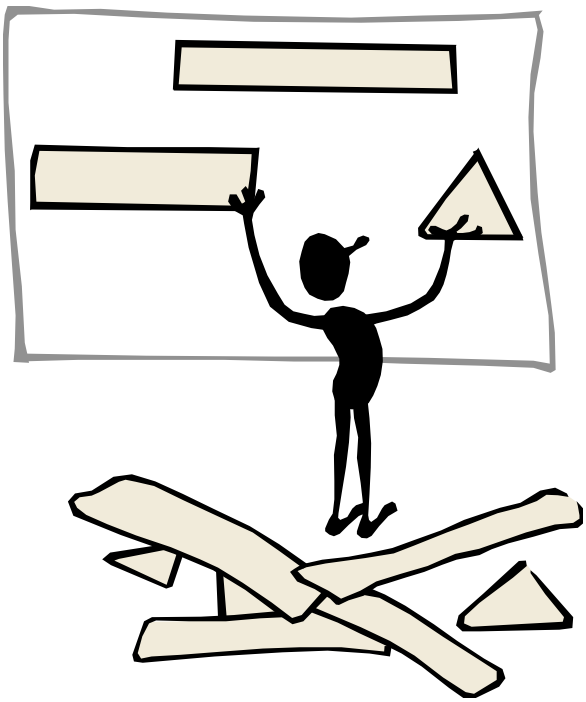
COMP33711

Agile Software Engineering

(Agile Methods Theme)

Test Doubles: Basic Concepts

2014/2015 Academic Session



Suzanne M. Embury
Room KB 2.105
School of Computer Science
University of Manchester
Oxford Road
Manchester M13 9PL
U.K.

Telephone: (0161) 275 6128
Fax: (0161) 275 6236
E-Mail: S.Embury@cs.manchester.ac.uk

1. Introduction

We cannot leave the topic of test automation without making some mention of "test doubles". This document gives a brief introduction to this important concept in modern testing practice. The basic terms are covered, but anyone wishing to apply these ideas on a real project should expect to have to read further to gain the depth of knowledge needed.

The material in this document is examinable.

2. Test Doubles: What and Why?

When we write automated tests for a software system, our goal is for each test to execute the code-under-test in a manner that is as close to the eventual deployment environment as we can, given the testing resources available. The reason for this is hopefully obvious: the greater the difference between the test environment and the actual environment of use, the greater the chance that a problem will occur in operation that was not found by our test suites. (And the greater the chance of us wasting our time fixing spurious errors that only occur in the test environment.)

Unfortunately, replicating the deployment environment for automated testing is not always possible. Consider for example an e-commerce system that allows payment by credit card through myPay, a (fictional) third-party system that manages credit card payments. We cannot access the real myPay system, and make a real payment, every time we want to run our automated test suite. If we are lucky, myPay will provide a test interface, but such test interfaces typically merely accept valid requests and reject malformed ones, without carrying out any actual processing behind them. So, using it would be quite different from the real deployed system. We might find some way around the problem of the expense of using the real myPay system (by paying tiny amounts from one of our own accounts into another of our accounts, for example), but might still encounter a range of other problems:

- We want to test across a whole range of payment amounts. The money for this would have to be found from somewhere, and the amount may be prohibitive.
- The addition of some new test cases might cause our credit balance to be reduced to zero unexpectedly, so that some tests which formerly passed now fail (and possibly *vice versa*). This would mean that we cannot rely on our test suites to tell us where the deviations from the specification actually are.
- When we want to test situations where our credit balance is zero, we have to have yet another credit card account with a zero balance, or else have the test transfer all the money out of the account before the test, and back again at the end of the test. This is all very slow and likely to be error prone, making the test itself unreliably. The same is true for many other edge cases we might want to test (an expired card, for example).
- The myPay system is third-party and we have little control over its behaviour. If becomes unavailable for a period, we are unable to run any of the tests that involve it.
- We want to run our acceptance test suite several times each day, and our unit test suites hundreds of times each day. We need our tests to be lightning fast. We cannot afford the time needed to connect to the myPay system, send the request, wait for the response and to disconnect, in each test case.
- We want to be able to test the behaviour of our software against all the valid and invalid states of the myPay system. That is, we want to be able to test how our system behaves when myPay is unavailable whenever we like, without having to

wait for it to go offline of its own accord, and then rushing to run all the relevant tests before it comes back online again.

All these issues come down to the same basic point: when we write (and run) automated tests, we need the result to be reliable. That is, we want a test failure to mean *only* that there is a bug in our system and *nothing else*. To do this, we need to have full control over the environment in which the test case runs. That means we need to replace components in that environment which are not fully under our control with versions of the components that are under our control. We call these replacements "test doubles".

