface Action<M> {  
    default boolean precondition(M model) {return true;}  
    M run(M model);  
}
```

```
class PopAction implements Action<MyStringStack> {  
    @Override  
    public boolean precondition(MyStringStack model) {  
        return !model.isEmpty();  
    }  
    @Override  
    public MyStringStack run(MyStringStack model) {  
        int sizeBefore = model.size();  
        String topBefore = model.top();  
  
        String popped = model.pop();  
        Assertions.assertThat(popped).isEqualTo(topBefore);  
        Assertions.assertThat(model.size()).isEqualTo(sizeBefore - 1);  
        return model;  
    }  
}
```



```
static Arbitrary<Action<MyStringStack>> actions() {  
    return Arbitraries.oneOf(push(), clear(), pop());  
}  
private static Arbitrary<Action<MyStringStack>> push() {  
    return Arbitraries.strings().alpha().ofLength(5).map(PushAction::new);  
}  
private static Arbitrary<Action<MyStringStack>> clear() {...}  
private static Arbitrary<Action<MyStringStack>> pop() {...}
```

```
class MyStackProperties {  
  
    @Property  
    void checkMyStackMachine(@ForAll ActionSequence<MyStringStack> sequence) {  
        sequence.run(new MyStringStack());  
    }  
  
    @Provide  
    Arbitrary<ActionSequence<MyStringStack>> sequences() {  
        return Arbitraries.sequences(MyStringStackActions.actions());  
    }  
}
```

Benötigen wir Example-Based-Testing noch?

"[...] TDD helps design code to have it
conform to expectations and demands,
[...] *property-based testing forces the*
exploration of the program's behaviour
to see what it can or cannot do."

aus <http://propertesting.com/>

Lessons Learned beim PBT

- Manchmal sind konkrete **Beispiele hilfreicher** beim Code-Verstehen
- Interaktionen mit der Außenwelt machen PBT **langsam**
- Randomisierte Tests können **nicht-deterministisch** sein

Die Zukunft von jqwik

- Mehr Default-Providers
 - ▶ z.B. für `java.time.*`
- Vollständige statt zufällige Generierung
- Groovy-DSL

Vollständige Generierung

```
@Property(generation = GenerationMode.EXHAUSTIVE)
boolean allCallsToMethodWork(
    @ForAll boolean aBool,
    @ForAll MyEnum anEnum,
    @ForAll @IntRange(min = 0, max = 100) int anInt)
{
    return ...
}
```

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/property-based-testing>

Slides:

[http://johanneslink.net/downloads/
PropertyTesting-Jax2018.pdf](http://johanneslink.net/downloads/PropertyTesting-Jax2018.pdf)