

CHAPTER 1

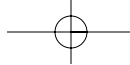
Introduction to AOP

Software development has come a long way since the days of toggle switches. Once the usefulness of software development was realized, its advancement became tied to finding techniques to more efficiently model real-world problems. Years ago, the common methodology for solving a problem was to break it into smaller and smaller modules of functionality, which in turn consisted of a dozen or more lines of code. This methodology worked, but it suffered from a system state being controlled through a large number of global variables that could be modified by any line of code in the application. The advent of object-oriented methodologies pulled the state of the system into individual objects, where it could be made private and controlled through access methods and logic.

This leads to the current situation: Developers are still having difficulty fully expressing a problem into a completely modular and encapsulated model. Although breaking a problem into objects makes sense, some pieces of functionality must be made available across objects. Aspect-oriented programming (AOP) is one of the most promising solutions to the problem of creating clean, well-encapsulated objects without extraneous functionality. In this chapter, we will explore what object-oriented programming (OOP) did right for computer science, problems that arise from objects, and how AOP can fill in the blanks.

Where OOP Has Brought Us

Object-oriented analysis, design, and programming (OOADP) is no longer the new kid on the block; it has been proven successful in both small and large



projects. As a technology, it has gone through its childhood and is moving into a mature adult stage. Research by educational establishments as well as audits by companies have shown that using OOP instead of functional-decomposition techniques has dramatically enhanced the state of software. The benefits of using object-oriented technologies in all phases of the software development process are varied:

- Reusability of components
- Modularity
- Less complex implementation
- Reduced cost of maintenance

Each of these benefits (and others you can think of) will have varied importance to developers. One of them, modularity, is a universal advancement over structured programming that leads to cleaner and better understood software.

What OOADP Did for Computer Science

The object-oriented methodology—including analysis, design, and programming—brought to computer science the ability to model or design software more along the lines of how you envision a system in the real-world. The primary tool used for this modeling is the object. An *object* is a representation of some primary component of the problem domain. The object has attributes representing the state of the object and behaviors that act on the object to change its state. For example, if you were tasked with designing a system to handle selling DVD products, an OO design might include objects like a product, a DVD, and a Boxset, as well as many others.

The objects must be filled out with attributes and behaviors specific to their roles. A product might have a context defined as follows:

- Attributes
 - Price
 - Title
 - Suppliers
- Behaviors
 - Assign price
 - Assign title
 - Get suppliers

Of course, a production system would include many more attributes and behaviors, but those added to the product object here will suit our purpose. In defining the product, we create or acknowledge a relationship between the product and a supplier object. After further decomposition of the problem, DVD objects are created as well as Boxset objects, as shown in Figure 1.1.

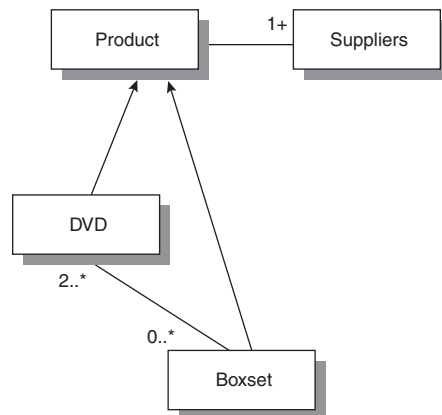
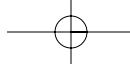


Figure 1.1 Example class model.

One of the goals in object design is encapsulating all the data and methods necessary for manipulating that data fully within the object. There shouldn't be any outside functions that can directly change the product object, nor should the product object make changes to any other object. Instead, a supplier object might send a message to a product object asking it to change its state by adding the supplier object to a list of suppliers in the product object. When a message is sent from one object to another, the receiving object is fully in control of its state. All the attributes of the object are encapsulated in a single entity, which can only be changed through an exposed interface. The exposed interface consists of the methods of the object having a public access type. The object could have internal private methods, but those methods aren't exposed to other objects. The encapsulation of the object is achieved by exposing an interface to other objects in the system. The interface defines the methods that can be used to change the object's state. The functionality behind the exposed interface is kept private.

Designing an object-oriented system in this manner aids in the functioning of the system, debugging if problems arise, and the extension of the system. All the objects in the system know their roles and perform them without worrying about malicious changes being made to their state. From a simplistic view, the system is just a group of objects that execute and send messages to each other, requesting information and changes in the other objects.



