

# Property-Based Testing mit Java

Softwerkskammer Karlsruhe  
2. Mai 2019

**@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

# Zeitplan

19:00 - 20:00: Theorie

20:00 - 20:15: Pause

20:15 - 20:45: Praxis

20:45 - ? : Diskussion

# 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?
```

## Vorbedingungen

- ▶ Beliebige Liste
- ▶ Nicht null

## Invarianten

- ▶ Länge der Liste bleibt gleich
- ▶ Alle Elemente bleiben erhalten
- ▶ Nach reverse ist das erste Element 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();  
}
```

@Property

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

@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);  
}
```

@Property

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

```
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: **1.1.3**

# 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**)



```
@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

```
@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
```

```
@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
```

@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: A **factory of factories** for arbitrary values

```
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);  
}
```

```
@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]]**



```
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 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 Schrumpfungversuche
  - ▶ 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**

# Werte generieren

**Arbitraries** sind der Anfang von allem...

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

# Generatoren Konfigurieren

## Fluent Interfaces

**Arbitraries** können konfiguriert werden

```
@Provide
StringArbitrary fluentString() {
    return Arbitraries.strings()
        .alpha()
        .numeric()
        .withChars('?', '!', '.')
        .ofMinLength(2)
        .ofMaxLength(10);
}
```

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

```
@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);
}
```

## ▼ ✓ Test Results

- ▼ ✓ Calling fizzBuzz with...
  - ✓ multiple of 5 contains 'Buzz'
  - ✓ number that is not a multiple of 3 nor 5 returns the number itself
  - ✓ multiple of 3 contains 'Fizz'
  - ✓ a multiple of 3 and 5 returns 'FizzBuzz'

# 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



# 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());  
    }  
}
```

# Demo

- *MyStackProperties*

# Lessons Learned beim PBT

- Beispiel-basierte Tests sind oft nützlicher, um **Verhalten des Codes zu verstehen**
- Interaktionen mit der Außenwelt machen PBT **langsam**
- Randomisierte Tests können **nicht-deterministisch** sein
- Investiere in **domänen-spezifische** Generatoren!



# Die Zukunft von jqwik

- Mehr Default-Providers
  - ▶ z.B. für `Map`, `java.time.*`, Funktionen, Tuples
- Groovy /Kotlin DSL
- Mehr Unterstützung für **Contract Testing**

# 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/pbt-workshop>

# Slides:

<http://johanneslink.net/downloads/PropertyTesting-SWKKA.pdf>

# Blog:

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

# Ein bisschen Praxis

Bitte kclone dieses Git-Repository:

<https://github.com/jlink/pbt-workshop>



# Vorbereitung

## WLAN: gwn!1539

### 1. Klone das Repo:

```
git clone https://github.com/jlink/pbt-workshop
```

### 2. Importiere das Repo in deine IDE

**Intelli J:** New → Project from Existing Sources... : build.gradle

**Eclipse:** Import → Gradle → Existing Gradle Project

### 3. Starte alle Tests im Package **pbt.examples.reverse** "10 of 10 tests passed"

jqwik's user guide:

<https://jqwik.net/user-guide.html>

# Übung:

`collections.sort(List aList)`

1. Sammelt alle Properties mit Preconditions, Postconditions und Invarianten
2. Implementiert die Properties in Klasse `pbt.exercise1.SortingProperties`

# Übung:

## `Collections.sort(List aList)`

1. Brainstorming: Properties mit Preconditions, Postconditions und Invarianten
2. Implementiert die Properties in Klasse `pbt.ex1.SortingProperties`



# Properties von `Collections.sort(List aList)`

- Zahl der Elemente bleibt gleich
- Alle Elemente bleiben erhalten
- Es kommen keine neuen Elemente hinzu
- Mehrmaliges Sortieren verändert das Ergebnis nicht
- Überprüfe „Sortiertheit“ durch **Vollständige Induktion**:
  1. If list size  $< 2 \rightarrow$  true
  2. If first element  $\leq$  second element  
then  $\rightarrow$  check tail of list  
else  $\rightarrow$  false

# Übung:

## `Collections.sort(List aList)`

1. Brainstorming: Properties mit Preconditions, Postconditions und Invarianten
2. Implementiert die Properties in Klasse `pbt.ex1.SortingProperties`