2014

Using Class Interfaces and Mock Objects to Unit Test Aspects

Michael Bryan Snider
Nova Southeastern University, mbsnider@comcast.net

Using Class Interfaces and Mock Objects to Unit Test Aspects

by

Michael Bryan Snider

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

2014

We hereby certify that this dissertation, submitted by Michael Snider, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

_____ _____

Francisco J. Mitropoulos, Ph.D. Date
Chairperson of Dissertation Committee

_____ _____

Sumitra Mukherjee, Ph.D. Date
Dissertation Committee Member

_____ _____

Michael J. Laszlo, Ph.D. Date
Dissertation Committee Member

Approved:

_____ _____

Eric S. Ackerman, Ph.D. Date
Dean, Graduate School of Computer and Information Sciences

Graduate School of Computer and Information Sciences
Nova Southeastern University

2014

An Abstract of a dissertation submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Using Class Interfaces and Mock Objects to Unit Test Aspects

by
Michael B. Snider
May 2014

Aspect oriented programming (AOP) has garnered a great deal of research interest in the past decade. The research has centered on the ability to implement aspects and to identify cross-cutting concerns in existing software. Little work has been done on the ability to test software that is created using AOP constructs.

In object oriented programming (OOP) class objects are individual units of code that encapsulate the desired functionality of each object. AOP is an attempt to handle the cross-cutting concerns that represent functionality needed by a class, but is not specific to that class. The cross-cutting functionality is implemented in AOP by using a class-like structure, the aspect. Aspects do not have their own context and as such are dependent upon other objects for their context. By not having their own context it is difficult to test the functionality of aspects. This study investigated the effectiveness of using class interfaces and mock objects to unit test aspects. This was accomplished by having the mock object inherit from the same interface as the base code, so that the mock object could be swapped in for the aspect.

Acknowledgements

This dissertation would not have been possible without the support and understanding of many individuals.

I would like to start by thanking my wife Sharon, if it were not for her loving support this would not have been possible. She put up with all of the hours of studying and countless trips to Florida that have allowed me to reach this achievement. I also wish to dedicate this work to my daughters Alexa and Madelyne who continually pushed me to keep going by asking when I was going to get done and by telling everyone that I was working on my doctorate.

Next I wish to thank Dr. Frank Mitropoulos. When I started my dissertation I was a ship lost at sea. Through his guidance I was able to identify a topic of research. He continued to guide me and advice me so that I continued to make steady progress until reaching the end. I would also like to thank my committee members Dr. Laszlo and Dr. Mukherjee.

I wish to thank my father, Bryan William Snider, for his support and faith in me. He was always there when I needed him no matter where I was or what the need. While he did not get to see me finish this project he always knew that I would.

Finally I would like to mention my Nova Southeastern doctoral colleagues and study group. The individuals in this group were there for me providing support and encouragement during course work and the dissertation process. I would particularly like to mention Dr. Raymond Halper for being a friend and study partner during the majority of my course work and Dr. Ron Krawitz for continually providing help and support to keep me going during the dissertation process.

Table of Contents

List of Figures

Figures

List of Tables

Tables

Chapter 1

Introduction

Background

Aspect Oriented Programming (AOP) was first introduced by Gregor Kickzales in 1996 (Haque, 2011). It was not proposed as a replacement to Object Oriented Programming (OOP), but as an enhancement. AOP is a methodology that deals with the problem of cross-cutting concerns found within a software application (Lemos, Ferrari, Masiero, & Lopes, 2006). In OOP classes are created to represent functional objects that are specified within the design requirements. However, there exist requirements that cannot be confined to a single object and do not represent needed functionality of that object. When a requirement is not a core concern of a class or method and that concern is needed in numerous places it is considered to be a cross-cutting concern (Monteiro & Aguiar, 2007). As an example, logging of actions within a method is typically a cross-cutting concern. Within the method, logging is not a core concern, but it is a requirement of the application that actions are to be logged. The logging takes place within numerous methods within the application, but the concern of logging is a functional requirement of the application.

AOP is a programming paradigm that addresses the issue of modularization of cross-cutting concerns (Wedyan & Ghosh, 2010). AOP extends programming by introducing four new concepts: joinpoints, pointcuts, advice and aspects (Wloka, Hirschfeld, & Hansel, 2008). The aspect is the construct for encapsulating a cross-cutting concern. Each aspect contains the functionality for a cross-cutting concern. An AOP aspect is very similar to a class found within an OOP application. Aspects have the same constraints as an OOP class (Parizi & Ghani, 2007). The aspect is implemented by weaving it into the base application code, see Figure 1.

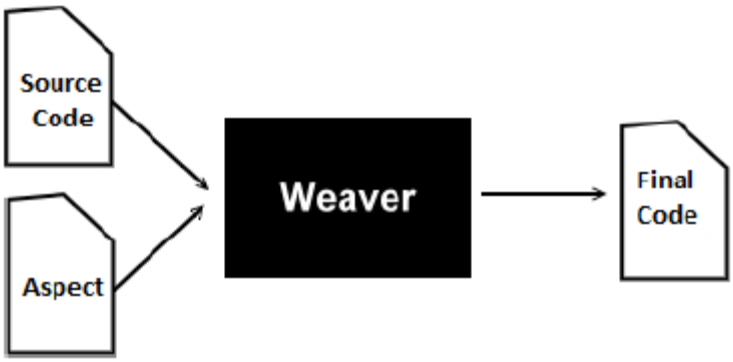

Figure 1: AOP Weaving Process

A joinpoint is a well-defined point within an application. A pointcut selects specific joinpoints and values at those joinpoints. The advice is code that will be executed when the joinpoint is reached during execution. The most common type of joinpoint is a method call. As AOP has advanced, the joinpoint types have grown to include attribute access and the initialization of an object (Parizi & Ghani, 2007). The pointcut is a set of patterns that designate how to recognize a joinpoint. By using a pointcut to identify several joinpoints within a program a software developer is able to implement a single piece of functionality at several places within an application (Parizi & Ghani, 2007). The

number of places the pointcut is implemented is dependent upon the joinpoint definition. The definition can be tight to look for a specific method within a specific class or, on the opposite, end it can be broad to include all methods in all classes.

In AOP there are three types of advice available; before, after and around. The before advice will execute as the joinpoint is reached and prior to the base code at the joinpoint being executed. The after advice will be executed at the joinpoint after the base code is executed. The around advice will execute instead of the base code at the associated joinpoint. The base code is not always skipped by the around advice. By calling the proceed method from within the around advice the base code will be executed (Monteiro & Aguiar, 2007).

As with any software construct, to ensure that the code works properly it must be tested for correctness at the time of development. This testing should be done before and during integration and repeated during continuing development. It is through the process of verification that one is able to increase the confidence that a program functions as intended (Olan, 2003). A programmer cannot assume that because their code compiles that it will execute properly or as intended. Unfortunately, too many programmers assume that because there are no syntax errors the code will work correctly (Olan, 2003).

Problem Statement

This study proposes to research the benefit of, or lack thereof, using class interfaces and mock objects in unit testing of aspects.

Parizi and Ghani (2007) noted that the testing of AOP and aspects is very difficult, due to aspects not having their own context and thus being dependent upon other objects. This difficulty includes the following (Alexander, Bieman, & Andrews, 2004):

- Aspects do not have their own identity or existence. They are dependent upon another class for their identity and execution context.
- Aspects are tightly coupled to the classes with which they are woven. Aspects are dependent on the internal representation and configuration of the methods to which they are woven. Any change to these methods is likely to have a direct effect on the aspect.
- When looking at source code the control and data dependencies are not readily visible for either the aspects or the base classes. The developer of the base code and the aspects are likely to be different personnel and, as such, it is very likely that neither developer will know the resulting control flow of the woven code. Finding failures in the resulting woven code may be difficult.
- There are many possible locations of a fault. The location of a fault may lie within the implementation of either the aspect or the base code method, or it could be the result of the weaving order of multiple aspects.

Unit testing was first introduced by Kent Beck in the 1970's in association with Smalltalk (Osherove, 2009). A unit test is a piece of code that invokes another piece of code. The desired effect of a unit test is to check for correctness of the invoked code by making assumptions about what the invoked code should or should not do. If the

assumption fails to happen then the unit test has failed and a change must be made to either the code tested or the unit test (Olan, 2003).

When testing units of code it is often required to provide additional information to handle external dependencies. Stubs are one device for solving the dependency problem in unit tests. A stub is a controllable replacement for a system dependency (Osherove, 2009). The stub allows for the testing of code without having to worry about an external dependency. The stub code will simulate what is actually done by the dependency code.

In unit testing, a unit of code is being tested in isolation. To help achieve this isolation, stubs have been used to create artificial dependencies that mimic a unit's actual dependencies. There are also times when the unit test interacts with other objects by sending inputs or receiving outputs from the other objects. In Mackinnon, Freeman, and Craig (2001) a new technique is proposed to handle the interaction testing by the usage of mock objects.

With normal unit testing a tester is trying to test the units from the outside. With mock objects one is replacing the domain code with dummy code (Mackinnon, et al., 2001). As stated previously, this is similar to stubs but with two differences: 1) the testing has a finer granularity and 2) the stubs and tests are used to drive the development of the domain code.

When one first looks at stubs and mock objects the difference seems to be very small. However, the distinction is quite important (Osherove, 2009). A stub will replace an object so that one is able to test another object without problems. A stub is used for doing action-driven testing, meaning that a particular action an object takes is being tested. A mock object is used for result-driven testing. In result-driven testing the test is looking to

see that an end result is now true (Osherove, 2009). The easiest method for determining if one is dealing with a stub is that the stub never fails a test, while mock objects will be used to verify whether a test is passed or failed. In using a stub, the operations assert is performed against the class being tested. The stub assists in making sure the test runs smoothly, see Figure 2. When using a mock object the test communicates with the mock object and the mock object records the communications. The test is using the mock object to verify that the test passes, see Figure 3.

