Integrazione e Test di Sistemi Software

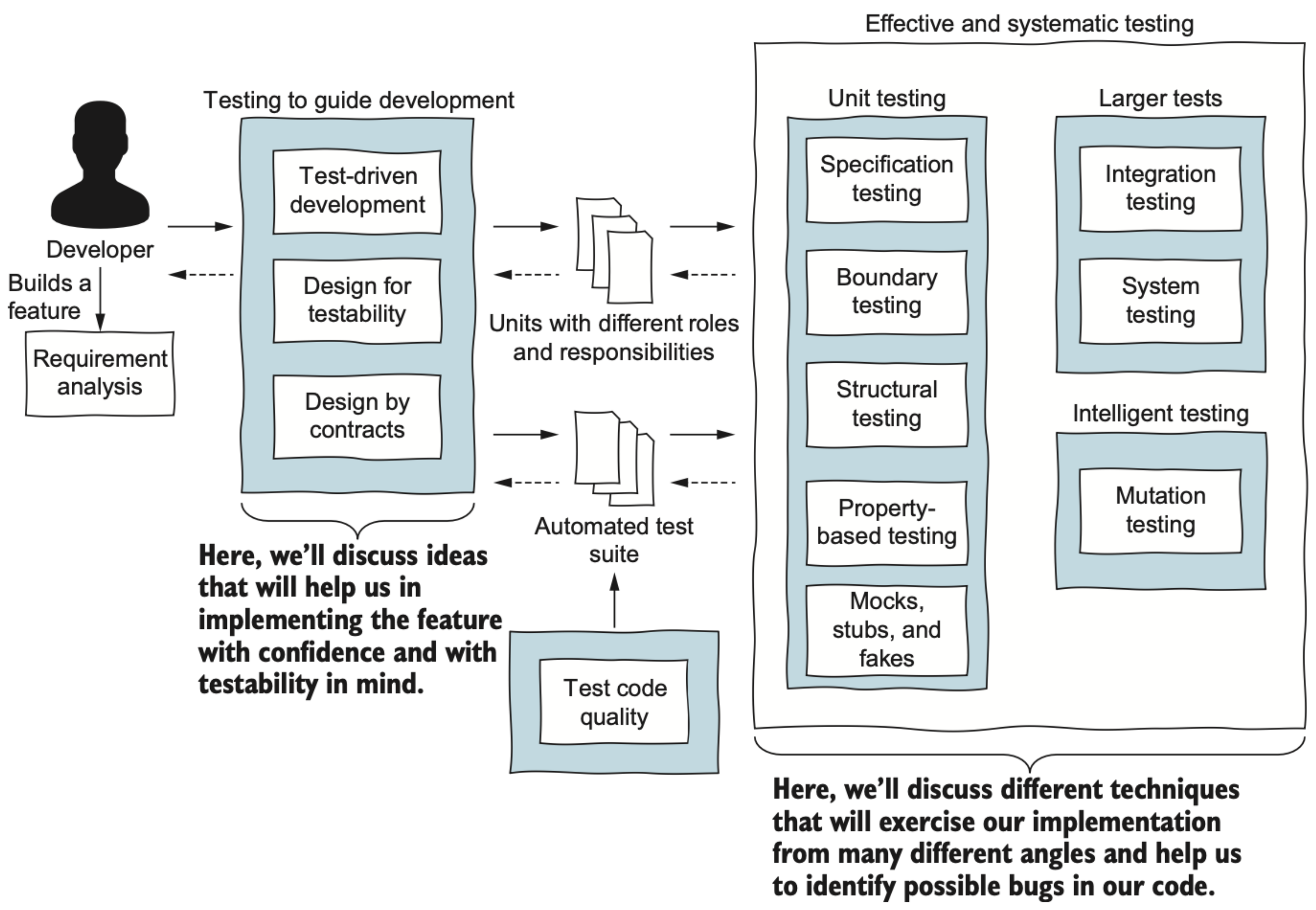
Property-based testing

Azzurra Ragone

Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 | Fax: +39.080.5442536
serlab.di.uniba.it

What is Property-based Testing?

Effective and systematic testing (workflow)



Developer
Builds a feature
Requirement analysis
Testing to guide development
Test-driven development
Design for testability
Design by contracts
Here, we'll discuss ideas that will help us in implementing the feature with confidence and with testability in mind.
Units with different roles and responsibilities
Automated test suite
Test code quality
Effective and systematic testing
Unit testing
Specification testing
Boundary testing
Structural testing
Property-based testing
Mocks, stubs, and fakes
Larger tests
Integration testing
System testing
Intelligent testing
Mutation testing
Here, we'll discuss different techniques that will exercise our implementation from many different angles and help us to identify possible bugs in our code.

Example vs Property-based testing

Example-based:
identify input partitions, pick one concrete example & write the test case

vs

Property-based:
Express the property we want to test & let the test framework choose several examples

Benefit of Property-based testing

- We define a property (or a set of properties) our program should adhere to, no need to pick a concrete example
- Our tests will be less dependent on a specific concrete example
- The test framework will call the method under test **multiple times with different input parameters** (with zero effort from us)

Example 1: The passing grade program

- A student passes an exam if he get a grade >= 5.0.
- Grades below that are a failure.
- Grades fall in the range [1.0, 10.0].

```
public class PassingGrade {
  public boolean passed(float grade) {
    if (grade < 1.0 || grade > 10.0)
      throw new IllegalArgumentException();
    return grade >= 5.0;
  }
}
```

Note the pre-condition check here.

Example 1: The passing grade program

- A student passes an exam if he get a grade >= 5.0.
- Grades below that are a failure.
- Grades fall in the range [1.0, 10.0].

Specification-based testing:

Devise partitions:

1. "passing grade," -> **pass**
2. "failing grade," -> **fail**
3. "grades outside the range." -> **invalid**

Then devise a single test case per partition, plus boundary cases.

Example 1: The passing grade program

- A student passes an exam if he get a grade >= 5.0.
- Grades below that are a failure.
- Grades fall in the range [1.0, 10.0].

<u>Property-based Testing (PBT):</u>

Identify properties based on the requirements:

- **fail**—For all numbers ranging from 1.0 (inclusive) to 5.0 (exclusive), the program should return *false*.
- **pass**—For all numbers ranging from 5.0 (inclusive) to 10.0 (inclusive), the program should return *true*.
- **invalid**—For all invalid grades (number below 1.0 or greater than 10.0), the program must throw an *exception*.

PBT: jqwik a PBT framework for Java

- We will use **jqwik** to write PBT
- The testing library jqwik will try to generate many value sets that fulfill the *precondition* hoping that one of the generated sets can *falsify* a wrong assumption.
- https://jqwik.net/docs/current/user-guide.html

Example 1: The passing grade program

- A student passes an exam if he get a grade >= 5.0.
- Grades below that are a failure.
- Grades fall in the range [1.0, 10.0].

Property-based Testing (PBT):

Let’s write PBT for the fail property:

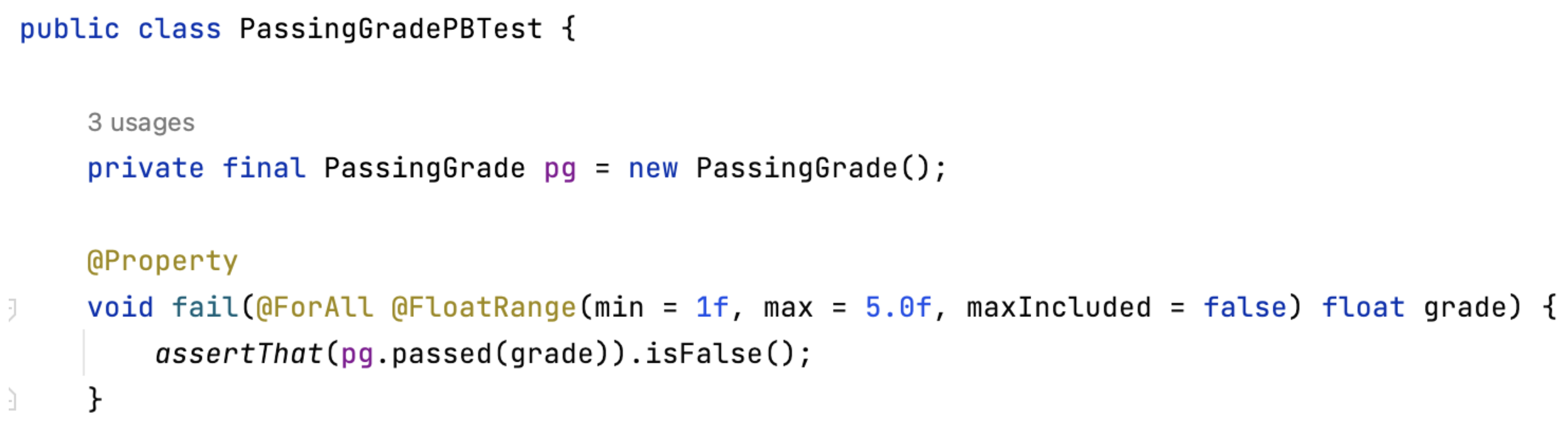
- **fail**—For all numbers ranging from 1.0 (inclusive) to 5.0 (exclusive), the program should return *false*.

Example 1: The passing grade program

Property-based Testing (PBT):

Let’s write PBT for the fail property:

- **fail**—For all numbers ranging from 1.0 (inclusive) to 5.0 (exclusive), the program should return *false*.



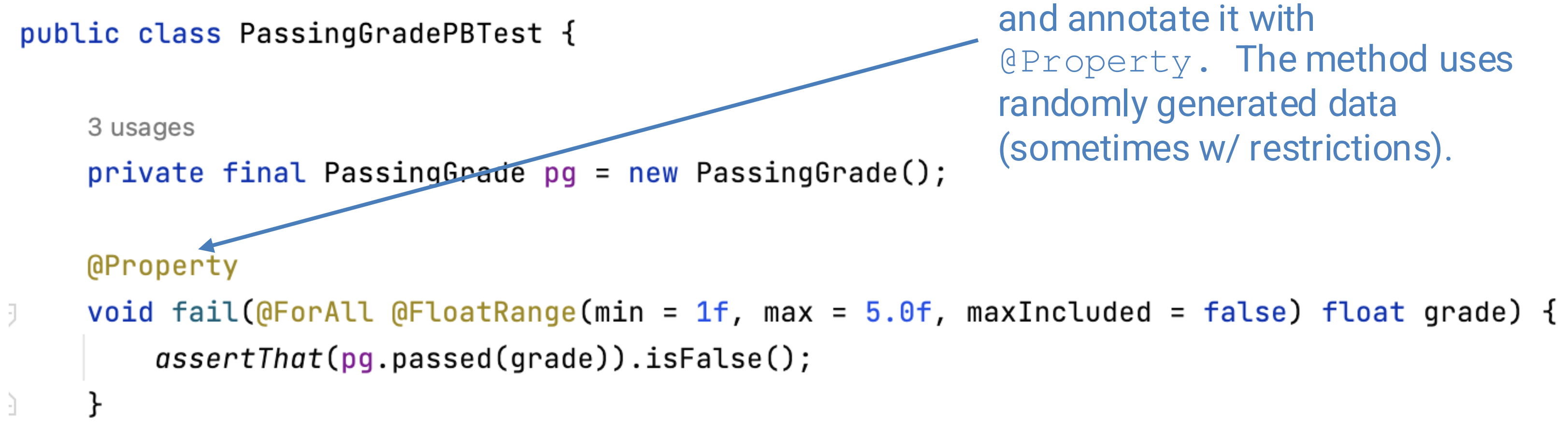
```
public class PassingGradePBTest {

    3 usages
    private final PassingGrade pg = new PassingGrade();

    @Property
    void fail(@ForAll @FloatRange(min = 1f, max = 5.0f, maxIncluded = false) float grade) {
        assertThat(pg.passed(grade)).isFalse();
    }
```

Example 1: The passing grade program

Property-based Testing (PBT):

Let's write PBT for the fail property:

- **fail**—For all numbers ranging from 1.0 (inclusive) to 5.0 (exclusive), the program should return *false*.



```
public class PassingGradePBTest {

    3 usages
    private final PassingGrade pg = new PassingGrade();

    @Property
    void fail(@ForAll @FloatRange(min = 1f, max = 5.0f, maxIncluded = false) float grade) {
        assertThat(pg.passed(grade)).isFalse();
    }
```

For each property we want to express, we create a method and annotate it with `@Property`. The method uses randomly generated data (sometimes w/ restrictions).

