International Journal of Computer Science and Informatics

Volume 3 | Issue 1 Article 9

July 2013

# Improving Software Modularity using AOP

B Vasundhara
*Dept. of Computer Science, AMS School of Informatics, Hyderabad, India*, vasu_venki@yahoo.com

KV Chalapati Rao
*CVR College of Engineering, Ibrahimpatnam, India*, chalapatiraokv@gmail.com

# Improving Software Modularity using AOP

**B Vasundhara[1] & KV Chalapati Rao[2]**

[1]Dept. of Computer Science, AMS School of Informatics, Hyderabad, India
[2]CVR College of Engineering, Ibrahimpatnam, India
E-mail : vasu_venki@yahoo.com[1], chalapatiraokv@gmail.com [2]

***Abstract*** **-** Software modularity is a software design technique that increases the level to which software is composed of separate interchangeable components called modules. The modules are devised by breaking down the program functions. Each module accomplishes one function and contains all that is necessary to accomplish it. Modules represent a separation of concerns, and improve the maintainability by enforcing logical boundaries between the components. Languages that formally support the module concept include Java, AspectJ, etc. OOP supports modularity, i.e., the source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be passed around inside the system. When compared to OOP, AOP is more finely grained, making it more functional for software engineering. In OOP, a software module corresponds directly to a block of executable code. Whereas in AOP, a crosscutting concern, can be located in multiple code blocks. This can turn modules into a tangled mess of crosscutting concerns. AOP is a programming paradigm that increases modularity by allowing the separation of cross-cutting concerns. AOP does not replace OOP in the maintenance of the systems but adds certain decomposition features that address the domination of crosscutting concerns. The effectiveness of AOP is illustrated by discussing the logging function in the online shopping catalogue application.

***Keywords*** **-** *aspect; aspectJ; logging aspect; oop.*

## I. INTRODUCTION

Modularity is a key concept that programmers use in their effort against the complexity of software systems. Implementation of crosscutting concerns in traditional programming languages like C, C#, Java, etc., result in software that is difficult to maintain and reuse. Code tangling occurs when implementations of different concerns coexist within the same module. Code scattering occurs when similar code fragments responsible for the same concern, spread through many modules. Code tangling and scattering are damaging to the software architecture.

Object Oriented Programming (OOP) allows us to use objects to encapsulate units of functionality in classes that provide for hierarchical inheritance. However, objects do not help much in dealing with systemic, crosscutting issues that are not confined to a single class, a software module, or a hierarchical location.

Aspect Oriented Programming (AOP) is a technique that aids in re-engineering systems, because it modularizes crosscutting concerns without actually modifying the original source code. AOP saves us from writing this code [4].

The ideas and practices of OOP stay relevant. Having a good object design will probably make it easier to extend it with aspects. Therefore AOP should not be seen as a replacement of OOP, but as an approach that makes your code more loosely-coupled, clean and focused on the business logic.

AOP introduces a new programming element: the *aspect* [4]. An aspect is defined as a piece of code that describes a recurring property of a program. Aspects provide the crosscutting modularity. We can use aspects to create software modules for issues that cut across various parts of an application. Aspects thus have the potential to make programmers' work easier, less time consuming, and less error prone. Also aspects can lead to less expensive applications, shorter upgrade cycles, and software that is flexible and more customizable.

With AOP, a segment of source code doesn't map linearly to a section of compiled code but maps to all instances in which a particular aspect appears. Changing a line of AOP code can thus have a widespread effect(s) on the compiled code. This makes AOP a potentially powerful programming methodology.

The main idea of AOP is that while the hierarchical modularity mechanisms of object-oriented languages

are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems [2]. The objective of AOP is to supplement object-oriented programming, and not to replace it, by facilitating another type of modularity that brings together the scattered implementation of a crosscutting concern into a single unit.

