

# Property-Based Testing mit Java

Accento Karlsruhe  
24. September 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?
```

## Vorbedingungen

- ▶ Beliebige Liste - aber 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);  
}
```

# 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: **1.2.0**

# 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:

## "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);  
}
```

```
@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

# 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));  
}
```

# How to Specify it

## Patterns and Strategies of PBT

- Fuzzing
- Postconditions
- Metamorphic Properties
- Black-box Testing
- Inductive Testing
- Test Oracle
- Stateful Testing
- Model-based Properties

# Fuzzing:

## Your 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

# Postconditions

- Häufig können wir Nachbedingung für die Ausführung von Operationen benennen
- Beispiel: Einfügen in Binary Search Tree
  - ▶ *"Nach dem Einfügen von Key X mit Value Y in irgendeinen beliebigen Binärbaum, sollten wir bei Suche nach X den Wert Y finden"*

```
@Property
boolean inserted_value_can_be_found(
    @ForAll Integer key, @ForAll Integer value,
    @ForAll("trees") BST<Integer, Integer> bst
) {
    Optional<Integer> found = bst.insert(key, value).find(key);
    return found.equals(Optional.of(value));
}
```



# Invarianten sind auch Nachbedingungen...

Manche Dinge ändern sich nie...

- ▶ Die Größe einer Liste nach dem Mapping
- ▶ Die Inhalte einer Liste nach dem Sortieren
- ▶ Die gesamte Geldmenge nach einer Überweisung

# Metamorphic Properties

"... even if the expected result of a function call [...] may be difficult to predict, we may still be able to express an expected relationship between this result, and the result of a related call."

John Hughes in *How to Specify It*

<https://johanneslink.net/how-to-specify-it>

# Metamorphic Property: 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(URLEncoder.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 = ["鯛斗뵙뵙,짚蚨뵙뵙뵙뵙뵙뵙뵙", "x-IBM1098"]

```

```

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

```

# Häufige "Metamorphic Properties"

- **Idempotenz**
  - ▶ Mehrfache Anwendung einer Operation verändert nichts
- **Kommutativität**
  - ▶ Veränderung der Reihenfolge von Operationen ist egal
- **Symmetrie**
  - ▶ Veränderung von Ausgangsdaten verändert das Ergebnis immer in gleicher Weise
- **Robustheit**
  - ▶ Einfügen von "Rauschen" oder Fehler in Datenströme lässt die Verarbeitungsergebnisse unverändert

# Black-box Testing

Hard to compute, easy to verify

- ▶ Primzahlermittlung
- ▶ Pfad durch ein Labyrinth

# Inductive Testing:

## Solving a smaller problem first

- Manchmal können wir die fachliche Spezifikation durch eine **Menge von Properties** vollständig beschreiben
- Beispiel: 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()));  
}
```



# **Test Oracle:** Verifizieren ein Ergebnis gegen alternative Implementierung

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

# Stateful Testing

Bei einem zustandsbehafteten Objekt...

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

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());  
    }  
}
```

# Model-based Properties

- Wir benutzen ein **vereinfachtes Modell** der zu testenden Logik, und vergleichen das Verhalten des Modells mit dem Verhalten unserer Software
  - ▶ z.B. Benutze Key-Value-Map als Modell für Datenpersistenz
- Stateful und model-based werden häufig kombiniert

# Lessons Learned

- Beispiel-basierte Tests...
  - ▶ sind oft bessere Treiber für **das funktionale Verhalten**
  - ▶ helfen beim **Verstehen der Fachlichkeit**
- Interaktion mit der externen Welt  
machen Properties **langsam**
- Randomisierte Tests werden häufiger  
**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