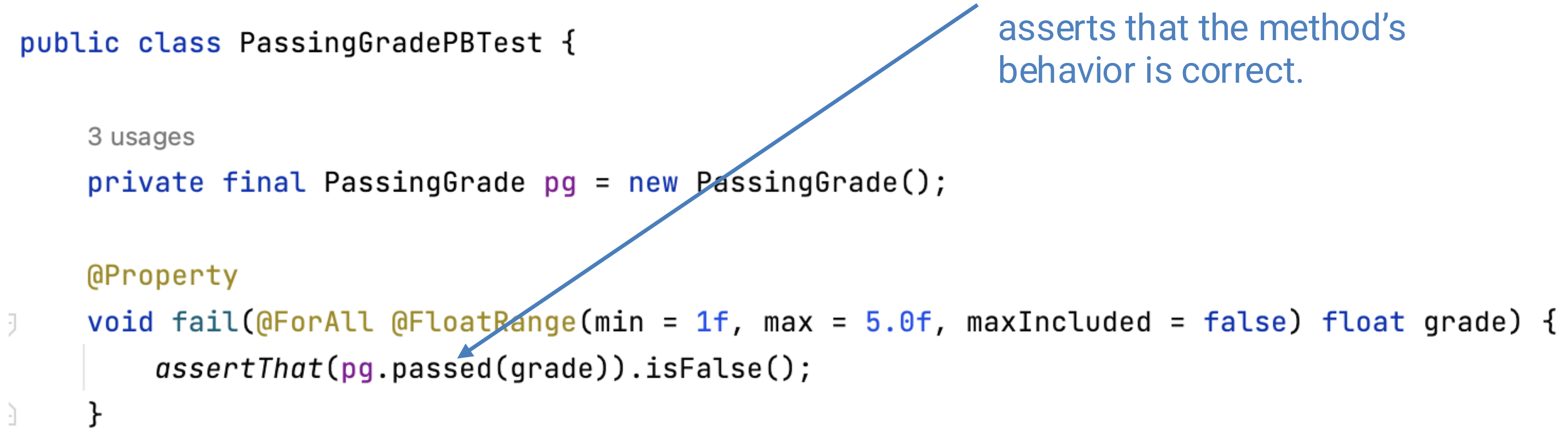
Example 1: The passing grade program

Property-based Testing (PBT):

Let’s write PBT for the fail property:

- **fail**—For all numbers ranging from 1.0 (inclusive) to 5.0 (exclusive), the program should return *false*.

The `@Property` method calls the method under test and asserts that the method’s behavior is correct.



```java
public class PassingGradePBTest {

    3 usages
    private final PassingGrade pg = new PassingGrade();

    @Property
    void fail(@ForAll @FloatRange(min = 1f, max = 5.0f, maxIncluded = false) float grade) {
        assertThat(pg.passed(grade)).isFalse();
    }
```

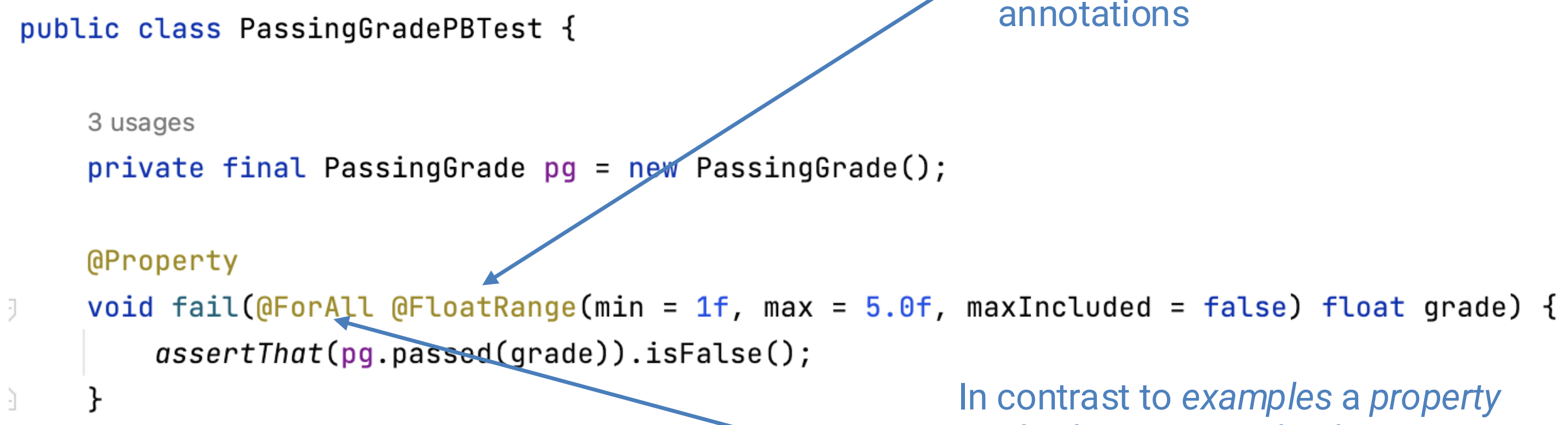
Example 1: The passing grade program

<u>Property-based Testing (PBT):</u>

Let’s write PBT for the fail property:

- **fail**—For all numbers ranging from 1.0 (inclusive) to 5.0 (exclusive), the program should return *false*.

Define the characteristics of the values we want to generate via annotations



```
public class PassingGradePBTest {

    3 usages
    private final PassingGrade pg = new PassingGrade();

    @Property
    void fail(@ForAll @FloatRange(min = 1f, max = 5.0f, maxIncluded = false) float grade) {
        assertThat(pg.passed(grade)).isFalse();
    }
```

In contrast to *examples* a *property* method is supposed to have one or more parameters, all of which must be annotated with `@ForAll`. We are saying this property should hold for all grades

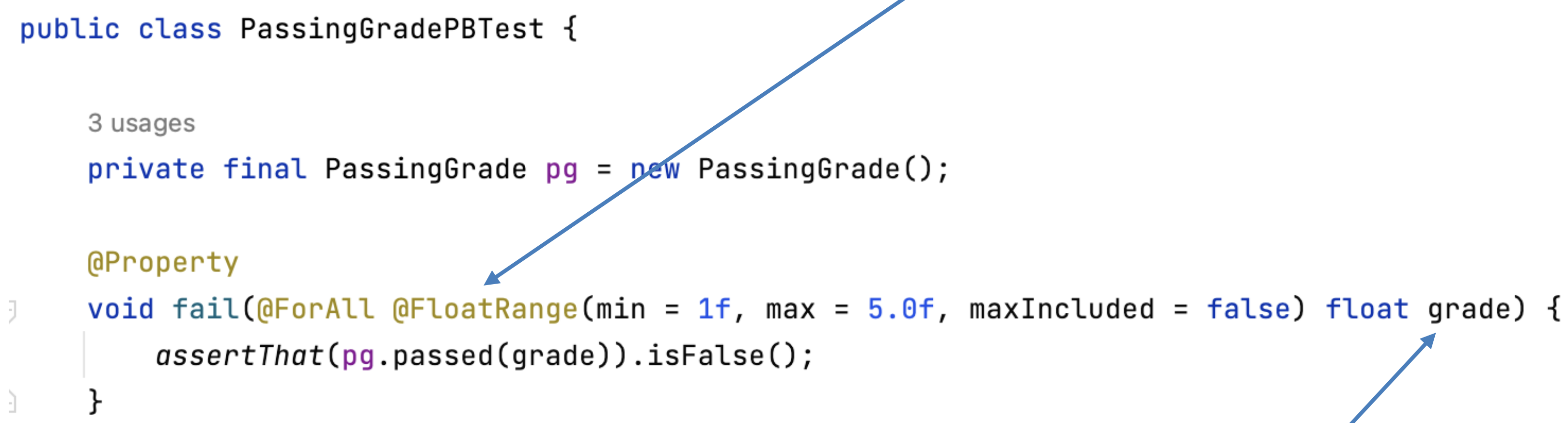
Example 1: The passing grade program

<u>Property-based Testing (PBT):</u>

Let’s write PBT for the fail property:

- **fail**—For all numbers ranging from 1.0 (inclusive) to 5.0 (exclusive), the program should return *false*.

We want to generate random floats in this range



```
public class PassingGradePBTest {

    3 usages
    private final PassingGrade pg = new PassingGrade();

    @Property
    void fail(@ForAll @FloatRange(min = 1f, max = 5.0f, maxIncluded = false) float grade) {
        assertThat(pg.passed(grade)).isFalse();
    }
```

The test method `fail` receives a `grade` parameter that jqwik will set following the rules specified in the annotation

PBT with jqwik

In contrast to examples, a property method is supposed to have one or more parameters, all of which must be annotated with `@ForAll`

- Just like an *example* test a **property method** has to either return:
  - a *boolean* value (true or false) of this property
  - return nothing (void). In the case of assertions
- jqwik generates a large amount of **random data** (following the characteristics you defined) and calls the test for it, looking for an input that would break the property.
- If not specified differently, jqwik will run **1000 tries**, i.e. a 1000 different sets of parameter values and execute the property method with each of those parameter sets. The first failed execution will stop value generation and be reported as failure. The tester has an **example of an input that breaks** the program.

PBT with jqwik

Property method uses randomly generated data.

Jqwik includes several generators for various types (including `Strings`, `Integers`, `Lists`, `Dates`, etc.).

Jqwik allows you to define *different sets of constraints* and *restrictions* to these parameters, for example, generate:

- only positive Integers
- only Strings with a length between 5 and 10 characters.

The property method receives all the required data for that test as parameters.