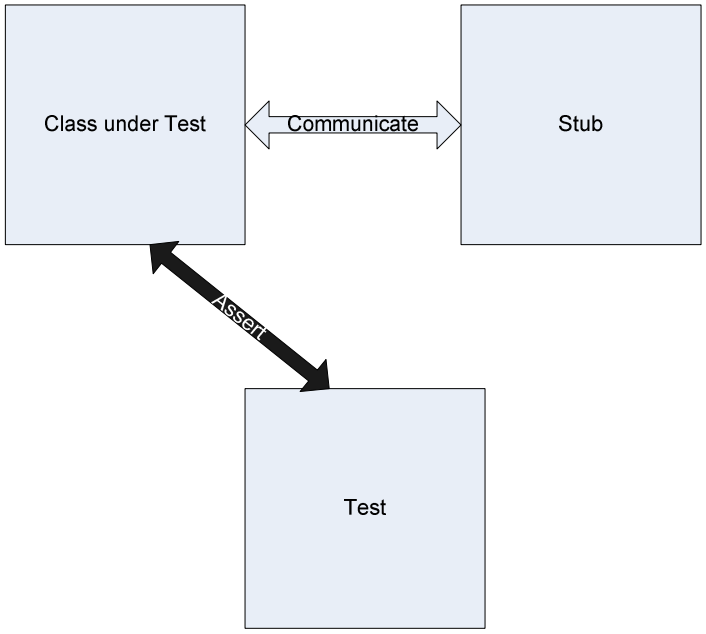

Figure 2: Testing with a stub

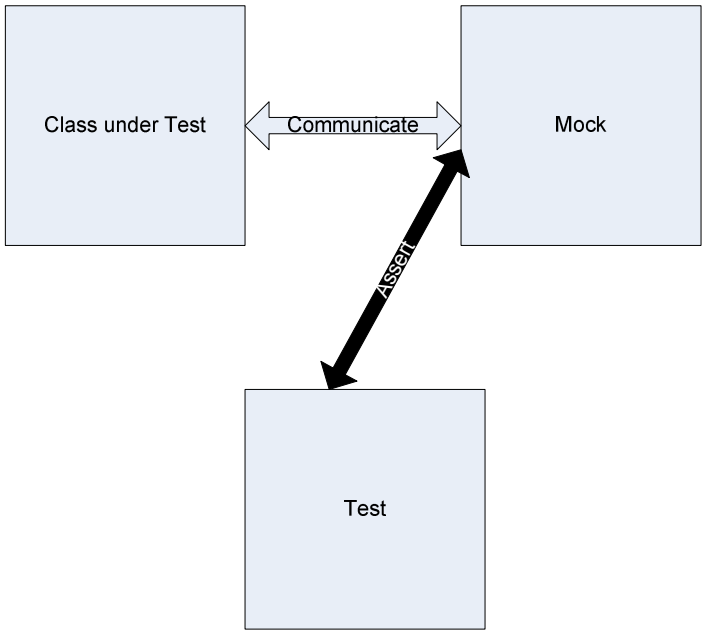

Figure 3: Testing with a mock object

The advantage to using a mock object is that the usage will assist in making the test more readable since a developer can identify the interactions by setting the constraints to the mock object in the test. The mock object also has the advantage of assisting the test to execute rapidly and smoothly (Kim, Park, & Wu, 2006). When developing code one of the most important advantages to mock objects is the ability to create the interaction between an object and its neighbors. It is typical for the neighbor objects to not exist at the time a developer is creating an object. By implementing mock objects to act as placeholders for the missing neighbors a developer is able to continue working on the object at hand. When creating the mock object the developer would only need to implement those methods used by the object under development. Mock objects are usually used in the following cases:

- Tests are taking too long to execute

- Tests do not run consistently
- When it may not be possible to run all of the tests
- When there exists a large dependency chain
- When a collaborator could cause events that cannot be easily recreated within a test.

In their follow up work Freeman, Mackinnon, Pryce, and Walnes (2004) updated the work that had been done in their original paper, Mackinnon, Freeman, and Craig (2001), on mock objects based upon their experiences with using mock objects. In this work they presented the usage of interfaces with mock objects. Interfaces are one of the foundations of an object-oriented system (Nandigam, et al., 2009). Interface based systems show three key characteristics: flexibility, extensibility and plug-ability. In OOP an interface is a language construct that will specify a set of method signatures that will define the objects behavior. The signature of a method will include the name of the method, any input parameters and a return type. Interfaces are only a specification construct and do not contain any implementation. The implementation is left to the object that is declaring usage of the interface (Nandigam, et al., 2009). By using an interface one is able to decouple the classes within a system. This decoupling will help reduce the dependencies between the classes and subsystem. The reduction comes from an adhering to the reusability principle design of "*program to an interface, not to an implementation"* (Gamma, Helm, Johnson, & Vlissides, 1995).

The following is an example of an interface inherited by two different classes. In the Main procedure each of the classes is instantiated and stored into an array based upon the

class interface. The elements of the array can be accessed either as their base class or by the interface.

```
public interface IAnimal
{
   public void Name( string filename);
   public string Speak( );
   public void Birth( );
}

public class Bird : IAnimal
{
   Private string name;
   public void Name( string filename)
   {
      name = filename;
   }
   public string Speak( )
   {
      return "chirp, chirp";
   }
   public void Birth( )
   {
      //code to lay an egg
   }
}
public class Dog : IAnimal
{
   Private string name;
   public void Name( string filename)
   {
      name = filename;
   }
   public string Speak( )
   {
      return "bark, bark";
   }
   public void Birth( )
   {
      //code to have puppies
   }
}
```

```
public static void Main(string[] args)
{
   Bird bird = new Bird();
   Dog dog = new Dog();
   IAnimal[] animals = new IAnimal[2];

   animals[0] = bird;
   animals[1] = dog;

   dog.Name = "Fred";
   bird.Name = "Tweety";
   string output = "bird says " + animals[0].speak + " dog
   says " + animals[1].speak;
}
```

Figure 4: Interface Example

Instead of creating mock objects in isolation that duplicate the domain code Freeman, et al. (2004) propose using interfaces that are thus inherited by both the domain code and the mock object code. By inheriting the interface the work to maintain the mock objects is reduced. The reduction in work was achieved by coding to an interface, the mock objects then only need to implement that functionality needed to complete the testing. The usage of an interface allows a developer to create a class that needs another class that does not yet exist. The interface is created and used by the dependent class, then at some later time the interface is inherited by a class and its functionality is created. An interface also assists in keeping the mock code and base code synchronized. If a new construct is added to the interface it must also be added to the base and mock classes, if not the code will not compile.

In the example that follows we can see the difference in the usage of a stub and a mock object. Both the stub and the mock object are created by inheriting from an interface that readily allows for substitution of the actual object within the test code.

```
public interface IExtensionMgr { public bool IsValid(
   string filename); }
public class ExtensionMgrMock : IExtensionMgr
{
      public string lastError;
      public bool IsValid(string filename)
      {
         …
         if(invalidFilename) lastError =
            String.Format("Filename {0} is not valid",
            filename);
      }
}
public class ExtensionMgrStub : IExtensionMgr
{
      public void IsValid( string filename)
      {
        Return true;
      }
}
public class FileLogAnalyzer
{
   private IExtensionMgr manager;
   FileLogAnalyzer ()
   {
      manager = new FileExtensionMgr();
   }
   FileLogAnalyzer (IExtensionMgr mgr)
   {
      manager = mgr;
   }
   public bool setFilename (string name)
   {
      Return mgr.IsValid(name);
   }
}
```

Figure 5: Stub versus Mock

A stub would test if a filename is correct by calling the base class's method, in this case FileLogAnalyzer's log method. The code is:

```
public class TestFileLogAnalyzer …
{
   public void testIsNameValid()
   {
      FileLogAnalyzer log = new FileLogAnalyzer ();
      ExtensionMgrStub mailer = new ExtensionMgrStub ();
      log.setMailer(mailer);
      bool rtn = log. setFilename ("afile");
      Assert.IsTrue(rtn);
   }
}
```

Figure 6: A stub

A mock object would test if a message is sent by calling the mock object, as shown:

```
public class TestFileLogAnalyzer …
{
   public void testOrderMsgSent()
   {
      FileLogAnalyzer log = new FileLogAnalyzer ();
      ExtensionMgrMock mailer = new ExtensionMgrMock ();
      log.setMailer(mailer);
      log. setFilename ("afile");
      Assert.AreEqual("Filename {0} is not valid",
         mailer.SentCount());.
   }
}
```

Figure 7: A mock

Mortensen, Ghosh, and Bieman (2006) looked at how to test during refactoring of existing OOP code so that AOP and aspects are used to handle cross-cutting concerns. The authors proposed using a mock system to reduce the time needed for the unit testing of the aspects. In their work a mock system was created that duplicated the functions and classes that existed in the actual system. It was found that using a mock system reduced the compile and weave times. This decrease in compilation time allowed a developer to quickly experiment with different pointcuts and advice. The mock system provided for a fast and effective way to develop and test the pointcuts and aspects. However, in this

work and their following work Mortensen, Ghosh, and Bieman (2012) acknowledge a problem with maintaining the mock system to match the real system.

Mock objects as introduced by Mackinnon, et al. (2001) have introduced a way to handle external interactions with objects outside the unit being tested. In their follow up work Freeman, et al. (2004) further explored mock objects but added the usage of interfaces. The authors showed that a developer is able to create an interface to handle the unit's external interactions, but as their goal was to look at refactoring code from OOP to AOP they did not explore the effectiveness of testing aspects with mock objects using class interfaces.

Goal

This paper proposes to determine the effectiveness of developer testing of aspects using mock objects that inherit from a class interface. The class interface will be inherited by both the base code and the mock object code. By using mock objects in the testing it is proposed that the aspects will be given context outside of the base code, thus the aspects can then be independently tested for joinpoint cover and advice correctness. By using a class interface that is common to both the base code classes and the test code mock objects classes it is proposed that the effort needed to keep the two systems synchronized will be simplified. The study will be using coverage criteria as proposed by Mortensen and Alexander (2005) to determine the effectiveness of the tests and determine the effectiveness of the class interface in the developer testing.

Research Question

A developer of aspect classes needs to be able to test the code that is written. There are numerous questions about how this testing can be accomplished. This study intends to look at the questions of:

1. The testing of the aspect consists of validating joinpoint coverage and advice code correctness. With advice not having their own context, how capable are mock objects at supplying the needed context?
2. What is the benefit to a developer of using a class interface for keeping a mock object and base application synchronized?
3. How many object constructs are available within a class interface that can have joinpoints?

Relevance and Significance

As presented in Parizi and Ghani (2007), software testing has many definitions. Testing can be defined as the act of executing a portion of code or an entire application with the intent of finding errors. A second definition presented is that software testing has the goal of evaluating any attribute. A third definition is that testing can be the capability of an application or system to determine if it meets its functional requirements.

In Zhu, Hall, and May (1997), the authors state that the testing of an application can show the presence of defects, but never the absence of defects. Parizi and Ghani (2007) state software testing has the goal of quality assurance, verification, validation and reliability estimation. Quality assurance refers to the ability of an application to perform as required by its specifications. Verification refers to the accuracy with which all

software development documentation was transformed into the application. Validation is the evolution of the accuracy of model input/output with respect to the input/output of the application. The reliability estimation is a probability value representing how likely the application is fault free (Parizi & Ghani, 2007).

The usage of AOP to create a software application, as with object-oriented and procedure-oriented, has the goal of creating high quality software. As discussed by Alexander et al. (2004) what AOP brings is new questions and challenges to the testing of applications. Due to the uniqueness of aspects the challenges of testing include:

- Aspects do not exist independently.
- Aspects tend to be tightly woven with their coupled classes.
- When reading through the source code of an application, the control and data dependencies that exist between a class and an aspect are not always obvious.
- When a defect occurs there exist many possible locations for the defect to occur.

As Alexander et al. (2004) point out; AOP adds to the modularity and cohesion, thereby increasing understanding and easing the maintenance burden. With the growth of AOP, the authors raise the following questions:

- How do we properly test AOP?
- How do we know when it is properly tested?
- Is there a way to tell when we have tested enough?

