Thoughts on Random Number Generators - QuickCheck

Frank Jung

15 October 2020

Figure 1: Photo by Frank H Jung

Part 1 of this series explored pseudo-random values—statistically random values derived from a known starting point. This article explores using random values in testing. Randomness in test invocation is common; for instance, JUnit5 provides an annotation to randomise the order of test execution. This article, however, examines a testing style using randomly generated input values to test *properties* of code, known as "Property Based Testing".

Why use random values in testing? Defining suitable positive and negative test cases to exercise code is often difficult. Automating the execution of many randomly selected tests covers a broader range of input values. Furthermore, recording and reporting failing tests allows for replay and debugging.

Property-based testing verifies program code using a large range of relevant inputs by generating a random

sample of valid values. For example, given a utility method to convert a string field into uppercase text, a unit test uses expected values:

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

In comparison, property-based tests examine the behaviour of any field matching the input type:

```
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.
```

Randomness provides the "for any" component.

This article reviews the history of these ideas, outlines the core principles of property-based testing, introduces Generators and Shrinkage, and discusses approaches to reproducing test results.

History

These concepts are not new. Tools like Lorem Ipsum have modeled text since the 1960s.

Kent Beck developed a unit testing framework for Smalltalk in 1989. These hand-crafted tests introduced key concepts now considered standard, organising and providing recipes for unit tests. For each test case, the framework created test data and discarded it upon completion. Aggregated test cases formed a test suite, part of a framework producing a report—an example of literate programming.

In 1994, Richard Hamlet wrote about Random Testing. Hamlet posited that computers could efficiently test a "vast number" of random test points. He also identified that random testing provided a "statistical prediction of significance in the observed results". Essentially, this quantifies the significance of a test that does *not* fail, determining whether the test merely exercises trivial cases.

In 1999, the influential paper by Claessen and Hughes, QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, provided a new method for running tests using randomised values. Written for the functional programming language Haskell, it inspired 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, and any input outside this range should return an appropriate failure. Systematic tests typically check the return value for a *specific* input. However, this requires selecting correct and *sufficient* input values. Tools checking test coverage verify the existence of a test for a control flow path, not the *adequacy* of the inputs.

Property-based testing uses randomly generated values selected over the input range, focusing on the properties of functions under test (i.e., inputs and expected outputs). Testing function properties over a large range of values often uncovers bugs ignored by specific unit tests. Uncovering edge cases with unexpected inputs often reveals overlooked bugs.

In summary:

- Unit testing provides fixed inputs (e.g., 0, 1, 2, ...) and yields a fixed result (e.g., 1, 2, 4, ...).
- Property-based testing declares inputs (e.g., all non-negative `int`s) and conditions that must hold (e.g., result is an `int`).

Property-based testing requires the production of randomised input test values using *generators*.

Generators

Generators are specific functions producing random values. Common generators manufacture booleans, numeric types (e.g., floats, ranges of integers), characters, and strings. Both QuickCheck and JUnit-QuickCheck provide many generators. Primitive generators can be composed into elaborate generators and structures like lists, maps, or bespoke structures.

Beyond custom values, custom distributions are often necessary. Random testing is most effective when test values closely match the actual data distribution. Standard generators typically produce a uniform distribution. Controlling the data distribution requires writing a custom generator. Fortunately, this is straightforward. Haskell:QuickCheck, Java:JUnit-QuickCheck, and Python:Hypothesis have rich libraries of extendable generators.

Shrinkage

Generators can produce large test values. Upon failure, finding a smaller example is desirable. This is known as *shrinkage*.

On failure, QuickCheck reduces the selection to the minimum set. From a large set of test values, it finds the minimal case failing the test. In practice, this concentrates tests on input value extremes. The *generator* can modify this behaviour.

Shrinkage is a critical feature; a minimal example facilitates understanding the failure's cause.

Test Reproduction

While useful, random testing requires reproducibility for debugging. Tools such as Python's Hypothesis record failed tests, automatically including them in future runs.

Other tools, such as Java's JUnit-QuickCheck, allow repetition by specifying a random seed. When a test fails, the system reports the random seed, allowing reproduction of the same test inputs.

Code Examples

The following examples demonstrate implementation using Java and the JUnit-QuickCheck package.

This generator creates an alphanumeric "word" with a length between 1 and 12 characters.

```
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:

```
/**
  * 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)

This example uses the custom generator and increases trials to 1000. The expected property of the word count utility is that given this input string, the output counts one word.

The following code uses this generator to build a list of strings delimited by a space. The code under test contains two word count methods accepting different input types. The custom generator composes test data for both input types to verify agreement between the word count methods:

```
/**
  * 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)

I use Ansible for automation, with custom modules written in Python. Hypothesis is a robust QuickCheck library for Python. An equivalent generator to the Java example above uses the text strategy:

```
@given(text(min_size=1, max_size=12, alphabet=ascii_letters + digits))
```

```python
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)

These examples demonstrate how this testing style complements systematic tests, enabling a larger number of test cases. Property-based tests focus on generalised code behaviour rather than specific use cases, making them a powerful addition to a test suite.

Summary

This article reviewed property-based testing, which uses random inputs to improve test quality and coverage.

Property-based tests do not replace unit tests; they augment existing tests with unforeseen values. Generating a large number of tests offers false security if test cases are trivial; choosing correct inputs, whether randomly generated or systematically selected, remains critical.

Property-based tests are efficient to write and help identify bugs that traditional testing approaches might miss.

Links

- Beyond Unit Tests
- JUnit5
- JUnit-QuickCheck (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)