Oriental Journal of Computer Science & Technology Vol. **2**(2), 113-118 (2009)

Modularizing the cross cutting concerns through aspect-oriented programming

SYED IMTIYAZ HASSAN[1] and S.A.M. RIZVI[2]

[1]Department of Computer Science, Jamia Hamdard, New Delhi (India).
[2]Department of Computer Science, Jamia Millia Islamia, New Delhi (India).

(Received: September 30, 2009; Accepted: November 09, 2009)

ABSTRACT

No one process, technique, language, or platform is good for all situations. Object oriented is not an exception. There are many situations, which the traditional Object-Oriented Programming (OOP) can't deal and implement as it should be. One of them is cross-cutting concerns like logging or security that affects multiple implementation modules. Using object-oriented techniques cross-cutting concerns are difficult to map in a single class and hence they are scattered throughout the code. Due to scattering resulting systems are harder to design, understand, implement, and evolve. Aspect-oriented programming (AOP) is one of the ideas that can be used to modularize the crosscutting concerns better than previous methodologies. The present paper discusses the problems caused by crosscutting concerns and then shows how AOP can solve the said problem with the help of a sample code implemented in AspectJ. This paper also demonstrates how one can use Eclipse IDE to implement AOP.

Key words: AOP, Cross-Cutting Concerns, Aspect, Aspect, OO.

INTRODUCTION

The design and evolution of programming languages is one of the most important areas of computer science. People sought to formalize methods for constructing correct, efficient and easily modified programs. Languages have continued to evolve in order to support their ever increasing usage and support new requirements. The history is full of with examples of improvements in our approach to building software, from the introduction of high-level languages, structured programming, and the object-oriented approach. Today, object-oriented is the leading programming paradigm for software development. One of the major aims of object-oriented development is to organize the data of an application and its associated methods into coherent entities to encourage software reuse[1]. However, object-oriented paradigm is not able to manage a complex program effectively, when it contains Cross-cutting concerns and Code scattering. A typical system may consist of several kinds of concerns, including business logic, performance, data persistence, logging and debugging, authentication, security, multithread safety, error checking, and so on. Aspect-Oriented Programming (AOP)[2,3], which represents one facet of Aspect-Oriented Software Development (AOSD) is the new entrant that offers a solution to of the said problem that has plagued software developers for years. AOP introduces the notion of Aspects, and shows how we can take crosscutting concerns out of modules and place them in a centralized place. The person most commonly associated with AOP is Gregor Kiczales who worked on AOP from 1984 to 1999 at the Xerox Palo Alto Research Center (PARC) and was a leader in developing implementations of it[2].

AOP is not replacement of OOP, but it complements OOP by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns that cut across multiple types and objects. AspectJ[4,5] is language for AOP in Java whereas Spring framework[6] for Java provides different API for implementing AOP.

AOP concepts

Some of the concepts related to AOP is discussed briefly for understanding of readers.

Aspect

An aspect is modular units that cross-cut the structure of other units. Aspects are elements such as security policies and synchronization, optimization, communication or integrity rules that crosscut traditional module boundaries[7]. Transaction management is a good example of a crosscutting concern in J2EE applications[8]. An aspect is similar to a class by having a type, it can extend classes and other aspects, and it can be abstract or concrete and have fields, methods, and types as members. It encapsulates behaviours that affect multiple classes into reusable modules.

Join point

A point in the program flow where something happens is called join point. AOP languages use Join points to modularize crosscutting concerns. The join points are well-defined points in the execution of a program like method calls, field access, conditional checks, loop beginnings, assignments and object constructions. Advice: Advice is an action taken by an aspect at a particular join point. Different types of advice include around, before and after (returning, throwing, finally) advice. Before advice is an advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception). An after returning advice is the advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception. After throwing advice is to be executed if a method exits by throwing an exception. After finally advice is to be executed regardless of the means by which a join point exits (normal or exceptional return).