In Mortensen, Ghosh, and Bieman, (2006) the ability to use mock objects for testing of aspects was shown to be feasible. The study used AspectC++ in the refactoring of existing C++ applications to use aspects. To validate the aspect code mock objects were

used to test the aspect code. While the usage was successful the types of joinpoints was limited and the author's attested to a problem with keeping the mock object code synchronized with the base class code.

Barriers and Issues

AOP as a methodology has been around since its introduction in 1996. During this time advances have been made in the support of this methodology within various languages and development environments. The best supported implementation of AOP is AspectJ in the Eclipse IDE. There are implementations in other languages, such as C++ and C#. However, a number of the various language implementations were done as graduate research projects. Once the project was completed continued support and extension of the implementation stopped (Parizi & Ghani, 2007).

During the past decade there have been a variety of research projects on the ability to test AOP applications. The existing research has focused mainly on the ability to detect whether joinpoints are properly woven into the base application. The work by Mortensen, et al. (2206) did look at the unit testing of aspects at the developer level. This work still focused on the proper weaving of joinpoints and the testing of the advice code. As noted by Kollanus (2010), while there have been numerous studies on the usage of unit testing and mock objects the current empirical evidence on their effectiveness has yielded contradictory results.

Limitations

This study created a Java software application to use as the base application for the aspects. JUnit was used to run the unit tests. The advantage to creating a software application as opposed to using an existing application is that the study will not be restricted to only those objects needed to meet functional requirements, but can instead create objects as needed for the sole purpose of the study. This approach does have the limitation of not being a very robust application and there are limited function requirements for the application to meet.

This study used mock objects within unit tests to validate that joinpoints are correctly woven with the desired pointcuts. There are a multitude of ways to write a pointcut and an equally large number of ways to create a joinpoint. This study did not attempt to look at all combinations of joinpoints and pointcuts for any given object. The scope of the study was limited to using each of the various types of pointcuts supported creating instances that are generic in nature. Against these pointcuts, joinpoints were created that ranged from a very exacting definition to a very broad definition. The desire was to have a wide sampling of joinpoints and pointcuts to test, validating that the pointcuts were woven to the desired joinpoints.

This study also used mock objects for the testing of an aspects advice code. The code within the advice can vary greatly depending on the needs of the application. The code written did not try to be all inclusive, but to be a general representation of the types of functionality that can exist within an advice module. In an attempt to overcome these limitations the study was not only executed against the test application being created, but

also contained testing against existing open source applications. By executing the study against open source code the desire was to have a more robust sampling of actual production software.

Definitions of Terms

Term	Definition
Advice	A piece of code within an aspect that is executed when a joinpoint is reached.
Aspect	A unit of modularity similar to a class. They contain pointcuts, advice and inter-type declarations
Cross-cutting concern	Software features that are spread across multiple code modules resulting in a 1 to n implementations of design implementations to feature requirement.
Interface	A programming construct that enforces certain properties upon inheriting classes.
Joinpoint	A well-defined point in the program flow
Mock object	Dummy implementation of domain code that emulates the real code.
Pointcut	Picks out specific joinpoints in the program flow.
Stub	A controllable replacement for an external dependency.
Unit	A method or function.
Unit test	A piece of code that invokes another piece of code with the intent of checking the correctness of the invoked code
Weaving	The task undertaken by the AOP compiler to link the classes and aspects together to produce an executable.

Summary

AOP is a methodology that allows for the handling of cross-cutting concerns found within a software application. AOP is able to modularize cross-cutting concerns so that

there is a one-to-one mapping of design concepts and functional requirements. This increases the encapsulation of the design requirements into separate modules, aspects.

To ensure that the requirements are met an application must be tested at the developer and at the integration levels. This testing applies to code contained within classes as well as the code implemented within an aspect. While AOP has been around for over a decade there is still a lack of research on the ability to test aspects at the developer level or at the integration level. Mortensen, Ghosh, and Bieman, (2006) used unit testing and mock objects to test aspects created in the refactoring of C++ applications to use aspects, this study extended their work to include the usage of class interfaces with the mock objects for the testing of aspects.

In Mortensen, et al. (2008) the study's main focus was the ability to refactor C++ applications to use aspects. As part of the study, unit testing with mock objects was used in the testing of the aspect code. That study was limited in the type of joinpoints to only those needed by the existing applications. The authors also noted that there was a problem in trying to keep the base code synchronized with the mock objects. This study expanded on the work done by Mortensen, et al. to include constructors, exceptions, object instantiations, and properties. This study evaluated the effectiveness of having the mock object classes inherit from a class interface common to the base code.

Chapter 2

Review of the Literature

The existing research done on AOP has focused on software development; e.g. requirements, design, implementation and discovery of crosscutting concerns (Kumar, Sharma, & Garg, 2009). While the research done is important, testing of aspects has received very little research attention. The research that has been done falls into four categories; data-flow based unit testing, joinpoint testing, state based testing and unit testing of aspects. Prior to delving into a review of the research of AOP testing, a brief review of the literature on unit testing will be done.

Unit Testing

As noted earlier unit testing was developed in the 1970's in association with SmallTalk. A unit is a small piece of code and thus works well with OOP and AOP. In OOP the software is divided into units called classes which contain methods and AOP software adds units called aspects which contain advice. In unit testing, each unit is tested in isolation and the results are compared against the expected results. Because unit tests are completed in isolation, additional dependencies are often required by the unit under test. The usage of mock objects was first introduced to the testing community in the work by Mackinnon, Freeman, and Craig (2001) as a way of solving the missing dependency problem. The authors proposed that domain objects be replaced with

dummy implementations to simulate the real code. The mock object will be used by the test object instead of the actual class object. The usage of a mock object is similar to the usage of stub objects except for two differences; 1) the mock object is able to test to a finer granularity and 2) the tests plus stubs are used in the development.

In the work by Thomas and Hunte (2002), the authors build upon the work done by MacKinnon, et al (2001). The study attempts to further clarify what constitutes a mock object. The authors offered the following six reasons to use a mock object:

- The real object has nondeterministic behavior
- The real object is hard to set up
- It is hard to trigger the real object
- There exists a user interface with the real object
- The real object has not been developed
- The real object needs to be queried about usage

In Sobering, Cook, and Anderson (2004) it is suggested that a mock object is more than what is needed. Instead of using a mock object they proposed to use a pseudo-class. The starting point of the pseudo-class is the type-declaration interface. To actually create a class the interface would be inherited and only the desired methods to be tested would be implemented. The remaining methods in the interface would simply throw an unimplemented exception. By taking this approach the creation time of the pseudo-class is greatly reduced. It was found that by using the interfaces with mock objects the creation and maintenance of the mock objects was greatly reduced.

In Freeman, Mackinnon, Pryce, and Walnes (2004) the authors refine and adjust their earlier work in MacKinnon et al (2001). Their first work showed how mock objects encouraged better structured tests and improved domain code by increasing encapsulation. In their new work, the authors looked at the usage and better understandability provided by the usage of interfaces. They observed that mock development is similar to Lean Development. In both, a core principle is that an object exists out of demand and is not pulled into existence. By testing objects as units, the objects are tested in isolation from the rest of the application. This isolation will force a tester to consider an object's interactions with its collaborators in the abstract and possibly before the collaborators even exist. The collaborators that an object requires will be filled by the usage of interfaces in the mock objects.

Data-flow Based Unit Testing

In Zhao (2003) and Wedyan and Ghosh (2010) data flow unit testing was studied as a method of testing AOP. Data flow testing focuses on the value assignment for each variable in the code. This is done by tracking the variable from instantiation to a desired point in the code where the variable is used. To do the testing the desired variables were first identified within the source code. The source code was then compiled to get the byte code and then using the byte code the selected variables are tracked.

The study by Zhao (2003) was able to work with computational Def-Use associations in a three level testing approach. The three levels covered the perspectives of intra-module, inter-module and intra-aspect/class. To build the def-use pairs the source code into five categories:

1. Clustering aspect – an aspect that affects some class or classes.
2. Clustering class – a class that is affected by one or more aspects.
3. Clustering method – a method that is affected by one or more advice.
4. Normal class – a class that is not affected by any aspects.
5. Normal method – a method that is not affected by any aspect.

Flow graphs would be constructed for each of the categories and tested separately. In Zhao (2003) only the clustering aspects (c-aspect) and clustering classes (c-class) are studied.

The construction of the flow graph required a three step process beginning with a call graph. A call graph represents the call relationships that exist between the various modules. For a c-aspect the call graph will contain only vertices that represent modules within the c-aspect and arcs that represent the calling relationships between the modules. The call sequence is shown by using directed edges between the modules of the c-aspect. The second step consists of adding vertices to the call graph. These vertices consist of an entry vertex, loop vertex, exit vertex, return vertex and a call vertex. These vertices combined create a frame that allow for the simulation of randomly calling certain modules within a c-aspect or c-class.

The last step in the construction has each vertex that represents a module in the c-aspect replaced by its own control flow graph. The result of the above steps results in the construction of what Zhao calls a Framed Control Flow Graph. For the modules identified a three level testing approach was performed, the levels consisted of:

- Intra-module - testing is performed on variables that have def-use pairs within a single module.

- Inter-module – testing is performed on variables that have def-use pairs across modules
- Intra-aspect/class – testing is performed on variables that have def-use pairs within a module but can be called in any order by user of the module

The study found that it was possible to handle testing problems that are unique to aspect-oriented programming. While study was able to show that it could identify def-use pairs for testing, the study did not cover a variable with a predicate usage. The study also failed to handle around advice, join points with multiple advice or dynamic pointcuts. Around advice have the ability to change the behavior, control or data dependencies of a method resulting in a change of def-use pairs.

The work by Wedyan and Gosh (2010) expanded on the work of Zhao (2003) by including in the study the predicate usage of variables and around advice. The research started with the tool AJANA, created by Xu and Rountev (Wedyan & Ghosh, 2010). AJANA creates CFG's of the methods and advice in the program. The CFG's of the advised methods are merged with the DFG's of the related advice uning interactive graphs. There will exist one interactive graph for each joinpoint. At each joinpoint that matchs a before or after advice the : (1) call-site and return-site nodes are added to the CFG, (2) the call-site node is connected to the entry node in the DFG of the desired advice, and (3) the exit node of the advice CFG is connected to the return-site node. If an around advice is called, its CFG replaces that of the method. Should the around advice contain a proceed statement, the advice CFG is connected to the method CFG using call-

site and return-site nodes. The resulting ICFG shows what methods and advice are invoked from any single call to a class or method.

The AJANA tool does not use a frame as proposed by Zhao (2003). To follow in Zhao's work the study modified the AJANA tool to include frames for ICFG's to create the frame the following nodes and edges were added:

- Frame entry node – the entry to the frame with edges to the entry nodes of CFG's for public constructors
- Frame exit node – exit points from the frame.
- Frame edges – connections between the exit node of a CFG representing either a public method or constructor and the entry node of a CFG of a public method or constructor.

