Property Based Testing (sort of) with Spock

Bruce Cota

January 5, 2016

What is Property Based Testing?

- Property testing is a style of testing from the functional programming world.
- First Approximation: Property based testing is sort of like classical unit testing but with *randomly generated inputs*.

Example Based Testing

- Traditional Unit Testing is basically *example based testing*.
- A traditional unit test
  1. Constructs a test input. (The example.)
  2. Calls a method on the test input.
  3. Checks a postcondition on the result of the method.

ExampleBased Testing: Example

```
public void testAssembleDisassemble() {
  /* Get example */
    TestLocation testLocation = getTestLocation();

    /* Construct method call */
    TestLocationDtoAssembler assembler = new TestLocationDtoAssemblerImpl();
    TestLocationDto testLocationDto = assembler.assembleDto(testLocation);
    TestLocation testLocationAssembled = assembler.disassembleDto(testLocationDto);

    /* Assert postcondition */
    assertEquals(testLocation, testLocationAssembled);
}

/* Construct Example */
public static TestLocation getTestLocation() {
    TestLocation testLocation = new TestLocation();
    testLocation.setCapacity(100);
    ... /* Lots more */
}
```

Property Based Testing

- A (universally quantified) *property* about a function, say "f", is a statement about $f(x)$ that must be true for *any* input x of the correct type .
  - Actually we might add a precondition so the property only applies to *some* inputs x.
- A property based testing framework tries to *disprove* properties by randomly generating inputs. A property passes if it is *not* disproved.
- Typically, only a small number of inputs are generated. (We want the tests to finish quickly.)

Property Based Testing: Example

A simple example using Scalacheck

```
property("startsWith") = forAll {
  (x: String, y: String) =>
  (x + y).startsWith(x)
}
```

A property about string concatenation: for any strings, x and y, the string $x + y$ should start with x.

Examples vs Randomly Generated Inputs

Property testing frameworks know how to randomly generate strings, but

- For complex business objects, we have to write our own generators.
- We write generators *instead of* constructing example inputs.
- We still have to make identify and generate "corner cases".

Actual Example: SearchParameters

- I wrote a search method that takes, as input, 11 search parameters (`userId`, `locationId`, `creatorId`, `minStartDate`, `etc`, all of which were optional, but one of the first four had to be present.
- Which combinations should we test?
- It was much easier to randomly generate legal SearchParameter objects.

Why Does Property Based Testing Matter

- We can *specify* the behavior of a service with properties.
- (It might be impractical to *completely* specify, but we can get pretty close.)
- Essentially, a property-based testing framework generates tests from the specification!
- You can't even *try* to do this with traditional example based testing.

Test Driven Disappointment

Three things I hate about doing test driven development:

1. The tests pass, but my code has a bug

   "hmm.. Need more examples..."

2. The tests *fail*, but my code is *correct*!

   "The tests and I seem to disagree about what the code should *be doing"*

3. The tests are tedious to write and are hard to maintain.

   Part *of the problem is that the construction of example inputs, and getting this construction mixed up with the actual tests.*

Trying to Separate Example Generation from Test

Problem: There is no standard way to do this.

```
public void testAssembleDisassemble() {

  /* Get example */
    TestLocation testLocation = getTestLocation();

    /* Construct method call */
    TestLocationDtoAssembler assembler = new TestLocationDtoAs
    TestLocationDto testLocationDto = assembler.assembleDto(te
    TestLocation testLocationAssembled = assembler.disassemble

    /* Assert postcondition */
    assertEquals(testLocation, testLocationAssembled);
}

/* Construct Example */
public static TestLocation getTestLocation() {
    TestLocation testLocation = new TestLocation();
    testLocation.setCapacity(100);
    ... /* Lots more */
}
```

Test Driven Disappointment

My personal (repeated) experience with TDD

1. Write down an interface
2. *Tediously* write a bunch of tests (based on examples) for that interface.
3. Get half the tests to pass when I realize the interface isn't going to do exactly what I want.
4. Discard the tests and realize that I don't have time to write new ones, so proceed with no tests.

Property Driven Development

What if, before implementing a service, we take the following steps

1. Try to identify *all* (most of?) the properties the service must satisfy to be correct. (Requires a lot more thought in advance.)
2. Write these properties down as executable code. (Defer the work of generating inputs.)
3. Write appropriate generators for the test inputs.
4. Plug these into a property based testing framework.

Property Driven Development:

What would we get?

1. Specifications that are *documented* in code.
2. A (sorta) *complete* set of tests for the specifications.
3. The tests *are* the specification, so no more "disagreeing" with them.
4. A clear and consistent separation of input generation and tests.

Nothing's Perfect

But we're only human, so

- The specifications probably aren't really complete
- The coding of the specification may have bugs
- The quality of the tests depends on the random generation of test inputs.

Still, this could be a big improvement...