Around advice is an advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behaviour before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception. Around advice is the most general kind of advice.

Pointcut

A Pointcut is the predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP. Weaving: The process of combining aspect and object models to create the desired runtime behaviour is called weaving. This can be done at compile time, load time, or at runtime. In compile-time weaving, the weaver is a program that, prior to any execution, produces an application code in which the classes are extended by the aspects. AspectJ is the most well-known compile-time aspect weaver. A compile-time weaver is very similar to a compiler and is often referred to as an aspect compiler or even as a compiler. Whereas in run-time weaving, the distinction between application objects and aspects is clearly established during the execution. A run-time weaver executes either the application code or the aspect code, depending on the defined weaving directives. The process of weaving aspects at run time can be compared to maintaining a relationship between a set of application objects and a set of aspect instances. The advantage of run-time weaving is that the relationships between objects and aspects can be dynamically managed.

Development steps of AOP

AOP involves three distinct development steps: Aspectual decomposition, Concern implementation and Aspectual recomposition. Aspectual decomposition decomposes the requirements to identify crosscutting and common concerns. Module-level concerns are separated from crosscutting system-level concerns. Concern implementation implements each concern separately. While in Aspectual recompositions step**,** an aspect integrator specifies recomposition rules by creating modularization units — aspects. The recomposition process, also known as weaving or integrating, uses this information to compose the final system.

Advantages of AOP

The advantages of AOP are

Cleaner responsibilities of the individual module

AOP allow a module to take responsibility only for its core concern; a module is no longer liable for other crosscutting concerns. For example, a module accessing a database is no longer responsible for pooling database connections as well. This results in cleaner assignments of responsibilities, leading to improved traceability.

Higher modularization

AOP provides a mechanism to address each concern separately with minimal coupling. These results in modularized implementation even in the presence of crosscutting concerns. Such implementation results in a system with much less duplicated code. Because the implementation of each concern is separate, it also helps avoid code clutter. Modularized implementation results in an easier-to understand and easier-to-maintain system.

Easier system evolution

AOP modularizes the individual aspects and makes core modules oblivious to the aspects. Adding a new functionality is now a matter of including a new aspect and requires no change to the core modules. Further, when we add a new core module to the system, the existing aspects crosscut it, helping to create a coherent evolution. The overall effect is a faster response to new requirements.

More code reuse

The key to greater code reuse is a more loosely coupled implementation. Because AOP implements each aspect as a separate module, each module is more loosely coupled than equivalent conventional implementations. In particular, core modules aren't aware of each other-only the weaving rule specification modules are aware of any coupling. By simply changing the weaving specification instead of multiple core modules, we can change the system configuration. For example, a database module can be used with a different logging implementation without change to either of the modules.

Improved time-to-market

Late binding of design decisions allows a much faster design cycle. Cleaner separation of responsibilities allows better matching of the module to the developer's skills, leading to improved productivity. More code reuse leads to reduced development time. Easier evolution allows a quicker response to new requirements. All of these lead to systems that are faster to develop and deploy.

Reduced costs of feature implementation

By avoiding the cost of modifying many modules to implement a crosscutting concern, AOP make it cheaper to implement the crosscutting feature. By allowing each implementer to focus more on the concern of the module and make the most of his or her expertise, the cost of the core requirement's implementation is also reduced. The end effect is a cheaper overall feature implementation.

Example

The code snippet 1 is used to model a case depicted in Figure 1 using Aspect.