Example 1: The passing grade program

Property-based Testing (PBT):

Identify properties based on the requirements:

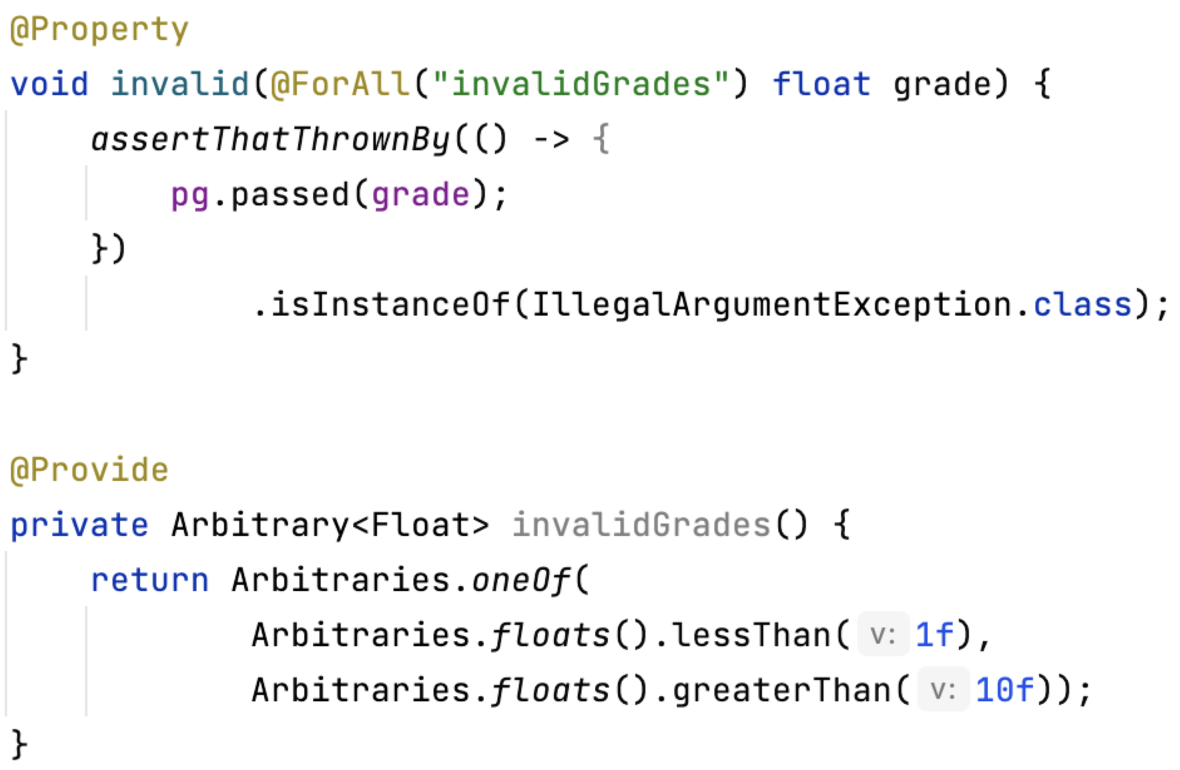
- **pass**—For all numbers ranging from 5.0 (inclusive) to 10.0 (inclusive), the program should return *true*.

```java
@Property
void pass(@ForAll @FloatRange(min = 5.0f, max = 10.0f, maxIncluded = true) float grade) {
    assertThat(pg.passed(grade)).isTrue();
}
```

Example 1: The passing grade program

Property-based Testing (PBT):

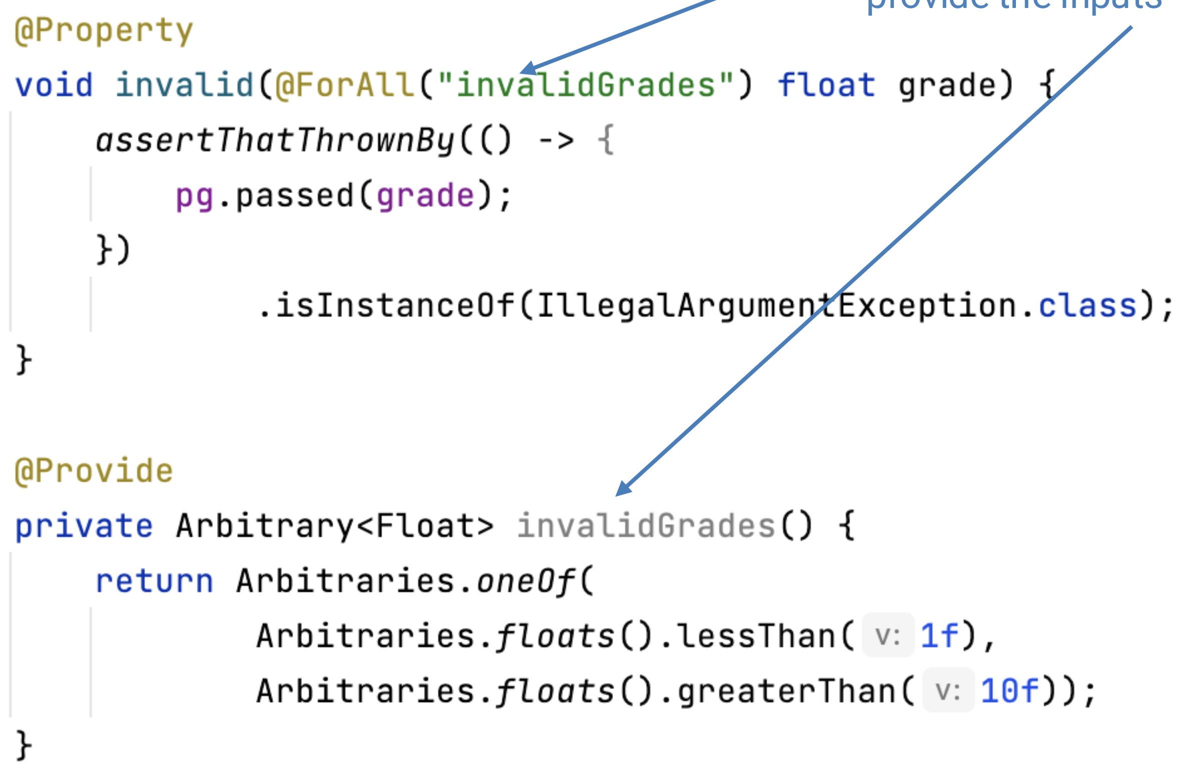
- **invalid**—For all invalid grades (number below 1.0 or greater than 10.0), the program must throw an *exception*.



```
@Property
void invalid(@ForAll("invalidGrades") float grade) {
    assertThatThrownBy(() -> {
        pg.passed(grade);
    })
            .isInstanceOf(IllegalArgumentException.class);
}

@Provide
private Arbitrary<Float> invalidGrades() {
    return Arbitraries.oneOf(
            Arbitraries.floats().lessThan( v: 1f),
            Arbitraries.floats().greaterThan( v: 10f));
}
```

Example 1: The passing grade program

The `@ForAll` annotation has the name of the method that provide the inputs



```
@Property
void invalid(@ForAll("invalidGrades") float grade) {
    assertThatThrownBy(() -> {
        pg.passed(grade);
    })
            .isInstanceOf(IllegalArgumentException.class);
}

@Provide
private Arbitrary<Float> invalidGrades() {
    return Arbitraries.oneOf(
            Arbitraries.floats().lessThan( v: 1f),
            Arbitraries.floats().greaterThan( v: 10f));
}
```

Example 1: The passing grade program

`Arbitrary` is how jqwik handles arbitrary values that need to be generated. For arbitrary floats, your provider method should return an `Arbitrary<Float>`

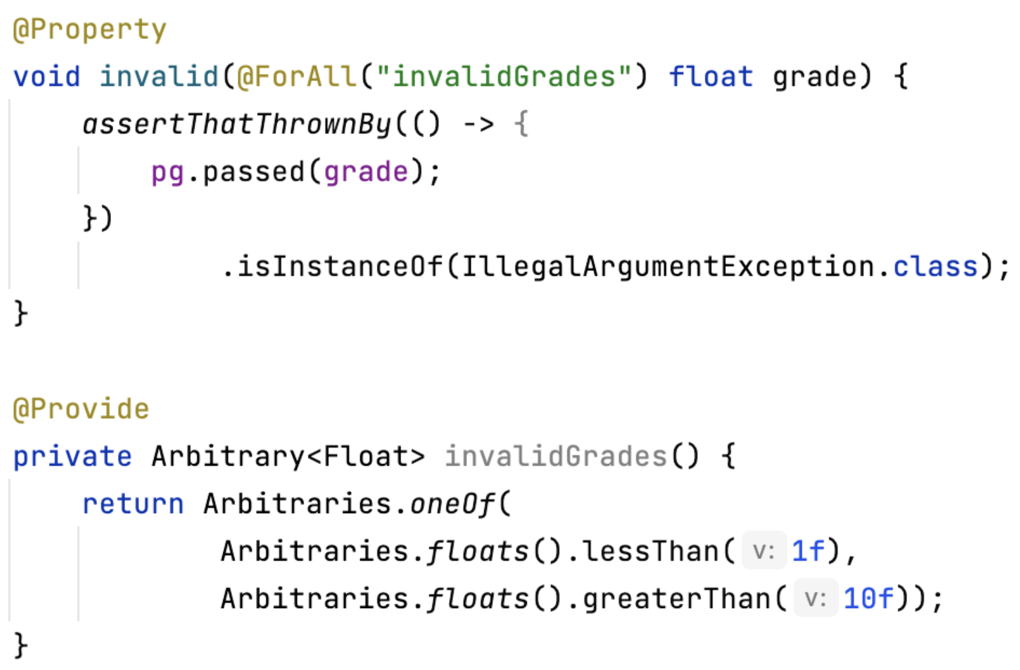
```
@Property
void invalid(@ForAll("invalidGrades") float grade) {
    assertThatThrownBy(() -> {
        pg.passed(grade);
    })
            .isInstanceOf(IllegalArgumentException.class);
}

@Provide
private Arbitrary<Float> invalidGrades() {
    return Arbitraries.oneOf(
            Arbitraries.floats().lessThan( v: 1f),
            Arbitraries.floats().greaterThan( v: 10f));
}
```

Example 1: The passing grade program

The `Arbitraries` class contains dozens of methods that helps you build arbitrary data. The `oneOf()` method receives a list of arbitrary values it may return.

The `Arbitraries.floats()` method generates random floats.

The `lessThan()` and `greaterThan()` methods generate numbers <1 and >10



```java
@Property
void invalid(@ForAll("invalidGrades") float grade) {
    assertThatThrownBy(() -> {
        pg.passed(grade);
    })
            .isInstanceOf(IllegalArgumentException.class);
}

@Provide
private Arbitrary<Float> invalidGrades() {
    return Arbitraries.oneOf(
            Arbitraries.floats().lessThan( v: 1f),
            Arbitraries.floats().greaterThan( v: 10f));
}
```

Example 1: The passing grade program

Property-based Testing (PBT):

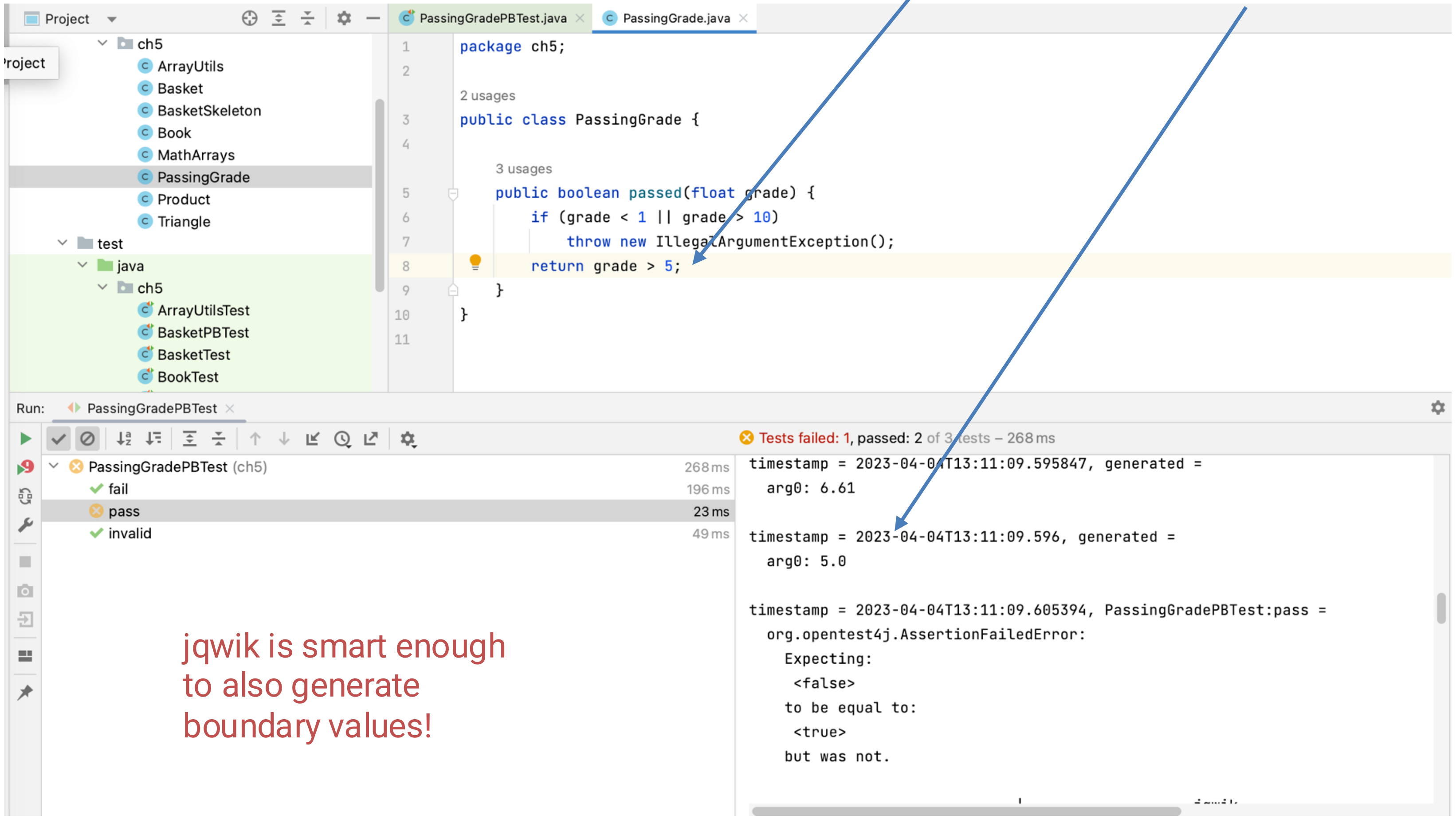
Identify properties based on the requirements:

- **pass**—For all numbers ranging from 5.0 (inclusive) to 10.0 (inclusive), the program should return *true*.