As the state of object-oriented technology has evolved, the vocabulary has, as well. As you know, an object is an instantiation of a class. The class is an abstract data-type used to model the objects in a system. A *class* is built based on a requirement extracted through an analysis phase (assuming there is an analysis phase). The class might be built on the fly during coding of a solution, with the requirement written in the comments of the class. These requirements and classes can be linked by a concern.

A *concern* is some functionality or requirement necessary in a system, which has been implemented in a code structure. This definition allows a concern to not be limited to object-oriented systems—a structured system can also have concerns. In a typical system, a large number of concerns need to be addressed in order for the system to accomplish its goals. A system designer is faced with building a system that uses the concerns but doesn't violate the rules of the methodologies being used. When all the concerns have been implemented with system code as well as related functional tests, the system is complete.

Problems Resulting from OOP

If you read books and articles about object orientation, they commonly say that OOP allows for the encapsulation of data and methods specific to the goal of a specific object. In other words, an object should be a self-contained unit with no understanding of its environment, and its environment should be aware of nothing about the object other than what the object reveals. A class is the cookie cutter for objects in a system, and it implements a concern for the system. The goal of the class is to fully encapsulate the code needed for the concern. Unfortunately, this isn't always possible. Consider the following two concerns:

Concern 1: The system shall keep a price relating to the wholesale value of all products.

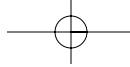
Concern 2: Any changes to the price shall be recorded for historical purposes.

The first concern dictates that all products in the system must have a wholesale price. In the object-oriented world, a Product class can be created as an abstract class to handle common functionality of all products in the system:

```
public abstract class Product {
    private real price;

    Product() {
        price = 0.0;
    }

    public void putPrice(real p) {
        price = p;
    }
}
```



```
    }

    public int getPrice() {
        return price;
    }
}
```

The Product class as defined here satisfies the requirement in concern 1. The principles of OO have been maintained, because the class encapsulates the code necessary to keep track of the price of a product. The same functionality could easily be created in a structured environment using a global array.

Now let's consider concern 2, which requires that all operations involved in changing the price be logged. In itself, this concern does not conflict with the first concern and is easy to implement. The following class defines a logging mechanism:

```
public class Logger {
    private OutputStream ostream;
    Logger() {
        //open log file
    }
    void writeLog(String value) {
        //write value to log file
    }
}
```

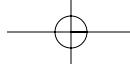
A logger object is instantiated from the Logger class by the application's constructor or other initialization function, or individual logger objects are created within those objects needing to log information. Again, the fundamental object-oriented concepts remain in the Logger class.

To use the logger, you add the writeLog() method to code where the product price might be changed. Because you only have one other class, Product, its methods should be considered for logging inclusion. As a result of the class analysis, a new Product class emerges:

```
public abstract class Product {
    private real price;
    Logger loggerObject;

    Product() {
        price = 0.0;
        loggerObject = new Logger();
    }

    public void putPrice(real p) {
        loggerObject.writeLog("Changed Price from" + price + " to " +
p);
        price = p;
    }
}
```

**6****Introduction to AOP**

```
    }  
  
    public int getPrice() {  
        return price;  
    }  
}
```

The change made to the Product class is the inclusion of the logging method calls in the setPrice() method. When the price is changed using this method, a call is made to the logger object, and the old/new prices are recorded. All objects instantiated from the Product class have a local logger object to handle all logging functionality.

Let's look at the idea of encapsulation and modularity within object-oriented methodologies. By adding code to the Product class to handle a second concern in the system, it would appear that we've broken the idea of encapsulation. The class no longer handles only its concern, but also must fulfill the requirements of another concern. The class has been crosscut by concerns in the system.

Crosscutting represents the situation when a requirement for the system is met by placing code into objects through the system—but the code doesn't directly relate to the functionality defined for those objects. (Crosscutting is discussed in more detail in the next section.) A class like Product, which is defined to represent a specific entity within the application domain, should not be required to host code used to fulfill other system requirements.

Consider what would happen to the Product class if you added timing information, authentication, and long-term data persistence. Are all these concerns supposed to be designed into the Product object? Structured and object-oriented languages leave you no other choice when addressing crosscutting concerns. The additional concerns are forced to be part of another concern, thus breaking many of the rules of our favorite methodology.