With the modified AJANA tool the study was able to then create Def-Use Associations for the targeted AOP application. These DUA's are then monitored in the bytecode to measure their coverage when executed against a test suite.

The research used the AjMutater application to seed faults into the test application written for the study. The study found that it was possible to use the automated test generation tool AjMutater if the aspects are written in @AspectJ annotation style. The study does have the limitation of using a small test application and the mutation operators used might not be representative of AO program faults.

State Based Testing

Xu and Xu, (2006) did work in the area of state-based testing. The advices within an aspect have the ability to change the state model of the core concerns. The new states

that are introduced to the system can cause the classes to violate their state invariants. Flattened Regular Expressions (FREE) are comparable to UML state models. By using FREE tests from OOP the authors are able to build state models called Aspectual State Model (ASM). By adding in FREE the authors were able to specify both classes and aspects. The entities were then placed in a transitional tree and from the transitional tree they were able to identify abstract test cases.

An example of an ASM is shown in Figure 8. State A has received a message to go to State B. There are three advice that are associated with the creation of State B, a before, an around and an after. At the first condition point the message is affected by the before advice moving the state to State C. At condition 2 the around advice is encountered and moving the state to State D. The last condition encountered is for the after advice which moves the message to State E. This is a representation of the conditional transition tree and represents both legal and illegal events.

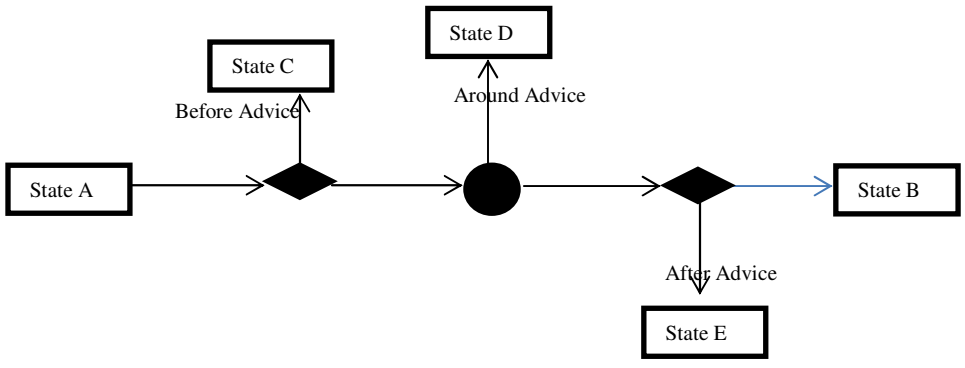

Figure 8: Example of an ASM

The study found that when treating aspects as incremental modifications to a base class, it is possible to extend base class state models to show the impact of aspects on

states and transitions of base class objects and generate abstract test cases. Additionally, the study showed that it was possible to reuse base class tests for conformance with AOP. The main issue with this approach is that the state based diagram can suffer from state explosion. While the study showed it is possible to reuse the base class tests a tester needs to be aware that a positive (or negative) test can become a negative (or positive) aspect test.

Joinpoint Testing

Joinpoint testing was done in the work by Lemos and Masiero (2008) and Wedyan and Ghosh (2008). In joinpoint testing, the work is designed to look at the coverage of a class by an aspect. The objective of the research is to determine if the joinpoints are being executed against the desired pointcuts and only the desired pointcuts. To make the coverage determination the authors inspected the byte code. The authors were looking for four types of pointcut faults:

- Selecting only a subset of the adequate joinpoints
- Selecting a superset of the adequate joinpoints
- Selecting a set of joinpoints with no intersection with the adequate set of joinpoints
- Selecting a subset of the adequate joinpoints and unintended joinpoints.

In Lemos and Masiero (2008) the focus is on structured integration testing. The method and the advice are considered to be the smallest units to be tested. In structured testing the CFG is used to represent the flow of control in the program with a node being a block of code and an edge representing the flow from one node to another. The base

unit for testing in this paper was the aspect-oriented def-use graph (AODU). An AODU is generated for each unit to be tested, either a method or an advice. A graph consists of a set of nodes (N) – composed of blocks of bytecode instructions that are executed. Edges (D) that connect the nodes, showing transfer of control. Entry points (s), exit points (T) and the nodes (C) that are affected by advice. The graphs also differentiated regular edges from exception edges, exception edges represent a flow resulting from an exception handler. A completed AODU graph has the following conventions: single circle node for regular blocks, double circle nodes for method calls, bold nodes for exit nodes, dashed ellipses for advice, regular edges are for regular flow and dashed edges for exception flow.

With the AODU's the study creates pointcut-based CFG's for each advice-pointcut pair. The PCCFG's consist of nodes and edges of the base units which are selected by a pointcut. Each PCCFG is labeled based on its bytecode offset and corresponding block. Then all PCCFG nodes and edges should be exercised at least once by a test case. If this is not the case then an error can occur since there is not 100% coverage of the pointcuts by test cases. With the PCCFG's the study was able to help in gaining confidence that all advice paths were properly executed during testing. The authors did note that there exists a scalability problem with advice that affect a large portion of the program and thus creating a large PCCFG.

Wedyan and Gosh (2008) present a tool that will assist in the measurement of joinpoint coverage for given test suite. The tool has the ability to measure from two perspectives; by advice, which measures the execution of an advice at each joinpoint and

by class, which measures the execution of all advice within the class. The tool uses the bytecode only of the post woven application.

The tool works in two phases. In the first phase the bytecode is parsed looking for joinpoint shadows. A joinpoint shadow is inserted by the AspectJ weaver in the locations of possible joinpoints. The located joinpoint shadow information is stored by the tool in an XML based file. The bytecode is also parsed for advice information which is stored in an XML file. Once the bytecode has been parsed the test runner component of the tool executes the test driver of the test suite. The runner component tracks the covered joinpoints whenever a shadow is executed. When the test is completed there is a listing consisting of joinpoints in the class, number of executed joinpoints, how often each joinpoint is executed and overall joinpoint coverage. When the tool was tested against the Apache BCEL library it was able to locate joinpoints within the tests executed against the library.

Unit Testing of Aspects

In the research by Lopes and Ngo (2005), the authors claim that unit testing of AOP code is difficult. In their work they look to see if a pointcut definition matches anything or if it includes and excludes only those joinpoints that are supposed to be captured. To do the tests the authors used Java Aspect Markup Language (JAML) for performing their unit tests. JAML is a markup language created by the authors for implementing aspects. The language uses core modules and aspect modules that contain a developer's code written in Java. The aspect binding is handled with the XML-based specifications specific to JAML. The specifications contain the binding instructions that determine how

the core and aspect modules are composed. The binding rules are the same as found in AspectJ.

With the JAML system in place the authors created applications to show that effectiveness of their system. To show that the bindings were properly made the authors used an extension of JUnit that would handle the JAML applications. The authors tested the aspect bindings, such as pointcut definitions, but state that the testing could be done on such constructs as interceptors and introductions.

It was found that the system created was able to provide a cost-effective means of testing and finding faults in aspect programs. It was also noted that there was success with the approach, there exist two challenges:

- Selecting the appropriate mock joinpoints. The JAML required that the testers select and mock the joinpoints to be tested. If an aspect has numerous joinpoints within a base class it would be necessary for a tester to find and create mock objects for each occurrence. The authors felt that creating mock objects for all joinpoints to be too daunting of a task and thus recommended identifying the most meaningful joinpoints and select only those for testing.
- Creating mock execution content. As noted an aspect can have numerous joinpoints and thus need a mock object for each joinpoint. With each mock object code must be created to test the joinpoint, which the authors felt would be a lengthy and difficult task. The authors see a need for support tools that would create the mock object content.

In the works by Mortensen, Gosh and Bieman (2006, 2008, 2012) the authors refactored three commercial VLSI CAD applications created at the Hewett-Packard

Company. The refactoring goal was to replace cross-cutting concerns within the three applications with aspects. Since the original applications were created with C++, the aspects were created using Aspect-C++. The steps taken to refactor the applications were:

- Identify the cross-cutting concerns.
- Create the aspects and unit test
- Remove the cross-cutting code in the applications and weave in the aspects
- Perform regression testing on the applications using existing regression tests.

To identify the cross-cutting concerns within the applications, the code was manually reviewed to find candidates. The manual review was used due to a lack of tools available for C++ analysis. Once the cross-cutting candidates were found aspects were created to mimic the existing functionality.

To test the aspects that were created, the authors developed a set of mock objects. The mock system, to be useful, must contain joinpoints that use naming conventions and structures consistent with the real system. The mock system created contained a subset of the methods, functions and classes found within the actual applications. When the aspects were created they were woven with the mock system. Using unit tests the aspects were tested until satisfactory joinpoint coverage was reached. Satisfactory joinpoint coverage was determined by checking logs recording whether all pointcuts had been woven with joinpoints in the mock system. They also checked the logs to determine if all advice methods had a pointcut woven to them.

Once testing with the mock system was completed, the advices were woven with the real applications code. Regression tests were executed to validate that the applications were executing as they had before the refactoring. To validate the weaving of the pointcuts to the joinpoints the authors created tools that would parse the woven code. The tools would check for unused aspects or unadvised methods, determine joinpoint coverage and if any advice or joinpoints with advice had not been tested.

Mortensen, et al. (2008) found that they were able to improve testing with the mock system and unit testing. The largest gain was with compilation and weave times. With the mock system being a smaller system the developer was able to quickly test, make error corrections, recompile and retest using the mock system. The study did have problems with keeping the mock system synchronized with the actual application. When a developer made a change in the real system they had to make sure the mock system was also updated to reflect the change. The study was also limited in scope to joinpoints consisting of classes and methods.

The research by Xie, Zhao, Marinov, and Notkin (2006) studied testing AOSD applications with Unit tests. In the study the authors:

- Present Raspect, a tool created to detect redundant unit tests in AspectJ projects.
- Presented an implementation of Raspect for detecting redundant unit tests on advised methods, advice and intertype methods.
- Evaluated Raspect using twelve AspectJ applications ranging from simple aspect sample applications to open source applications.

To use the Raspect tool the desire aspects are created then unit tests are constructed to test the aspects. The Raspect tool works by dynamically monitoring the executing tests. With each test execution, sequences of method calls are produced. The method execution's behavior is dependent upon the state of the receiving object and the method arguments. Raspect identifies and collects these method-entry states. The method-entry states are then linearized to allow for ease of comparison. Linearization's that are determined to b equivalent are deemed to be redundant.

In the study the authors used the AspectJ compiler to compile and weave the twelve applications being tested. To create unit tests the generated byte-code was feed into Purasoft Jtest, a tool for creating unit tests. The unit tests were then executed with Raspect monitoring the executions. Raspect was able to detect and determine that Jtest was producing a number of redundant unit tests. The authors feel that their tool is capable of assisting developers and testers by helping to identify redundant unit tests and reduce the number of unit tests being executed, thus reducing the amount of time needed to test AspectJ applications.

Chapter 3

Methodology

Research Methods

This research investigated developer testing of aspects, validating and extending the work done by Mortensen, et al. (2006). In testing of software there exist different phases of testing, two of which are developer and integration (Parizi & Ghani, 2007). Developer testing is done by the person writing the software and it is intended to validate that the code works correctly. Integration testing is done after developer testing and is designed to assure that all of the modules work together correctly.