To see all the generated inputs, add a `@Report` annotation to the property annotation

```
@Property
@Report(Reporting.GENERATED)
void pass(@ForAll @FloatRange(min = 5.0f, max = 10.0f, maxIncluded = true) float grade) {
    assertThat(pg.passed(grade)).isTrue();
}
```

If we change the condition on grade from >= to > the pass test will fail, and the report will show all the inputs tried until the one that fails (5)



```java
package ch5;

public class PassingGrade {

    public boolean passed(float grade) {
        if (grade < 1 || grade > 10)
            throw new IllegalArgumentException();
        return grade > 5;
    }
}
```

Tests failed: 1, passed: 2 of 3 tests – 268 ms

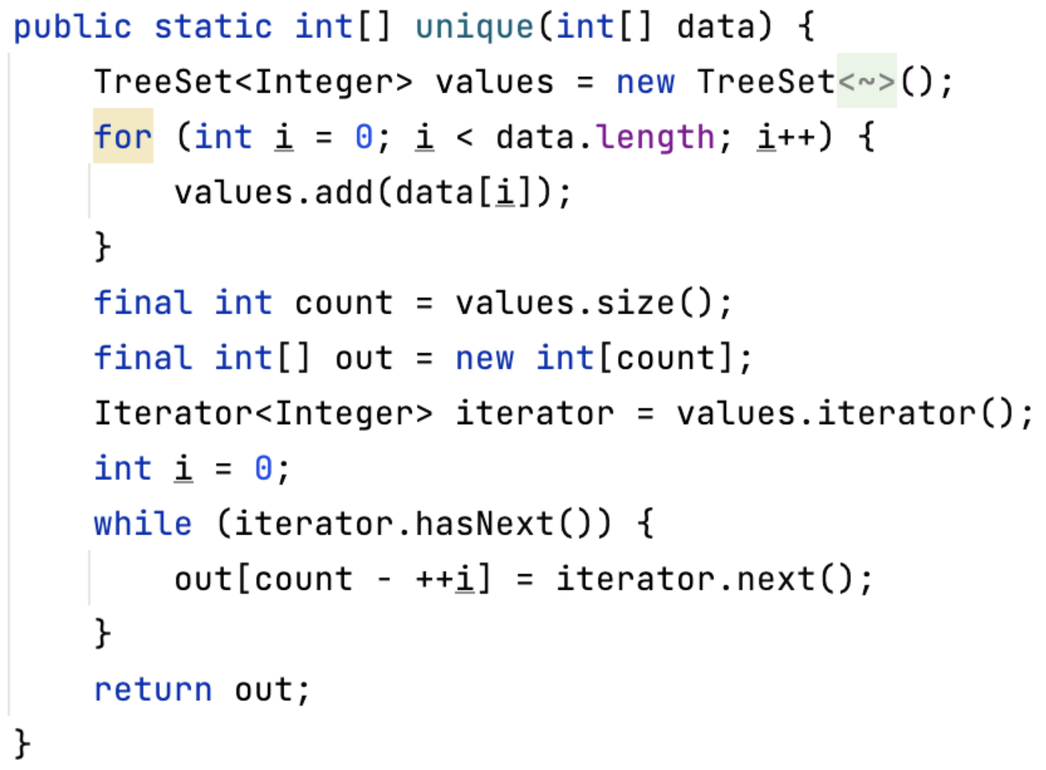
```
PassingGradePBTest (ch5)   268 ms
    fail                   196 ms
    pass                    23 ms
    invalid                 49 ms
```

```
timestamp = 2023-04-04T13:11:09.595847, generated =
  arg0: 6.61

timestamp = 2023-04-04T13:11:09.596, generated =
  arg0: 5.0

timestamp = 2023-04-04T13:11:09.605394, PassingGradePBTest:pass =
  org.opentest4j.AssertionFailedError:
    Expecting:
     <false>
    to be equal to:
     <true>
    but was not.
```

jqwik is smart enough to also generate boundary values!

Example 2: Testing the unique method

Returns an array consisting of the **unique** values in data. The return array is sorted in **descending order**. *Empty* arrays are allowed, but *null* arrays result in a `NullPointerException`.



```java
public static int[] unique(int[] data) {
    TreeSet<Integer> values = new TreeSet<~>();
    for (int i = 0; i < data.length; i++) {
        values.add(data[i]);
    }
    final int count = values.size();
    final int[] out = new int[count];
    Iterator<Integer> iterator = values.iterator();
    int i = 0;
    while (iterator.hasNext()) {
        out[count - ++i] = iterator.next();
    }
    return out;
}
```

Note on TreeSet

The `TreeSet` is a sorted collection that extends the `AbstractSet` class and implements the `NavigableSet` interface.

Most important aspects of this implementation:

- It stores **unique** elements
- It doesn't preserve the insertion order of the elements
- It sorts the elements in **ascending order**

The `TreeSet iterator()` method returns an iterator iterating in the *ascending order* over the elements in the Set.

Example 2: Testing the unique method

The method returns a **descending list of unique values** included in the input array

```
public class MathArraysPBTest {

  @Property
  void unique(
   @ForAll
   @Size(value = 100)
   List<@IntRange(min = 1, max = 20) Integer>
   numbers) {

    int[] doubles = convertListToArray(numbers);
    int[] result = MathArrays.unique(doubles);

    assertThat(result)
        .contains(doubles)
        .doesNotHaveDuplicates()
        .isSortedAccordingTo(reverseOrder());
  }

  private int[] convertListToArray(List<Integer> numbers) {
    int[] array = numbers
      .stream()
      .mapToInt(x -> x)
      .toArray();

    return array;
  }
}
```

An array of size 100

With values in [0, 20]. Given the size of the array (100), we know it will contain repeated elements.

Contains all the elements

No duplicates

In descending order

Utility method that converts a list of integers to an array

Example 2: Testing the unique method

AssertJ assertions:

`contains()`: verifies that the actual iterable/array contains the given values in any order

`doesNotHaveDuplicates()`: verifies that the actual array does not contain duplicates.

`isSortedAccordingTo()`: the `Comparator.reverseOrder()` can be used to assert that it should be the reverse of it's natural ordering

```
public class MathArraysPBTest {

  @Property
  void unique(
   @ForAll
   @Size(value = 100)
   List<@IntRange(min = 1, max = 20) Integer>
   numbers) {

    int[] doubles = convertListToArray(numbers);
    int[] result = MathArrays.unique(doubles);

    assertThat(result)
        .contains(doubles)
        .doesNotHaveDuplicates()
        .isSortedAccordingTo(reverseOrder());
  }

  private int[] convertListToArray(List<Integer> numbers) {
    int[] array = numbers
      .stream()
      .mapToInt(x -> x)
      .toArray();

    return array;
  }
}
```

An array of size 100

With values in [0, 20]. Given the size of the array (100), we know it will contain repeated elements.

Contains all the elements

No duplicates

In descending order

Utility method that converts a list of integers to an array

Example 3: Testing the indexOf method

Finds the index of the given value in the array starting at the given index. This method returns −1 for a null input array. A negative `startIndex` is treated as zero. A `startIndex` larger than the array length will return −1.

Input parameters:

- `array`: Array to search for the object. May be null.
- `valueToFind`: Value to find.
- `startIndex`: Index at which to start searching.

The method returns the index of the value within the array, or −1 if not found or null.

Example 3: Testing the indexOf method

Implementation of the `indexOf` method

```
class ArrayUtils {
  public static int indexOf(final int[] array, final int valueToFind,
    ➥ int startIndex) {
    if (array == null) {
      return -1;
    }

    if (startIndex < 0) {
      startIndex = 0;
    }

    for (int i = startIndex; i < array.length; i++) {
      if (valueToFind == array[i]) {
        return i;
      }
    }
    return -1;
  }
}
```

The method accepts a null array and returns -1 in such a case. Another option could be to throw an exception, but the developer decided to use a weaker pre-condition.

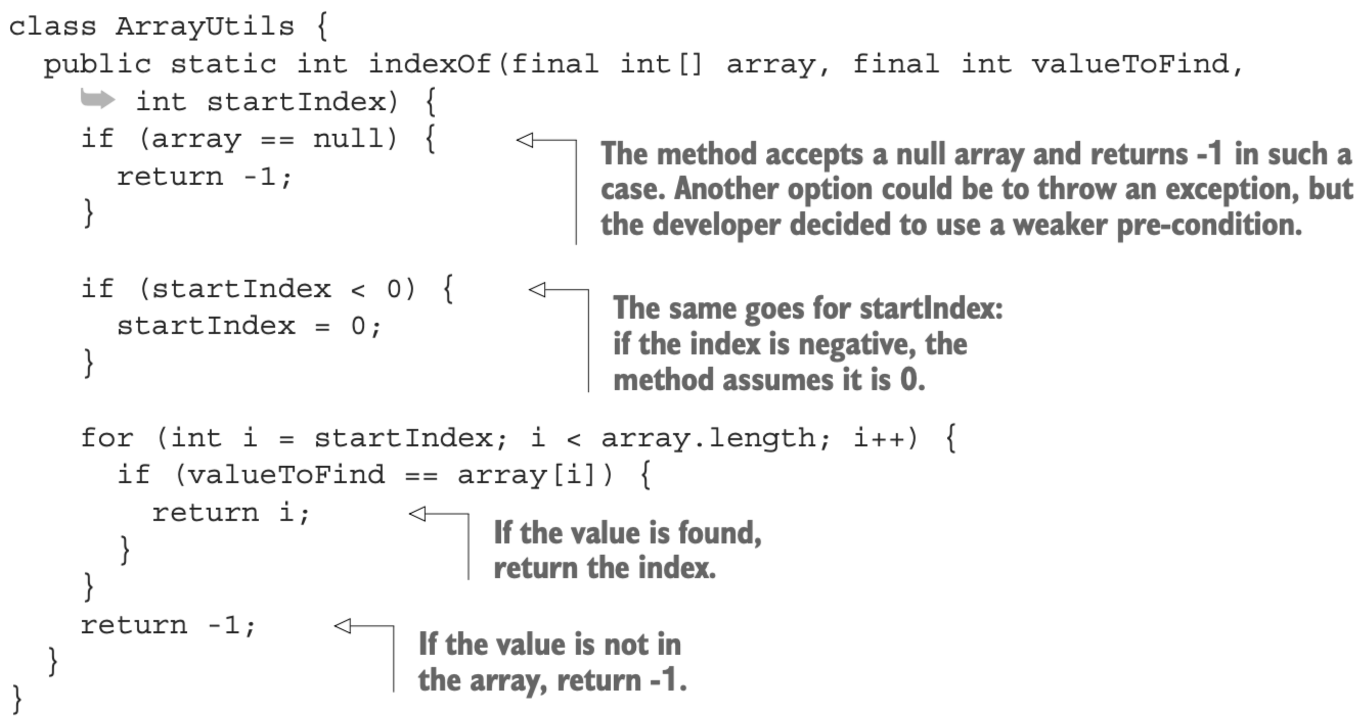
The same goes for startIndex: if the index is negative, the method assumes it is 0.

