TESTING

Know for Sure with Property-Based Testing

How do you test your code against thousands of values?

by Johannes Link

August 19, 2019

Writing unit tests with a tool such as JUnit is an essential technique to ensure the quality of your code. However, when a function requires many test cases to check for all potential problems, testing becomes burdensome and error-prone. *Property-based testing* (PBT) can rescue and relieve you from writing dozens of test cases. In this article, I explain what PBT is, how to use PBT on the JUnit 5 platform, and how it can enhance and sometimes even replace example-based testing.

Example-Based Testing

Let's say you're working on a class `Aggregator` that's supposed to `receive` individual measurement values and count their respective frequency in a tally. To check if your object works as expected, you can use the following simple JUnit 5 test:

```
import java.util.*;
import org.junit.jupiter.api.*;

class AggregatorTests {

    @Test
    void tallyOfSeveralValues() {
        Aggregator aggregator = new Aggregator();
        aggregator.receive(1);
        aggregator.receive(2);
        aggregator.receive(3);
        aggregator.receive(2);

        Map<Integer, Integer> tally = aggregator.ta
        Assertions.assertEquals(1, (int) tally.get(
        Assertions.assertEquals(2, (int) tally.get(
        Assertions.assertEquals(3, (int) tally.get(
    }
}
```

This kind of test is often called *example-based* because it uses a concrete input example and checks whether the produced output matches expectations for a specific situation. Most developers have written similar tests for years or even decades—usually with good success for detecting common programming errors. There is one thought, though, that has always been nagging in the back of my mind: How can I be confident that `Aggregator` also works for five measurements? Should I test with 5,000 elements, with none, or with negative numbers? On a bad day, there is no end to the amount of doubt I have about my code—and about the code of my fellow developers.

Properties

You can, however, approach the question of correctness from a different angle: Under what preconditions and constraints (for example, the range of input parameters) should the functionality under test lead to particular postconditions (results of a computation), and which invariants should never be violated in the course?

The combination of preconditions and qualities that are expected to be present is called a *property*.

Let's formulate some properties for `Aggregator` in plain English:

- All measured values show up as keys in a tally.
- Values that are never measured do not show up in a tally.
- The sum of all tally counts is equal to the number of received measurements.
- The order of measuring does not change a tally.

All four sentences make rather general statements except for the last one, which requires at least two measurements to make sense; each property can be applied to any list of elements, regardless of whether the list is empty or of considerable length. The measurements could fill the whole range of type `Integer` and they could be duplicated.

Automating Property Testing

How can properties be used for automatic testing? The statements themselves seem to be translatable into code. Formulating the first property as a Java method is straightforward, as follows:

```
boolean allMeasuredValuesShowUpAsKeys(List<Integer>
    Aggregator aggregator = new Aggregator();
    measurements.forEach(aggregator::receive);
    return measurements.stream()
        .allMatch(m -> aggregator.tally().containsK
}
```

What's missing for a real automated test is a way to generate a set of input lists, feed those lists to the method, and fail the test as soon as the condition returns `false`. This could all be done using vanilla JUnit but it would require a lot of dedicated generation logic. And it can be done for simple cases but will become burdensome as soon as the values to generate become more complicated and more domain-specific. That's why I use another test engine: jqwik.

If you use jqwik, the previous code needs only the following minor tweaking to become an executable property:

```
import java.util.*;
import net.jqwik.api.*;

class AggregatorProperties {

    @Property
    boolean allMeasuredValuesShowUpAsKeys(
        @ForAll List<Integer> measurements)
   {
        Aggregator aggregator = new Aggregator();
        measurements.forEach(aggregator::receive);
        return measurements.stream()
         .allMatch(m -> aggregator.tally().containsK
   }
}
```

Let's take a closer look at this code:

- A property is a method inside a container class. The method should have an informative name. `allMeasuredValuesShowUpAsKeys`

is a reasonable summary of the property's intent.

- To mark a method as a property method, it must be annotated with `@Property` so that IDEs and build tools will recognize it as such—that is, if they support the JUnit platform.
- Adding parameters and annotating them with `@ForAll` tells jqwik that you want the framework to generate instances for you. A parameter's type, `List<Integer>`, is considered to be the fundamental precondition.
- Returning a boolean value is the simplest form of communicating a property's necessary condition. Alternatively you can use any assertion library such as AssertJ or JUnit 5 itself.