In the work by Mortensen, et al. (2006) the authors validated that they could use a mock system to test the aspects created in refactoring a C++ application. A mock system was created that contained functions and classes as found in the real system, but on a much simpler scale. This mock system was created by copying a subset of the classes, methods and functions found in the real system to create the mock system. The mock system was created so that it contained the joinpoints used in the real system. In the study the main object was to study the refactoring of a C++ application to use aspects. Because of this the types of joinpoints used consisted of method calls and executions. In creating the mock system it was noted by Mortensen et al. (2008) that there was a cost in

maintaining the mock system. Every time there were changes made to the real system these changes had to be mirrored in the mock system.

In using mock objects a mock is easiest to create and use if one adheres to an interface based design (Nandigam, Gudivada, Hamou-Lhadj, & Tao, 2009). If one does not commit to a concrete implementation and instead uses an interface mock objects can be substituted for the real objects since both adhere to the same interface. By using an Interface Segregation Principle clients and mock objects are not forced to implement methods that are not required. The use of the interface will help reduce the complexity of the mock object, with only the needed interfaces implemented by the mock object (Bender & McWherter, 2011).

This study added the usage of a class interface inherited by the base application classes and the mock object classes. The study evaluated the effectiveness of the class interface in making available the type of objects that can be used as joinpoints. By using interfaces the resulting systems plug ability is increased allowing for gracefully substituting of identical objects (Nandigam, et al., 2009).

In refactoring existing systems it was shown that mock objects could be used to test the joinpoints for class methods and functions. In this study the types of objects used as joinpoints was expanded to include constructors, exceptions, object instantiations, and properties. After creation of the expanded list of joinpoints the study evaluated whether it is possible to test the joinpoints and advice using the mock objects.

The study also looked at using a class interface to assist in developing the mock object. The study tested to see if the joinpoints used by the aspects can be created in the

class interface. The effectiveness of the class interface construct was determined by the number and variety of joinpoints that could be defined in the interface.

Test Criteria

In Zhu, Hall, and May (1997), the authors look at the concept of test adequacy for software applications. To determine if an application is adequately tested, there has to exist a set of test criterion. The criterion for testing is one of four types:

- Statement Coverage – Testers are expected to create tests that will execute all of the statements in the application.
- Branch Coverage – Testers are expected to test all locations where control transfers.
- Path coverage – All paths from start to finish through the application are expected to be tested.
- Mutation Coverage – Tests are designed to find faults in the software.

In Mortensen and Alexander (2005) the coverage criteria were modified and expanded to encompass aspects. The coverage definitions are:

- Statement Coverage – Every path through the aspect code fragment is executed after weaving.
- Insertion Coverage – Each aspect code fragment is tested at the points where it is woven into the base code.
- Context Coverage – This is a combination of statement and insertion coverage, so that each aspect code fragment is executed at its point of insertion.

- Def-use Coverage – this is the testing of def-use pairs within advice, between different advice, between advice and methods, and where flow control has been changed because of an advice.

With a set of criteria one is able to determine when a program has been adequately tested. Zhu et al. (1997) created definitions that quantify Stopping Rules and Measurements based upon the criteria set for test data adequacy. In classifying adequacy criteria the usage of the underlying testing approach is one method available. When classifying by testing approach there are three types of approaches to the testing:

- Structural testing – testing requirements are based in terms of coverage for a particular set of elements in the program.
- Fault based testing – testing centers on finding defects within the software.
- Error based testing – testing requires that test cases are checking the program for certain error-prone points.

To validate the testing of the aspects this research followed the work of Mortensen and Alexander (2005) in using coverage criteria. The criteria used to determine adequacy of advice tested within an aspect, as used by Mortensen and Alexander (2005) and then again by Wedyan and Ghosh (2008), was statement coverage. To determine adequacy of pointcuts tested the research used insertion coverage. As with Wedyan and Ghosh (2008) the ability to determine the insertion coverage was determined by analyzing the bytecode generated from the weaving process.

This research focused on developer testing of the aspect code and not the integration testing of the aspects to the base application. As such the research focused on the testing

done with the mock objects. The mock objects were used to validate that the aspect code worked as designed.

As with Mortesen and Bienem (2006), to test the aspects with the mock objects the research ensured that each aspect is covered by test cases. In the testing there are two criteria to be met for having all the advice within an aspect tested. The coverage criteria of the advise, but the criteria does not include having the advice woven against all joinpoints only that the advice itself is tested. Joinpoint coverage is achieved by having the advice tested and that the tests find all of the intended joinpoints.

The last criteria used in the research measured the effectiveness of the class interface in assisting in keeping the base classes and the mock object classes synchronized. This criterion consisted of a comparison between the number of possible joinpoint types available to an advice and the joinpoints that can exist in a class interface.

With the criteria listed above the research questions will be evaluated using with the following metrics:

- Insertion coverage, the percentage of joinpoint that were correctly woven against the mock pointcuts.
- Code coverage, the percentage of code that was covered by unit tests.
- Development time, the time needed to develop classes using and not using a class interface within the mock project.
- Class objects that are declarable within a class interface that can be matched by joinpoint.

Implementation Strategy

This research used Java, AspectJ and the Eclipse IDE. The study selected Java and AspectJ so that it was testing a different language than C++ used by Mortensen et al (2006). Additionally AspectJ has a number of open source projects that provided for a diverse collection of code to test. The study consisted of studying the ability to use unit tests at the developer level to test the correctness of an advice. The correctness included the ability to test the code contained within an aspects advice as well as determining whether an aspects joinpoints were woven with the correct pointcuts. Mock objects were used to give the aspects context and allow for the testing of the joinpoints and advice code within the aspects. The testing was conducted on a sample application created for this study and then repeated on existing open source code. In Zhu, Hall and May (1997), it is pointed out that while the testing of an application can show the presence of defects, it can never show the absence of defects.

A test set is a set of test cases for testing an application or part of an application. A test set should have a goal of executing all of the statements within an application. The percentage of statements executed is a measure of adequacy (Zhu et al., 1997). How well the unit tests are exercising the application code is expressed by the code test coverage or the code coverage (Bender & McWherter, 2011). Code coverage of 100% does not guarantee good code only that all of the code has been subjected to a test.

In creating the unit tests for this study, the optimal goal was to have all of the aspects fully covered by a test set. While the achievement of 100% coverage of the joinpoints

and the advice would be optimal it was not a goal of this research. This research instead attempts to set a bench mark level for aspect testers to use as a comparison.

In the research a software application was written that simulates a banking application. The application is be able to create new accounts, allow for withdrawals, accept deposits and close an account. Aspects were created to check for sufficiency of funds and to log transactions. Approximately half of the classes in the application were inherit from a class interface. The class interfaces created were inherited by a mock object for the testing of the aspects and the classes without interfaces were duplicated in the mock system. Once the mock objects were created, one set of unit tests were created to test the aspect functionality and to test the joinpoints of the aspects. The effectiveness of the unit tests written to test aspect functionality was determined using statement coverage. To do the testing of the aspects mock object classes were created inheriting from the classes interfaces, thus simulating the actual application classes. Unit tests, using the JUnit framework, were created with the goal of covering all execution paths through each aspect. Each unit test contained at least one assert statement that validated a portion of the aspect code. An assert statement evaluated to true when the desired value is obtained from the aspect code under test and at that point, it was be considered to have passed.

The effectiveness of the joinpoint unit tests was evaluated using insertion coverage metric. Using the same mock objects created for the statement coverage tests, a set of unit tests was created that validated that the before, after, and around advice executed against the desired joinpoints. These unit tests each had at least one assert statement that

was capable of validating that each before, after and around advice executed against advice's joinpoints.

To further validate the results of the testing done using the banking application, testing was performed against additional existing applications, see Table 1. The applications contain aspects containing all three types of advice; before, after and around. The advices have pointcuts that range from being narrow in scope to broad in scope. The pointcuts have joinpoints using many different class object types.

Table 1: Overview of AspectJ programs evaluated

Name	Lines of Code	Source	Description
Introduction	200	AspectJ example[1]	Simple example using aspects
Telecom	725	AspectJ example[1]	A telephony simulation
Spacewar	2,700	AspectJ example[1]	
AJHotDraw	22,104	open source project[2]	2D Graphics Framework
PetStoreAspectJ	17,800	open source project[2]	
Bean	246	AspectJ example[1]	Makes point objects into Java Beans

The testing of the aspects, as done by Mortensen et al. (2008), consisted of validating the pointcuts were correctly finding their intended joinpoints and that the code within the advice worked as intended. The joinpoint coverage was checked by using three different types of patterns within the mock system.

The first pattern type tested joinpoints that are not affected by the advice creation. An example of such an aspect would be an aspect that logs the joinpoint was called. To test

[1] http://www.eclipse.org/aspectj/index.php
[2] http://www.kevinjhoffman.com/tosem2012/

this pattern a mock object was created such that the naming convention of the pointcut in the mock object matches that in the real system. This pattern tested joinpoint coverage. Such as when an IAccount class is modified the aspect will log a message indicating that the account was modified. The aspect does not have an effect on the actual IAccount object.

```
pointcut accountModified(Account acct) : call(*
   Account+.*(..)) && target(acct);

after (Account acct) : accountModified(acct){
   System.out.println("Account modified, account number-" +
      acct.IdNum());
}

@Test
public void testAccountModified(){
   mockAccount tmp = new mockAccount();
   tmp.Balance(5000.00F);
   assertEquals("Check balance",5000.00F, tmp.Balance(),
      0.009);
}
```

Figure 9: Unit Test Account Id updated

The second pattern type tested that the advice does not change the existing joinpoint functionality. This pattern was tested by creating joinpoints that match between mock and real system. Simple functionality is added to the mock that validated the advice does not change the joinpoint functionality. The mock joinpoint was executed with differing parameters values to validate that the joinpoint functionality was not altered. The pattern test validated joinpoint coverage and the woven mock system validated the advice statements were executed for statement coverage. The personAdded pointcut is woven to those instances where the AddPerson routine is called. When the advice executes the

person object is assigned an identification number. In the test code there is change to the functionality, but the object is updated by the aspect.

```
pointcut personAdded(Person per) : call(void AddPerson(..))
         && args(per);

before(Person per): personAdded(per){
      per.IdNum(People.MaxId++);
}

@Test
public void testAddPerson(){
   Person tmp = new Person();
   tmp.FirstName("Fred");
   tmp.LastName("Jones");
   tmp.City("Paradise");
   tmp.State("pa");
   tmp.Zip("02038");

   mockPeople people = new mockPeople();
   people.AddPerson(tmp);
   Person chk = people.GetPerson("1");
   assertEquals("Set id to 1", 1, chk.IdNum());
   assertEquals("Set maxid to 1001", 1000,
   mockPeople.MaxId);
}
```