If the value is found, return the index.

If the value is not in the array, return -1.

Example 3: Testing the indexOf method

Esercitazione: *testare il metodo* `indexOf` *scrivendo un test di tipo parametrico*



```
class ArrayUtils {
  public static int indexOf(final int[] array, final int valueToFind,
    ➥ int startIndex) {
    if (array == null) {
      return -1;
    }

    if (startIndex < 0) {
      startIndex = 0;
    }

    for (int i = startIndex; i < array.length; i++) {
      if (valueToFind == array[i]) {
        return i;
      }
    }
    return -1;
  }
}
```

The method accepts a null array and returns -1 in such a case. Another option could be to throw an exception, but the developer decided to use a weaker pre-condition.

The same goes for startIndex: if the index is negative, the method assumes it is 0.

If the value is found, return the index.

If the value is not in the array, return -1.

Example 3: Testing the indexOf method

PBT steps:

- Insert a **random value** in a **random position** of a **random array**.
- The `indexOf()` method will look for this random value.
- The test will **assert** that the method returns an **index** that matches the random position where we inserted the element.

NOTE THAT: We should ensure that the random value we add in the array does not already exist in the random array. Ex: in the list `[1, 2, 3, 4]` we cannot insert 4 on index 1, otherwise the response will be different depending on whether `startIndex` is 0 or 3.
While the response should depend only on the condition: `startIndex <` or `> indexToAddElement`

Example 3: Testing the indexOf method

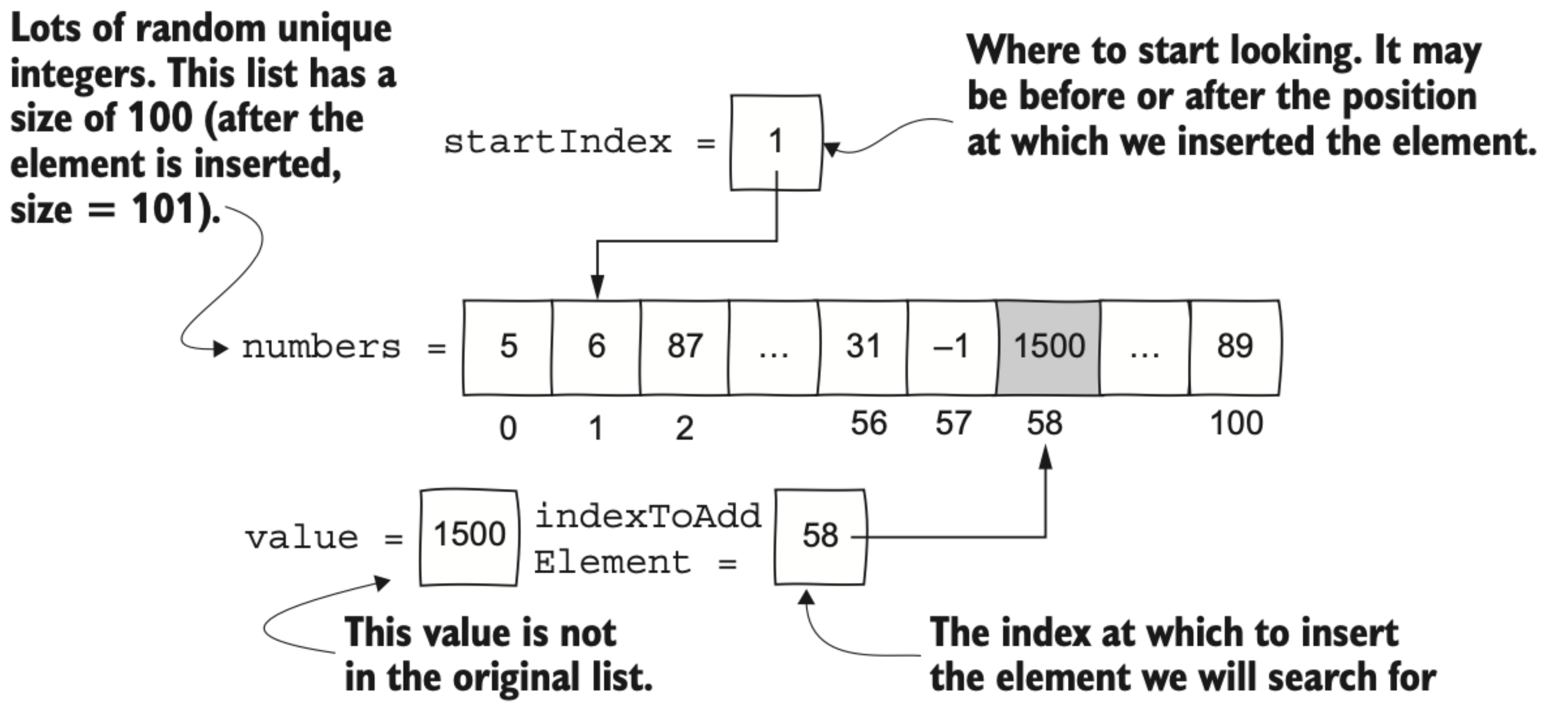
The PBT needs at least four parameters:

- `numbers`—A list of random integers (*list not array!*) with size of 100 and with values between [–1000, 1000].
- `value`—A random integer that is the value to be inserted into the list, with values between [1001, 2000], so the value generated will not exist in the list.
- `indexToAddElement`—A random integer that is the index where to add the `value`. The index range is [0, 99] (the list has size 100).
- `startIndex`—A random integer that represents the index where to start the search. The index range is [0, 99] (the list has size 100).

We assert that the method returns `indexToAddElement` if `indexToAddElement >= startIndex` (that is, the element was inserted after the start index) or –1 if the element was inserted before the start index.

Example 3: Testing the indexOf method

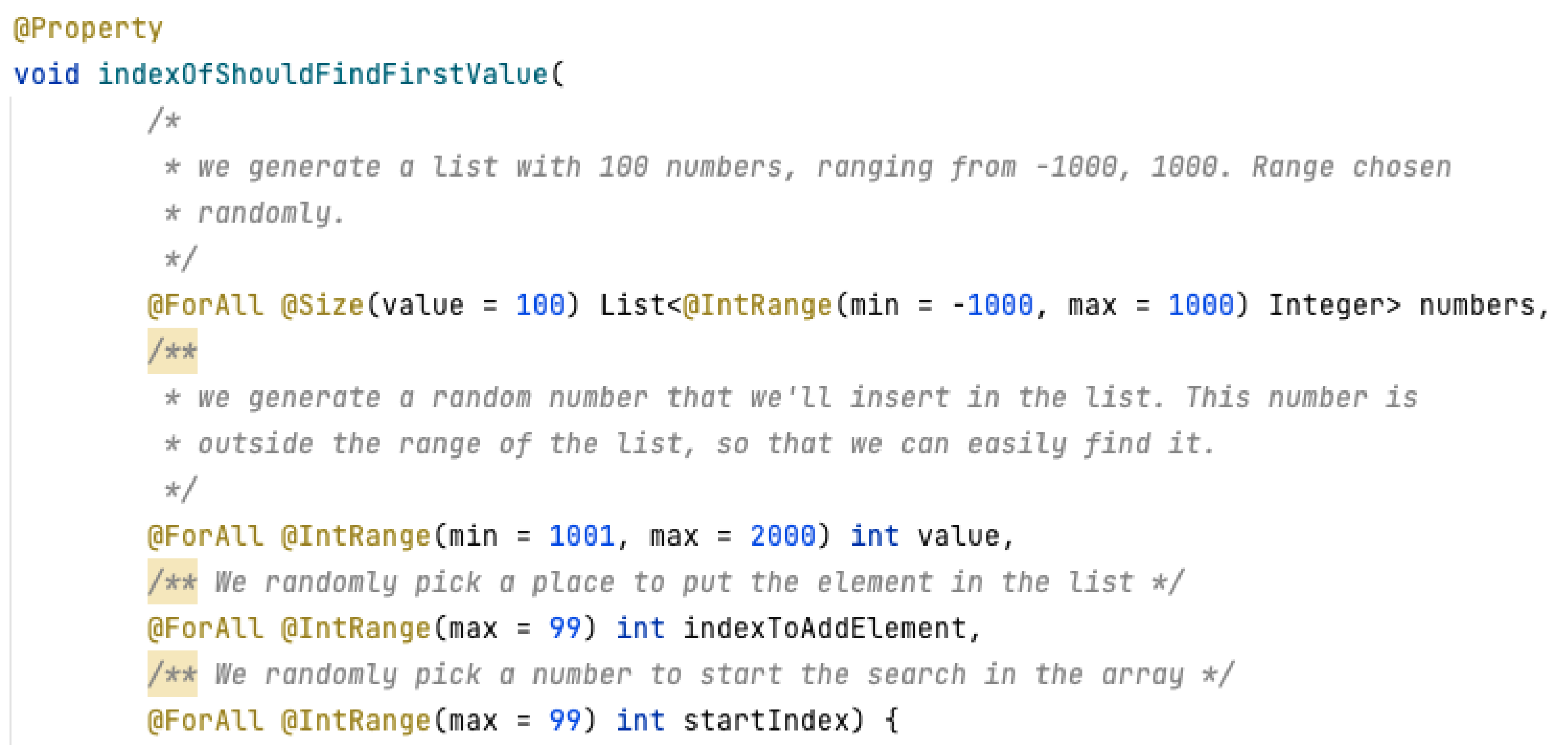
We assert that the method returns `indexToAddElement` if `indexToAddElement >= startIndex` (that is, the element was inserted after the start index) or −1 if the element was inserted before the start index.



