

Property-Based Testing

Using jqwik for Java

André Restivo / Hugo Sereno

Index

Testing 101

Property Based Testing

jqwik

Testing 101

The Novice Programmer

Everyone likes to reimplement stuff from scratch; in that spirit, let us code our own sum function:

```
public int mySum(int a, int b) {  
    int accumulator = a;  
    while(b > 0) {  
        a++;  
        b--;  
    }  
    return accumulator;  
}
```

How should one proceed to test it? Many will write something like this:

```
@Test  
public int mySumTest() {  
    assertEquals(2, sum(1, 1));  
    assertEquals(4, sum(2, 2));  
    assertEquals(8, sum(4, 4));  
}
```

Everything is awesome! All tests are passing!...

The Lurking Bug

There's indeed a bug in the implementation. Look at the code very carefully:

```
public int mySum(int a, int b) {  
    int accumulator = a;  
    while(b > 0) {  
        a++;  
        b--;  
    }  
    return accumulator;  
}
```

What happens when you try something like `mySum(2, -3)`?

Expected `-1`; got `2`.

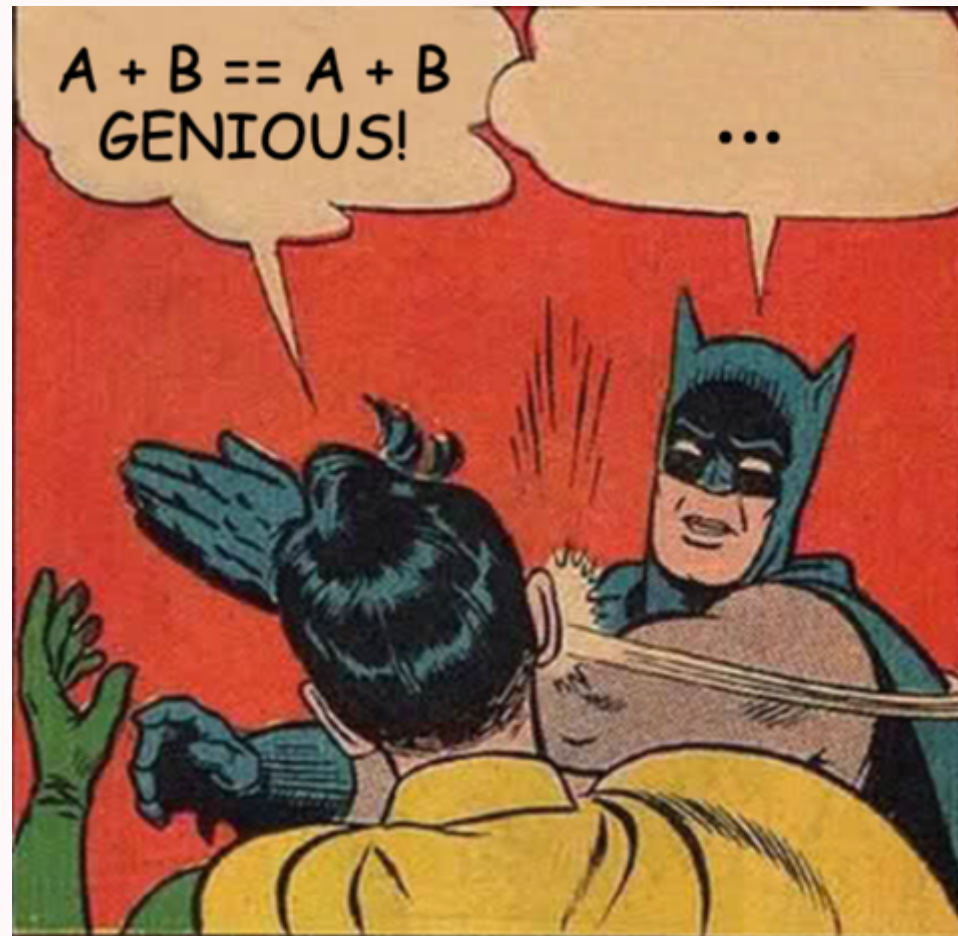
The Intermediate Programmer

But that is stupid! Why don't we use the + operator? — an older student.

```
public int mySum(int a, int b) {  
    return a + b;  
}
```

... and proceed to test for a range! — the same older student.

```
@Test  
public int mySumTest() {  
    for (int a = 0; a < 1000; a++)  
        for (int b = 0; b < 1000; b++)  
            assertEquals(a + b, sum(a, b));  
}
```



The problem of testing

- ... we test what we know. Because if we knew what we didn't know, we would do it right.
- ... so how can we test what we don't know?

