Thoughts on Random Number Generators - QuickCheck

frankhjung@linux.com

15 January 2026

Figure 1: Photo by Frank H Jung

In part 1 of this series, we explored pseudo-random values. These are values that are statistically random, but are derived from a known starting point and is typically repeated over and over. In this article we explore how random values can be used in testing. You may already be familiar with randomness in test invocation. For instance JUnit5 provides an annotation to randomise the order of test execution. Here, however, we are looking at a style of testing that uses randomly generated input values that test *properties* of your code. This is known as "Property Based Testing".

You may be asking: *Why would you use random values in testing? Doesn't that defeat the whole purpose of unit testing, where known values are tested to known responses?* Well, no. Often it is hard to think of suitable positive and negative test cases that exercise your code. In addition: what if many randomly selected tests can be automatically run? And what if these tests cover many different input values? And what if instances where tests fail are automatically recorded so they can be reported and replayed later? These are just some of the benefits of this approach to testing.

Property-based testing verifies your program code using a large range of relevant inputs. It does this by generating a random sample of valid input values. Perhaps an example may help. Given a utility method to convert a string field into uppercase text, a unit test would use some expected values. In pseudo-code this may look like:

```
Given "abc123" then expect "ABC123"
Given "ABC" then expect "ABC"
Given "123" then expect "123"
```

In comparison, property-based tests look at the behaviour of any field that matches the input type. In pseudo-code this looks like:

```
For any lowercase alphanumeric string then expect the same string but in uppercase.
For any uppercase alphanumeric string then expect the same string, unchanged.
For any non-alphabetic string then expect the same string, unchanged.
```

The "for any" is where the randomness comes in.

This article will review where these ideas came from, before outlining the core principles of property-based testing. It will introduce the concepts of Generators and Shrinkage and discuss approaches to reproducing tests results.

History

These concepts aren't new. Tools like Lorem Ipsum have been around since the 1960s to model text.

Kent Beck developed a unit testing framework for Smalltalk in 1989. Those tests had to be hand crafted. This introduced a number of key concepts that we now take for granted. It organised and provided recipes for unit tests. For each test case, the test data was created then thrown away at the end. Test cases were aggregated into a test suite. The test suite formed part of a framework that also produced a report — an example of what is known as literate programming.

In 1994, Richard Hamlet wrote about Random Testing . Hamlet posited that computers could efficiently test a "vast number" of random test points. Another benefit he identified was that random testing provided a "statistical prediction of significance in the observed results". This last point is somewhat technical. In essence it describes the ability to quantify the significance of a test that does *not* fail. In other words: is this just testing trivial cases?

A few years later, in 1999, the influential paper by Claessen and Hughes, QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, provided a whole new way to run tests using randomised values. Their paper was written for the functional programming language, Haskell. It inspired a suite of property-based testing tools for many other languages. A list of current implementations appears on the QuickCheck Wikipedia page.

Introducing Property Based Testing

The core principle of property-based testing is that for a function or method, any valid input should yield a valid response. Likewise, any input outside this range should return an appropriate failure. Compare this to how systematic tests are normally written: Given a *specific* input, check the program's return value. Therein lies the problem: you need to be sure you have chosen correct and *sufficient* input values to test your code. The tools we use to check test coverage do not check the *adequacy* of your tests — just that you have a test for a control flow path. So the quality of a test is dependent upon the quality of the inputs. Property-based testing provides tools to test using randomly generated values selected over the range of input. This changes the focus of our tests. We concentrate on the properties of functions under test. i.e. What the inputs are and what the outputs are expected to be. Testing the properties of a function or method over a large range of values can help find bugs otherwise ignored in specific unit tests. We have experienced this first hand. It is a true “aha” moment when these tests uncover a use-case with input we hadn’t thought of.

In summary:

- With unit testing we provide fixed inputs (e.g. 0,1,2,. . .) and get a fixed result (e.g. 1,2,4,. . .).
- With property-based testing we provide a declaration of inputs (e.g. all non-negative `ints`) and declaration of conditions that must be held (e.g. result is an `int`).

At its core, property-based testing requires the production of randomised input test values. These test values are produced using *generators*.

Generators

Random values are produced using *generators*. These are specific functions that produce a random value. Some common generators are used to manufacture booleans, numeric types (e.g. floats, ranges of integers), characters and strings. Both QuickCheck and JUnit-QuickCheck provide many generators. Once you have a primitive generator, you can then compose these into more elaborate generators and structures like lists and maps, or other bespoke structures.

Apart from custom values, you may also want a custom distribution. Random testing is most effective when the values being tested closely match the distribution of the actual data. As the provided generators know nothing of your data, they will typically produce a uniform distribution. To control the data distribution you will need to write your own generator. Luckily, this is not a difficult task. The test tools we have used (Haskell:QuickCheck, Java:JUnit-QuickCheck and Python:Hypothesis) have rich libraries of generators that can be easily extended.

Shrinkage

Generators can produce large test values. When a failure has been detected it would be nice to find a smaller example. This is known as *shrinkage*.

On failure, QuickCheck reduces the selection to the minimum set. So, from a large set of test values, QuickCheck finds the minimal case that fails the test. In practice, what this does is concentrate tests to the extremes of an input value. However, this behaviour can be modified by the *generator*.

Shrinkage is an important feature of property based testing. Having an example of failure is good. Having a minimal example of failure is better. With a minimal example you are more likely to understand the reasons for the failure.