Lots of random unique integers. This list has a size of 100 (after the element is inserted, size = 101).
startIndex =
1
Where to start looking. It may be before or after the position at which we inserted the element.
numbers =
5
6
87
...
31
–1
1500
...
89
0
1
2
56
57
58
100
value =
1500
indexToAddElement =
58
This value is not in the original list.
The index at which to insert the element we will search for

Example 3: Testing the indexOf method

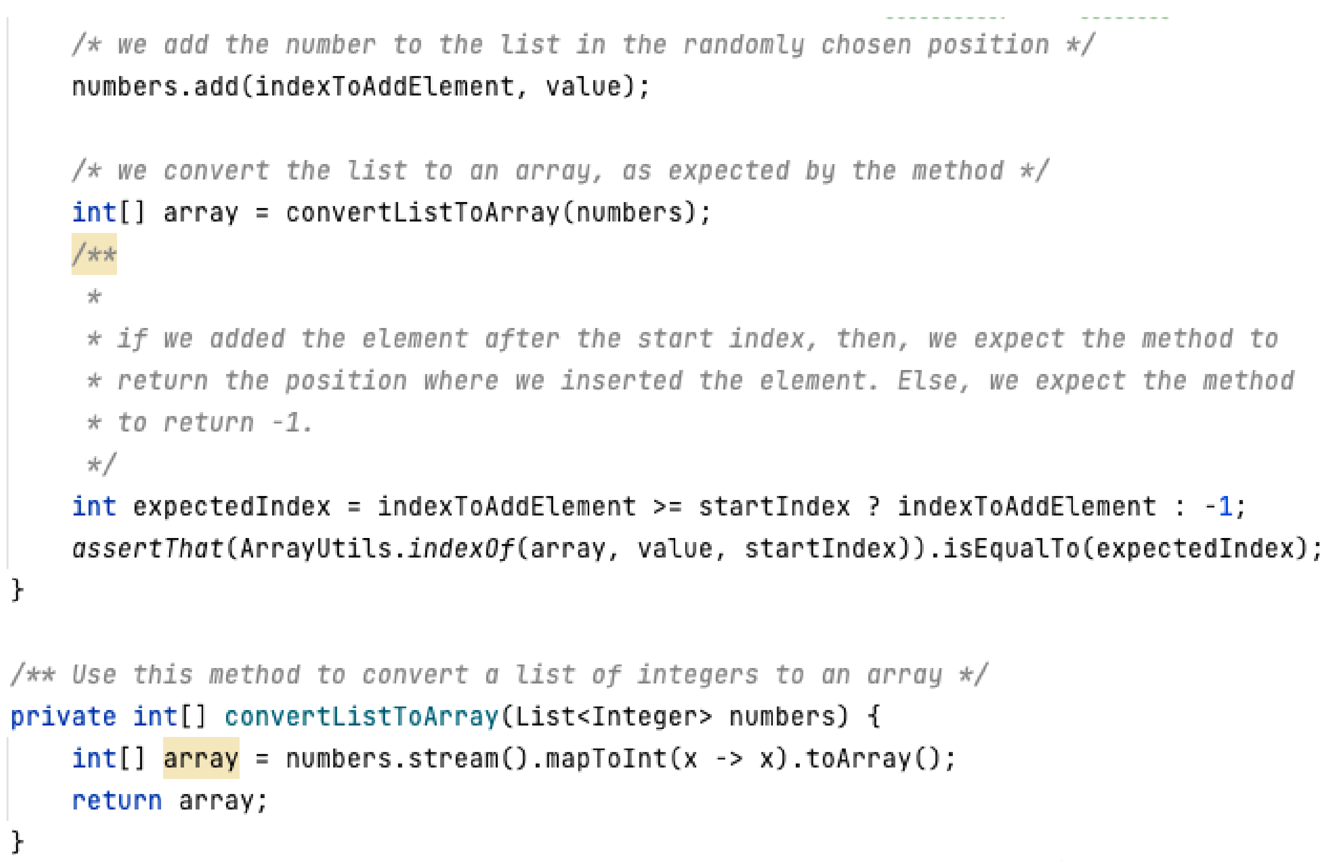
The PBT needs at least four parameters:



```java
@Property
void indexOfShouldFindFirstValue(
        /*
         * we generate a list with 100 numbers, ranging from -1000, 1000. Range chosen
         * randomly.
         */
        @ForAll @Size(value = 100) List<@IntRange(min = -1000, max = 1000) Integer> numbers,
        /**
         * we generate a random number that we'll insert in the list. This number is
         * outside the range of the list, so that we can easily find it.
         */
        @ForAll @IntRange(min = 1001, max = 2000) int value,
        /** We randomly pick a place to put the element in the list */
        @ForAll @IntRange(max = 99) int indexToAddElement,
        /** We randomly pick a number to start the search in the array */
        @ForAll @IntRange(max = 99) int startIndex) {
```

Example 3: Testing the indexOf method

We assert that the method returns `indexToAddElement` if `indexToAddElement >= startIndex` (that is, the element was inserted after the start index) or –1 if the element was inserted before the start index.



```
    /* we add the number to the list in the randomly chosen position */
    numbers.add(indexToAddElement, value);

    /* we convert the list to an array, as expected by the method */
    int[] array = convertListToArray(numbers);
    /**
     *
     * if we added the element after the start index, then, we expect the method to
     * return the position where we inserted the element. Else, we expect the method
     * to return -1.
     */
    int expectedIndex = indexToAddElement >= startIndex ? indexToAddElement : -1;
    assertThat(ArrayUtils.indexOf(array, value, startIndex)).isEqualTo(expectedIndex);
}

/** Use this method to convert a list of integers to an array */
private int[] convertListToArray(List<Integer> numbers) {
    int[] array = numbers.stream().mapToInt(x -> x).toArray();
    return array;
}
```

Example 3: Testing the indexOf method

Note on PBT:

- PBT better exercises the properties of the method than parameterized testing method
- *Creativity*:
  - generate a random value that is never in the list
  - Assertion w.r.t. `expectedIndex`
- Random generated data should be as much as possible similar to what you would expect in real cases.
- Partitions should have the same likelihood of being exercised by your test (You can check this with Jqwik statistics)

Collecting and Reporting Statistics

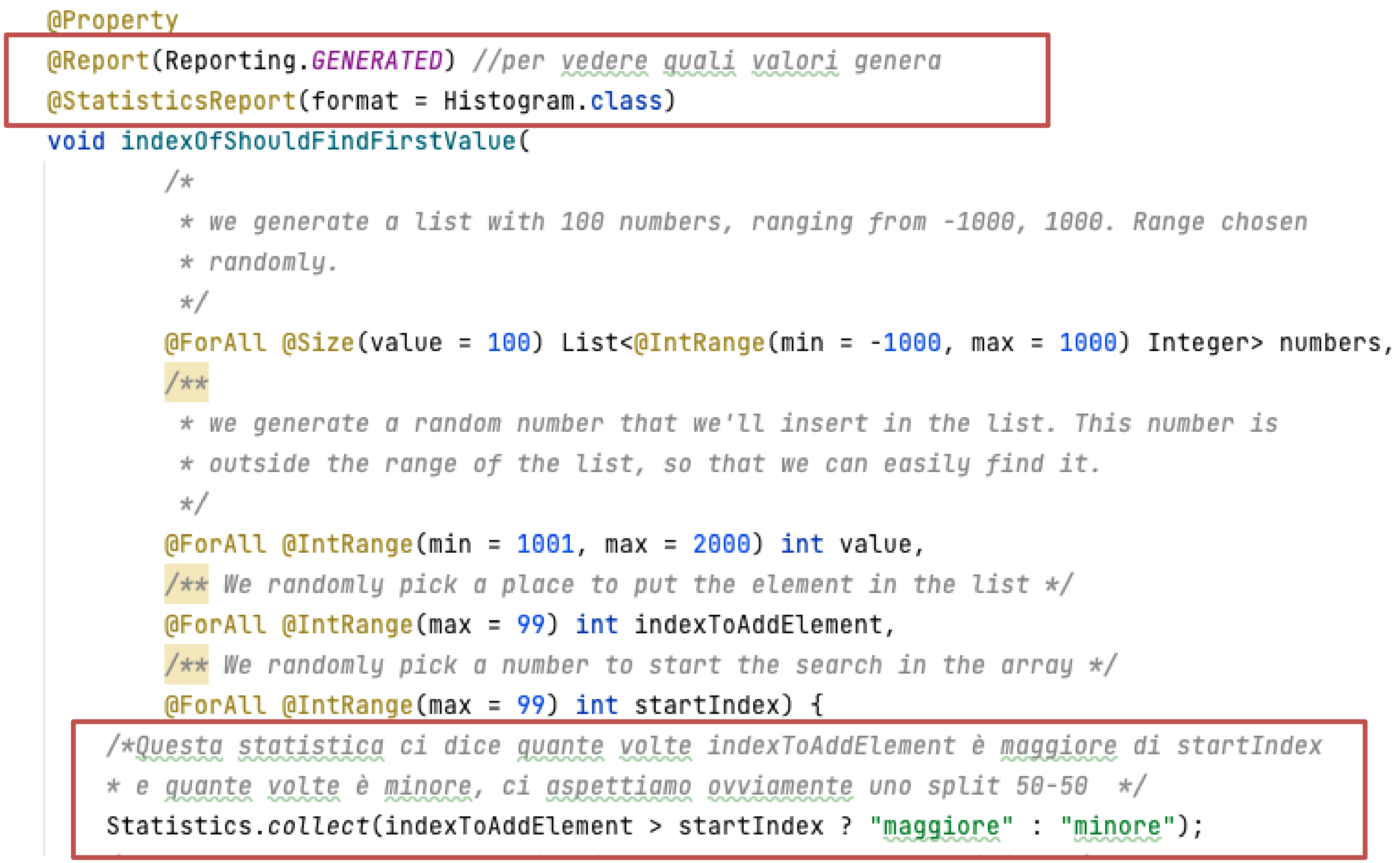
In some cases, you may want to know if:

- jqwik really generates the kind of **values you expect**
- if the **frequency** and **distribution** of certain value classes meets your testing needs

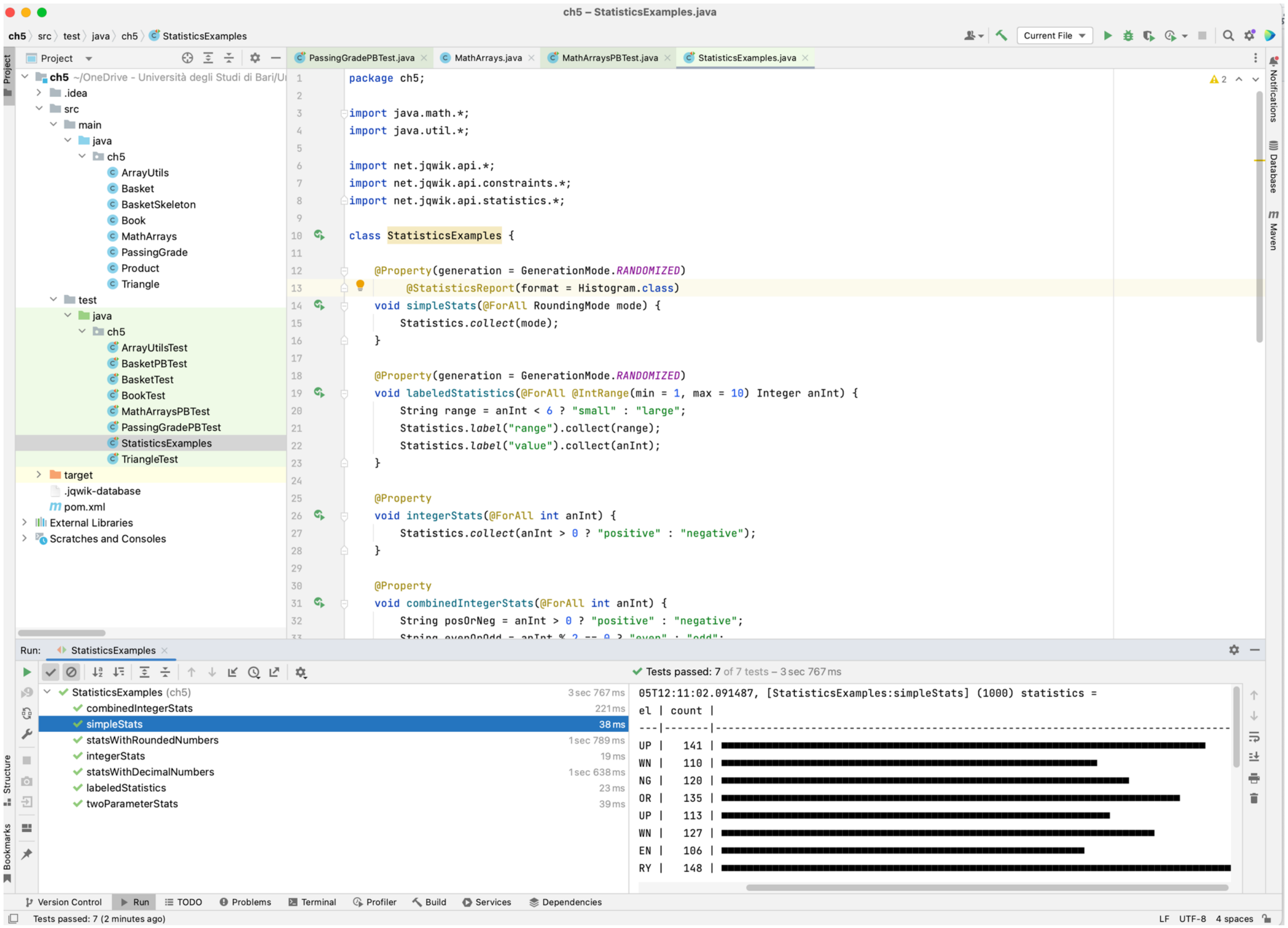
`Statistics.collect()` is made for this exact purpose.

