Test Driven Development using Aspect Oriented Programming Techniques

Einar Boson
Johan Lundquist
D00, Lund Institute of Technology, Sweden.
einar.boson@dsek.lth.se
d00jl@efd.lth.se

May 2003

Abstract We explore the possibilities of using AspectJ for easing unit testing. Doing this we write aspects that facilitate the test patterns Mock Objects, Log String, Crash Test Dummy and Self Shunt.

1 Introduction

One problem with conventional object oriented programming is that functionality which spans a whole system, and therefore affects several parts of the code, is hard or even impossible to modularise. This prevents code from being reused, and it complicates development and refactorisation of the common code. Lately there have been many different techniques introduced to solve this problem. One of them is Aspect Oriented Programming (AOP).

AOP lets the developer develop and change the functionality without changing the present code in the system. Functionality and changes are instead added as free-standing modules, called aspects, which can span multiple classes. The most mature aspect oriented programming language to this date is AspectJ [AJE].

Our intention with this study was to find out if AOP, and especially AspectJ, could make unit testing easier and to see if it could help separate test code from the production code. We found that it indeed can make unit testing easier and that it in some cases can hinder test code from affecting the production code.

1.1 Structure

We begin this paper with an introduction to AspectJ followed by a description of our methodology. After that we walk you through the development of aspect based solutions to common testing problems. Then we state our results and thoughts about AspectJ after which we finish off with a brief description of our references and of relevant reading.

In Appendix A we list the code of the aspect SelfShuntProxy which we develop in Section 2.4.
1.2 Introduction to AspectJ

AspectJ is an extension of the Java language and is compiled with a separate compiler written in Java. It produces normal .class files that can be run in the Java Virtual Machine. The compiler can also output Java source code (.java files), so that one can see how things are implemented. The new unit of modularisation introduced by AspectJ is the aspect.

One big advantage of aspects is that they can change the behaviour of the existing code without it being aware of the changes. This makes it possible to compile a system with, or without, different aspects included. This gives us the possibility to produce systems which behave differently based on which aspects were included at compile time.

An aspect consists, apart from normal methods, of point cuts, advice and introduction. (We will not use introduction at all in this paper, and will therefore give no explanation of it. The interested reader is directed to [AJE].) A point cut is a set of join points where each join point is a well-defined point in a program’s execution. It is possible to specify regular java code to be run before, after or while a point cut is reached. This is called advice.

Let us show an example. If we would like a message written to the standard output every time MyClass.myMethod() is called we write:

```java
aspect MyAspect {
    pointcut myPointcut() : call(public MyClass.myMethod(..));
    before() : myPointcut() {
        System.out.println("now calling MyClass.myMethod()");
    }
}
```

The point cut is read like this: “a public method, in MyClass, named myMethod is called, with any number of arguments of any type” (the two dots indicate any arguments). The before advice is then read like this: “before the point cuts defined by myPointcut, execute the following code”. The whole aspect is thus interpreted like this: “every time the program reaches a method call to myMethod in the class MyClass, print out “now calling MyClass.myMethod()” before the program continues the “normal” execution”.

If we instead would like to trace when an attribute is changed, the set() point cut can be used. Here is how myPointcut would look like then:

```java
Pointcut myPointcut() : set(int MyClass.myAttribute);
```

It is also possible to access and change the attributes and objects referenced in the point cuts as one sees fit. It is even possible to throw and/or catch exceptions. This can be used, for instance, to make sure arguments supplied to methods are legal.
aspect MyAspect {
    pointcut myPointcut(int val) : call(myClass.myMethod(int))
        && args(val);
    around(int val) : myPointcut(val) {
        if (val > 0) {
            proceed(val);
        } else {
            // do something
        }
    }
}

The aspect above checks whether the argument to myMethod() is greater than zero before myMethod() is called. If so, proceed is called. The word proceed is a reserved word in AspectJ; it is used to proceed with the method that was intercepted by the around advice. The arguments passed to proceed must match the arguments to the enclosing around advice in number and types.

Another special construct in AspectJ is the method signature MyClass.new(). This join point is reached when the constructor of MyClass is called, i.e. when the object is created.

For a more extensive introduction to AspectJ we recommend the official AspectJ site [AJE].

1.3 Methodology

To illustrate how to use Aspect Oriented Programming (AOP) to ease the unit testing process, we use the classic scenario of bank transactions. Our examples will consist of a CashMachine communicating with a Bank via a BankConnection. We pretend that BankConnection and Bank are given and that our job is to implement CashMachine.

First of all we would like to clarify that we henceforth will not include irrelevant details about our test first strategy. Neither will we show all the classes’ code inline in the document, since much of it is irrelevant to the current scope - it is the possibilities of AspectJ we want to illustrate, not create an operational bank system.