Property-Based Testing

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing **a** with 0 (or to 0), is the same as doing nothing:

$$a + 0 = 0 + a = a$$

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing **a** with 0 (or to 0), is the same as doing nothing:

$$a + 0 = 0 + a = a$$

- Summing **a** with **b** is the same as summing **b** with **a**:

$$a + b = b + a$$

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing a with 0 (or to 0), is the same as doing nothing:

$$a + 0 = 0 + a = a$$

- Summing a with b is the same as summing b with a :

$$a + b = b + a$$

- Summing a with b is the same as summing $a-1$ with $b+1$:

$$a + b = (a-1) + (b+1)$$

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing a with 0 (or to 0), is the same as doing nothing:

$$a + 0 = 0 + a = a$$

- Summing a with b is the same as summing b with a :

$$a + b = b + a$$

- Summing a with b is the same as summing $a-1$ with $b+1$:

$$a + b = (a-1) + (b+1)$$

- Summing a with b and then with c is the same as:

$$a + (b + c) = (a + b) + c$$

Challenge

Suppose you don't have access to the `+` operator. How can we implement a test that uses the above properties to verify if our sum is working?

So what is PBT?

- ... usage of Arbitraries;
- ... usage of Statistics to cover the search space and provide confidence;
- ... usage of Properties to specify the external behavior of our system and search for counter-examples;

So nice things:

- ... Reproducibility (via paths and seeds);
- ... Shrinking (smallest cases that reproduce the bug).

Arbitraries

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x , where x is a natural number*
- In code, we can think of `x` an object of `Arbitrary`;

Arbitraries

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x , where x is a natural number*
- In code, we can think of x an object of **Arbitrary**;
- Most frameworks provide us **Arbitrary** for several **common cases**:
 - Numbers (**Integers**, **Double**...)
 - Booleans
 - Collections (**ArrayList**, **HashSet**...)

Arbitraries

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x , where x is a natural number*
- In code, we can think of x an object of **Arbitrary**;
- Most frameworks provide us **Arbitrary** for several **common cases**:
 - Numbers (**Integers**, **Double**...)
 - Booleans
 - Collections (**ArrayList**, **HashSet**...)
- You can also define your own Arbitraries, either by:
 - **Mapping built-in** Arbitraries;
 - Creating them from **scratch**.

Arbitraries

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x , where x is a natural number*
- In code, we can think of x an object of **Arbitrary**;
- Most frameworks provide us **Arbitrary** for several **common cases**:
 - Numbers (**Integers**, **Double**...)
 - Booleans
 - Collections (**ArrayList**, **HashSet**...)
- You can also define your own Arbitraries, either by:
 - **Mapping built-in** Arbitraries;
 - Creating them from **scratch**.
- There are things that make an Arbitrary more useful than being **merely a random generator**, which is (next slide)...

... Statistics

- The **small-scope hypothesis** claims that **most inconsistent models have counterexamples within *small bounds***;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...

... Statistics

- The **small-scope hypothesis** claims that **most inconsistent models have counterexamples within *small bounds***;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - -128, 129, 256, 32769, -32768, 65536... why?

... Statistics

- The **small-scope hypothesis** claims that **most inconsistent models have counterexamples within *small bounds***;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - -128, 129, 256, 32769, -32768, 65536... why?
 - And **Infinity** and **-Infinity**...
- So, while you would need to **cover all the space** to gain **perfect knowledge**, in practice, a **small number** of instances of certain Arbitraries are responsible for most bugs;
- Can you think of **ArrayList**'s that usually triggers buggy code?