So Let's Try it!

Activation Objects

Consider a DAO for a big ugly business object called *Activation*

Never mind the structure of *Activation* – it has subobjects and is mapped to multiple tables in an RDBMS.

Activation Dao

```
public interface ActivationDao {

  /** Returns an id */
  String create(Activation activation);

  Activation read(String activationId);

  void update(Activation activation);

  /* Returns true if activation was found and deleted */
  boolean delete(String activationId);

  Set<Activation> search(SearchParameters parameters);
}
```

Properties for ActivationDao

How do I say something like "The read method should fail unless the activationId exists?"

- A *complete* set of properties probably requires tracking the state of the database.
- Some frameworks can generate random sequences of state changing operations!
- But it's easier (and probably good enough) to think of the effects of each operation on an empty database.

Properties for ActivationDao

Basic "happy path" properties I came up with

- After creating an activation, if I retrieve the activation by id I get the original activation back. (Specifies `create` and `find` in one property.)
- If I `update` an activation and then `find` by id, the changes are reflected.
- After calling `delete(activationId)`, `find` fails on that id.

We also need properties for the "unhappy paths". e.g. `find` should return null or throw an exception if the id doesn't exist, etc. but never mind right now.

Property Based Testing Frameworks

So what do we use to code this?

- `@Property` annotation in Junit — but it's not very flexible.
- JavaQuickCheck is the Property Based Testing Framework for Java – but I didn't find it in time.
- Spock – a Groovy-based framework that supports *data driven testing*. (Hence the "sort of" in the title.)

Spock Basics

- A Spock test is a file with the name "`*Spec.groovy`" extending the `Specification` class.
- A Spock test can have the usual fixture methods — `setup`, `cleanup`, etc.
- A *feature method* is a method that returns a boolean (or maybe just not `void`?)
- Feature methods can have *blocks* like `setup:`, `when:`, `expect:`, `where:`, and some more.

Spock Data Driven Testing Example

```
def "computing the maximum of two numbers"() {
    expect:
    Math.max(a, b) == c

    where:
    a << [5, 3]
    b << [1, 9]
    c << [5, 9]
}
```

- Note the truly self-documenting method name :)
- The `where:` block specifies the test inputs.
- This test is executed twice. Once where $a = 5,\ b = 1,\ c = 5$, and once where $a = 3, b = 9, c = 9$.

Data Driven Tests ⇒ Property Based Tests

My approach:

1. Create a `Generators` class with some fields that are lists of test data.
2. On initialization, the `Generators` initializes it's fields with random data.
3. The `setupSpec` method in the `*Spec.groovy` class initialize the `Generator`.

Initializing the Test Generator

```
class ActivationSpec extends Specification {

    //@Shared across feature methods
    @Shared ActivationDao dao
    @Shared ActivationTestGenerator generator

    //called once for all feature methods
    def setupSpec() {
        dao = //vague for now
        generator = new ActivationTestGenerator()
        generator.initialize(service)
    }
    ...
```

Testing Create/Find

```
def "After creating an activation, \
     the activation can be found by activationId"() {

  given:
  String id = service.create(activation)
  //to make  equality test work
  activation.activationId = id

  when: Activation newActivation = service.find(id);

  //This calls the Activation.equals method
  then: newActivation == activation

  where: activation << generator.activationsToCreate
}
```

Notes on Testing Create/Find

- `generator.activationsToCreate` is a randomly generated list of objects ready to create.
- I hijacked the equals method here:

```
then: newActivation == activation
```

I usually auto-generate them using Intellij Idea, and they work well for simple testing.

What if a test fails?

If we use randomly generated test data, how can we reproduce our failures?

What if a test fails?

Spock reports failures like this:

```
newActivation == activation
|             |  |
|             |  ProtoActivation{userId='e4d8be07-6b93-4985-ad94-ba4d349763a8', \
                   assessmentId='2342a70a-6576-4a98-8c1d-8fed757940dd', locationId='86', \
                   startDate=Mon Jan 04 17:53:03 EST 2016, \
                   endDate=Mon Jan 04 17:53:03 EST 2016, \
                   attributes={7ceb411a-c50f-4b38-97bc-bcaf1de86306=1984548939}}
|             false
Activation{activationId='d4ecb5e5-b90b-4972-b72b-22c8e3ba1b86', \
userId='e4d8be07-6b93-4985-ad94-ba4d349763a8', \
assessmentId='2342a70a-6576-4a98-8c1d-8fed757940dd', \
assessmentTitle='assessment 2342a70a-6576-4a98-8c1d-8fed757940dd', \
locationId='86', locationLabel='location 86', startDate=2016-01-04 17:53:03.753, \
endDate=2016-01-04 17:53:03.753, attributes={7ceb411a-c50f-4b38-97bc-bcaf1de86306=1984548939}, \
statusChangeHistory=[], status=UNKNOWN}
```

(I reformatted a little, it's a bit worse than that.)

