Test Double Notes

Mathieu Pauly

July 2015

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()));
}
```