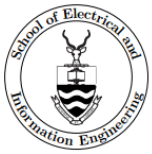

UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Electrical and Information Engineering
ELEN7045 - SD Methodologies, Analysis and Design

Student Name: Adrian Freemantle
Student number: 1558184

Individual Assignment II

Mock Objects

Date: 30 June 2016

Abstract

Test Driven Development provides rapid feedback to the developer on the quality of the system's design which results in continuous refactoring of the code base in order to make it expressive, improve its design and remove duplication. State-based Unit Tests may result in an object exposing state in order to validate the outcome of the test, this breaks encapsulation which can lead to *train wreck* code that violates the Law the of Demeter. Encapsulation can be preserved by using Test Doubles which act as replacements for the Unit-Under-Test's collaborators. Mock Objects are behaviour-based Test Doubles which directly assert that the Unit-Under-Test interacted with it according to expectations defined by the test. While TDD and Mock Objects can be used as design aids, they cannot compensate for a poor overall design and may amplify existing problems when used incorrectly.

Table of Contents

1 Introduction 1

2 Unit Testing 1

3 Test Doubles 2

4 Mock Objects 3

5 Critiques 4

6 Conclusion 5

References 6

Source Code Listings

Code Listing 1: Sociable testing 1

Code Listing 2: Train wreck code 2

Code Listing 3: Solitary testing with a Stub Object 3

Code Listing 4: Testing with Mock Objects 4

Code Listing 5: Needs-driven development with Mock Objects 4

1 Introduction

Test Driven Development (TDD) is the practice of developing features by first creating a failing Unit Test [1] and then developing the simplest possible functionality that will make the test pass. This iterative and incremental process is a design activity [2] as it provides rapid feedback to the developer on the quality of the design which results in continuous refactoring of the code base in order *to improve and simplify its design, remove duplication, and ensure that it clearly expresses what it does* [3].

Section 2 presents traditional state-based Unit Testing and examines the potential problems that may be encountered when including the Unit-Under-Test's collaborating objects. Next, section 3 examines how various types of Test Doubles [4] can be used to isolate the Unit-Under-Test from its collaborators.

Section 4 presents Mock Objects [5], a behaviour-based Test Double which directly asserts that the Unit-Under-Test interacted with it according to expectations defined by a Unit Test. State-based and behaviour-based testing are compared and the advantages and disadvantages of Mock Objects are presented.

Finally, in section 5, criticisms of TDD and the use of Mock Objects are considered and contrasted with opinions in favour of these practices.

2 Unit Testing

While the process of TDD is well documented and understood, the definition of what constitutes a *unit* within a Unit Test is contentious [1]. Within the context of Object-Oriented Design (OOD) it may be argued that the goals of a Unit Test are to verify the behaviour of the object being tested, also known as the Unit-Under-Test, and validate its design [3].

An Object-Oriented system's behaviour is *an emergent property of the composition of its objects* [3] and the way in which those objects interact with each other. This raises the question of whether to include the Unit-Under-Test's collaborating objects, so called sociable testing [1], or whether to isolate the Unit-Under-Test from all collaborating objects, so called solitary testing [1].

Code Listing 1 presents an example of a sociable test. Here the Unit-Under-Test is a *Vehicle* object and its collaborating objects are included in the test.

```
public void Sociable_test(){
    Engine engine = new Engine();
    Starter starter = new Starter(engine);
    Ignition ignition = new Ignition(starter);
    Vehicle vehicle = new Train(ignition);

    vehicle.Start();

    Assert.IsTrue(engine.IsRunning);
}
```

Code Listing 1: Sociable testing

Sociable tests work well when there are few collaborating objects which are not prohibitively expensive to instantiate, however this kind of testing frequently requires an object to expose its internal state in order to verify the outcome of a test. The '*Tell, Don't Ask'* [3] principle states that a calling object should not query the internal state of its neighbouring objects, but rather issue a directly instruction as to what functionality it wants to have performed. This is known as the Law of Demeter [2] which states that objects should:

> *Make their decisions based only on the information they hold internally or that which came with the triggered message; they avoid navigating to other objects to make things happen [3].*

Failure to adhere to the Law of Demeter can result in a series of objects being chained together to create *train wreck* [3] code. This results in brittle tests as changes to the internals of an object can ripple throughout the system.