... Statistics

- The **small-scope hypothesis** claims that **most inconsistent models have counterexamples within *small bounds***;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - -128, 129, 256, 32769, -32768, 65536... why?
 - And **Infinity** and **-Infinity**...
- So, while you would need to **cover all the space** to gain **perfect knowledge**, in practice, a **small number** of instances of certain Arbitraries are responsible for most bugs;
- Can you think of **ArrayList**'s that usually triggers buggy code?
 - The **empty list**!

... Statistics

- The **small-scope hypothesis** claims that **most inconsistent models have counterexamples within *small bounds***;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - -128, 129, 256, 32769, -32768, 65536... why?
 - And **Infinity** and **-Infinity**...
- So, while you would need to **cover all the space** to gain **perfect knowledge**, in practice, a **small number** of instances of certain Arbitraries are responsible for most bugs;
- Can you think of **ArrayList**'s that usually triggers buggy code?
 - The **empty list**!
- PBT frameworks call this **biased** search, and considers it for you;
- It is up to the Arbitrary to define their **bias**.

Shrinking

Imagine testing if your *hero* can walk out of the *arena*.

If we do a **random search**, we might end up with a counter example that is similar do this:

```
[UP, DOWN, RIGHT, LEFT, LEFT, UP, LEFT, UP, DOWN, DOWN, RIGHT, LEFT, UP, LEFT, UP,  
DOWN, RIGHT, LEFT, RIGHT, UP, LEFT, UP, DOWN, DOWN, LEFT, LEFT, UP, LEFT, UP, DOWN,  
LEFT, LEFT, LEFT, UP, LEFT, UP, DOWN, DOWN, RIGHT, LEFT, UP, LEFT, UP, DOWN, LEFT,  
LEFT, LEFT, UP, LEFT, UP, DOWN, DOWN, LEFT, LEFT, UP, LEFT, UP, DOWN, LEFT, LEFT,  
LEFT, UP, LEFT, UP, DOWN, DOWN, RIGHT, LEFT, UP, LEFT, UP, DOWN, LEFT, LEFT, LEFT,  
UP, LEFT, UP, DOWN, DOWN, RIGHT, LEFT, UP, LEFT, UP, DOWN, LEFT, LEFT, LEFT, UP]
```

Can we do **better** (what is better?)

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've **found something** that produces a **counter-example**, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've **found something** that produces a **counter-example**, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.
- How can we **shrink a number**? What about **decreasing its magnitude**? Does $-5234153245/2 = -2617076622$ also produces a counter example? **Yes**.

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've **found something** that produces a **counter-example**, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.
- How can we **shrink a number**? What about **decreasing its magnitude**? Does $-5234153245/2 = -2617076622$ also produces a counter example? **Yes**.
- If you apply **shrinking strategies recursively**, you'll eventually reach -1 . Which is a much nicer counter-example to deal with :)

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've **found something** that produces a **counter-example**, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.
- How can we **shrink a number**? What about **decreasing its magnitude**? Does $-5234153245/2 = -2617076622$ also produces a counter example? **Yes**.
- If you apply **shrinking strategies recursively**, you'll eventually reach -1 . Which is a much nicer counter-example to deal with :)

Can you think of **strategies** to shrink an **ArrayList**?

jqwik

jqwik

The main purpose of **jqwik** is to bring **Property-Based Testing (PBT)** to the **JVM**.

A **property** is supposed to describe a **generic invariant** or **post-condition** of your code, given some **precondition**.

jqwik will then try to **generate** many **value sets** that fulfill the precondition hoping that one of the generated sets can **falsify** a **wrong assumption**.

[jqwik documentation](#)

Gradle

To use **jqwik**, you just need to add the following to your build.gradle:

```
test {
    useJUnitPlatform {
        includeEngines ('junit-jupiter', 'jqwik')
    }
}

dependencies {
    testCompile group: 'org.junit.jupiter', name: 'junit-jupiter', version: '5.6.2'
    testCompile group: 'net.jqwik', name: 'jqwik', version: '1.2.7'
}
```

Notice that we need to use JUnit5 instead of JUnit4.