In the case of our e-commerce system example, we might deal with the testing challenges mentioned above by creating our own version of the myPay system. It is accessed using the same interface as the myPay system (as far as that is possible), but it is totally under our control. We can set it up to be available or unavailable, to be running fast or slow, to be behaving correctly or incorrectly. Each test case can configure its own instance of the pretend system so that it has the accounts expected, with the features expected of each account. This allows us to test against a full range of normal and exceptional account states without needing to create actual credit card accounts matching the states the test needs.

As well as needing test doubles for components that are not under the control of the project team, we also use them for components that have features that make them inconvenient for testing. For example, it is common to use a test double for a component that is very slow (such as a database server that must be communicated with over a network), or that is difficult to instantiate and configure. A further very common reason for needing a test double is when a dependent component doesn't yet exist. We don't want to have to wait until the dependent component is fully implemented before we can start to test the component we are currently working on. A test double can help us make progress on our own part of the system regardless of the state of development of the rest of the system.

Of course, testing with a test double is not the same as testing against the real component. It is important, therefore, that *some* acceptance tests are run against the true components as soon and as often as the project team can manage, though they are likely to be much fewer in number than the unit tests and acceptance tests that work with test doubles.

3. Types of Test Double

Three types of test double are used in automated testing today. Each one has its strengths and weaknesses, and the test automation engineer needs to know about all of them, since we might use all three in a single test suite, covering different types of testing need. The three types are:

- stubs,
- fakes, and
- mocks.

(Other authors may use their own terminology for these types of test double. The terms given here are the ones I will expect you to use in the exam.)

We'll give a brief introduction to each test double type below, along with an example of each from the Snake Game system we have been working on.

3.1 Stubs

A stub version of a component is a very simple and stupid implementation of just that portion of the interface of the component that is needed by the test case. Stubs often return hard-coded values, or else perform very simple computations on the input values to create the output values.

We used a stub in some of the unit tests created when making the first Snake Game acceptance tests pass (see the ATDD case study document, under week 11, for details). For example, the test below says that a `SnakeGame` instance should return the Field instance made for it by the `FieldFactory` supplied to its constructor, when the `getField()` method is called:

```
@Test
public void shouldReturnTheFieldCreatedByTheGivenFieldFactory() {
    final Field expectedField = new Field(5);
    SnakeGame game = new SnakeGame(new FieldFactory() {
        @Override
        public Field makeField() {
            return expectedField;
        }
    });

    assertEquals(expectedField, game.getField());
}
```

Here, we needed a very simple `FieldFactory` implementation that allows us to control the Field instance that is returned. A stub in the form of an anonymous class with a hard-coded return value is sufficient for what this test requires.

3.2 Fakes

A fake is a test double that is a genuine implementation of the component needed, in that it provides the same basic functionality as the component being doubled, but with some key differences. A fake often implements the same interface as the component it doubles, but adds additional methods or fields that allow easy programmatic control of the state of the component from an automated test script.

An example might be the use of an in-memory database in place of an on-disk database. The in-memory database is populated with the relevant data during the set-up of the test case. It functions like an ordinary database in almost all ways, except that the data is kept in memory (on the same JVM on which the test case is executing) and therefore can answer queries much more quickly than the on-disk database server located on another machine somewhere across the network (and shared by many other users).

We saw an example of a fake being used in the Snake Game implementation while making the first acceptance test pass. This was in the fixture code for the automatic movement tests, specifically in the constructor of the fixture code for the "Play Snake Game" script table:

```
public PlaySnakeGame() {
    List<List<String>> initialGameState =
                                FitnesseTestEnvironment.getInitialGameField();
    SnakeGame game = new SnakeGame(new FitnesseFieldFactory(initialGameState));
    FitnesseTestEnvironment.saveGame(game);
}
```

The constructor of the SnakeGame instance expects a FieldFactory instance to be passed. The real FieldFactory to be used by the game in normal play creates a field layout at random, with the snake located in one of the four corners of the playing field. That's not helpful to us when writing tests for the Snake Game, as we need precise control over what the starting state of the game field should be. We can't start from a randomly selected game state and then give precise and clear expectations about what the finished state should be in each situation tested.

So, for our acceptance tests, we use a fake FieldFactory implementation, called FitnesseFieldFactory. This does everything that a FieldFactory should do (it makes legal Field instances on demand) but it allows the caller to provide a FitNesse table data structure, and uses that to create the contents of the Field instance to be returned. Thus it provides the interface required by FieldFactory, by adds an additional element that allows the FitNesse table data structure to be passed in to the constructor, providing the control over the Field layout that the tests require.