Code Listing 2 presents an example of *train wreck* code. Instead of calling the *Start* method on the Vehicle object, the code navigates the object hierarchy and directly calls the *StartEngine* method on the *Engine* object.

```
Vehicle vehicle =
    new Vehicle(new Ignition(new Starter(new Engine())));

vehicle.Ignition.Starter.Engine.StartEngine();
```

Code Listing 2: Train wreck code

3 Test Doubles

Solitary testing isolates the Unit-Under-Test from all collaborators and replaces them with Test Doubles [4]. This ensures that only the Unit-Under-Test's behaviour is being evaluated and eliminates the need for real collaborators to expose their internal state in order to verify a test result.

Test Doubles [6] are test-specific objects which act as a replacements for the Unit-Under-Test's collaborators. These objects range from Dummy Objects which have no behaviour or state, to Fake Objects which provide reasonable simulations of the real collaborator's behaviour. Some of the more commonly used Test Double types include [4]:

- Dummy: Used as parameters but provide no functionality
- Fake: Provide reasonable simulations of the real object's behaviour.
- Stub: Provide hard-coded responses to predetermined inputs
- Spy: Extends the role of stubs by storing information on how they were called.
- Mocks: Verify that the Unit-Under-Test interacted with the Mock Object correctly.

Dummy, Fake, Stub and Spy objects are test-specific implementations of the Unit-Under-Test's collaborators. These objects may expose additional state so that the Unit Test can validate the outcome of a test, this is known as state-based testing.

Code Listing 3 illustrates the use of a Stub Object which implements the *IIgnition* interface in order to verify that the *vehicle* object called the *TurnOn* method.

```
public interface IIgnition{
    void TurnOn();
}

public class IgnitionStub : IIgnition{
    public bool TurnOnCalled { get; private set; }

    public void TurnOn(){
        TurnOnCalled = true;
    }
}

public void Solitary_test(){
    IgnitionStub stub = new IgnitionStub();
    Vehicle vehicle = new Vehicle(stub);

    vehicle.Start();

    Assert.IsTrue(stub.TurnOnCalled);
}
```

Code Listing 3: Solitary testing with a Stub Object

A disadvantage to state-based testing is that one or more Test Double classes need to be created for each collaborator. Creating and maintaining these Test Doubles can be time consuming as they are frequently hard coded for specific test scenarios [5].

4 Mock Objects

A Mock Object [5] is a specialised type of Test Double which directly asserts that the Unit-Under-Test interacted with it according to expectations defined by the Unit Test. These expectations include the order in which methods should be called and the parameters that will be passed to methods. A Mock Object may also assert how many times a specific method may be called and define what values the mocked method will return each time it is called by the Unit-Under-Test.

> *A Mock Object is a substitute implementation to emulate or instrument other domain code. It should be simpler than the real code, not duplicate its implementation, and allow you to set up private state to aid in testing [5]*

The example in Code Listing 4 illustrates the use of a Mock Object to mock the *IIgnition* interface and specify that the *vehicle* object is expected to call the *TurnOn* method at least once. Mock Objects test the behaviour of the Unit-Under-Test, this inside-out testing is known as Endo-Testing [5].

```
public interface IIgnition{
    void TurnOn();
}

public void Turn_vehicle_on(){
    Mock<IIgnition> mock = new Mock<IIgnition>();
    Vehicle vehicle = new Vehicle(mock.Object);

    vehicle.Start();

    mock.Verify(i => i.TurnOn(), Times.AtLeastOnce);
}
```

Code Listing 4: Testing with Mock Objects

Mock Objects allow the design of the system to be approached in a top-down manner as collaborators and the roles they perform can be defined on interfaces and immediately mocked for testing purposes. This process of interface discovery is known as Needs-Driven Development [2] in Lean Software as new roles and responsibilities are pulled into existence when they are needed.

Code Listing 5 illustrates how the *TurnOff* method on the *IIgnition* interface is pulled into existence when a *Turn vehicle off* test is implemented.

```
public interface IIgnition{
    void TurnOn();
    void TurnOff();
}

public void Turn_vehicle_off(){
    Mock<IIgnition> mock = new Mock<IIgnition>();
    Vehicle vehicle = new Vehicle(mock.Object);

    vehicle.Start();
    vehicle.Stop();

    mock.Verify(i => i.TurnOn(), Times.AtLeastOnce);
    mock.Verify(i => i.TurnOff(), Times.Exactly(1));
}
```

Code Listing 5: Needs-driven development with Mock Objects

Mock Objects provide advantages over state-based Unit Tests by reducing the cost of maintaining Test Doubles and enabling a more intuitive top-down design approach [5]. The process of interface discovery results in collaborators being more clearly defined and their roles more focused as they are only pulled into existence when needed [5].