Properties

To define a **property**, we just need to add the `@Property` annotation to our test:

```
class TestNumbers {  
    @Property  
    public void testSumAssociativity(@ForAll int a, @ForAll int b, @ForAll int c) {  
        assert((a + b) + c == a + (b + c));  
    }  
}
```

Notice that our test receives **three parameters** (a, b, and c) and that we are saying that the property should **hold for all** possible values using the `@ForAll` annotation.

Parameter Generation

jqwik is capable of generating parameters for a **wide range of types**: Strings, all kinds of numerical types, booleans, characters, Lists, Sets, Streams, Arrays, ...

Here we are testing if reversing any integer list twice results in the same list:

```
@Property
public void testDoubleReverse(@ForAll List<Integer> list) {
    assert(reverseList(reverseList(list)).equals(list));
}

public <T> List<T> reverseList(List<T> list) {
    ArrayList<T> reversed = new ArrayList<>();

    for (T e : list) reversed.add(0, e);

    return reversed;
}
```

Constraining Parameters

Sometimes we want to constrain the generated parameters. For example, the following test:

```
@Property
public void testDivision(@ForAll int number) {
    assertEquals(1, number / number);
}
```

Does not work if the number is zero, so we can use the `@Positive` annotation to constrain the number:

```
@Property
public void testDivision(@ForAll @Positive int number) {
    assertEquals(1, number / number);
}
```

Maybe this should also work for negative numbers!

Constraining Parameters

There are several types of constraints that can be applied:

- `@WithNull(value = 0.1)` If we want to generate null values, value is the percentage of null values to generate.
- `@Unique` Prevents repeated values in a list.
- `@StringLength(value = 0, min = 0, max = 0)` A fixed size string or between min and max characters.
- `@Chars(value = {})`, `@CharRange(from = 0, to = 0)`, `@NumericChars`, `@LowerChars`, `@UpperChars`, `@AlphaChars`, `@WhiteSpace` Several ways to constraint character generation.
- `@NotEmpty`, `@Size(value = 0, min = 0, max = 0)` To constraint the size of generated lists.
- `@Positive`, `@Negative`, `@IntRange(min = 0, max)`, `@DoubleRange(min = 0.0, max)`, ... To constraint generated numbers.

Constraining Parameterized Types

If we want to constrain the generation of **contained parameter types** we can annotate the parameter type **directly**:

```
@Property
public void testListSumPositive(@ForAll @NotEmpty List<@Positive Integer> list) {
    assert(sum(list) > 0);
}

private int sum(List<Integer> list) {
    int sum = 0;
    for (int e : list) sum += e;
    return sum;
}
```

This property does not hold, why?

Arbitrary

If the **default generators** are **not enough**, we can use the `@Provide` annotation and `Arbitrary` class to create **new generators**:

Fixing the `testDivision` test:

```
@Property
public void testDivision(@ForAll("notZero") int number) {
    assertEquals(1, number / number);
}

@Provide
Arbitrary<Integer> notZero() {
    return Arbitraries.integers().filter(n -> n != 0);
}
```

With **arbitraries** we can generate `integers()`, `strings()`, ... which we can then restrict using functions like `filter(f)`, `map(f)`, `greaterOrEqual(n)`, `alpha()`, `numeric()`, `ofLength(n)`, ...

Combining Arbitraries

We can even combine *arbitraries* using the `combine` and `as` methods, sprinkled with some Java lambda magic:

```
@Property void testWithPlates(@ForAll("carPlates") String plate) {  
    // do some testing using plate  
}  
  
@Provide  
Arbitrary<String> carPlates() {  
    return Combinators.combine(  
        Arbitraries.strings().alpha().ofLength(2),  
        Arbitraries.strings().numeric().ofLength(2),  
        Arbitraries.strings().numeric().ofLength(2)  
    ).as((s1, s2, s3) -> s1.toUpperCase() + "-" + s2 + "-" + s3);  
}
```