This mixing of concerns leads to a condition called code scattering or tangling. With code scattering, the code necessary to fulfill one concern is spread over the classes needed to fulfill another concern. Code tangling involves using a single method or class to implement multiple concerns. Both of these problems break the fundamentals of OO and cause headaches for designers (for more information, see the following section). Consider the following Product class, where the two concerns mentioned earlier have been added in pseudocode form. This additional functionality is necessary, but it shouldn't be part of the Product class:

```
public abstract class Product {  
    private real price;  
    Logger loggerObject;
```

```
Product() {
    price = 0.0;
    loggerObject = new Logger();
}

public void putPrice(real p) {
    //start timing
    //Check user authentication
    loggerObject.writeLog("Changed Price from" + price + " to "
+ p);
    price = p;
    // log if problem with authentication
    //end timing
    //log timing
}

public int getPrice() {
    //check user authentication
    return price;
}

public void persistIt() {
    //start timing
    //save this object
    //end timing
    //log timing
}
}
```

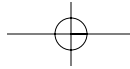
Once the Product class has been created, a DVD concrete class is formulated. The class inherits all the functionality found in the Product class and adds a few more attributes. The DVD class includes an attribute and associated methods for the number of copies currently available. This is important information that should be included in all logging activities:

```
public class DVD extends Product {

    private String title;
    private int count;
    private String location;

    public DVD(String inTitle) {
        super();
        title = inTitle;
    }

    private void setCount(int inCount) {
        //start timing
        //check user authentication
        count = inCount;
    }
}
```



```
        //end timing
        //log timing
    }

    private int getCount() {
        return count;
    }

    private void setLocation(String inLocation, int two) {
        //start timing
        //check user authentication
        location = inLocation;
        //end timing
        //log timing
    }

    private String getLocation() {
        return location;
    }

    public void setStats(String inLocation, int inCount) {
        //start timing
        //check user authentication
        setLocation(inLocation, 0);
        setCount(inCount);
        //end timing
        //log timing
    }
}
```

Do you notice any problems with the code? The logging hasn't been included in the methods that change the count information. Unfortunately, the developer missed this concern when creating the new class.

Results of Tangled Code

A developer doesn't have to be in the industry long to find out the effects of tangled and scattered code. Some of the effects are as follows:

- Classes that are difficult to change
- Code that can't be reused
- Code that's impossible to trace

Engineers and managers who need to refactor code commonly encounter one example of dealing with tangled code. If the code is written in clear components using well-defined objects, a relatively obvious cost-benefit ratio can be created. If the time and money can be justified, the components of the system

can be refactored. However, in most cases, the code for the components is intertwined, and factoring becomes too cost prohibitive under traditional means. However, AOP allows the refactoring to be performed on a different level and in a manner that helps to eliminate some of the tangled code.

In one of the original AspectJ Tutorial presentations (<http://aspectj.org/documentation/papersAndSlides/OOPSLA2002-demo.ppt>), you could analyze the Jakarta Tomcat project to determine where code that performed logging was located in the source code. The result of the project showed that the logging code wasn't in just one place in the code, and not even in a couple of small places—it's spread throughout the source code.

As the Tomcat analysis project showed, code tangling is a major problem. Just think about the nightmare if the code for logging needed to change. The tangled code clearly accomplishes some defined functionality, like logging. The code is tangled because it needs to be spread throughout the application. When a requirement results in tangled code, we say that it *crosscuts* the system. The crosscutting isn't always a primary requirement of the system, just as logging isn't required for the application software to function properly; but sometimes it is required in the case of user authentication.

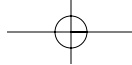
How AOP Solves OOP Problems

Aspect-oriented programming is a paradigm created to solve the problems discussed so far without the difficulties and complexities encountered with subject-oriented programming (SOP) and multidimensional separation of concerns (MDSOC). AOP isn't necessarily a new idea; its roots lie in the separation of concerns movement, but it has moved into the forefront through work by Gregor Kiczales and his colleagues at Xerox's PARC (www.parc.com/groups/csl/projects/aspectj/).