5 Critiques

Neither Mock Objects nor TDD can compensate for poor design and when these techniques are used incorrectly they amplify *problems such as tight coupling and misallocated responsibilities* [2]. However, the process of performing TDD can aid in learning how to correctly apply OOD principles which can have a lasting impact on the way developers think about software design [7], even when TDD is not used.

David Heinemeier Hansson, the creator of Ruby on Rails, refers to TDD as an *unrealistic, ineffective morality campaign* [8] used by test first *fundamentalists* to shame developers who do not adhere to the TDD *dogma.* He argues that the test-first approach results in software design that suffers from *needless indirection and conceptual overhead* [9] in order to achieve easy-to-mock objects and fast Unit Tests. In his view, tests should be driven by the system's design and not the other way round.

Robert C Martin argues that if *you aren't doing TDD, or something as effective as TDD, then you should feel bad* [10] as TDD is the only testing strategy that requires loose coupling between components. He further argues that tests should *act as accurate, precise, and unambiguous documentation* [10] of the system and that TDD is the best way of achieving this goal.

TDD and Mock Objects provide additional benefits other than aiding in the design of Object-Oriented software. These practices create a safety net that enable developers to *quickly and easily clean the code without fear* [10]. This enables continuous refactoring of the code base in order *to improve and simplify its design, remove duplication, and ensure that it clearly expresses what it does* [3].

6 Conclusion

Test Driven Development provides rapid feedback to the developer on the quality of the system's design. State-based Unit Tests may result in an object exposing state in order to validate the outcome of the test, this breaks encapsulation which can lead to *train wreck* code. Encapsulation can be preserved by using Test Doubles which act as replacements for the Unit-Under-Test's collaborators. Mock Objects are behaviour-based Test Doubles which directly assert that the Unit-Under-Test interacted with it according to expectations defined by the test. While TDD and Mock Objects can be used as a design aids, they cannot compensate for a poor overall design and may amplify existing problems when used incorrectly.

References

[1] M. Fowler, “Unit Tests,” martinfowler.com, 5 May 2014. [Online]. Available: http://martinfowler.com/bliki/UnitTest.html. [Accessed 21 June 2016].

[2] S. Freeman, T. Mackinnon, N. Pryce and J. Walnes, “Mock Roles, Not Objects,” in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Vancouver, 2004.

[3] S. Freeman and N. Pryce, Growing object-oriented software, guided by test, Boston: Pearson Education, 2010.

[4] G. Meszaros, “Test Double Patterns,” xUnit Patterns, 9 February 2011. [Online]. Available: http://xunitpatterns.com/Test%20Double%20Patterns.html. [Accessed 21 June 2016].

[5] T. Mackinnon, S. Freeman and P. Craig, “Endo-testing: Unit Testing with Mock Objects,” in *Extreme Programming Examined*, Boston, Addison-Wesley Longman Publishing Co., 2001, pp. 287-301.

[6] M. Fowler, “Test Double,” martinfowler.com, 17 January 2006. [Online]. Available: http://www.martinfowler.com/bliki/TestDouble.html. [Accessed 21 June 20116].

[7] K. Stobie, “Too Darned Big To Test,” *Acmqueue*, vol. 3, no. 1, pp. 32-37, 2005.

[8] D. H. Hansson, “TDD is dead. Long live testing.,” David Heinemeier Hansson, 23 April 2014. [Online]. Available: http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html. [Accessed 21 06 2016].

[9] D. H. Hansson , “Test-induced design damage,” David Heinemeier Hansson, 29 April 2014. [Online]. Available: http://david.heinemeierhansson.com/2014/test-induced-design-damage.html. [Accessed 21 June 2016].

[10] R. C. Martin, “Monogamous TDD,” 25 April 2014. [Online]. Available: http://blog.8thlight.com/uncle-bob/2014/04/25/MonogamousTDD.html. [Accessed 22 June 2016].