Link: https://jqwik.net/docs/current/user-guide.html#collecting-and-reporting-statistics

Statistics for the indexOf method



```
@Property
@Report(Reporting.GENERATED) //per vedere quali valori genera
@StatisticsReport(format = Histogram.class)
void indexOfShouldFindFirstValue(
        /*
         * we generate a list with 100 numbers, ranging from -1000, 1000. Range chosen
         * randomly.
         */
        @ForAll @Size(value = 100) List<@IntRange(min = -1000, max = 1000) Integer> numbers,
        /**
         * we generate a random number that we'll insert in the list. This number is
         * outside the range of the list, so that we can easily find it.
         */
        @ForAll @IntRange(min = 1001, max = 2000) int value,
        /** We randomly pick a place to put the element in the list */
        @ForAll @IntRange(max = 99) int indexToAddElement,
        /** We randomly pick a number to start the search in the array */
        @ForAll @IntRange(max = 99) int startIndex) {
    /*Questa statistica ci dice quante volte indexToAddElement è maggiore di startIndex
    * e quante volte è minore, ci aspettiamo ovviamente uno split 50-50  */
    Statistics.collect(indexToAddElement > startIndex ? "maggiore" : "minore");
```



ch5 – StatisticsExamples.java

ch5 › src › test › java › ch5 › StatisticsExamples

Project: ch5 ~/OneDrive - Università degli Studi di Bari/U
- .idea
- src
  - main
    - java
      - ch5
        - ArrayUtils
        - Basket
        - BasketSkeleton
        - Book
        - MathArrays
        - PassingGrade
        - Product
        - Triangle
  - test
    - java
      - ch5
        - ArrayUtilsTest
        - BasketPBTest
        - BasketTest
        - BookTest
        - MathArraysPBTest
        - PassingGradePBTest
        - StatisticsExamples
        - TriangleTest
- target
- .jqwik-database
- pom.xml
- External Libraries
- Scratches and Consoles

Tabs: PassingGradePBTest.java | MathArrays.java | MathArraysPBTest.java | StatisticsExamples.java

```java
package ch5;

import java.math.*;
import java.util.*;

import net.jqwik.api.*;
import net.jqwik.api.constraints.*;
import net.jqwik.api.statistics.*;

class StatisticsExamples {

    @Property(generation = GenerationMode.RANDOMIZED)
        @StatisticsReport(format = Histogram.class)
    void simpleStats(@ForAll RoundingMode mode) {
        Statistics.collect(mode);
    }

    @Property(generation = GenerationMode.RANDOMIZED)
    void labeledStatistics(@ForAll @IntRange(min = 1, max = 10) Integer anInt) {
        String range = anInt < 6 ? "small" : "large";
        Statistics.label("range").collect(range);
        Statistics.label("value").collect(anInt);
    }

    @Property
    void integerStats(@ForAll int anInt) {
        Statistics.collect(anInt > 0 ? "positive" : "negative");
    }

    @Property
    void combinedIntegerStats(@ForAll int anInt) {
        String posOrNeg = anInt > 0 ? "positive" : "negative";
        String evenOrOdd = anInt % 2 == 0 ? "even" : "odd";
```

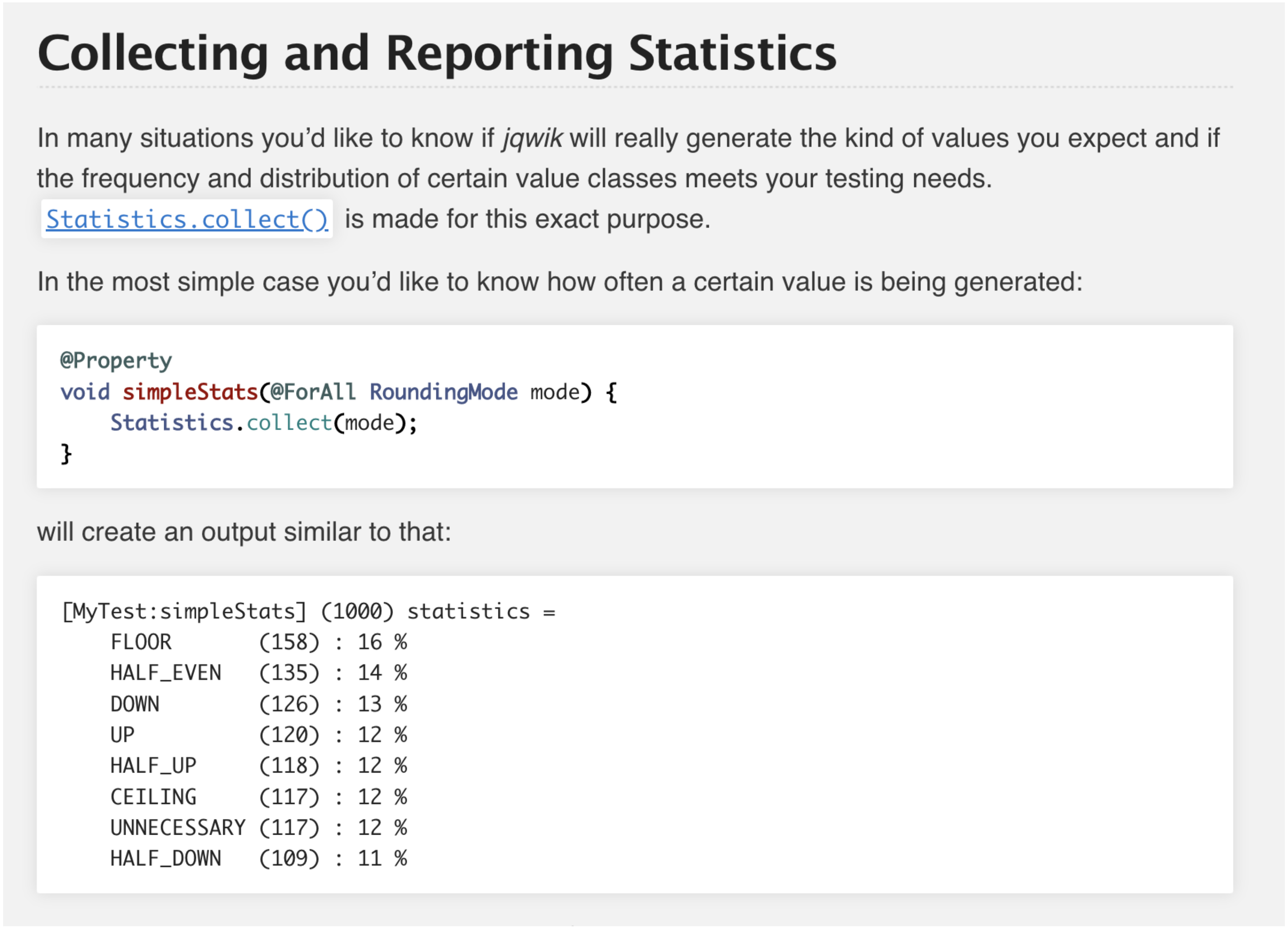
Run: StatisticsExamples

✓ Tests passed: 7 of 7 tests – 3 sec 767 ms

✓ StatisticsExamples (ch5)	3 sec 767 ms
✓ combinedIntegerStats	221 ms
✓ simpleStats	38 ms
✓ statsWithRoundedNumbers	1 sec 789 ms
✓ integerStats	19 ms
✓ statsWithDecimalNumbers	1 sec 638 ms
✓ labeledStatistics	23 ms
✓ twoParameterStats	39 ms

```
05T12:11:02.091487, [StatisticsExamples:simpleStats] (1000) statistics =
el | count |
---|-------|---------------------------------------------------------------
UP |   141 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
WN |   110 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
NG |   120 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
OR |   135 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
UP |   113 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
WN |   127 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
EN |   106 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
RY |   148 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

Tests passed: 7 (2 minutes ago)



Collecting and Reporting Statistics

In many situations you'd like to know if *jqwik* will really generate the kind of values you expect and if the frequency and distribution of certain value classes meets your testing needs. `Statistics.collect()` is made for this exact purpose.

In the most simple case you'd like to know how often a certain value is being generated:

```
@Property
void simpleStats(@ForAll RoundingMode mode) {
    Statistics.collect(mode);
}
```

will create an output similar to that:

```
[MyTest:simpleStats] (1000) statistics =
    FLOOR       (158) : 16 %
    HALF_EVEN   (135) : 14 %
    DOWN        (126) : 13 %
    UP          (120) : 12 %
    HALF_UP     (118) : 12 %
    CEILING     (117) : 12 %
    UNNECESSARY (117) : 12 %
    HALF_DOWN   (109) : 11 %
```

PBT: Inputs distributions

In PBT inputs should be fairly distributed among all the possible options.

Ex. A: Integer between 0 and 10

Ex. B: generate an array of 100 elements in which the numbers have to be unique and multiples of 2, 3, 5, and 15.

PBT: Inputs distributions

EX. C: testing method that receive 3 `int`: `a`, `b`, and `c` and return a `boolean` indicating whether these three sides can form a **triangle**.

```
public static boolean isTriangle(int a, int b, int c)
```

What could be the trick here?

PBT: Inputs distributions

To write a PBT for the `isTriangle` method, we need to express two properties: **valid** triangles and **invalid** triangles.

If the developer generates three random integer values:

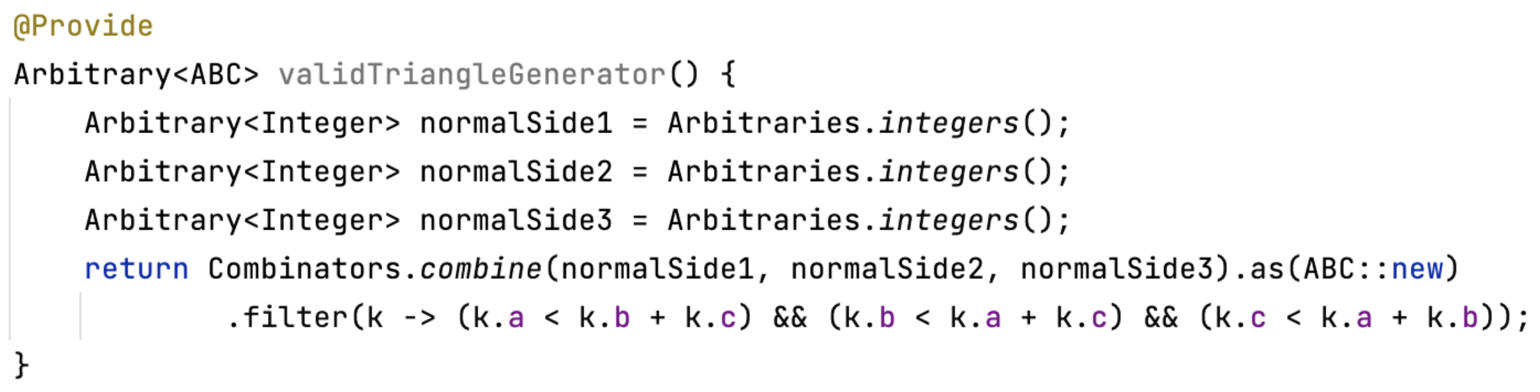
```
@Property
void triangleBadTest(@ForAll @IntRange(max = 100) int a,
                     @ForAll @IntRange(max = 100) int b,
                     @ForAll @IntRange(max = 100) int c) {
    // ... test here ...