Another Arbitrary Example

A prime number cannot be divided by any(?) other number:

```
@Provide
Arbitrary<Integer> primeNumbers() {
    return Arbitraries.integers().greaterOrEqual(2).filter(n -> isPrime(n));
}

private boolean isPrime(Integer n) {
    for(int i=2; i<=Math.sqrt(n); i++)
        if(n%i==0) return false;
    return true;
}

@Property void testWithPrimes(
    @ForAll @IntRange(min = 2) int number,
    @ForAll("primeNumbers") int prime) {
    assert(prime == number || prime % number != 0);
}
```

Output

The result of a test in jqwik looks something like these:

```
tries = 1000          | -----jqwik-----  
checks = 1000         | # of calls to property  
generation-mode = RANDOMIZED | parameters are randomly generated  
after-failure = PREVIOUS_SEED | use the previous seed  
seed = 529692752344469023 | random seed to reproduce generated values
```

In this report we can see the number of test runs for this property (**tries**), number of calls that were not rejected (**checks**), how values were generated (**generation-mode**), if we should keep using the same seed if a property check fails (**after-failure**), and which seed was used (**seed**).

Configuring Runs

We can change some configuration parameters for each test:

```
@Property(tries = 2000,  
          seed="259083988309207343",  
          afterFailure = AfterFailureMode.RANDOM_SEED)  
public void testDoubleReverse(@ForAll List<Integer> list) {  
    assert(reverseList(reverseList(list)).equals(list));  
}
```

Shrinking

The advantage of using arbitraries instead of just using random data generators, is that arbitraries know how to **shrink**:

```
@Property
public void testDifferenceAssociativity(
    @ForAll int a,
    @ForAll int b,
    @ForAll int c) {
    assert((a - b) - c == a - (b - c));
}
```

tries = 1	-----jqwik-----
checks = 1	# of calls to property
generation-mode = RANDOMIZED	# of not rejected calls
after-failure = PREVIOUS_SEED	parameters are randomly generated
seed = -1077203421743176744	use the previous seed
sample = [0, 0, -1]	random seed to reproduce generated values
original-sample = [-304, -133, -84]	

This allows us to find smaller examples that are easier to understand.

An Hero example...

Testing if the arena bounds are correctly checked:

```
@Property
public void testArenaBounds(@ForAll @IntRange(min = 1, max = 100) int width,
                             @ForAll @IntRange(min = 1, max = 100) int height,
                             @ForAll int x,
                             @ForAll int y) {
    Arena arena = new Arena(width, height, null);

    assert(x >= 0 || !arena.isInBounds(new Position(x, y)));
    assert(y >= 0 || !arena.isInBounds(new Position(x, y)));
    assert(x < arena.getWidth() || !arena.isInBounds(new Position(x, y)));
    assert(y < arena.getHeight() || !arena.isInBounds(new Position(x, y)));
}
```

...or two!

Testing if the hero never leaves the arena:

```
@Property
public void testMovingBounds(@ForAll List<ArenaView.ACTION> actions)
    throws IOException {

    Hero hero = new Hero(new Position(2, 2));
    Arena arena = new Arena(5, 5, hero);
    ArenaViewMock view = new ArenaViewMock(actions);
    GameController controller = new GameController(arena, view);

    while (view.hasMoreActions()) {
        controller.step();

        assert (hero.getPosition().getX() >= 0);
        assert (hero.getPosition().getY() >= 0);

        assert (hero.getPosition().getX() < arena.getWidth());
        assert (hero.getPosition().getY() < arena.getHeight());
    }
}
```

Mocking for PBT

In this second example we had to create a specialized **mock** for the `ArenaView` class:

```
public class ArenaViewMock extends ArenaView {
    private List<ACTION> actions;

    public ArenaViewMock(List<ACTION> actions)
        throws IOException {
        super(null, null);
        this.actions = actions;
    }

    @Override
    public ACTION getAction() throws IOException {
        ACTION action = actions.get(0);
        actions.remove(0);
        return action;
    }

    @Override
    public void draw() throws IOException { // Do nothing }

    public boolean hasMoreActions() { return actions.size() > 0; }
}
```