Two figure elements Point and Line are considered here that can be moved in a plane. Since moveBy() and set..() methods are used whether a Point or a Line is moved, these methods are considered for pointcut (refer code snippet 1).

```
class Line {
  privatePoint p1, p2;
  Point getP1() {
    return p1;
  }
  Point getP2() {
    return p2;
  }
  void setP1(Point p1) {
    this.p1 = p1;Display.update(this);
  }
  void setP2(Point p2) {
    this.p2 = p2;Display.update(this);
  }
  void moveBy(intdx, intdy) {
     …
  }
}
```

```
class Point{
  privateintx = 0, y = 0;
  int getX() {
    return x;
  }
  int getY() {
    return y;
  }
  void setX(int x) {
     this.x= x;Display.update(this);
  }
  void setY(int y) {
    this.y= y;Display.update(this);
  }
  voidmoveBy(intdx, intdy) {
    …
  }
}

Aspect DisplayUpdating{
  pointcut move(FigureElementfigElt):target(figElt)
&&
    (call(voidFigureElement.moveBy(int, int)
     ||call(voidLine.setP1(Point))
     ||call(voidLine.setP2(Point))
     ||call(voidPoint.setX(int))
    ||call(voidPoint.setY(int)));
  after(FigureElementfe) returning:
    move(fe){Display.update(fe);
  }
}
```

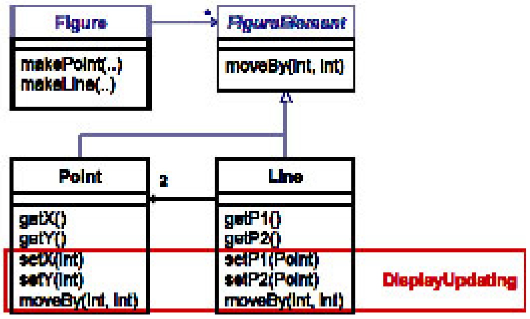
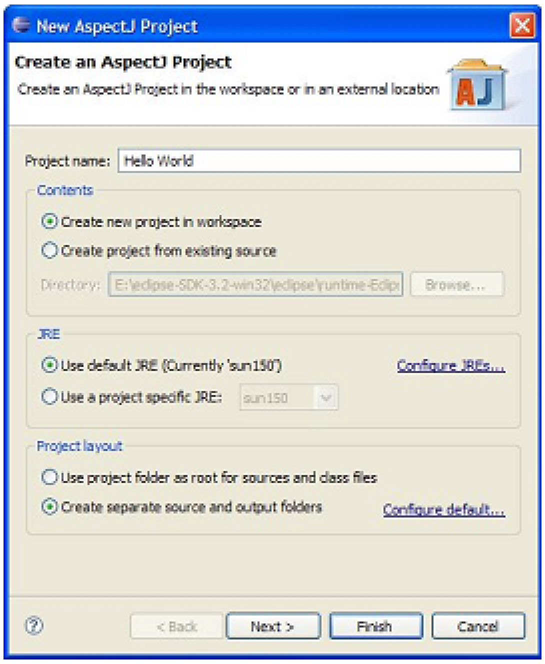


Fig. 1: Line and Point class

Code snippet 1: Demonstrating Aspect

Using Eclipse for AspectJ

In this section, creation of a new AspectJ project and add a package and a Java class is discussed. A very simple application has been



New AspectJ Project

Create an AspectJ Project

Create an AspectJ Project in the workspace or in an external location

Project name: Hello World

Contents

[✓ Create new project in workspace]
Create project from existing source
Directory: E:\eclipse-SDK-3.2-win32\eclipse\runtime-Eclip

