The value of Aspect Oriented Programming

Gordon Kwong, Eric Kingori, Szymon Czarnota
Department of Computer Science
University of Calgary
Alberta, Canada

Abstract—The Object-Oriented Programming paradigm has effectively revolutionized how software engineers approach the issue of modularity. Object-Oriented Programming does an effective attempt at organizing the majority of software into objects. In regards to the remaining portions of software that are not easily separated into their own objects, there is Aspect Oriented Programming. This programming paradigm is an extension of Object-Oriented Programming and assists in modularizing secondary and supporting functions from the main functions. In this paper, we will outline the value that Aspect Oriented Programming provides and demonstrate how to use Aspect Oriented Programming to increase Modularization using AspectJ.

Keywords—Aspect Oriented Programming (AOP); Aspect; Advice; Join Point; Pointcuts; Cross-Cutting Concerns

I. INTRODUCTION

A. Overview of Object-Orientation

The development of the object-oriented (OO) paradigm provided software engineers and developers with the tools required to create more complex systems and solve more complex problems by modeling real world objects. As a result, it has effectively revolutionized how software engineers approach the issue of modularity by allowing them to divide the activities into independent groups of common or related functionalities and allowing for and encouraging code reuse. The adaptation and continued use of objects gave way to the creation of design patterns, solutions to common, recurring problems in software design. Objects also introduce a layer of security, by having methods solely for the purpose of accessing and modifying its internal data, the chance of unpredictable corruption by the user is decreased significantly [5]. Despite the benefits of using OO and its advantages in software design and development, it still has its shortcomings. One of the main goals of object-orientation is to increase maintainability of the end product by making redesign affect and modify as little classes as possible. However as systems get larger and more complex it is not that easy to clearly separate objects and concerns into classes and modules.

B. Introduction into Aspect-oriented programming

Aspect-oriented programming (AOP) was developed with the intention to address these and other issues with OO by removing cross-cutting concerns from the individual classes and putting them into a single place; keeping track and dealing with them at the same time. Cross-cutting concerns are elements of a software program that are dependent on and influence other parts of the system, effectively cross-cutting across the architectural layers and being responsible, amongst many other things, for tangled or inter-dependent code. Aspect-oriented programming is aimed at improving on the areas that object-orientation lacks. It reduces coupling between classes and allows for more modular code. To show the advantages of AOP this paper will explain the idea of what AOP is and the major concepts behind it. It will also talk about the differences between OO and AOP then finally give a brief introductory tutorial on how to use AspectJ to write aspect-oriented code.

II. WHAT IS AOP

A. Aspect-oriented programming

Similar to object-oriented programming, aspect-oriented programming is a technique that aids in separating the functionality of a program into distinct modules. Unlike object-oriented programming that separates a program using the concept of an object which has a distinct state and communicates through creating instances and passing messages, AOP monitors the execution of the program and when a specified object is instantiated or a specific point in the code is reached, the right action takes place, similarly to the injection of code during runtime. Although this insinuates that aspect-oriented programming is competing with the much more popular OO, they are actually quite complementary and produce great results when used together.

B. Aspects

Aspect-oriented programming works on the basis of aspects. Aspects are analogous to classes, but are more abstract and less concrete [7]. They represent the crosscutting functionality of the system but are separated from the main implementation making it easier to deal with.

III. HOW AOP WORKS

A. Understanding by analogy

To understand how AOP programming works, consider the following analogy. There is a neighborhood that represents a program written using the object-oriented paradigm. This neighborhood has a purpose and within it there are houses which represent different classes within the program. Within the houses there are families and their possessions, similarly to the methods and variables within a class. Each member of the house has a responsibility to do within the neighborhood. In this neighborhood the houses must be maintained by construction workers. Since the construction workers are not contributing to the purpose of the neighborhood, they do not belong in houses within the neighborhood. Therefore similar to an aspect, the construction workers monitor the neighborhood without affecting the neighborhood’s purpose. When they see a
house that must be repainted a construction worker comes into the neighborhood and paints the house without disrupting the family members. Sometimes the house must be painted before a family member does a particular task, after a family member does a particular task, or both.

B. Definitions

To better understand aspect-oriented programming and make the tutorial easier to dissect, there are a few definitions within the AOP model which are beneficial to define [7]:

- Advices: advice are used within aspects to provide additional behavior upon meeting certain conditions. Pointcuts are used in the definition of an advice. In AspectJ there are three kinds of advices: before advice, after advice and around advice.

- Aspects: as previously mentioned these are class-like entities which represent the cross-cutting concern and separate them from the main program. In AspectJ they are defined in terms of joinpoints, pointcuts and advices.

- Joinpoints: these are clearly stated points within the execution of the program, for example method calls. They specify when to apply an advice. In AspectJ, when writing an aspect, you are required to specify which joinpoint to run when defining pointcuts.

- Pointcuts: program constructs to designate joinpoints.