## II. ASPECT ORIENTED PROGRAMMING

AOP enables a clear separation of concerns in software applications. One of the main promises of AOP is to promote improved modularization of crosscutting concerns, thereby enhancing the software stability in the presence of changes. Following the principle of separation of concerns, the idea of AOP is to separate the component code from the aspect code. The aspect code can consist of several aspect programs, each of which implements a specific aspect in a problem-oriented language. An aspect weaver then takes the component and the aspect code, interprets both of them, finds join points and weaves all together to form a single entity. AOP aims at achieving a better separation of concerns by localizing cross-cutting features, for example, the code implementing non-functional requirements such as memory management, debugging, failure handling, synchronization, etc. An aspect weaver, a compiler-like entity, composes the final system by combining the core and crosscutting modules through a process called weaving [4].

Instead of developing code for each module where a functional component is encountered, an aspect is developed and then code is injected in the appropriate locations using an aspect weaver. The aspect weaver may increase the code size but the advantage obtained by removing redundant code from the different modules outweighs this disadvantage. By identifying the core concerns and crosscutting concerns of a system, we can focus on each individual concern separately, thus reducing the overall complexity of design and implementation. By separating the crosscutting and core concerns, the effect of the requirements contained in such concerns can be reasoned easily as there is no longer scattering or tangling. This separation of crosscutting concerns also supports extensibility, since changes are brought about in aspect code without altering base code. The early identification of concerns will result in untangled and non-scattered design and code. This will reduce the cost of implementation and enhances maintainability.

AOP provides a mechanism we can use to write code representing crosscutting concerns once and have it appear wherever needed. We can write references to aspects at join points, the appropriate places in code where the aspects belong. The references then call the required aspects. The AOP compiler reads the reference and weaves the aspects into the application where they belong. The compiler also combines the separate aspect descriptions into an executable form. This occurs in the same way that an OOP compiler makes sure that method calls happen properly. Aspects thus eliminate many lines of scattered code that programmers would otherwise have to spend considerable time writing, tracking, maintaining, and changing. This makes changing and upgrading applications more accurate. AOP allows us to change an aspect once and have it affect the aspect wherever it occurs in an application.

Aspects can even allow code reuse for the extra-functional requirements they implement, which usually crosscut the whole system. Thus, they make system implementation easier and faster. AOP and AspectJ have an impact on virtually every kind of programming, including enterprise applications, desktop clients, real-time systems, and embedded systems [1,4,5]. AOP contributes to reuse because it is modular. That is, we can use aspect modules wherever necessary in an application without rewriting code.

## III. AspectJ

AspectJ is a simple and practical aspect-oriented extension to Java with a few new constructs. AspectJ provides support for modular implementation of a range of crosscutting concerns. In AspectJ, join points are well-defined points in the execution of the program; pointcuts are collections of join points; advices are special method-like constructs that can be attached to pointcuts; and aspects are modular units of crosscutting implementation, comprising pointcuts, advice, and ordinary Java member declarations [5]. AspectJ is powerful, and programs written using it are easy to understand. AspectJ is the basis for a practical assessment of AOP with Java compatibility.

Aspect-based languages are aspect enhancements to current languages such as Java, C, and C++. Thus, developers can use AOP in various types of applications, especially highly configurable programs in which the methodology's power would generate the programming and financial benefit. These programs frequently have many recurring issues and also require changes that enable reconfiguration, which are both addressed by aspects.

The AOP approach is discussed with reference to the logging aspect in the Online Shopping Catalogue application. Online Shopping Catalogue allows the web site owner to setup online store so that customers can buy the product online. It assists online shopping customers in selecting their purchasing items and paying bills through online payment by using their credit cards or any other means of transaction.

Logging is the exemplary example of a crosscutting concern because a logging strategy affects every single logged part of the system. Logging crosscuts all logged classes and methods. Suppose we do logging both at the beginning and at the end of each function body. This will result in crosscutting all classes that have at least one function. Other typical crosscutting concerns include error handling, performance optimization, security, etc [1].

In its simplest form, logging prints messages describing the operations performed. For example, in a banking system, we would log each account transaction with information such as type of transaction, account number, and the transaction amount. By examining the log, one can spot unexpected system behavior and correct it. A log also helps us to see the interaction between different parts of a system and acts as a diagnostic assistant in order to detect exactly where the problem might be [3].

