Test Double Notes

Mathieu Pauly

July 2015

Disclaimer: draft.

The Little Mocker (Uncle Bob)

This section is a summary of the Uncle Bob article around mocking.

There are many types of test doubles. We split them into two categories: a hierarchy of dummies and fakes. Precisely we have:

- Dummy > Stub > Spy > Mock;
- Fake.

Dummy

No message should be sent to a dummy in the tested scenario but the class under test (the API, constructor or interface) requires it.

```
class Dummy {
  Boolean authenticate(String id) {
    return null;
  }
}
```

Stub

A stub implements the context for one kind of scenario. It specifies the inputs.

```
class Stub {
  Boolean authenticate(String id) {
    return true;
  }
}
```

Spy

Spy registers the interaction with the class under test. It specifies the outputs. Test verifies the expected interactions. Spying may increase coupling between the test and the class under test (test/tested coupling).

```
class Spy {
  Boolean authenticate(String id) {
    authenticateWasCalled = true;
    return true;
  }
}
```

Mock

A mock is programmed to react to non expected behavior. Mock verifies the expected invocations.

```
class Mock {
  Boolean authenticate(String id) {
    authenticateWasCalled = true;
    return true;
  }

  boolean verify() {
    return authenticateWasCalled;
  }
}
```

Fake

Simulation of a stereotyped business case. For example, we can say that “admin” can always authenticate. Here is another example: saying that “/tmp/does-not-exist” by convention (in tests) does not exist.

```
class Fake {
  Boolean authenticate(String id) {
    return "admin".equals(id);
  }
}

class FsFake {
  Boolean exists(String path) {
```

```
    return "/tmp/does-not-exist".equals(path);
  }
}
```

Fakes can be remarkably more complicated. They can mimic the real system and be put in a test harness.

Remarks

What are the consequences of separating commands from queries on the coupling between test and tested?

If stubbing increases the test/tested coupling it may indicate that we are stubbing private collaborators (i.e.: hidden implementation) not public collaborators.

Should injected dependencies systematically be considered as public collaborators (i.e.: part of the contract)?

What are the fondamental differences between I/O implemented by method calling and I/O implemented by argument/return?

```
String logInfo(String message) {
  return String.format("[INFO] %s", message);
}
```

or

```
void loginfo() {
  logger.log(String.format("[INFO] %s", queue.getMessage()));
}
```

State Versus Interaction Testing

Verifying state on collaborators is equivalent to making an integration test. By verifying state, you acknowledge (test) the semantic of the collaborators. That is, you test the collaborator itself.

We assume we jave the following implementation:

```
class AppLock {
  void lock() {
    fs.createNewFile("/tmp/app-lock");
  }
}
```

Example of state testing

```
void test() {
  new AppLock(fs).lock();
  assertTrue(fs.exists("/tmp/app-lock"));
}
```

We see that it seems that `fs` is tested. Because we test that if `createNewFile` is called then `exists` should return `true`.

Example of interaction testing

```
void test() {
  expectThat(fs).createNewFile("/tmp/app-lock");
  new AppLock(fs).lock();
  verify(fs);
}
```

This version is more focused on the `AppLock` class. We see that because we check that app lock interacts correctly with its collaborator. Which can be a good thing if we do pure unit testing (i.e., test in isolation).

But we can see that if the implementation changes, for example we change the path of the lock file, the test will be broken. In that situation, this means that path shouldn't be exposed.

Avoid Broken Test: first try

A first solution can be to break the `AppLock` class into two parts: the one that knows which path is configured (yes this is configuration) and the other which creates a file given a configured path. This quite is complex. Here a path provider (a.k.a. config) is injected in `AppLock`.

Second solution is to make the lock path a constant in `AppLock`. Then we will use that constant in the test. Here `AppLock` defines the path itself.

We show a third solution where path is injected as a value to `AppLock`:

```
void testFileLock() {
  expectThat(fs).createNewFile("/tmp/app-lock");
  new FileApp(fs, "/tmp/app-lock").lock();
  verify(fs);
}
```

In that version we bound path between test steps. We could write using gherkins syntax:

```
Given /tmp/app-lock as configured lock
When I lock app
Then /tmp/app-lock is created
```

Path in `Given` clause is implemented using lock path in constructor. Path in `Then` clause is implemented using expectThat (which is actually invoked on `verify`). Expect declare what will be verified.

The coupling between the test and its class it better because path is parameterized and is explicitly subject to change.

Comparison between state & interaction tests: second try

Back to the state-style test. We could also parameterize the lock path to bind the `Given` and the `Then` clauses. If we compare state-style and interaction-style test we have:

```
assertTrue(fs.exists("/tmp/app-lock"));
```

versus

```
expectThat(fs).createNewFile("/tmp/app-lock");
verify (fs);
```

Role	State	Interaction
Check	assertTrue	expectThat
Method	exists	createNewFile
Data	“/tmp/app-lock”	“/tmp/app-lock”

Even then the syntax has a different form in the first and the latter forms (i.e.: 1 method for assertTrue call versus 2 for expectThat & verify) we can see one shared point: `"/tmp/app-lock"`. Thus, we can say that, `fs.exists("/tmp/app-lock")` is an equivalent for `createNewFile("/tmp/app-lock")`. In that example, the two expressions are interchangeable.

Think in term of effect

We can then implement app lock differently. AppLock is not interested in creating a file. It's goal is to ensure that file will exist.

So far, we assume we had the following implementation:

```
class AppLock {
  void lock() {
    fs.createNewFile(lockPath);
  }
}
```

Now we could have that implementation:

```
class AppLock {
  void lock() {
    fs.exists(lockPath);
  }
}
```

In the latter form, the file system implementation itself will ensure that the file is created when `exists` is called. File existence is a property. An algorithm to reach that property could be to create a file.

This is like saying the following:

```
int x = 21;
x == 21;
```

The two form are equivalent. To have the actual equallity between `x` and `21` I have to assign `21` to `x`.

In that case, it is no more a problem to write interaction-style tests. We do no more rely on an interaction which could be an implementation detail (the algorithm) that could change. We focus on the collaborator properties (state). Here the implementation of `exists` is moved into the file system. We just defined a contract and we used it.

Interaction Return

We can make it more confusing if return value of a function gives an indication of the interaction.

For example, `File.createNewFile(String path)` returns true if and only if the file was created (i.e.: it did not exist before).

Thus, the returned value of `createNewFile` does not express a state (actually it expresses the previous state). Instead it expresses the interaction of the method. By interaction I mean the how-to, the description of the action (the algorithm) that makes is possible to pass from the state before call to the state after call.

State machine

We can see applications as state machines. States (represented as nodes) are connected by interactions (represented with arrows). With state-style testing we put the system in a given state, we trigger an event and we check the resulting state. With interaction-style testing, we place system in a given state, we trigger an event and we check the resulting interactions.

The distinction between the two is confusing when we define arrows by the source and destination state. It is like defining a function by enumerating the mapping between input and output elements:

```
f 0 = 1
f 1 = 0
```