- Tangled code: functions are tangled when they deal with concerns that do not relate to the main purpose of the function such as logging, error handling, printing, security, etc. These are referred to as Cross-Cutting Concerns.

IV. WHY USE AOP

Object-oriented programming provides software engineers with the capability to design and create complex programs which positively impact the development of software. It does so by allowing programs to become more capable, reusable and more maintainable [5]. In an object-oriented program, each class can be viewed as the skeleton of an object [5]. It contains each object’s attributes and the methods to manipulate data in one form or another to achieve some goal. One of the significant benefits is modularity of different objects. This allows objects to grow and change independently of the rest of the program. Each object can focus on a specific task making it easier to debug, test and improve a program. OO also introduces important and extremely useful programming practices such as information hiding and inheritance as well as design patterns. Information hiding leads to more durable systems since data manipulation is controlled in a predictable way. Inheritance allows the code to be recycled in an efficient manner which gave developers the ability to create software without re-implementing existing code. For example the class java.util.ArrayList inherits code from java.util.AbstractList which inherits from java.util.AbstractCollection. This allows for arraylist to have a wide variety of methods without reimplementing them. Despite these and other great advantages of object oriented programming, the paradigm is plagued with the curse of growing more complex as programs become larger, and as a result for the code to become more tangled and interdependent [6]. This can and does introduce many cross-cutting concerns and often makes the product unmaintainable and very resistant to change and evolution. Aspect-oriented programming helps minimize the presence of cross-cutting concerns, and their impact on the modularity and maintainability of the system making it easier for other engineers to understand, improve and build upon later on.

V. DIFFERENCES AND SIMILARITIES BETWEEN AOP AND OOP

A class in OOP is similar to an aspect in the sense that both act as templates for objects. In OOP classes communicate by calling mutually public methods or variables and exchanging variables through parameters and return statements [5]. In AOP the two objects do not communicate to each other directly, instead there is an aspect acting as a third party third party that communicates on behalf of the objects.

The following image compares the execution of a method in AOP to the execution in OOP[5].

To better understand this image let ObjectA represent an object named ‘oldSecurity’ and let ObjectB represent an object named ‘login’. The arrow going from the oldSecurity object to the login object means that within the login object there exist the instantiation of oldSecurity into a variable then the use of the variable within the class, similar to the following code.

```
oldSecurity foo= new oldSecurity();
...
```

...20 lines later
foo.doSomething();
...10 lines later
foo.doSomething()” [6]

This means in the login object there is code that is invokes the oldSecurity object scattered along the login object. Depending on how many calls are made to the oldSecurity object and other objects within the system, the login code will contain lines of code within it that are not related to login which reduces modularity of the login class. This also leads to highly coupled code because the oldSecurity object is not free to be updated to betterSecurity object without tracing through the login class and editing all the code related to security. This means that in OOP there is no way to make perfectly modular code with no coupling, since without the instantiation of other objects the objects will not be able to communicate.

In the case of the AOP within the aspect there is no initialization of security within login. Instead there exist the lines for the joinpoint that tell the aspect when to interject the advice which in this case would be the doSomething() method. The pointcut would be lines telling the aspect where the advice is to be interjected similar to the point where the method was called with the line “foo.doSomething().” The code within the login object will have no lines related to the oldSecurity object which allows for highly modular code. In the case where the oldSecurity object is replaced with betterSecurity object, only the lines relating to the join points and point cuts within the aspect class need to updated. This leads to less coupled code since now objects have the freedom to evolve without affecting the system in a significant way. In conclusion despite the diagram making AOP look far more complicated than OOP the difference between AOP and OOP is AOP far more elegant because it allows for less coupled and highly modular code.

VI. Step-by-Step Tutorial on Writing AOP Code in AspectJ

AspectJ is an extension to the Java Programming language. It can be downloaded both as a standalone or as a plugin for well established IDEs such as Eclipse, NetBeans or Emacs.

Aspect Oriented Programming is not limited to AspectJ or the Java programming language, similar to how Object Oriented Programming is not limited to a language such as C++ or Java. However, AspectJ has been widely used as the standard for an Aspect Oriented Programming tool, thus this tutorial will make use of AspectJ to demonstrate Aspect Oriented Programming.

A. Prerequisites

Since this tutorial will use the Eclipse IDE, the following three requirements are necessary.

1. JDK and JRE: A Java Development Kit(JDK) and a Java Runtime Environment(JRE) installation is required to run AspectJ. Instructions on how to setup and install these can be found at Oracle’s official site.

2. Eclipse IDE: An Eclipse Development Environment. Instructions on how to setup and install Eclipse can be found at the Eclipse’s official site.

3. AspectJ plugin: The AspectJ plugin can be downloaded from the official AspectJ website at eclipse.org/aspectj/.

B. AspectJ Tutorial in Eclipse

Before we are able to write any code in AspectJ, we have to setup an AspectJ project in Eclipse. This can be accomplished within the Eclipse Development Environment by selecting File->New->Other->AspectJ Project.