Currently used mechanisms implement logging along with the operation's core logic, which is tangled with the logging statements. Since logging is a crosscutting concern, AOP and AspectJ can help modularize it. With AspectJ, we can implement the logging mechanism independent of the core logic. AspectJ simplifies the logging task by modularizing its implementation and obviating the need to change many source files when requirements change. AspectJ not only saves lots of code, but also establishes centralized control, consistency, and efficiency.

Consider implementing logging without AspectJ. Since logging is a common requirement, APIs and libraries are available that allow us to perform logging consistently. Notable are the Java logging API and log4j from Apache. Although these logging APIs are a significant improvement over the use of System.out.println() or other solutions, in the complete perspective, they provide only a part of the answer. To illustrate we will look at how we can add logging to the Online Shopping Catalogue example.

*A. Logging the Conventional Way*

First, we implement each method of the Product class to log the entry into it as in Figure 1. We choose to log each method at the level Level.INFO because, we are simply writing informational entries to the log when we enter the methods. The Product class has methods for querying product identifier and its price. Later, we change the Product class by adding code to obtain the logger object and log each method.

Next, as done with the Product class, we implement logging into the ShoppingCatalogue class's methods, as shown in Figure 2. The ShoppingCatalogue a list of products and allows us to add and remove products to the list. We notice that the changes needed for logging in both classes are the same, i.e., every method needs to make an additional call to the logp() method.

```
import java.util.logging.*;
public class Product
{
   static Logger _logger = Logger.getLogger("trace");
   private String _productid;
   private float _productprice;
     public Product(String productid, float productprice)
{          _productid = id;
      _productprice = price;    }
    public String getID()
{_logger.logp(Level.INFO,        "Product",        "getID",
  "Entering");
      return _productid;    }
    public float getPrice()
{_logger.logp(Level.INFO,"Product","getPrice","Enteri
ng");
      return _productprice;    }
    public String toString()
{_logger.logp(Level.INFO,"Product","toString",
"Entering");
      return "Product: " + _productid;    }
}
```

Fig. 1 : The Product class with logging enabled

```
import java.util.*;
import java.util.logging.*;
public class ShoppingCatalogue
{
  static Logger _logger = Logger.getLogger("trace");
   private List _products = new Vector();
    public void addProduct(Product product)
{_logger.logp(Level.INFO,"ShoppingCatalogue","addP
roduct", "Entering");
    _products.add(product);    }
    public void deleteProduct(Product product)
{_logger.logp(Level.INFO,"ShoppingCatalogue",
"deleteProduct", "Entering");
```

```
    _products.delete(product);    }
    public void empty()
{_logger.logp(Level.INFO,"ShoppingCatalogue","empty", "Entering");
    _products.clear();    }
    public float producttotalValue()
{_logger.logp(Level.INFO,"ShoppingCatalogue", "producttotalValue", "Entering");
   return 0;    }}
```

Fig. 2 : The Shopping Catalog class with logging enabled

The logging implementation for the Inventory, Shopping Catalogue Operator, and Test classes is done very similarly. We now have an implementation with an entry for each method to be logged, using the standard Java logging toolkit. Then we compile the classes and run the program.

The job here is just mechanical, i.e., copy and paste code and modify the arguments to logp() methods. How long it would take to introduce logging in a real system with hundreds of classes? How sure could one be that the methods would log the right information?

*B. Logging the Apect Oriented Way*

With AspectJ-based logging, we need not modify the classes.

```
import java.util.logging.*;
import org.aspectj.lang.*;
 public aspect TracingAspect
{
   private Logger _logger = Logger.getLogger("trace");
  pointcut tracingMethods() : execution(* *.*(..)) &&
  !within(TracingAspect);
  before() : tracingMethods()
{ Signature sig = thisJoinPointStaticPart.getSignature();
  _logger.logp(Level.INFO,
   sig.getDeclaringType().getName(),sig.getName(),
   "Entering");   }
}
```

Fig. 3 : TraceAspect performing the same job

We then compile this aspect together with the shopping catalogue classes and run the test program using the AspectJ compiler. We observe that by using AspectJ we have saved on the amount of code. Such modularization is possible because of AOP and AspectJ's support for programming crosscutting concerns [4, 5].