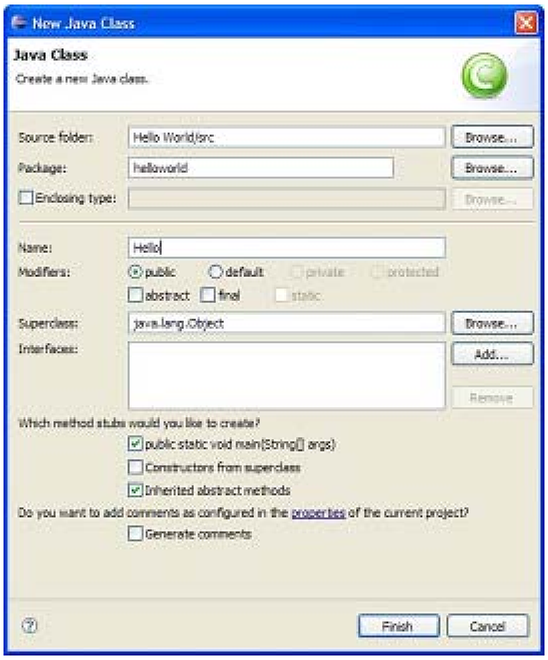
JRE

[✓ Use default JRE (Currently 'sun150')]
Use a project specific JRE: sun150

Project layout

Use project folder as root for sources and class files
[✓ Create separate source and output folders]

Fig. 2: Creating an AspectJ project



New Java Class

Java Class

Create a new Java class.

Source folder: Hello World/src
Package: helloworld
Enclosing type:
Name: Hello
Modifiers: [✓ public] default private protected
abstract final static
Superclass: java.lang.Object
Interfaces:

Which method stubs would you like to create?
[✓ public static void main(String[] args)]
Constructors from superclass
[✓ Inherited abstract methods]
Do you want to add comments as configured in the properties of the current project?
Generate comments

Fig. 3: Creating a new Java Class

demonstrated here.

´ Inside Eclipse select the menu item File > New > Project.... to open the New Project wizard.

´ Select AspectJ Project then click Next. On the next page, type “Hello World” in the Project name field and click Finish (refer Figure 2).

The Java perspective opens inside the workbench with the new AspectJ project in the Package Explorer.

´ Select the “Hello World” project in the package explorer then select File > New > Package.... to open the New Java Package wizard. Type “helloworld” in the name field and click finish.

´ Select the package that you just created in the package explorer then select File > New > Class.... to open the New Java Class wizard. Type “Hello” in the name field, select the option to allow Eclipse to create a main method and click finish (refer Figure 3).

´ Type the following into your new class.

```
public static void main(String[] args) {
   sayHello();
}
public static void sayHello() {
    System.out.print(“Hello”);
}
```

Save the file.

To create a new aspect and to add a pointcut do the following:

´ In the Package Explorer view, select the helloworld package. From the package’s context menu, select New > Aspect.

´ Make sure that Hello World appears in the Source Folder field and that helloworld appears in the Package field. In the Name field, type World. Click Finish to create the new aspect (refer Figure 4).

´ The new file is opened in the editor. It contains the new aspect, the constructor and comments.

´ Change the body of the aspect to the following:

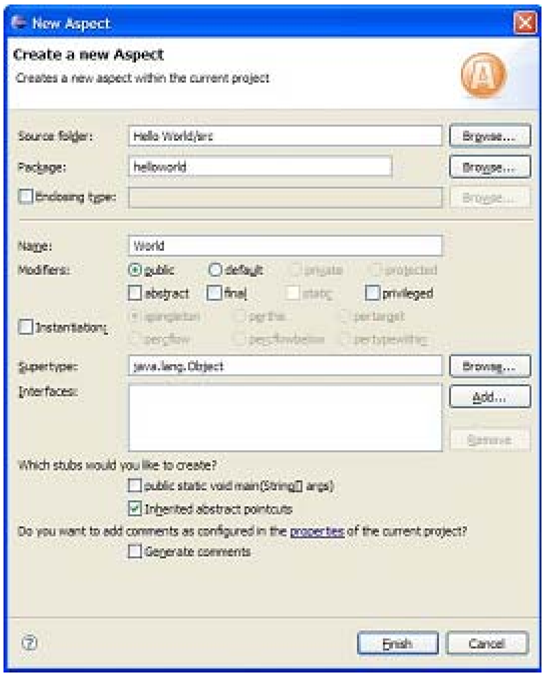
```
public aspect World {
          pointcut  greeting()  :  execution(*
Hello.sayHello(..));
    after() returning() : greeting() {
        System.out.println(“ World!”);
    }
}
```

Save the file.