}
```

The test exercises the invalid triangle property more than the valid triangle property.
Solution: have two tests, one for valid and one for invalid triangles!

PBT: Inputs distributions

Given three segment a, b, c they can form a **valid** triangles if and only if:

(a < b + c) AND (b < a + c) AND (c < a + b)



```
@Provide
Arbitrary<ABC> validTriangleGenerator() {
    Arbitrary<Integer> normalSide1 = Arbitraries.integers();
    Arbitrary<Integer> normalSide2 = Arbitraries.integers();
    Arbitrary<Integer> normalSide3 = Arbitraries.integers();
    return Combinators.combine(normalSide1, normalSide2, normalSide3).as(ABC::new)
            .filter(k -> (k.a < k.b + k.c) && (k.b < k.a + k.c) && (k.c < k.a + k.b));
}
```

Ref.:

1 - https://jqwik.net/docs/current/user-guide.html#combining-arbitraries-with-combine

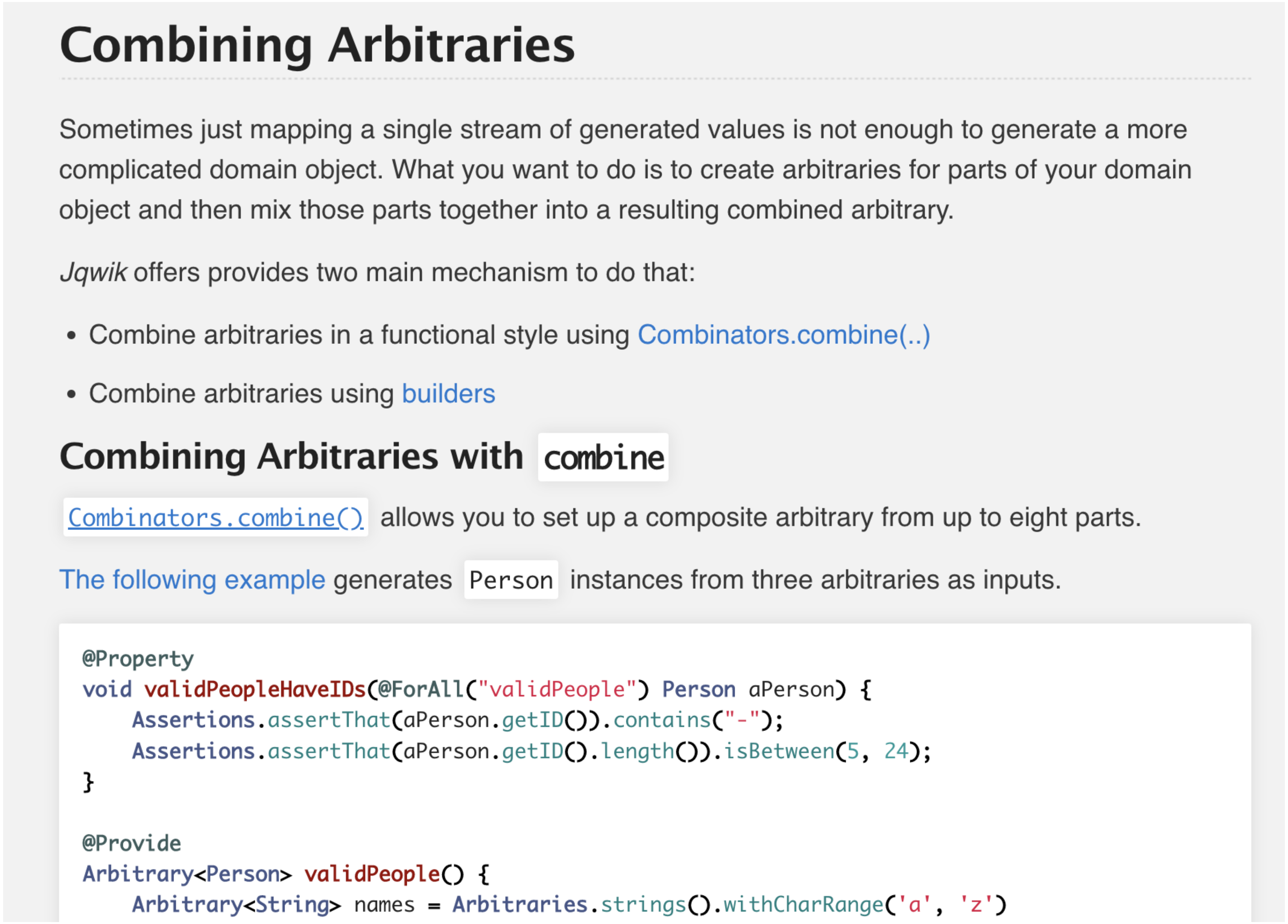
2 - https://jqwik.net/docs/current/user-guide.html#filtering-combinations

PBT: Filter combinations

If you want to combine two or more arbitraries, but not all combinations of values makes sense: filter the combinations.

Ex: combining a pair of values from the same domain, but the values should never be the same: you add a filter step between `combine(..)` and `as(..)` so you sort out unwanted combinations

```
@Property
void pairsCannotBeTwins(@ForAll("digitPairsWithoutTwins") String pair) {
    Assertions.assertThat(pair).hasSize(2);
    Assertions.assertThat(pair.charAt(0)).isNotEqualTo(pair.charAt(1));
}

@Provide
Arbitrary<String> digitPairsWithoutTwins() {
    Arbitrary<Integer> digits = Arbitraries.integers().between(0, 9);
    return Combinators.combine(digits, digits)
                      .filter((first, second) -> first != second)
                      .as((first, second) -> first + "" + second);
}
```



Combining Arbitraries

Sometimes just mapping a single stream of generated values is not enough to generate a more complicated domain object. What you want to do is to create arbitraries for parts of your domain object and then mix those parts together into a resulting combined arbitrary.

Jqwik offers provides two main mechanism to do that:

- Combine arbitraries in a functional style using Combinators.combine(..)
- Combine arbitraries using builders

Combining Arbitraries with `combine`

`Combinators.combine()` allows you to set up a composite arbitrary from up to eight parts.

The following example generates `Person` instances from three arbitraries as inputs.

```
@Property
void validPeopleHaveIDs(@ForAll("validPeople") Person aPerson) {
    Assertions.assertThat(aPerson.getID()).contains("-");
    Assertions.assertThat(aPerson.getID().length()).isBetween(5, 24);
}

@Provide
Arbitrary<Person> validPeople() {
    Arbitrary<String> names = Arbitraries.strings().withCharRange('a', 'z')
```

Example 4: Testing the Basket class

How to create a bunch of jqwik `Actions` implementing `Action interface`

Then let jqwik generate a random sequence of action to test the method

Practice exercise

Reference: chapter 5 pag. 129

Example 5: Creating complex domain objects

When you want to create complex objects, you can use jqwik `Combinators` feature.

```
public class Book {

  private final String title;
  private final String author;
  private final int qtyOfPages;

  public Book(String title, String author, int qtyOfPages) {
    this.title = title;
    this.author = author;
    this.qtyOfPages = qtyOfPages;
  }

  // getters...
}
```

Example 5: Creating complex domain objects

The `Combinators` API lets us combine different generators to build a complex object.

It is enough to build specific `Arbitrary`s for each of the attributes of the complex class we want to build:

- `Arbitrary<String>` for the title,
- `Arbitrary<String>` for the author,
- `Arbitrary<Integer>` for the number of pages.

After that, we use the `Combinators.combine()` method, which receives a series of `Arbitrarys` and returns an `Arbitrary` of the complex object.

Practice exercise. Reference: chapter 5 pag. 136

Example 5: Creating complex domain objects

```
public class BookTest {

  @Property
  void differentBooks(@ForAll("books") Book book) {
    // different books!
    System.out.println(book);

    // write your test here!
  }

  @Provide
  Arbitrary<Book> books() {
    Arbitrary<String> titles = Arbitraries.strings().withCharRange(
      ➥ 'a', 'z')
        .ofMinLength(10).ofMaxLength(100);
    Arbitrary<String> authors = Arbitraries.strings().withCharRange(
      ➥ 'a', 'z')
        .ofMinLength(5).ofMaxLength(21);
    Arbitrary<Integer> qtyOfPages = Arbitraries.integers().between(
      ➥ 0, 450);

    return Combinators.combine(titles, authors, qtyOfPages)
        .as((title, author, pages) -> new Book(title, author, pages));
  }
}
```

Instantiates one arbitrary for each of the Book's fields

Combines them to generate an instance of Book

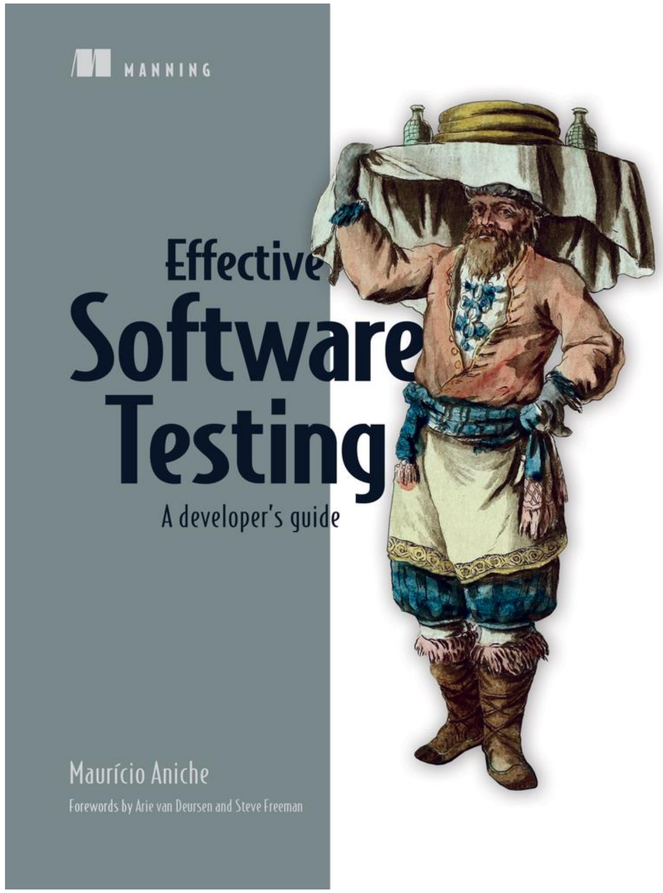
Property-based testing wrap up

- PBT requires lots of **creativity**: you have to find ways to generate random data, expressing the property, etc.
- In PBT instead of producing concrete examples, we **express the property that should hold** for that method.
- PBT is one more tool to have in your belt. Sometimes traditional example-based testing is enough. Sometimes is good to **mix** techniques.
- Jqwik will do its best to mix **random** data points with **edge** cases (but you have to express property and boundary correctly).
- The input data you pass to the test method should be **fairly distributed** (triangle example).

Reference book:

Effective Software Testing. A developer's guide. Mauricio Aniche. Ed. Manning. (**Chapter 5**)

.


MANNING
Effective Software Testing
A developer's guide
Maurício Aniche
Forewords by Arie van Deursen and Steve Freeman

References

- Jqwik – Propert—based testing in java: https://jqwik.net/docs/current/user-guide.html
- About PBT:
  - https://jqwik.net/property-based-testing.html
- Java TreeSet: https://www.javatpoint.com/java-treeset
- Combining Arbitraries: https://jqwik.net/docs/current/user-guide.html#combining-arbitraries
- Collecting and Reporting Statistics: https://jqwik.net/docs/current/user-guide.html#collecting-and-reporting-statistics

SERLAB
Software Engineering Research

Azzurra Ragone

Department of Computer Science- Floor VI – Room 616
Email: azzurra.ragone@uniba.it