Figure 10: Person assigned id by advice

The third pattern type involved making sure the mock object has the functionality necessary for the advice to execute. There are times when the advice requires methods and data structures to be present to execute properly. The mock object must contain the needed methods and data structures. This pattern tested that the joinpoint exists in the mock object and matches that in the application class. Any data structures needed by the advice had to be identified and created within the mock object. As with the second pattern, this pattern validated that pointcut provides joinpoint coverage and was able to test the advice execution, giving statement coverage. An example of this is the

Customers class which stores a list of Customer objects; the mock Customers class does not need a full list. In the mock Customers class the list was replaced with a mock Customer variable. Then in the unit test the mock object needed was created. The joinpoints were woven into the mock Customers class and the additionally needed mock objects were available.

```
public class Customers implements ICustomers {

   static int MaxId = 1000;
   protected Customer holder;

   @Override
   public void AddCustomer(ICustomer toAdd) throws
      Exception {
      holder = (Customer)toAdd;

   }

   @Override
   public void UpdateCustomer(ICustomer toUpdate) throws
      Exception{
      holder = (Customer)toUpdate;
   }

   @Override
   public ICustomer GetCustomer(String Id) {
      return holder;
   }

   @Override
   public void CustomerLeaves(ICustomer toAdd) {
      holder.CloseCust();
   }

   @Override
   public int GetMaxId() { return MaxId;  }

   @Override
   public void IncrMaxId() {  MaxId++;    }
}
```

```
@Test
public void testAddMockCustomer(){
   try{
         mockCustomer tmp = new mockCustomer("Test Business
            Acct");
         mockAccount acct = new mockAccount();
         tmp.AddAccount(acct);
         customers.AddCustomer(tmp);
         mockCustomer rtn =
            (mockCustomer)customers.GetCustomer("1");
         assertEquals("Set id to 1000", 1000, rtn.IdNum());
         assertEquals("Set maxid to 1001", 1001,
            customers.GetMaxId());
   }
   catch (Exception ex){
         fail("Shouldn't be here");
   }
}
```

Figure 11: Unit Test Customer class

The testing of the usage of a class interface was accomplished by two methods. In Mortensen et al. (2008) a measurement of development time was used as one of the factors to determine if using mock objects is beneficial. This study also used a measurement of developer time. In creating the mock objects approximately half of the mock objects were created by inheriting from a class interface that the base code also inherits and approximately half of the mock objects were created by duplicating the base code. The amount of time to create and maintain each of the mock objects was kept so that a comparison of the time needed to create mock objects inheriting an interface and mock objects created by duplication could be made.

The second test for the class interface usage consisted of determining how many of the joinpoints needed could be created within a class interface. Not all data objects can exist within a class interface, but can still be joinpoints. The study tracked how many of

the joinpoints were able to be instantiated within the class interface structure. The usefulness of the class interface was determined by the percentage of joinpoints that could be instantiated within the class interface structure.

Procedure

Step 1: Created an AspectJ project that contained an application with class interfaces, application class code, and aspect code. The base application is a simple project that simulates a banking application. The base application is only for the purposes of testing the concept of using a class interface and mock objects for testing and as such has no real purpose. The Java project for the banking application was created containing one package, NovaSE. The package contained the application class interfaces, application classes and aspects. The class interfaces were created followed by the classes and the aspects. The IAccount interface was implemented by the Account class and the mock Account class.

```
package NovaSE;

public interface IAccount {
   public void AddUser(Person toAdd);
   public void RemoveUser(Person toAdd);
   public boolean NoUsers();
   public void IdNum(int nm);
   public int IdNum();
   public void Balance(float bal);
   public float Balance();
   public void CloseAcct();
   public boolean IsActive();
}
```

Figure 12: Class Interface

Step 2: Created the application class files within the NovaSE package. These files contained the functionality needed to for the application. The files included classes for customers (personal and business), accounts (savings, loans and checking) and additional classes Main and Setup to hand running the application. The Account class is one of the classes created:

```
public class Account implements IAccount{

   protected int IdNumber;
   private boolean acctActive;
   protected float balance = 0.0f;

   private HashMap<Integer, Person> acctUsers = new
      HashMap<Integer, Person>();

   public Account(){
      super();
      acctActive = true;
   }

   public void IdNum(int nm) {
      IdNumber = nm;
   }

   public int IdNum() {
      return IdNumber;
   }

   public boolean IsActive(){
      return acctActive;
   }

   public void CloseAcct(){
      if(acctActive == false) throw new
         BankingException("Account already closed.");
      acctActive = false;
   }
```

```
    public void AddUser(Person toAdd){
        if(toAdd == null) throw new BankingException("No
            Person to add.");
        acctUsers.put(toAdd.IdNum(), toAdd);
    }

    public void RemoveUser(Person toAdd){
        Person tmp = acctUsers.remove(toAdd.IdNum());
        if(tmp == null) throw new BankingException("Person
            not found");

        tmp = null;
    }

    @Override
    public boolean NoUsers() {
        return acctUsers.isEmpty();
    }

    @Override
    public void Balance(float bal) {
        balance = bal;
    }

    @Override
    public float Balance() {
        return balance;
    }
}
```

Figure 13: Account class in application inheriting interface

Step 3: Created the aspects, an example is the ValidateAccount aspect. The aspect joinpoints and advice were written to interact with the class objects. The code was not modified to work with the unit tests in the test project.

```
public aspect ValidateAccount {

    pointcut accountHasPerson(Account acct) :
        (execution(void Accounts.AddAccount(..)) ||
        execution(void Accounts.UpdateAccount(..)))  &&
        args(acct);
```

```
   pointcut accountModified(Account acct) : call(*
      Account+.*(..)) && target(acct) &&
      !within(ValidateAccount);

   pointcut newLoan(): call(ILoan+.new(float, int, float));

    pointcut loanBalanceChanged(ILoan acct):
       set(float balance) && target(acct) &&
          !withincode(new(..));
    after (Account acct) : accountModified(acct) &&
          !target(ValidateAccount){
       System.out.println("Account modified, account number-
       " + acct.IdNum());
    }

    before(Account acct) throws Exception :
          accountHasPerson(acct){
       if(acct.NoUsers())
          throw new Exception("Must Contain at least 1
             account");
    }

    after() returning(ILoan acct) : newLoan(){
       float monthlyRate = acct.InterestRate()/12;
       double a = monthlyRate / (1-1/Math.pow(1+monthlyRate,
       acct.LoanTerm()));
       acct.LoanBalance(acct.LoanBalance() +
       (acct.LoanBalance() * (float)a));
       acct.Payment(acct.LoanBalance()/acct.LoanTerm());
    }

    after(ILoan acct) : loanBalanceChanged(acct){
       if(acct.LoanBalance() <= 0){
          acct.LoanPaid();
          acct.Payment(0);
          System.out.println("Loan is paid off");
       }
    }
}
```

Figure 14: Aspect to validate Account classes

Step 4: Created the test project containing the exact same package, NovaSE, as the base application. The aspect class files and the class interface files were then linked from

the base application project to the test application project. By linking the files the exact same code will be used by both the base application and the test project. The linked files were placed in the NovaSE package as they were in the base application.

```
public class Account implements IAccount{

   private int idNum;
   protected float balance;
   private boolean active;
   Person holder;

   public Account(){
      super();
      active = true;
   }

   public void IdNum(int nm) {
      idNum = nm;
   }

   public int IdNum() {
      return idNum;
   }

   public void CloseAcct() {
      if(active == false) throw new
         BankingException("Account already closed.");
      active = false;
   }

   public boolean IsActive(){
      return active;
   }

   public void AddUser(Person toAdd) {
      if(toAdd == null) throw new BankingException("No
         Person to add.");
      holder = toAdd;

   }
```

```
    public void RemoveUser(Person toAdd) {
        if(holder == null) throw new BankingException("No
            Person to remove.");
        holder = null;
    }

    @Override
    public boolean NoUsers() {
        return holder == null;
    }

    @Override
    public void Balance(float bal) {
        balance = bal;
    }

    @Override
    public float Balance() {
        return balance;
    }
}
```

Figure 15: Mock Account class in test project

Step 5: Unit tests were created that are from one of the four patterns listed in the implementation section. The unit test files were next created in the NovaSE package. These unit tests tested the joinpoint coverage of the pointcuts and the statement coverage of the advice. As the unit tests were created the needed mock objects were added to the NovaSe package. The mock objects were created by either inheriting from a class interface or by duplicating the base code. In both cases the mock object only contained enough code to allow the unit tests to execute. An example is the mock Account class use to mock the Account class.

There is at least one unit test per aspect. Each unit test contains at least one assert statement to validate a portion of the aspect code. The testAddMockAccount, in the testAccount class, is an example of a unit test that was created. Its purpose was to

validate that when an account was created the account id's were properly added by the aspect.

```
@Test
public void testAddAccount(){

   try{
      Account tmp = new Account();
      Person per = new Person();
      tmp.AddUser(per);
      accounts.AddAccount(tmp);
      mockAccount rtn =
         (mockAccount)accounts.GetAccount(1);
      assertEquals("id set to 1000", 1000, rtn.IdNum());
      assertEquals("maxid set to 1001", 1001,
         accounts.GetMaxId());
   }
   catch (Exception ex){
      fail("Shouldn't be here");
   }
}
```

Figure 16: Unit test before aspect weaving

Step 6: The coverage criterion is used to evaluate the effectiveness of the testing. Coverage of 100% would indicate that all paths through the aspect code were testable using the unit tests running against the mock object.

Step 7: The insertion criterion was used to evaluate the effectiveness of the advice testing. Coverage of 100% indicates that all advice are associated with their desired joinpoints. The decompile testAddAccount from the decompile testAccount class is shown below. The insertion of an joinpoint can be found by the search for the string "aspectOf().ajc$".

```
/*    */   @Test
/*    */   public void testAddAccountNoPerson()
/*    */   {
/*    */     try
```

```
/*    */     {
/*    */       Account localAccount1;
/* 42 */
                CreateID.aspectOf().ajc$afterReturning$NovaS
                E_CreateID$7$a53f022e (localAccount1);
                Account tmp = localAccount1 = new Account();
/* 43 */        Account localAccount2 = tmp;
                Accounts localAccounts = this.accounts;
/*    */        try {
                  CreateID.aspectOf().ajc$before$NovaSE_Crea
                  teID$8$2aefc8b6 (localAccounts,
                  localAccount2);
                  localAccounts.AddAccount(localAccount2); }
                catch (BankingException
                      localBankingException) {
                   Validate.aspectOf().ajc$afterThrowing$Nov
                   aSE_Validate$2$bd737665(localBankingExcep
                   tion);
                      throw localBankingException; }
/*    */     }
/*    */     catch (Exception ex) {
/* 46 */       Assert.assertEquals("Should have exception",
                      "Must Contain at least 1 account",
/* 47 */       ex.getMessage());
/*    */     }
/*    */   }
```

Figure 17: Unit test after aspect weaving

Step 8: Once all of the unit tests were run once, the other criterion was evaluated to determine the effectiveness of the class interfaces. The effectiveness of the class interface was based upon the comparison of time needed to code the mock objects using a class interface verses the time needed to code the mock objects not using a class interface. The effectiveness of the interface was determined by the percentage of joinpoints that can be instantiated within a class interface. The fewer the number of additional items needed the more effective the class interface is at minimizing the work needed to keep the mock objects and the application synchronized.