Test Reproduction

Random testing is useful, but once a failure has been identified and fixed, we would like to repeat the failed tests to ensure they have been resolved. Tools such as Python’s Hypothesis record all failed tests. Future runs automatically include prior failed tests.

Other tools such as Java’s JUnit-QuickCheck allow the tests to be repeated by specifying a random seed. When a test fails, the random seed used to generate that test is reported, and can then be used to reproduce the same test inputs.

Code Examples

So, what does this look like in real code? I have used Java for development of digital solutions, so the first examples are based on the Java JUnit-QuickCheck package.

The following generator will create a alphanumeric “word” with length of between 1 and 12 characters.

```java
import com.pholser.junit.quickcheck.generator.GenerationStatus;
import com.pholser.junit.quickcheck.generator.Generator;
import com.pholser.junit.quickcheck.random.SourceOfRandomness;
import java.util.stream.IntStream;

/** Generate alpha-numeric characters. */
public final class AlphaNumericGenerator extends Generator<String> {

  /** Alphanumeric characters: "0-9A-Za-z". */
  private static final String ALPHANUMERICS =
      "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

  /** Maximum word length. */
  private static final int MAX_WORD_LENGTH = 11;

  /** Inherit form super class. */
  public AlphaNumericGenerator() {
    super(String.class);
  }

  /** Generate a alphanumeric word of length 1 to 12 characters. Do not create null words. */
  @Override
  public String generate(final SourceOfRandomness randomness, final GenerationStatus status) {
    final int stringSize = randomness.nextInt(MAX_WORD_LENGTH) + 1; // non-empty words
    final StringBuilder randomString = new StringBuilder(stringSize);
    IntStream.range(0, stringSize)
        .forEach(
            ignored -> {
              final int randomIndex = randomness.nextInt(ALPHANUMERICS.length());
              randomString.append(ALPHANUMERICS.charAt(randomIndex));
            });
    return randomString.toString();
  }
}
```

(source)

To use this generator in a unit test:

```java
/**
  * Test alphanumeric word is same for stream as scanner using Alphanumeric generator. Trials
  * increased from the default of 100 to 1000.
  *
  * @param word a random alphanumeric word
  */
@Property(trials = 1000)
public void testAlphanumericWord(final @From(AlphaNumericGenerator.class) String word) {
  assertEquals(1, WordCountUtils.count(new Scanner(word)));
  assertEquals(1, WordCountUtils.count(Stream.of(word)));
}
```

(source)

Here we are using our custom generator, and have increased the trials to 1000 from the default of 100. The expected property of our word count utility is that given this input string, its output would indicate that it counted one word.

The following code uses this generator to build a list of strings that are delimited by a space. The code to be tested contains two word count methods accepting different input types. Using our custom generator we can compose test data for both input types. Then test to see if the word count methods agree:

```
/**
  * Test a "sentence" of alphanumeric words. A sentence is a list of words separated by a space.
  *
  * @param words build a sentence from a word stream
  */
@Property
public void testAlphanumericSentence(
    final List<@From(AlphaNumericGenerator.class) String> words) {
  final String sentence = String.join(" ", words);
  assertEquals(
      WordCountUtils.count(new Scanner(sentence)), WordCountUtils.count(Stream.of(sentence)));
}
```

(source)

At Marlo we also use Ansible for automation, with some custom modules written in Python. An excellent QuickCheck library for Python is Hypothesis. An equivalent generator to the Java example above is the text strategy. Used in a test, it looks like:

```
@given(text(min_size=1, max_size=12, alphabet=ascii_letters + digits))
def test_alphanumeric(a_string):
    """
    Generate alphanumeric sized strings like:
        'LbkNCS4xl2Xl'
        'z3M4jc1J'
        'x'
    """
    assert a_string.isalnum()
    a_length = len(a_string)
    assert a_length >= 1 and a_length <= 12
```

(source)

While the above are trivial examples, they do demonstrate how this style of testing is a valuable complement to systematic tests. They enable a larger number of test cases to run against your code. The style of tests are different, in that they focus on the generalised behaviour of code rather than specific use cases. This makes them powerful addition to a test suite.

Summary

In this article we took a brief look at the features of property-based testing, which uses random inputs to improve the quality and coverage of tests.

However, it is important to note that property-based tests don't replace unit tests. Instead they should be used to augment your existing tests with values that you may not have thought about. Generating a large number of tests may, however, give a false sense of security if most of these test cases are trivial. So, choosing the correct inputs, whether randomly generated or systematically selected is important.

Property-based tests are easy to write and can help identify bugs that traditional testing approaches might miss. So, why not use randomness to your advantage?

Links

There are many resources online resources available if you want to learn more:

- Beyond Unit Tests
- JUnit5

- JUnit-QuickCheck a Java QuickCheck style tool (GitHub)
- Lorem Ipsum
- Property Testing
- Python Hypothesis
- QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs by Koen Claessen & John Hughes (1999) (PDF)
- QuickCheck: As a test set generator
- QuickCheck: A tutorial on generators
- QuickCheck: Automatic testing of Haskell programs
- QuickCheck (Wikipedia)
- Random Testing by Richard Hamlet (1994) (PDF)
- Simple Smalltalk Testing: With Patterns by Kent Beck (1989)
- Source for this article (GitHub)