AOP doesn't require the user to learn a host of new techniques, but instead relies on the features of its host language to solve crosscutting of concerns. Depending on the implementation of AOP, you need to learn only a handful of new keywords. At the same time, AOP supports reuse and modularity of code, to eliminate code tangling and scattering. With the advent of Java and the AspectJ support language, AOP is on the verge of becoming the next big thing in computer science since the adoption of OOP.

What Is AOP?

Aspect-oriented programming is a paradigm that supports two fundamental goals:



- Allow for the separation of concerns as appropriate for a host language.
- Provide a mechanism for the description of concerns that crosscut other components.

AOP isn't meant to replace OOP or other object-based methodologies. Instead, it supports the separation of components, typically using classes, and provides a way to separate aspects from the components. In our example, AOP is designed to support the separation of the example concerns and to allow both a *Logger* and a *Product* class; it also handles the crosscutting that occurs when logging is required in the components supporting another concern.

Development Process with AOP

To get an idea of how AOP helps with crosscutting, let's revisit the example concerns:

Concern 1: The system shall keep a price relating to the wholesale value of all products.

Concern 2: Any changes to the price shall be recorded for historical purposes

The two classes built to implement these concerns separated their functionality, as would seem appropriate. However, when concern 2 is fully implemented, it becomes clear that calls from the *Product* class will need to be made to the *Logger* class. Suddenly the *Product* class isn't completely modular, because it needs to incorporate within its own code calls to functionality that isn't part of a product.

AOP provides several tools that can help with this problem. The first is the language used to code the requirements or concerns into units of code (either objects or functions). The AOP literature commonly calls this the component language. The secondary or support requirements (aspects) are coded as well, using an aspect language. Nothing in the paradigm states that either language needs to be object-oriented in nature, nor do the two languages need to be the same. The result of the component and aspect languages is a program that handles the execution of the components and the aspects. At some point, the respective programs must be integrated. This integration is called weaving, and it can occur at compile, link, run-, or load time.

Using this information, let's look at how AOP handles the issue of putting logging code directly into the *Product* class. AOP is designed to respect the idea that some requirements can be modularly coded and others will crosscut the previously modular classes. In our example, concern 1 can be implemented in the *Product* class without violating the modularity of the class. Concern 2 cannot be implemented in a modular fashion within the *Product* class because it

needs to be implemented in different spots throughout the *Product* class and other classes in the software system.

If we step back from the implementation details behind both concerns, we find that concern 2 doesn't necessarily need to be coded directly in the *Product* class (and the *DVD* class, the *Boxset* class, and so on). Instead, it would be ideal if the logging code could be called when the system calls any log-worthy methods.

For this to occur, an aspect must be created specifying that when the system encounters a call to the method `setPrice()`, it should first execute code defined in the aspect language. Here's an example of what the aspect might look like in a (fictional) object-oriented aspect language:

```
define aspect Logging{
  Logger loggingObject = new Logger();
  when calling set*(taking one parameter) {
    loggingObject.writeLog("Called set method");
  }
}
```

This aspect is compiled along with the component *Product* class using a compiler provided by the AOP system. The compiler weaves the aspect code into the component code to create a functioning system. Figure 1.2 shows graphically how the weave looks.

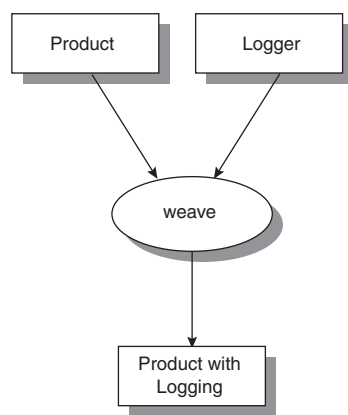
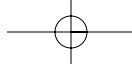


Figure 1.2 A graphical illustration of a weave.

The weave occurs based on the information provided in line 3, where the aspect is defined to act when a call is made to any method having a name starting with `set` and taking a single parameter. Once the system begins to execute, a call is made to the `setPrice()` method of the *DVD* object. Just before control is given to the `setPrice()` method of a target object, the code in line 4 executes and produces the statement "Called set method" in the system log. As a result of using



AOP, any call matching the aspect criteria produces an entry in the log—you don't have to scatter code throughout the entire program to support the concern.

What's Next

In Chapter 2, we will look at some of the details behind implementing a language extension to support the functionality required in AOP. The primary consideration in any AOP tools is the weaver. We'll discuss the current state of AOP weavers as well as future implementations.