What if a test fails?

- If a condition in a `then:` block fails, the arguments are printed using `toString()`
- `toString()` is *very important* because that might be all the information you get!
- I've been auto-generating `toString` with Intellij

Update Property

```
def "After updating an activation,\
     the changes are reflected in a find"() {

  given:
  Activation newActivation = generator.randomActivation()
  newActivation.activationId = activationid
  dao.update(newActivation)

  when: Activation updated = dao.find(activationId)

  then: newActivation == updated

  where: activationId << generator.idsToUpdate
}
```

The generator needs to have *created* some activations for update..

Delete Property

```
def "After deleting an activation, \
     the activation cannot be found"() {

    given: dao.delete(activationId)

    when: dao.find(activationId)

    then: thrown(ActivationNotFoundException)

    where: activationId << generator.idsToDelete
}
```

Testing Different Daos

How can I use the same tests on two different Daos (say, Postgresql and Mongo)?

```
abstract class ActivationSpecBase extends Specification {

    abstract ActivationDao createDao()

    def doSetupSpec() {
        dao = createDao()
        generator = new ActivationTestGenerator()
        generator.initialize(service)
    }

    ...
```

Testing JPA Implementation

This is the entire Spec class for JPA

```
public class JpaActivationSpec
    extends ActivationSpecBase {

    @Shared ApplicationContext context
    @Shared EntityManagerFactory emf

    ActivationDao createDao() {
        JpaActivationDao dao = new JpaActivationDao()
        dao.setEntityManagerFactory(emf)
        return dao
    }

    def setupSpec() {
        context =  new ClassPathXmlApplicationContext("/testApplic
        emf = context.getBean("entityManagerFactory")
        super.doSetupSpec()
    }
}
```

Testing Mongo

Mongo is even simpler

```
public class MongoActivationDaoSpec
    extends ActivationSpecBase {

    @Shared ApplicationContext context

    ActivationDao createDao() {
        MongoOperations template
            = context.getBean("mongoTemplate")
        return new MongoActivationDao(template)
    }

    def setupSpec() {
        context =  new ClassPathXmlApplicationContext("/testApplic
        super.doSetupSpec();
    }
}
```

Properties for Search

What properties describe the correct behavior of search?

```
Set<Activation> search(SearchParameters parameters);
```

Properties for Search

I came up with two main properties:

1. Every activation returned should satisfy the SearchParameters
2. If an activation exists that satisfies the SearchParameters, it should be returned by search.

SearchParameters Class

```
public class SearchParameters {

    //Null parameters are to be ignored

    private Set<String> userIds;
    private Set<Integer> locationIds;
    private Set<String> creatorIds;
    private Set<String> assessmentIds;
    private boolean includeCanceled;
    private Date minStartDate;
    private Date maxStartDate;
    private Date minEndDate;
    private Date maxEndDate;
    private Date minCreateDate;
    private Date maxCreateDate;
    ...
}
```

First Search Property

```
def "Any activation returned by a search satisfies \
     the search parameters"() {

    when: Set<Activation> result = dao.search(qry)

    then:
    qry.userIds == null ||
        result.every {
            act ->  qry.userIds.contains(act.userId)
    }
    qry.assessmentIds == null ||
        result.every {
            act -> qry.assessmentIds.contains(act.assessmentId)
        }
    qry.locationIds == null ||
         result.every {
             act -> qry.locationIds.contains(act.locationId)
         }
    ...
    ...
    where: qry << generator.searchParameters
}
```

Second Search Property

```
def "Any activation that satisfies the search parameters is \
     returned by the search"() {

    when:
    tuple.activation.activationId =
          dao.create(tuple.activation)

    then:
    dao.search(tuple.parameters).contains(tuple.activation)

    where: tuple << generator.searchParameters .
             collect { parameters ->
                  [parameters: parameters,
                   activation: generator.activationForParameter
             }.

}
```

Limitations of Spock

Spock isn't really a property based testing framework :(

- Doesn't run tests concurrently. (Can't test for race conditions.)
- Preconditions on input don't really work – you can try filtering the data but if no data meets the precondition the test fails (not what you want.)
- Random generators have to be written from scratch.
- No support for recreating failed tests.

Summary

- Property Based Testing is a style of testing from Functional Programming
- Goal is to *specify* the behavior of a service and automatically generate tests from the specification.
- Support is scant in the Java community, but you can fake it with Spock.

Property Based Testing vs Example Based Testing

- I think the main advantage of Property Based Testing is that you actually specify behavior, not just a few examples.
- Is it easier to write test input generators or to construct examples of test input? I prefer writing the generators but YMMV.
- Another potential advantage is concurrent testing, which could test for race conditions. But Spock does not do that.

Questions?

?