To run the AspectJ programs do the following:

´ Right click on Hello.java in the Package Explorer and select Java Application from the cascading Run menu. This will launch the selected class as a local Java application. Note that unless you need to run a main method that is in an aspect or use an aspectpath you do not need to use the Run > AspectJ/Java Application option.

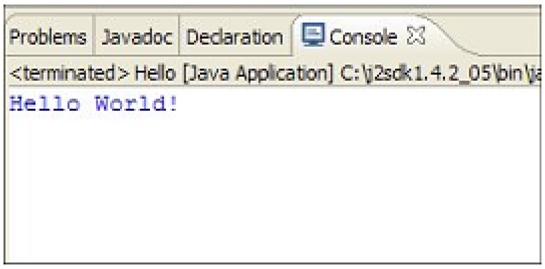
´ Notice that the program has finished



Create a new Aspect
Creates a new aspect within the current project

Source folder: Hello World/src
Package: helloworld
[ ] Enclosing type:
Name: World
Modifiers: [✓ public] ( ) default ( ) private ( ) protected
[ ] abstract [ ] final [ ] static [ ] privileged
[ ] Instantiation: [✓ singleton] ( ) perthis ( ) pertarget ( ) percflow ( ) percflowbelow ( ) pertypewithin
Supertype: java.lang.Object
Interfaces:
Which stubs would you like to create?
[ ] public static void main(String[] args)
[✓] Inherited abstract pointcuts
Do you want to add comments as configured in the properties of the current project?
[ ] Generate comments

Fig. 4: Creating a new Pointcut



```
<terminated> Hello [Java Application] C:\j2sdk1.4.2_05\bin\ja
Hello World!
```

Fig. 5: One possible output of Hello World

running and the following message has appeared in the console (refer Figure 5):

CONCLUSION

AOP addresses a problem space that object-oriented and other procedural languages have never been able to deal with. The key difference between AOP and other approaches is that AOP provides component and aspect languages with different abstraction and composition mechanisms. A special language processor called an aspect weaver is used to coordinate the co-composition of the aspects and components. Grady Booch describes aspect-oriented programming as one of three movements that collectively mark the beginning of a fundamental shift in the way software is designed and written. While this paradigm is still relatively new, it seems promising and perhaps given time will replace object-oriented paradigm. The present paper demonstrated a case fit for AOP and implemented it using AspectJ to modularize the cross-cutting concerns. This paper also guided the reader in using Eclipse for AOP.

REFERENCES

1. Booch Grady, “Object-Oriented Analysis and Design with Applications”, 2nd Edition, Addison-Wesley Object Technology Series.
2. Pollice Gary, “A look at aspect-oriented programming”, [Online]. Available: http://www-106.ibm.com/developerworks/rational/library/2782.html
3. Kiczales G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” in proc. of the 11th European Conference on Object-Oriented Computing (ECOOP’97), Jyva¨skyla¨, Finland, June 9–13 (1997). Lecture Notes on Computer Science, Vol. 1241, Springer-Verlag, New York 200-242 (1997).
4. Pahlsson N., “Aspect-Oriented Programming- An Introduction to Aspect-Oriented Programming and AspectJ”, Report for Software Engineering, University of Kalmar, SWEDEN (2002).
5. Kiczales G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An Overview of AspectJ,” in proc. of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), Budapest, Hungary, June 18–22, 2001, Lecture Notes on Computer Science, Vol. 2072, Springer-Verlag, New York 327-353 (2001).
6. Johnson Rod, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu, “Professional Java Development with the Spring Framework”, Wrox publication, JulyISBN: 978-0-7645-7483-2, 1-28 (2005).
7. Booch Grady, “Through the Looking Glass”, [Online]. Available: http://www.ddj.com/architect/184414752.
8. Pawlak R. et al., “Foundations of AOP for J2EE Development”, Apress 1-23 (2005).