*C. Problems with Conventional Logging*

Every place that needs to log an event needs to explicitly invoke a call to the log method. The logging calls will be all over the core modules. When a new module is added to the system, all of its methods that need logging must be implemented. Such implementation is invasive, causing the tangling of the core concerns with the logging concern. Further, if we happen to change the logging toolkit to a different API, we need to revisit every logging statement and modify it.

Consistency is the most important requirement of logging. It means that if the logging specification requires that certain kinds of operations be logged, then the implementation must log every invocation of those operations. Achieving consistency using conventional logging is a major goal, and while systems can attain it initially, it requires continuing observation to keep it so. For example, if we add new classes to the system or new methods in existing classes, we must ensure that they implement logging that matches the current logging strategy.

*D. Advantage of AspectJ-based Logging*

AOP and AspectJ overcome the above mentioned limitations [5]. AspectJ easily implements the invocation of logging statements from all the log points. The finery is that one does not need to actually instrument any log points; writing an aspect does it automatically. Also, as there is a central place to control logging operations, we achieve the consistency easily.

The fundamental difference between conventional logging and AspectJ-based logging is the modularization of the logging concern. Instead of writing modules that implement core concepts in addition to invoking logging operations, with AspectJ, we can write a few aspects that advise the execution of the operations in the core modules to perform the logging. That way, the core modules do not carry any logging-related code. Thus by modularization, the logging concerns are separated from the core concerns.

With AspectJ-based logging, the logger aspect separates the core modules and the logger object. Instead of the core modules embedding the log method invocations in their source code, the logger aspect weaves the logging invocations into the core modules when they are needed. AspectJ-based logging reverses the dependency between the core modules and the logger; it is the aspect that encodes how the operations

in the core modules are logged instead of each core module deciding for itself.

## CONCLUSION

AOP deals with code tangling and scattering [4]. Concerns can be mapped easily to different modules, if they are functional in nature. Such concerns are called core concerns. Many systems also contain different kinds of concerns that are hard to factor out in traditional units of decomposition, i.e. functions or classes. Such concerns are called crosscutting concerns. When they are implemented using a traditional language, their code spreads throughout the system. This is because the traditional languages provide only one dimension along which systems can be decomposed.

AspectJ-based logging results in low-investment and low-risk programming. Implementation of such logging concerns can now be nicely modularized. This solution leads to increased flexibility, improved accuracy, and better consistency. It saves us from the tedious task of writing nearly identical log statements in code all over the system. The use of AspectJ also makes the job of choosing logging toolkits an easy task. Further the ability to remove unwanted code from various processes could increase the effectiveness of the system in other areas, such as code size.

## REFERENCES

[1] Barreto, L. P. & Muller, G. (2002) Bossa: "A language based approach to the design of real-time schedulers". 10th International Conference on Real-Time Systems. (RTS'2002). Paris, France. Cooling, J. (2002) Software Engineering for real-time Systems, Addison-Wesley.

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997) "Aspect-oriented programming". European Conference on Object-Oriented Programming. Finland, Springer-Verlag.

[3] Tsang, S. L., Clarke, S. & Baniassad, E. (2004) "An Evaluation of Aspect-Oriented Programming for Java based Real-time Systems Development". Seventh IEEE International Symposium on Object-Oriented Real- Time Distributed Computing (ISORC'04). Vienna, Austria.

[4] "Aspect-oriented programming": http://aosd.net

[5] http://eclipse.org/aspectj

[6] Filman, R.; Elrad, T.; Clarke S.; Aksit, M. (eds.), "Aspect-oriented software development", Addison-Wesley Longman, Amsterdam, 2004.

❑❑❑