(The FitnesseFieldFactory implementation can be found in the Snake Game implementation on Moodle, in package comp33711.snakegame.fixtures.)

3.3. Mocks

The final type of test double is the most sophisticated: the "mock object" or just "mock" for short. When we write object-oriented code, we often want to specify our expectations of how the objects we are building interact, as well as just what the data values passed between the objects should be. This can't be done easily using standard assertions in tests, but it can be done with mock objects. When we need to use a test double for some class (typically conforming to an interface), we can request that a mock object be created for it. This instance looks like a member of the class (implementation of the interface) and we can inject it into our production code in place of the instance we want to double. We can also specify the method calls that we expect to see occurring between the mock object and the rest of our production code. We then execute the production code under test, and the mock object framework keeps track of what method calls are made, returns the expected values, and reports back on any deviations from the communication protocol between the objects that we have specified.

We don't have time or space to go into mock objects in detail here. To give a basic flavour of what is involved, here is a version of the unit test that we looked at in section 3.1, but using a mock FieldFactory rather than a stub:

```
01   @Rule public JUnitRuleMockery context = new JUnitRuleMockery();
02
03   @Test
04   public void shouldReturnTheFieldCreatedByTheGivenFieldFactory() {
05      final Field expectedField = new Field(5);
06      final FieldFactory fieldFactory = context.mock(FieldFactory.class);
07      context.checking(new Expectations() {{
08         oneOf (fieldFactory).makeField();
09         will(returnValue(expectedField));
10      }});
11      SnakeGame game = new SnakeGame(fieldFactory);
12      assertEquals(expectedField, game.getField());
13   }
```

This test is written using jMock, just one of several mocking tools available today. The first line of the test creates a "context" for the test to run. The context means the set of objects with which the object under test must interact. These objects don't actually exist

in this version of the test. jMock will create the doubles for them automatically, from the information we provide in the test itself.

The first line of the test case (line 05) creates the `Field` instance that we expect to get back from the `SnakeGame` once set-up. The next line asks jMock to create a mock `FieldFactory` for us, in the `context` of the test case. This mock object is stored in the variable `fieldFactory`, so that we can reference it in our expectations later in the test.

These expectations are stated on the next line (lines 07-10). This method call describes the interactions between the objects involved in the test that we expect to have happened by the end of the test. In this case, we set two expectations, one on line 08 and one on line 09. The first states that there should have been exactly one call to the `makeField()` method on our mock `FieldFactory` instance, by the end of the test. The second expectation is that the return value of that call will be the `Field` instance we create on line 05. jMock uses this information to create an implementation of FieldFactory that both checks that the expectations are met, and responds appropriately (i.e., like a real FieldFactory implementation) when the code under test interacts with it.

The final two lines (11 and 12) describe the execution of the code under test. We create a new `SnakeGame` instance, passing the mock `fieldFactory` through to its constructor. We then ask for the `Field` from the `SnakeGame` instance, and check that it is the field we expected.

At the end of the test, jMock inserts a check as to whether the FieldFactory instance has received the expected method calls. A test containing a mock object can therefore fail because the data values appearing in the assertions don't match the stated expected values, or because the expected method calls were not made to the mock objects in the context of the test case. Thus, the key difference between mock objects and the other kinds of test double mentioned here is that mock objects can verify expectations on the way they should be used, whereas stubs and fakes passively respond to the interactions they are involved with, without assessing whether it is correct or not.

4. Where Each Type of Double is Used

Although it is possible to use fakes in unit tests, it is more common to see stubs and mocks used in this kind of test. Fakes by definition tend to be fairly sizeable pieces of code, which often themselves need to be tested, since they are rarely "too simple to test". This makes them less suitable for unit tests, which have to be fast, simple and very easy to locate faults inside. Stubs and mocks, on the other hand, can both be very fast, and lend themselves to the type of lean test environment that we need for unit tests.

In acceptance tests, on the other hand, we rarely use stubs and mocks but often use fakes. An acceptance test does not have the limitation of having to test a single class that a unit test has; indeed, the whole point of an acceptance test is to assess (and describe) how a collection of objects or components work together to solve some meaningful task. Therefore, in acceptance tests, we try to use as much of the system-under-test as possible, and resort to test doubles only when we really have to, in order to achieve the test performance and coverage that we need.

5. Further Reading

There is an abundance of material on these topics on the Web. For a quick introduction, using words other than those above, see Koskela 2008.

"Test-Driven: Practical TDD and Acceptance TDD for Java Developers", by Lasse Koskela, Manning Publications, ISBN: 1-932394-85-0, 2008.