2 The Creation of the Bank System

We begin by creating the classes CashMachine and CashMachineTest. The first task is to make sure we can log in to the bank. This forces us to try to communicate with the bank. Since we do not have a live connection to a bank we have to somehow fake it. The standard way of faking functionality is to use Mock Objects1 [KB02], so that is what we will do. A Mock Object is an object you use to simulate a “real” object during tests. The reason for this varies, but a common one is that the real object is to complex to use during test, e.g. databases.

We would rather have used Self Shunt since that would have allowed us to gather all code in one place, which is easier to both write and understand [KB02]. Self Shunt is very similar to

---
1. For example, see www.mockobjects.com.
Mock Object but instead of creating a new mock class, the test class itself is used as the mock object.

Java does, however, not let us use Self Shunt. The reason is that `CashMachineTest` already extends `TestCase`, and therefore is unable to extend `BankConnection`, making it seemingly impossible to use the Self Shunt technique. In Java, it is only possible to use Self Shunt when the type to be replaced is defined by an interface since Java does not allow multiple inheritance.

A common disadvantage of both Mock Objects and Self Shunt is that one has to be able to somehow specify the `BankConnection` that the `CashMachine` is to use. Thus, the design is affected by the tests, and that is generally frowned upon. This aside, we implement a mock `BankConnection` named `MockBankConnection`.

### 2.1 The Introduction of Aspect Orientation

The `MockBankConnection` now acts as a `BankConnection` in our system, but the design is slightly flawed as stated above. This is where AspectJ can simplify and improve our code.

We now introduce the first aspect to our system. Since aspects are a very good way of stipulating how the code should behave, without actually changing it, we can now revert to our original design and remove the need to be able to externally specify the `BankConnection` that the `CashMachine` should use.

Using AspectJ we change the flawed design by adding an *around advice* to intercept when a `BankConnection` is created and to intervene by substituting the `BankConnection` with a `MockBankConnection` [NL02]. This allows us to simulate a functioning connection to the bank while retaining our original design. The advice looks like this:

```java
BankConnection around() : call(BankConnection.new()) {
    return new MockBankConnection();
}
```

This has the disadvantage that we have to make an ad hoc solution to get the instance of the `MockBankConnection` used. We need this to check that the correct methods were called from the `CashMachine`. However, the advantages outweigh the disadvantages: the design is unaffected by the tests.

### 2.2 Tracing the Program Flow

It seems reasonable to be able to log the bank system. If anything goes wrong it is imperative that a technician is able to trace the program flow. As it happens, this is also rather similar to the test pattern *Log String*. Log String is used to check that methods are called in the correct order on a specific object by appending information to a string every time a method is called. By modifying the following aspect a bit we could easily create a Log String aspect.
We could solve Log String with the pattern Decorator. Instead of changing the original code, a decorator acts as a wrapper for the code. To create a logger using a decorator, we could make a new decorator class for every class we would like to log. These would extend the class they log and override all methods that should be logged. The decorator class would first do the logging procedures, and then call the corresponding method in the original code. This would of course take quite some time to implement (in a real system), and every time a change is made to the system, the decorator class would have to be changed as well.

Since we are now thinking aspect oriented, we do not start to change the original code, and we do not create decorator classes. Instead we create an aspect that invokes logging methods before and after every call is made. Thus, we effectively create a logger. The aspect based logger will work even if the design and/or code were to be changed, and we do not have to use the decorator classes, saving us a lot of time and effort.

The important parts of the aspect Logger looks like this:

```java
public aspect Logger {

    pointcut allMethods() : execution(* *.*(..));
    pointcut testMethods() : within(junit..*+) || within(Logger) || within(TestAll);

    before() : allMethods() && !testMethods() {
        // call log()
    }

    after() : allMethods() && !testMethods() {
        // call log()
    }

    after() throwing (Throwable t) : allMethods() {
        // call log()
    }

    private void log(String msg) {
        // do logging
    }
}
```

We now have an efficient way of logging the system. If we do not want to log the system anymore, we simply exclude the aspect from the compilation.

### 2.3 Provoking Errors

What if the disc on the computer at which the system is running were to become full? What would happen to the logging functionality? This is something which is quite hard to test in reality, since it would require the developer to fill the disc before running the test. Instead we want to make the system believe that the disc is full, even though it is not.