In order to demonstrate the purpose of Aspect Oriented Programming, let us create a class with cross-cutting concerns. Adding the following class to the current package will introduce cross-cutting concerns. The main method below is contained within the DrawObject class so that the tutorial is short and easy, you may separate it out to increase modularity. Also, the system print lines represent actual functionality or calling a method. They are represented by System.out methods for simplicity and quick feedback when testing.

```java
public class DrawObject {
    public static void main(String[] args) {
        DrawObject object = new DrawObject();
        object.updateTexture();
        object.updateColor();
    }
    public void updateTexture() {
        System.out.println("do some logging");
        System.out.println("updating Texture");
        System.out.println("update Screen");
    }
    public void updateColor() {
        System.out.println("do some logging");
        System.out.println("updating Color");
        System.out.println("update Screen");
    }
}
```

Now from the DrawObject class we can identify the cross cutting concerns. This class handles how an object will appear. It deals with the objects color and texture. Within this class the object also handles logging and when the screen is updated. These two activities do not belong in a single object but should be called when any object require these functionalities. These are the cross cutting concerns in the example.

To address the cross cutting concerns, let us introduce an aspect. An aspect can be created in the Eclipse environment by selecting File->New->Other->Aspect. Let us name the aspect LoggingAspect.aj to follow good naming conventions.

Now within the Aspect we can define a pointcut. Below is the sample aspect with a “before pointcut” since the logging
occurs before the methods are called. The Pkg in the calls is the name of the package where the class DrawObject exists.

```java
public aspect LoggingAspect {
    pointcut loggingPointCut() :
        call(void Pkg.DrawObject.updateTexture()) ||
        call(void Pkg.DrawObject.updateColor());
    before():loggingPointCut {
        System.out.println("do some logging");
    }
}
```

In the aspect above we have declared a pointcut and an advice. The pointcut is highlighted in blue and indicates the points in the code where the advice should be run. The advice is highlighted in green and indicates what code should be run when a pointcut is made. This aspect increases the modularity in the DrawObject class because the DrawObject class is no longer concerned about logging. Let us update the DrawObject class to reflect the addition of this aspect.

```java
public class DrawObject {
    public static void main(String[] args) {
        DrawObject object = new DrawObject();
        object.updateTexture();
        object.updateColor();
    }
    public void updateTexture() {
        System.out.println("updating Texture");
    }
    public void updateColor() {
        System.out.println("updating Color");
    }
}
```

Running the code now should return the same results as the original DrawObject class, so the same functionality is maintained. The difference is that the DrawObject class is no longer concerned about keeping logs. Let us now create an additional aspect for updating the screen. The following aspect demonstrates an “after pointcut”.

```java
public aspect ScreenUpdateAspect {
    pointcut screenUpdatePointCut() :
        call(void Pkg.DrawObject.updateTexture()) ||
        call(void Pkg.DrawObject.updateColor());
    after():screenUpdatePointCut {
        System.out.println("update Screen");
    }
}
```

The two aspects that we created addresses the two cross cutting concerns in the DrawObject class. To maintain modularity, an aspect should be created to address each cross cutting concern. In this tutorial, logging and updating the screen have little to do with each other so separate aspects were created. Now let us complete this tutorial with the final updates to the DrawObject class.

```java
public class DrawObject {
    public static void main(String[] args) {
        DrawObject object = new DrawObject();
        object.updateTexture();
        object.updateColor();
    }
    public void updateTexture() {
        System.out.println("updating Texture");
    }
    public void updateColor() {
        System.out.println("updating Color");
    }
}
```

Running the final code should still return the same results and maintain the same functionality. The revised DrawObject class is now much more modularized which makes it more readable and understandable.

**VII. CONCLUSION**

Aspects consists of advices and pointcuts and help relieve the issues introduced by cross cutting concerns. Although the use of Aspect Oriented Programming introduces many new terminologies, the benefits of using Aspect Oriented Programming cannot be overlooked. Aspect Oriented Programming can be used in conjunction with Object Oriented Programming to improve the modularization of code, thus providing software engineers with code that is more readable, maintainable and coherent. For this reason despite AOP being a relatively new paradigm in comparison to OOP, its catching a lot of attention in the development world, enough that there are
several platforms designed for AOP such as Spring or AspectJ. Despite this paper’s focus on the application of AOP in conjunction to OOP, which is the most common application of AOP, it is not limited to OOP [2]. Other paradigms have the same low modularity and high coupling issues such as procedural and functional languages [2]. The application of AOP to procedural language is not common but it is possible.

ACKNOWLEDGMENTS

The authors would like to thank Robert Kremer and Tristan for their assistance and feedback throughout the development and research process. This research was also assisted by the University of Calgary for providing work space, access to academic papers and other resources that provided background knowledge and information about Aspect Oriented Programming.

REFERENCES