Running a successful jqwik property is as quiet as running a successful JUnit test. If it is not instructed otherwise, jqwik will invoke each property method 1,000 times with different input parameters. If needed, you can tune the number as high or as low as you want.

Failing Properties

To see a property fail, let's look at the third property from the list (the sum of all tally counts is equal to the number of received measurements):

```
@Property
boolean sumOfAllCountsIsNumberOfMeasurements(
          @ForAll List<Integer> measurements)
    {
        Aggregator aggregator = new Aggregator();
        measurements.forEach(aggregator::receive);
        int sumOfAllCounts =
            aggregator.tally().values()
         .stream().mapToInt(i -> i).sum();
      return sumOfAllCounts == measurements.size();
}
```

Currently, the tallying functionality contains a bug, so any value is counted only once. In this case, jqwik should find an example that will detect this bug. And indeed it does. Here is the output:

```
org.opentest4j.AssertionFailedError:
  Property [AggregatorProperties:sumOfAllCountsIsNu
    falsified with sample [[0, 0]]

                                |--------------------
tries = 11                      | # of calls to prope
checks = 11                     | # of not rejected c
generation-mode = RANDOMIZED    | parameters are rand
seed = -2353742209209314324     | random seed to repr

sample = [[0, 0]]
originalSample = [[2068037359, -1987879098, 1588557
```

That's quite a lot of information: You can see the number of test attempts (`tries`), the number of actually run tests (`checks`), the random seed (`seed`), the falsifying sample (`sample`), and other information that can sometimes be useful.

In this example, jqwik succeeded in revealing a bug by generating lists that have duplicate elements. If you had written the example tests yourself, you might or might not have thought about this variant. By using a PBT library, you gained test depth without needing to think up additional examples. You must, however, be aware of what property-based testing does not do: It cannot prove that a property is correct. All it does is try to find examples that falsify a property.

Integration of jqwik with JUnit 5

jqwik is not a standalone framework. Rather, it is a test engine that hooks into JUnit 5. JUnit 5 not only provides a modernized approach for writing and executing tests, but it is designed to be a platform for a large spectrum of different test engines. The advantage of jqwik's design is that IDEs and build tools need to integrate only the JUnit platform, not the individual test engines. This is a big advantage for test-engine developers who don't need to bother with aspects such as public APIs for discovering and running their test specifications. Test engines automatically inherit IDE and build-tool support.

Moreover, the platform enables developers to use any number of engines in parallel. All you have to do is add a single additional dependency in your Maven or Gradle setup. Currently, the two most used Java IDEs—IntelliJ and Eclipse—come with excellent support for JUnit 5, as do Gradle and Maven.

More jqwik Features

Currently, jqwik has all the essential features that property-based testers require. For example, many standard types can be generated out of the box. All `Number`, `String`, `Character`, and `Boolean` types as well as the built-in container types `List`, `Set`, `Stream`, `Iterator`, and `Optional` are recognized. Thus, you can have a parameter of type `Set<List<String>>` and jqwik will automatically generate sets of lists of strings for you.

There are many annotations that allow you to influence value generation directly in a property method's signature. Therefore, you might enhance the `Set<List<String>>` type like this:

```
@ForAll @Size(3) Set<List<
     @CharRange(min='a', max='f') String>>
        aSetOfListsOfStrings
```

Making that change generates only sets with a length of 3 that themselves contain lists with strings that use only the characters `a` through `f`.

The generation of values is not purely random. It considers typical edges and corner cases such as empty strings, the number 0, maxima and minima of ranges, and a few others. If your constraints are tight enough, jqwik will even exhaustively generate all possible value combinations.

Programmatic Value Generation

Sometimes you're dealing with classes for which jqwik does not have default generators. On other occasions, the domain-specific constraints of a primitive type are so specific that existing annotations are not powerful enough. In these cases, you can delegate the provision of parameter generators to another method in your test container class. The following example shows how to generate German postal codes by using a provider method:

```
@Property @Report(Reporting.GENERATED)
void letsGenerateZipCodes(@ForAll("germanZipCode") S

@Provide
Arbitrary<String> germanZipCode() {
    return Arbitraries.strings()
        .withCharRange('0', '9')
        .ofLength(5)
        .filter(z -> !z.startsWith("00"));
}
```

The `String` value of the `@ForAll` annotation is a reference to a method's name within the same class. This method must be annotated with `@Provide` and must also return an object of type `@Arbitrary<T>`, where `T` is the static type of the parameter to be provided.