# Code:

<http://github.com/jlink/property-based-testing>

# Slides:

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

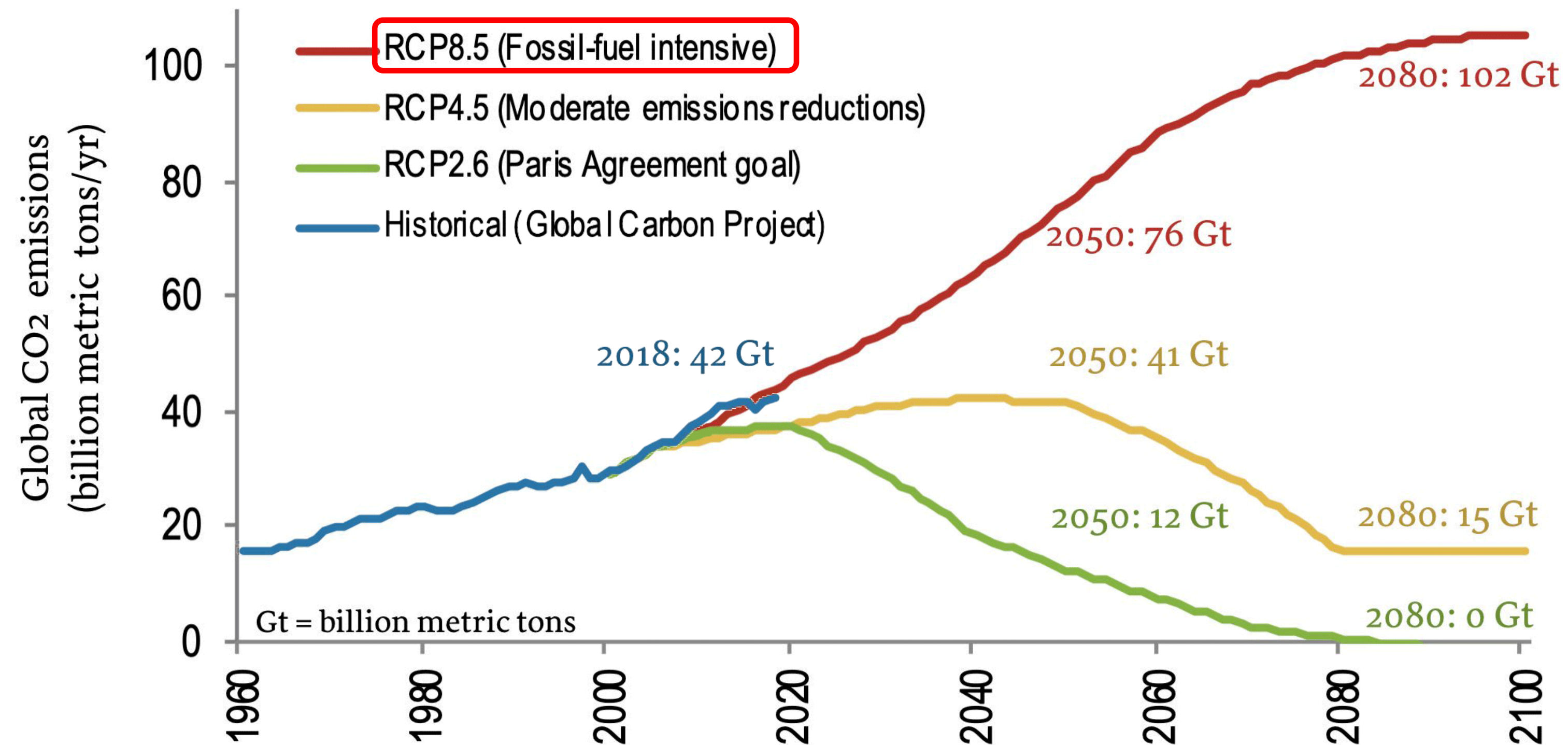
# Blog Series:

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

Das alles ist aber völlig ohne  
Bedeutung...

... wenn es morgen  
**unsere Welt** in dieser  
Form **nicht mehr gibt**

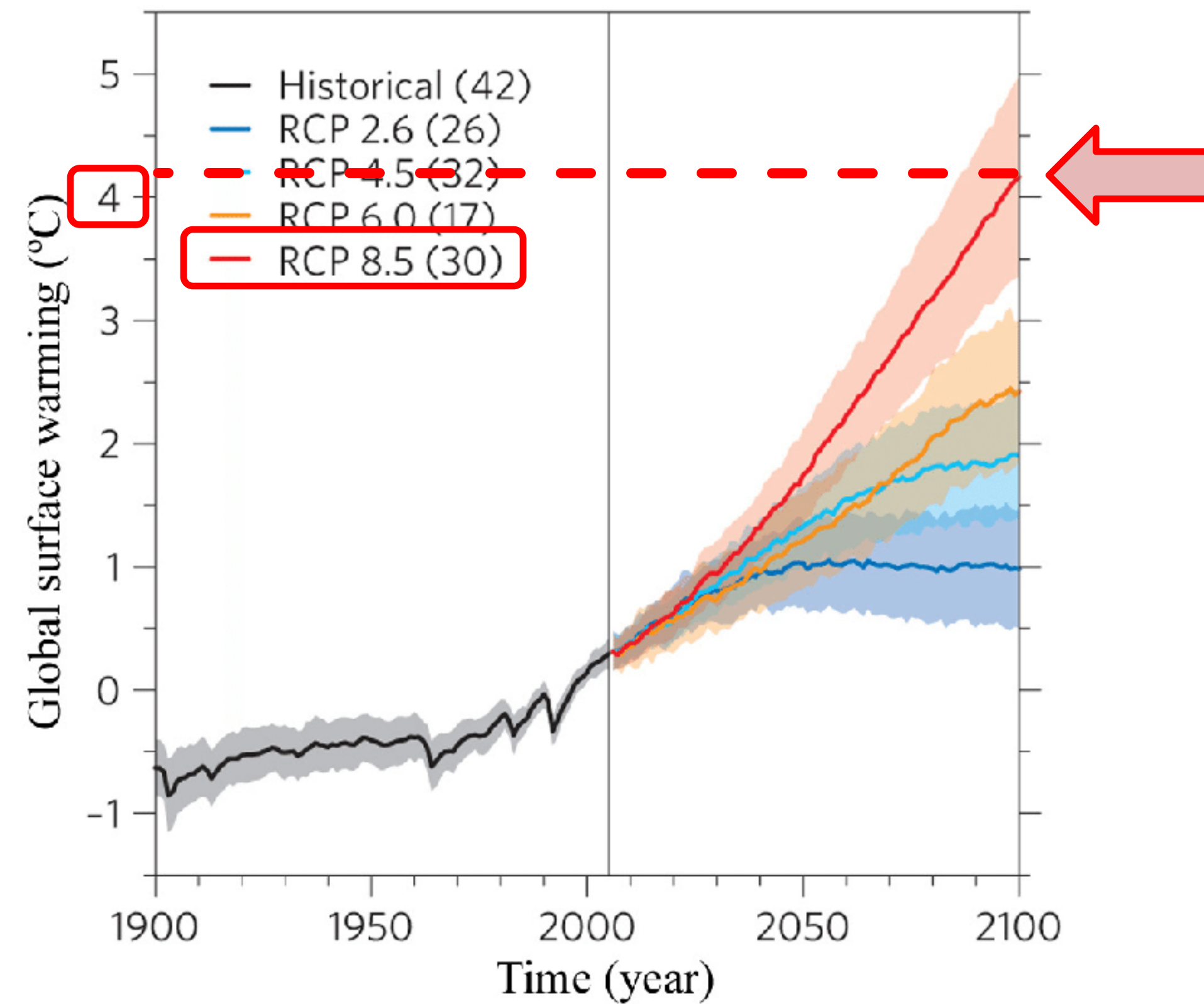
# CO<sub>2</sub> Emissionen



(R. Kopp, Rutgers Climate Institute; aktualisiert von R. Kopp nach Kopp et al. 2014)

Quellen und weitere Details: [www.ClimateFactsNow.org](http://www.ClimateFactsNow.org)

# Temperatur



Knutti and Sedláček (2013)

Quellen und weitere Details: [www.ClimateFactsNow.org](http://www.ClimateFactsNow.org)

# Wir sind auf dem Weg zum Worst-Case-Szenario!

Weniger als 2 Grad Erderwärmung sind  
vermutlich schon nicht mehr machbar.

**4 Grad noch in diesem Jahrhundert** sind  
wahrscheinlich.

- Massive Hitzewellen und Trockenzeiten
- Intensive Überflutungen
- Zerstörung großer Ökosysteme (Korallenriffe, Feuchtgebiete, Wälder)
- Weltweite Ernteeinbrüche
- Öffentliche Unruhen und politische Instabilisierung
- Kriege um Wasser, Land und Nahrungsmittel
- Bis 2050 ca. 400 Mio Flüchtlinge
- und und und und

Unsere Intuitionen aus der Softwareentwicklung taugen nicht bei der Eingrenzung der Klimakrise!

- Klimakrise Pro/Contra  
**ist nicht Emacs vs VIM**
- Klimapanik **ist keine willkürliche Deadline eines Projektleiters**
- **Technologie allein** wird das Problem nicht lösen
- Der "gesunde Mittelweg" hilft nicht, wenn ein Extrempunkt **Überleben** heißt



**Was können wir tun?**



# Was **müssen** wir tun?

- Die Fakten und wissenschaftlichen Erkenntnisse immer wieder aussprechen
- Die Politik(er) unter Druck setzen
  - ▶ Wählen
  - ▶ Briefe, Anrufe, Emails, Petitionen
  - ▶ Demos, Streiks, Blockaden

ScientistsForFuture

FridaysForFuture

ParentsForFuture

EXtinctionRbellion