The common way of achieving this is to use a test pattern called Crash Test Dummy. A Crash Test Dummy is a special object that throws an exception instead of doing any real work. This
involves setting up a mock object that throws an exception on disc write access. The crucial parts of the aspect are as follows:

```java
public aspect EmulateDiscIsFull {
    pointcut callOfFileWriterWrite() : call(* FileWriter.write(..));
    pointcut callOfFileWriterFlush() : call(* FileWriter.flush(..));
    pointcut callOfFileWriterCreate() : call(FileWriter.new());
    void around() throws IOException : callOfFileWriterWrite()  {
        // throw an exception if writeException is enabled
    }
    void around() throws IOException : callOfFileWriterFlush()  {
        // throw an exception if flushException is enabled
    }
    FileWriter around() throws IOException : callOfFileWriterCreate() {
        // throw an exception if createException is enabled
    }
    private void enableWriteException(boolean enable);
    .
    .
}
```

2.4 Self Shunt After All

After gaining insights into the AspectJ language, we realise that by combining this new tool with the Java reflection package, we can now implement the Self Shunt we yearned for earlier. We do this by implementing an aspect called SelfShuntProxy.

```java
public aspect SelfShuntProxy {
    pointcut callAnyMethod(Object target) :
        call(* *.*(..)) && target(target);
    pointcut withinThis() :
        within(SelfShuntProxy) || target(SelfShuntProxy);
    public static void initialize(Object newTarget,
                   Class classToOverride) {
        // populate a HashMap with redirection information
    }
    Object around(Object target) :
        callAnyMethod(target) && !withinThis() {
            // checks if target is of the type specified in initialize()
            // if so, call the corresponding method in our SelfShunt
        }
    }
```

This aspect acts as a proxy (hence the name) and redirects all calls to methods of objects of the class `realTarget` to a method of the object `newTarget`. The scheme to choose method name in `newTarget` is the following:
A method `myMethod(argType1, argType2, ...)` is called on an object of class `classToOverride`. The method really invoked is instead `_myMethod(argType1, argType2, ...)` on the object `newTarget`. Please note the underscore preceding the method name - it is there to indicate that this is the fake method and also to enable interception of methods already present in the new target. If no such method is found then `_default()` is called with the arguments `methodName`, `args[]` and `callerObject`.

Our bank system might not look much to the world but the `SelfShuntProxy` is a reusable aspect that can be used to facilitate the test pattern Self Shunt in Java. It circumvents the limits of the Java language and allows us to dynamically substitute one object for another of a different class.

### 3 Results

We have shown that the test patterns Mock Object, Log String, Crash Test Dummy and Self Shunt can be implemented using AspectJ.

#### 3.1 Mock Object

We found several advantages of AspectJ vis-à-vis Java in implementing Mock Object and Self Shunt. The great advantage of our implementation of Mock Object, using AspectJ, was that we did not have to change the API or the method calls in the original code. Instead we added an aspect that substituted all `BankConnection` with `MockBankConnection` at creation time.

Mock Object using AspectJ had, however, a couple of disadvantages as well. One drawback, for example, is that there is no obvious way of gaining access to the mock object to manipulate it. A solution to this problem is shown in [MH02].

#### 3.2 Log String

We have not actually implemented a true Log String. We have, however, solved a very similar problem by logging all activity in our system. Our `Logger` logs every method call and prints the program flow to a file. It also logs all `Exceptions` and `Errors` thrown. This is very helpful when it comes to debugging and optimising the code.

#### 3.3 Crash Test Dummy

Another helpful testing technique we have discussed is the Crash Test Dummy. The technique is used for provoking errors that normally are difficult to generate, and testing that the program responds in the appropriate way. This is usually done with a Mock Object but can be done with an around advice instead. This has the same advantages as when we do our Mock Object with an around advice - we do not have to change the API.
3.4 Self Shunt

The obvious advantage of our Self Shunt implementation is that it is implementable at all. Using our SelfShuntProxy aspect one can now easily test classes using Self Shunt. Since one can use the SelfShuntProxy to redirect method calls from arbitrarily chosen classes to arbitrarily chosen objects one can set it up to emulate Mock Objects.

The disadvantage of this is of course the performance overhead. SelfShuntProxy screens all method calls whereas our other solution for Mock Object does not generate unnecessary method calls. This is not a big problem though. It can in most cases be disregarded, since performance is not a key issue in tests.
4 References and Related Work

4.1 References

The official web site of AspectJ. Contains extensive information about the AspectJ language including a tutorial and a semantics index. This is the place to start if you want to learn more about AspectJ. This is also where the AspectJ compiler is available.

[MH02] Simon Monk, Stephen Hall: Virtual Mock Objects using AspectJ with JUNIT.
A truly elegant solution for the mock object test pattern. A new subclass of JUnit’s TestCase is created where you easily can specify new return values for different methods of different classes.

An introduction, with good examples, to all the basic test patterns. Here you can read more about Mock Objects, Self Shunt, Log String and Crash Test Dummy.

[NL02] Nicholas Lesiecki: Test Flexibly with AspectJ and Mock Objects
An article about how to implement mock objects using AspectJ without the test code affecting the design.

4.2 Related Work

If you are interested in reading more about aspect oriented programming with AspectJ, we recommend the following articles:

Michael Kircher: XP + AOP = Better Software.
A short article which discusses the use of aspect oriented programming in an XP environment on a general basis.

Nicholas Lesiecki: Improve Modularity with Aspect-Oriented Programming.
An introduction to AspectJ and how it improves modularity in Java. It includes many good code examples.
Appendix A

The source code for our SelfShuntProxy:

```java
import org.aspectj.lang.*;
import java.lang.reflect.*;
import java.util.*;
import junit.framework.*;

public aspect SelfShuntProxy {

    pointcut callAnyMethod(Object target) : call(public * *.*(..))
        && target(target);

    pointcut withinThis() : within(SelfShuntProxy)
        || target(SelfShuntProxy);

    private static Object shuntObject;
    private static Class captureClass;
    private static HashMap methodsToRelay;

    public static void initialize(Object newTarget, Class classToOverride) {
        shuntObject = newTarget;
        captureClass = classToOverride;
        methodsToRelay = new HashMap();
        Method[] realMethods = classToOverride.getMethods();
        Method[] fakeMethods = newTarget.getClass().getMethods();
        for (int i = 0; i < realMethods.length; i++) {
            for (int j = 0; j < fakeMethods.length; j++) {
                if (methodsMatch(fakeMethods[j], realMethods[i])) {
                    String methodDesc =
                        buildMethodDescription(realMethods[i].getName(),
                                               realMethods[i].getParameterTypes());
                    methodsToRelay.put(methodDesc, fakeMethods[j]);
                }
                if (fakeMethods[j].getName().equals("_default")
                    && methodsToRelay.get("default") == null) {
                    methodsToRelay.put("default", fakeMethods[j]);
                }
            }
        }
    }
}
```
Object around(Object target) : callAnyMethod(target) &amp;&amp; !withinThis() {
  if (target != null &amp;&amp; shuntObject != null &amp;&amp; target.getClass().equals(captureClass)) {
    Object[] targetArgs = thisJoinPoint.getArgs();
    String targetMethodName = thisJoinPoint.getSignature().getName();
    StringBuffer strbuf = new StringBuffer();
    strbuf.append(targetMethodName + "(");
    int i = 0;
    for (; i < targetArgs.length - 1; i++) {
      strbuf.append(targetArgs[i].getClass().getName() + ", ");
    }
    if (i < targetArgs.length) {
      strbuf.append(targetArgs[i].getClass().getName());
    }
    strbuf.append(")");
    Method redirectMethod = (Method) methodsToRelay.get(strbuf.toString());
    if (redirectMethod != null &amp;&amp; targetArgs != null) {
      try {
        return redirectMethod.invoke(shuntObject, targetArgs);
      } catch (Exception e) {
        e.printStackTrace();
        throw new Error("Could not invoke " + targetMethodName);
      }
    } else if (methodsToRelay.get("default") != null) {
      try {
        Method relayMethod = (Method) methodsToRelay.get("default");
        Object[] relayArgs = new Object[] {targetMethodName, targetArgs, thisJoinPoint.getThis()};
        return (relayMethod.invoke(shuntObject, relayArgs));
      } catch (Exception e) {
        e.printStackTrace();
        throw new Error("Could not invoke " + targetMethodName);
      }
    } else {
      throw new Error("Could not invoke " + targetMethodName);
    }
  } else {
    return proceed(target);
  }
}

private static String buildMethodDescription(String name, Class[] arguments) {
  StringBuffer strb = new StringBuffer();
  strb.append(name + "(");
  int i = 0;
  for (; i < arguments.length - 1; i++) {
    strb.append(getArgType(arguments[i]) + ", ");
  }
  if (i < arguments.length) {
    strb.append(getArgType(arguments[i]));
  }
  strb.append(")");
  return strb.toString();
}
// for compatibility with AspectJ's autoboxing
private static String getArgType(Class arg) {
    if (arg.isPrimitive()) {
        if (arg.getName().equals("byte")) {
            return "java.lang.Byte";
        } else if (arg.getName().equals("double")) {
            return "java.lang.Double";
        } else if (arg.getName().equals("float")) {
            return "java.lang.Float";
        } else if (arg.getName().equals("int")) {
            return "java.lang.Integer";
        } else if (arg.getName().equals("long")) {
            return "java.lang.Long";
        } else if (arg.getName().equals("short")) {
            return "java.lang.Short";
        } else if (arg.getName().equals("char")) {
            return "java.lang.Character";
        } else if (arg.getName().equals("boolean")) {
            return "java.lang.Boolean";
        } else {
            return arg.getName();
        }
    } else {
        return arg.getName();
    }
}

private static boolean methodsMatch(Method fake, Method real) {
    return fake.getName().equals("_" + real.getName())
        && Arrays.equals(fake.getParameterTypes(),
                        real.getParameterTypes());
}
}