Resources Requirements

The code development and experiments were run on a personal computer using Eclipse IDE, Java, AspectJ, JUnit (for unit testing), EclEmma (for code coverage) and JD-GUI[3] (a Java decompiler).

[3] http://jd.benow.ca/jd-gui/downloads/jd-gui-0.3.5.windows.zip

Chapter 4

Results

Analysis

Four metrics were used in evaluate the effectiveness of using a class interface to unit test aspects. The metrics are:

- Code coverage, the percentage of code within a project covered by unit tests.
- Insertion coverage, the percentage of pointcuts that were correctly woven against the mock joinpoints. This value is only intended joinpoints and not unintended joinpoints.
- Development time, the time needed to develop the classes with the mock project being used to test the aspect code.
- Class objects declarable within a class interface that can be matched by joinpoints.

In the testing of the mock object usage the metric of insertion coverage was used. To determine the percentage of coverage the test project was compiled to create the bytecode. The bytecode should contain the woven joinpoints. A decompiler, jd-gui, was run against the bytecode to convert the bytecode back into readable Java code. To find the joinpoints a simple text search was conducted on each decompiled Java file looking

for the marker “aspectof().ajc$” which would identify where a weaving had taken place. As each marker was identified the joinpoint being woven was identified and recorded.

The joinpoints in the aspects that were tested all had 100% coverage. In all cases the joinpoints were woven into the pointcut as expected. In Table 2 below, the number of aspects tested per project, with the number of possible joinpoints per project. As such the mock objects within the test projects were capable of supplying context to the aspects and allowed for the testing of the aspect code.

Table 2 Insertion Coverage

Name	Aspects Tested	Joinpoint Available	Joinpoints Found
Introduction	3	1	1
Telecom	3	6	6
Spacewar	5	7	7
AJHotDraw	4	6	6
PetStoreAspectJ	4	17	17
Bean	1	1	1
NovaSE Bank	3	12	12

The second metric used for evaluating mock objects for the unit testing of aspects was code coverage; the percentage of aspect code that could be unit tested using a mock object in a test project. To determine the amount of aspect code that was covered by the unit tests the study used EclEmma, a Java code coverage tool for Eclipse. The code coverage is outlined in Table 3.

The tool EclEmma was designed to look at code coverage of Junit tests against Java code. When running the code coverage tool against AspectJ code the coverage percentage is artificially low. The coverage rates in most of the projects are below 50% because the EclEmma counts joinpoint declarations as lines of code. The coverage rate

in Spacewar was the lowest of all of the applications tested. The Spacewar application had a lower coverage rate due to the way three classes were developed. In each of the three class an aspect was created within a class. The unit tests tested only the aspect portion of the three classes and not the remaining class code. However, the EclEmma tool counted all of the lines of code in each of the three files resulting in a skewed coverage rate. The joinpoints are not testable code except when associated with an aspect. The code coverage results shows that it is possible to test the aspect code effectively using a mock object to supply the context for the aspects.

Table 3 Code Coverage

Name	**Aspects Tested**	**Code Coverage %**
Introduction	3	49.8
Telecom	3	43.3
Spacewar	5	33.3
AJHotDraw	4	46.2
PetStoreAspectJ	4	39.8
Bean	1	72.7
NovaSE Bank	3	54.5

To evaluate the effectiveness of using a class interface in the construction of the mock objects there were two metrics used. The first of the metrics was the time needed to build and maintain the mock objects. This metric involved keeping track of the approximate amount of time needed to create and maintain the mock objects.

In creating the mock objects to be used in testing the aspects, the mock objects were divided into two groups. One set of mock objects was created by inheriting a class interface that was also inherited by the application code class being mocked. The other

set of mock objects was created without using a class interface. The number of mock objects per test project and the time needed is shown in Table 4.

Table 4 Time Comparison

Name	Mock Objects w/ Interface	Time to create and maintain	Mock Objects w/o Interface	Time to create and maintain
Introduction	1	1 hr	0	N/A
Telecom	2	2 hrs	4	4 hrs
Spacewar	2	4 hrs	2	4 hrs
AJHotDraw	8	10 hrs	15	20 hrs
PetStoreAspectJ	6	8 hrs	5	7 hrs
Bean	1	1 hr	0	N/A
NovaSE Bank	6	8 hrs	3	5 hrs

When creating the mock objects only the necessary functionality needed was added. The mock objects that inherited from a class interface had all interface methods created. The methods within the class actually being used had code added else the method would throw a not implemented exception. The mock objects that did not inherit from a class interface only had the methods needed created. In both cases an empty class of the appropriate name was created in the test project. Then as functionality was needed it would be added to the mock object. An additional amount of time was needed in the maintenance of the class interface. As a new public method was needed in the base application to keep the class interface updated.

While the class interface shortened the time to a small degree there wasn't a great difference in the amount of time between the two methods. In the creation of the mock objects only functionality needed to successfully test an aspect was needed. Thus the mock functionality was copied into or reproduced from the application object piece by

piece. If new functionality was added to the base application object, the mock object only needed the functionality if needed by a unit tester aspect being tested.

The second metric to be evaluated was the number of declarable class objects, such as methods, constructors, exception handlers, and attributes, within the class interface that could be joinpoints. The aspect pointcuts would not actually be woven against the interface joinpoints, but the interface joinpoints would give definition to the Java objects and thus be available within any class inheriting from the interface.

Of the available Java object types such as attributes, methods, etc., the only object that could be used in the interface is the public method. The class interface in Java is not capable of containing any private attributes or method declarations. It is also not capable of having public attributes or constructors. From a percentage rate evaluation the usage of the class interface was not helpful in the testing of aspects using mock objects.

Summary of Results

The goal of this research was to determine the effectiveness of testing aspects using mock objects with class interfaces. The testing was done with the Java language using AspectJ and within the Eclipse IDE.

To determine whether mock objects and interfaces are beneficial three questions were asked:

- With advice not having their own context will mock objects supply that context?
- Does the class interface benefit the user for keeping the mock object and application class synchronized?

- How many class constructs that can have joinpoints can a class interface contain?

How many aspect joinpoint types are available in a class interface

In Java a class interface is able to contain public method declarations and public constants. Of these two types aspects are able to have joinpoints against only the public method declarations. In contrast AspectJ is able to create joinpoints against public and private methods, public and private attributes and constructors.

The inability of the interface to have more constructs hampers the usefulness of the class interface with mock objects testing aspects. With the method declarations one is able to use the interface declaration within the joinpoint and the advice. By using the interface declaration one is able to match against either the application class or the mock object, with resolution at runtime. The code below, taken from the Spacewar and Telecom applications, shows the usage of an interface within a joinpoint and advice.

```
pointcut helmCommandsCut(IShip ship):
    target(ship) && ( call(void rotate(int)) ||
           call(void thrust(boolean)) ||
           call(void fire()) );

    after(ITimer t): target(t) && call(* ITimer.start()){
      System.err.println("Timer started: " +
         t.getStartTime());
      start = true;
}
```

Figure 18: Interface usage in an Aspect

Is a class interface beneficial for synchronization

The class interface was able to supply just public methods of the application class for use as possible pointcuts. The usage of a class interface was not valuable at keeping the application class and the mock object synchronized. As a result when a change was made to the application class the developer had to remember to provide the new functionality to the mock object.

The usefulness of the class interface was determined by keeping track of the time needed to create and maintain a mock object inheriting from a class interface versus the time needed to create and maintain a mock object not inheriting from a class interface. As with the class objects there was not a distinct advantage in using the class interface. The time needed to create and maintain the mock code was roughly the same whether an interface was used or not used. The time needed for the mock objects in the Telecom example was on average one hour for the non-interface code and the interface code. The time needed for the NovaSE code the time needed was on average one hour and fifteen minutes for the interface code and one hour and forty-four minutes for the non-interface code.

Can mock objects supply aspect context

As was seen by Mortenson et al. (2006), the usage of mock objects is a viable method of testing aspect code. By creating a separate project for the unit tests a tester is able to create mock objects that are contained in the same package and have the same class name as those in the base application. The tester is then able to add only the code needed for

the testing. The research was able to test pointcuts that had joinpoints of method calls, object instantiations, exceptions and constructors.

Summary

The ability to use mock objects for the testing of Java aspect code was shown by the study to be a viable method for testing. To properly test, the mock objects must be in a mock project with the package structure the same as the application project. The aspect code is able to use the mock objects for pointcuts and object instantiations. The unit tests were able to use the mock objects to test that the pointcuts are found and that the advice code properly executes.

The addition of an interface to the creation of a mock object, while helpful, did not supply an overwhelming addition to the process. The class interface did help in keeping public methods synchronized between the classes in the two projects. With only public methods being available as possible pointcuts combined with the negligible difference in the time needed to create a mock object inheriting from a class interface versus a mock object that does not inherit from a class interface, the usage of a class interface when creating mock objects to unit test aspects is not warranted. A tester would spend as much time creating and maintaining the mock objects with a class interface as without.

Chapter 5

Conclusions

Conclusions

This dissertation found that using mock objects in a mock project it is possible to test aspect class code. The study was able to create mock objects that were able to give aspects context by supplying the joinpoints with pointcuts identical to those found in the application classes. The pointcuts were verified by using a decompiler and verifying that the advice code was woven into the pointcut.

By using the mock objects unit tests were able to be written that tested the functionality of the aspect advice. With the unit testing the study showed that a developer would be able to test their aspect code to validate that it was working as expected.

The usage of a class interface was also incorporated into the study to see if a class interface would ease the problem of keeping application class code and mock object code synchronized. When using a class interface it was found that the interface code was not a detriment, conversely it was also found to not be advantageous. The time needed to create and maintain the mock objects with a class interface verses the mock objects without a class interface was almost the same. In both cases a tester would have to

continually refer to the application code to manually copy or reproduce code in the mock object to duplicate the code found in the application class.

Implications

This dissertation investigated whether the usage of a class interface with the construction of mock objects was beneficial. The class interface did not cause any adverse effects, but in the ability to save time there was no worthwhile advantage. The mock object, in a Java environment, was shown to be able to provide context to aspect classes. With the mock objects a tester is able to validate that an aspect performs as expected.

Recommendations

The usage of AOP as a method of developing software not only requires the tools for effectively and efficiently creating aspects, but the ability to make sure the code works and is maintainable. With this work developers have a way of testing their code to make sure that it works and to test that it is woven where desired. While it is possible to test that the joinpoints are woven where desired it is still a problem of detecting that joinpoints are not incorrectly woven into the code.

This study was concerned with the ability to effectively test aspects by a developer to ensure that the aspect code written would work as expected. The ability for testers at the integration level to test that aspect code is working correctly and not adversely affecting other modules still needs to be studied.

Cross-cutting concerns are a problem and AOP is a possible solution. Singleton classes are an additional solution to cross-cutting concerns. Is one a better solution than the other?