> One problem that comes with random generation is that the relationship between a randomly chosen falsifying example and the problem underlying the failing property is often buried under a lot of noise.

Parameter provision methods usually start with a static method call to `Arbitraries` and are often followed by one or more filtering, mapping, or combining actions, as described in the next section.

Filtering, Mapping, and Combining

As the base type of all value generation, the `Arbitrary` class comes with a few default methods that can be used to modify generation behavior. You usually start with one of the static generator functions of the class `Arbitraries`. Most generator functions return a specific subtype of `Arbitrary` that gives you additional configuration possibilities through a fluent interface.

Let's assume that you want to generate integers between 1 and 300 that are multiples of 6. Here are two ways to achieve that:

```
Arbitraries.integers()
           .between(1, 300)
           .filter(anInt -> anInt % 6 == 0)
```

Or

```
Arbitraries.integers().between(1, 50)
           .map(anInt -> anInt * 6)
```

Which way is better? Sometimes it is only a matter of style or readability. At other times, however, the way you choose can influence performance. If you compare the two previous options, you can see the former is closer to the given specification, but it will—through filtering—throw away five out of six of all the generated values. The latter is, therefore, more efficient but also less comprehensible. Usually, generating primitive values is so fast that readability trumps efficiency.

Real domain objects often have several distinct and mostly unrelated parts—for example, a `Person` would need a first name and last name. That's why it can be a good idea to start from unrelated base generators and combine them. The following example creates an `Arbitrary` for domain class `Person` by combining two `Arbitrary` entities into one:

```
@Provide
Arbitrary<Person> validPerson() {
    Arbitrary<String> firstName = Arbitraries.strin
        .withCharRange('a', 'z')
        .ofMinLength(2).ofMaxLength(10)
        .map(this::capitalize);
    Arbitrary<String> lastName = Arbitraries.string
        .withCharRange('a', 'z')
        .ofMinLength(2).ofMaxLength(20);
    return Combinators
        .combine(firstName, lastName).as(Person::ne
}
```

You can combine up to eight `Arbitrary` entries in one go by using this technique. If you want, you can register your own `Arbitrary` entries so that they will be applied automatically to all parameters of your domain type.

The Importance of Shrinking

One problem that comes with random generation is that the relationship between a randomly chosen falsifying example and the problem

underlying the failing property is often buried under a lot of noise. A simple example can illustrate this concern:

```
@Property(shrinking = ShrinkingMode.OFF)
boolean rootOfSquareShouldBeOriginalValue(
     @Positive @ForAll int anInt )
{
    int square = anInt * anInt;
    return Math.sqrt(square) == anInt;
}
```

The property states the trivial mathematical concept that the square root of a squared value should be equal to the original value. The first line switched off shrinking by using the `shrinking` annotation attribute. Running this property fails with a message similar to this:

```
originalSample = [1207764160],
sample = [1207764160]

org.opentest4j.AssertionFailedError:
    Property [rootOfSquareShouldBeOriginalValue]
        falsified with sample [1207764160]
```

The failing sample found by jqwik is random. The number itself does not give you an obvious hint about the cause of the failure. Even the fact that it is rather large might be a coincidence. At this point, you will either add additional logging or start up the debugger to get more information about the problem at hand.

PBT is based on the idea that you can find general and desired properties for functions, components, and whole programs, and often those properties can be falsified by the randomized generation of test data.

Let's take a different route by turning shrinking on with (`ShrinkingMode.FULL`) and rerunning the property. The failure will be the same, but reporting will show a change of the found falsifying `sample`:

```
sample = [46341]
originalSample = [1207764160]
```

The number 46,341 is much smaller and it is different from the original sample. After failing with 1,207,764,160, jqwik kept on trying to find a simpler example that would also fail. This searching phase is called *shrinking* because it starts with the original sample and tries to make it smaller and smaller.