Summary

AOP has its origins in 1996 in a proposal by Gregor Kickzale (Haque, 2011). AOP was proposed as an enhancement to OOP. The AOP methodology is a possible solution to the occurrence of cross-cutting concerns found within OOP applications (Lemos, et al., 2006).

In OOP classes are created to emulate functional objects that are specified within the application design requirements. In design requirements there exist requirements that are not specific to a single object. Instead these requirements are needed by multiple objects, but are not core functionality of any one object. These requirements are known as cross-cutting concerns (Monteiro & Aguiar, 2007). A common example of a cross-cutting concern is the logging of information by an application is it executes.

AOP is a software development paradigm that addresses the problem of cross-cutting concerns (Wedyan & Gosh, 2010). AOP solves the problem with the introduction of four new concepts: joinpoints, pointcuts, advice and aspects (Wloka, et al., 2008). An aspect is a class like structure that encapsulates the pointcuts and advice. Like a class an aspect is able to contain attributes and methods along with the pointcuts and advice. The aspect is implemented in the application by having the aspect code woven into the application code during the compilation process.

A joinpoint is an aspect construct that defines where within an application an advice is to be woven. The location within the application that the weaving will take place is known as the pointcut. An advice is similar to a class method. The advice contains code that will be executed at the corresponding pointcuts (Parizi & Ghani, 2007).

As with any software construct the code written must be tested for correctness. This testing takes place at the time of development to ensure it is functionally correct. It is tested again at integration time to ensure that the code interfaces properly with other parts of the application. A programmer cannot assume that because an applications code compiles, that it is correct. Unfortunately too many developers make this mistake (Olan, 2003).

This study researched the benefit of using class interfaces with mock object to do unit testing of aspects.

Unit testing was introduced by Kent Beck in the 1990's in association with Smalltalk (Osherove, 2009). A unit test is code that will test a specific piece of code in isolation from the overall application. The test is conducted by making an assumption of what the application code should do. The unit test will execute and if the assumption occurs the test passed (Olan, 2003)

Since unit tests are executing in isolation there is often additional information or objects needed for the application code under test to properly execute (Osherove, 2009). A mock object is code written that will provide the additional information. A mock object will replace an actual application object by immolating the application object without duplication all of the application objects functionality, only that functionality which is needed for the test (Mackinnon, et al., 2001)

The ability to use mock objects to test AOP applications was tested by Mortensen et al. (2006). In their study C++ OOP applications were refactored to use AOP. To test the aspects being created a mock system was created that allowed the researchers to test the validity of the aspect code.

The authors found that the time needed to compile the mock system was significantly less than that needed for the full application system. With the quicker compilation times, the time needed for running of the unit tests was shortened. The mock system also allowed for quicker experimentation with the joinpoints and advice, due to the shorter compilation time. The authors noted that there existed a problem with keeping the mock system and the actual application objects synchronized. There was also a limitation in the types of joinpoints created, mainly being limited to method calls

This research looked to determine the effectiveness of using a class interface inherited by mock objects for the testing of aspects. To determine the effectiveness three questions were asked:

- Can mock objects supply context for aspects?
- Is there a benefit to using a class interfaces inherited by the application classes and the corresponding mock objects?
- How many object constructs that can as as joinpoints are available in an interface?

The research to answer these questions consisted of two main steps. The first step involved writing a Java based application that simulated a banking application. The banking application project was created containing a package, NovaSE. In the package

was placed the java class code, class interfaces and the aspect code. The aspect joinpoints were written to find pointcuts within the application classes and perform actions needed by the application.

A test project was created that duplicated the structure of the base application. In the case of the banking application the project Test_NovaSE Bank with a package of NovaSE was created. The class interfaces files and the aspect class files were linked into the test NovaSE package allowing for the test project and the application project to share the same files.

To test the aspects, unit tests were created that would validate correctness. The tests were designed to validate that joinpoints were being correctly found and that there were no joinpoints missed. Tests were also designed to validate that the code within the advice executed as expected. The mock objects were created by either inheriting from the class interface or then creating the needed functionality or they were created by duplicating the application class structure and then completed only the needed functionality.

In addition to testing with a sample application the study used six open source projects; AspectJ Introduction, Bean, Telecom, Spacewar, Petstore AspectJ and AJHotDraw. With each of these applications the procedures outlined above were followed:

- A test project was created with the same package names
- The aspect and interface code files were imported into the test project package
- Unit tests were written to test the aspects
- Mock objects were created as needed to test the aspect code.

The research for the class interface provided very little benefit. The joinpoint objects available in a class interface were limited to just public methods. Excluded were constructors, public variables and any private construct. With only the public method available in the class interface a time savings verses creating the mock object by copying the needed functionality from the application class was not seen. The main reason for the lack of a time difference was that the mock object functionality was only added as it was needed to properly test the aspect code. Thus the code for the mock inheriting an interface had to have the majority of its code created by duplicating the code found in the application object, same as the mock without the interface.

The mock object was created as an empty class shell. Functionality was first added to satisfy the aspect code and then code was added to allow the unit tests to run. The functionality for the mock objects was either cut and pasted straight from the application object or a subset of the application code was taken as needed. The aspect code from the base application was linked into the test project, placing the aspect code in the same package as in the base application. Unit tests created to test the pointcuts and advice were now able to execute as expected.

This dissertation found that using mock objects in a test project is a viable method of providing aspects context. With context provided by the mock objects unit testing of aspect within the Java language is possible. The usage of a class interface to ease the ability to keep application code and mock project code synchronized was shown to be neither a help nor a deterrent. The time need to create and maintain the mock was negligibly different between using and not using a class interface.

References

Alexander, R. T., Bieman, J. M., & Andrews, A. A. (2004). *Towards the systematic testing of aspect-oriented programs, Technical Report CS-4-105.* Colorado State University.

Bender, J., & McWherter, J. (2011). *Test Driven Development with C#.* Indianapolis, Indiana: Wiley Publishing Co.

Freeman, S., Mackinnon, T., Pryce, N., & Walnes, J. (2004). Mock roles, Not Objects. *19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '04)* (pp. 236-246). New York, NY: ACM.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley.

Haque, M. A. (2011). Problems in Aspect Oriented Design: Facts and Thoughts. *International Journal of Computer Science Issues, 8*(2), 552-556.

Kim, T., Park, C., & Wu, C. (2006). Mock Object Models for Test Driven Development. *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*, (pp. 221-228).

Kollanus, S. (2010). Test-Driven Development - Still a Promising Approach?,. *2010 Seventh International Conference on the Quality of Information and Communications Technology* (pp. 403-408). IEEE.

Kumar, M., Sharma, A., & Garg, S. (2009, October). A study of aspect oriented testing techniques. *Industrial Electronics & Applications*, pp. 996-1001.

Lemos, O. A., & Masiero, P. C. (2008). Using Structural Testing to Identify Unintended Join Points Selected by Pointcuts in Aspect-Oriented Programs. *32nd Annual IEEE Software Engineering Workshop*, 84-93.

Lemos, O. A., Ferrari, F. C., Masiero, P. C., & Lopes, C. V. (2006). Testing aspect-oriented programming Pointcut Descriptors. *In Proceedings of the 2nd workshop on Testing aspect-oriented programs* (pp. 33-38). New York, NY: ACM.

Lopes, C., & Ngo, T. (2005). Unit-Testing Aspectual Behavio. *4th International Conference on Aspect-Oriented Software Development.*

Mackinnon, T., Freeman, S., & Craig, P. (2001). Endo-testing: unit testing with mock objects. In *In Extreme Programming Examined* (pp. 287-301.). Boston, MA.: Addison-Wesle.

Monteiro, M. P., & Aguiar, A. (2007). Patterns for refactoring to aspects: an incipient pattern language. *In Proceedings of the 14th Conference on Pattern Languages of Programs (PLOP '07).* New York, NY: ACM.

Mortensen, M., & Alexander, R. T. (2005). An Approach for Adequate Testing of AspectJ Programs. *2005 Workshop on Testing Aspect-Oriented Programs in conjunction with AOSD 2005.*

Mortensen, M., Ghosh, S., & Bieman, J. (2012). Aspect-Oriented Refactoring of Legacy Applications: An Evaluation. *IEEE Transactions on Software Engineering, PP*(99).

Mortensen, M., Ghosh, S., & Bieman, J. M. (2006). Testing During Refactoring: Adding Aspects to Legacy Systems. *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, (pp. 221-230).

Mortensen, M., Ghosh, S., & Bieman, J. M. (2008). A Test Driven Approach for Aspectualizing Legacy Software Using Mock Systems. *Information and Software Technology, 50*, 621-640.

Nandigam, J., Gudivada, V. N., Hamou-Lhadj, A., & Tao, Y. (2009). Interface-Based Object-Oriented Design with Mock Objects. *2009 Sixth International Conference on Information Technology*, (pp. 713-718).

Olan, M. (2003). Unit testing: test early, test often. *J. Comput. Small Coll, 19*(2), 319-328.

Osherove, R. (2009). *The Art of Unit Testing with Examples in.Net.* Greenwich, CT: Manning Publications Co.

Parizi, R. M., & Ghani, A. A. (2007). A Survey on Aspect-Oriented Testing Approaches. *The 2007 International Conference Computational Science and its Applications*, 78-85.

Sobering, G., Cook, L., & Anderson, S. (2004). Pseudo-classes: very simple and lightweight mockObject-like classes for unit-testing. *In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '04)* (pp. 162-163). New York, NY, USA: ACM.

Thomas, D., & Hunte, A. (May/June 2002). Mock Objects. *IEEE Software, vol. 19, no. 3*, 22-24.

Wedyan, F., & Ghosh, S. (2008). A Joinpoint Coverage Measurement Tool for Evaluating the Effectiveness of Test Inputs for AspectJ Programs. *2008 19th International Symposium on Software Reliability Engineering*, 207-212.

Wedyan, F., & Ghosh, S. (2010). A Dataflow Testing Approach for Aspect-Oriented Programs. *2010 IEEE 12th International Symposium on High-Assurance Systems Engineering*, 64-73.

Wloka, J., Hirschfeld, R., & Hansel, J. (2008). Tool-supported Refactoring of Aspect-oriented Programming. *7th international conference on Aspect-oriented software development*, (pp. 132-143). Brussels, Belgium.

Xie, T., Zhao, J., Marinov, D., & Notkin, D. (2006). Detecting Redundant Unit Tests for AspectJ Programs. *17th International Symposium on Software Reliability Engineering*, 179-190.

Xu, D., & Xu, W. (2006). State-based incremental testing of aspect-oriented programs. *In Proceedings of the 5th international conference on Aspect-oriented software development* (pp. 180-189). New York, NY: ACM.

Zhao, J. (2003). Data-Flow-Based Unit Testing of Aspect-Oriented Programs. *27th Annual International Computer Software and Applications Conference*, 188-197.

Zhu, H., Hall, P. A., & May, J. H. (December 1997). Software unit test coverage and adequacy. *ACM Comput. Surv. 29, 4* , 366-427.