So what's the special thing about 46,341 in this case? As you might have guessed, the square of 46,341 equals 2,147,488,281, which is just a bit larger than `Integer.MAX_VALUE` and will, therefore, lead to an integer overflow. Conclusion: The property above holds only for integers up to the square root of `Integer.MAX_VALUE`.

Shrinking is an important topic in PBT because it makes the analysis of many failed properties much easier. It also reduces the amount of indeterminism in PBT. Implementing good shrinking, however, is a complicated task. From a theoretical perspective, you face a search problem with a potentially very large search space. Because deep search is time-consuming, many heuristics are applied to make shrinking both effective and fast.

Patterns for Finding Properties

When you take your first steps with PBT, finding suitable properties can feel like a challenging task. Compared to typical property examples, identifying real-world properties requires a different kind of thinking. A set of useful patterns to guide your property detection can be handy. Luckily, you do not need to discover all things on your own. PBT has been around for a while and there is a small but well-known collection of property-based testing patterns. My personal list is certainly incomplete, but here are some typical sources:

Business rule as property. Sometimes the domain specification itself can be interpreted and written as a property. Consider a business rule such as: For all customers with a yearly turnaround greater than X € we give an additional discount of Y percent, if the invoice amount is larger than Z €. This can be straightforwardly translated into a property by using arbitraries for X and Z and checking that the calculated discount is indeed Y.

Inverse functions. If a function has an inverse function, applying the function first and the inverse function second should return the original input.

Idempotent functions. The multiple application of an idempotent function should not change results. Ordering a list a second time, for example, shouldn't change it.

Invariant functions. Some properties of your code do not change after applying your logic. For example, sorting and mapping should never change the size of a collection, and after filtering out values from a list, the remaining values should still be in the original order.

Commutativity. If a set of functions is commutative, a change of order in applying the functions should not change the final result. For example, sorting and then filtering should have the same effect as filtering and then sorting.

A test oracle. Sometimes you know an alternative implementation of the function under test. You can then use this implementation as a *test oracle*: Any result of using the function should be the same for both the original and alternative implementations. Here are some sample alternatives:

- Simple and slow versus complicated but fast
- Parallel versus single-threaded
- Self-made versus commercial
- Old (prerefactoring) versus new (postrefactoring)

Hard to compute, but easy to verify. Some logic is hard to execute but easy to check. Consider, for example, the effort for finding prime numbers versus checking a prime number.

Induction (that is, solve a smaller problem first). You might be able to divide your domain check into a base case and a general rule derived from that base case.

Stateful testing. Especially in the object-oriented world, an object's behavior can often be described as a state machine with a finite set of states and actions to change state. Exploring the space of state transitions is an important use case for PBT, and that's why jqwik provides special support for it.

Fuzzing. Code should never explode, even if you feed it with lots of diverse and unforeseen input data. Thus, the main idea of this pattern is to generate a large variety of input, execute the function under test, and check the following:

- No exceptions occur, at least no unexpected ones.
- There are no 5xx return codes for HTTP requests; maybe you even require 2xx status all the time.
- All return values are valid.
- Runtime is under an acceptable threshold.

Fuzzing is often done in retrospect when you want to scrutinize the robustness of existing code and systems.

Applying these patterns to your code requires practice. The patterns can, however, be a good starting point for overcoming test writer's block. The more often you think about properties of your own code, the more opportunities you will recognize to derive property-based tests from your example-based tests. Sometimes they can serve as a complement; sometimes they can even replace the old tests.

Conclusion

PBT is not a new technique; it has been used effectively in languages such as Haskell and Erlang for more than a decade. PBT is based on the idea that you can find general and desired properties for functions, components, and whole programs, and often those properties can be falsified by the randomized generation of test data.

jqwik is a JVM-based property test engine. Because it is built for the JUnit 5 platform, integration into all modern IDEs and build tools is seamless. If you are not using JUnit 5 (yet), a couple of alternatives are available. This article only scratches the surface of PBT. If you want to dive a bit deeper, you might start with this blog series.

Also in This Issue

Johannes Link

Johannes Link (@johanneslink) has been developing software professionally for almost 25 years. As early as 2001, he got hooked on test-driven development and wrote a book about it. He was one of the core committers to JUnit 5 in its first year. He is also the main developer of jqwik.

Share this Page

Contact
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us
Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials
Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events
Acquisitions
Blogs
Events
Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices