[Aspect Oriented Programming (AOP)](http://en.wikipedia.org/wiki/Aspect-oriented_programming) refers to the programming paradigm which isolates secondary or supporting functions from the main program's business logic. AOP is a promising technology for separating crosscutting concerns, something usually hard to do in object-oriented programming. The application's modularity is increased in that way and its maintenance becomes significantly easier.  
  
The most prominent example of a crosscutting concern is [logging](http://www.javacodegeeks.com/2011/01/10-tips-proper-application-logging.html). Logging is used mainly for [debugging](http://www.javacodegeeks.com/2010/06/debug-production-server-eclipse.html) and troubleshooting issues mainly by tracing method calls and overall execution flow. It qualifies as a crosscutting concern because a logging strategy necessarily affects every logged part of the system. Logging thereby crosscuts all logged classes and methods.  
  
Note that AOP and OOP are not exclusive. In contrary, they complement each other and their combined use can help us produce robust and maintainable software. With AOP, we start by implementing our project using our OO language and then we deal separately with crosscutting concerns in our code by implementing [aspects](http://en.wikipedia.org/wiki/Aspect-oriented_programming#Terminology).  
  
AOP's usage and adoption was boosted with the help of our favorite [Spring framework](http://www.javacodegeeks.com/search/label/Spring). Spring makes AOP easier to integrate in our projects using an non-intrusive approach. Justin has talked before about Spring AOP in his post named “[Aspect Oriented Programming with Spring AspectJ and Maven](http://www.javacodegeeks.com/2010/07/aspect-oriented-programming-with-spring.html)” here at [JavaCodeGeeks](http://www.javacodegeeks.com/). However, our latest [JCG partner](http://www.javacodegeeks.com/p/jcg.html), Siva from [SivaLabs](http://sivalabs.blogspot.com/), has also written a [very nice article on Spring AOP](http://sivalabs.blogspot.com/2011/01/aspect-oriented-programming-using.html) and I wanted to share it with you, so here it is.

While developing software applications for a business we receive the project's requirements either from the requirements gathering team or from our business analysts. In general, those requirements are functional requirements which represent the activities that the business is doing. However, when developing software applications, apart from the functional requirements, we should also consider some other points like performance, transaction management, security, logging etc. These are called non-functional requirements.  
  
Let us consider a BookStore application which provides web access to a book store. A user can browse the various categories of books, add some books to a cart and finally checkout, do payment and get the books. For this app we might receive the requirements from a business analyst as follows:

* A login/registration screen to enter into BookStore.
* Users should be able to browse through various categories of books
* Users should be able to search books by name, author name, publisher
* Users should be able to add/remove the books to/from their cart
* Users should be able to see what items currently exist in their cart
* Users should be able to checkout and pay the corresponding amount through some payment gateway
* A successful message should be shown to users with all the details of their purchases.
* A failure message should be shown to users with the cause of failure.
* A BookStore administrator/manager should be granted access to add/remove/update book details.

All the above requirements fall under the “Functional Requirements” category. While implementing the above, we should also take care of the following things even though they are not explicitly mentioned:

* Role based access to the UI. Here, only Administrators/Managers should have access to add/remove/update book details. [Role based Authorization]
* Atomicity in Purchasing. Suppose a user logged into the BookStore and added 5 books into his cart, checked out and completed his payment. In the back-end implementation we may need to enter this purchase details in 3 tables. If after inserting the data into 2 tables the system crashed, the whole operation should be rolled-back. [Transaction Management].
* No one is perfect and no system is flawless. So if something went wrong and the development team has to figure it out what went wrong, logging will be useful. So, logging should be implemented in such a way that developer should be able to figure out where exactly the application failed and fix it. [Logging]

The above implicit requirements are called Non-Functional Requirements. In addition to the above, performance should obviously be a crucial non-functional requirement for all public facing websites.  
  
So, with all the above functional requirements we can build the system by decomposing it into various components band taking care of the non-functional requirements through out the components.

public class OrderService

{

private OrderDAO orderDAO;

public boolean placeOrder(Order order)

{

boolean flag = false;

logger.info("Entered into OrderService.placeOrder(order) method");

try

{

flag = orderDAO.saveOrder(order);

}

catch(Exception e)

{

logger.error("Error occured in OrderService.placeOrder(order) method");

}

logger.info("Exiting from OrderService.placeOrder(order) method");

return flag;

}

}

public class OrderDAO

{

public boolean saveOrder(Order order)

{

boolean flag = false;

logger.info("Entered into OrderDAO.saveOrder(order) method");

Connection conn = null;

try

{

conn = getConnection();//get database connection

conn.setAutoCommit(false);

// insert data into orders\_master table which generates an order\_id

// insert order details into order\_details table with the generated order\_id

// insert shipment details into order\_shipment table

conn.commit();

conn.setAutoCommit(true);

flag = true;

}

catch(Exception e)

{

logger.error("Error occured in OrderDAO.saveOrder(order) method");

conn.rollback();

}

logger.info("Exiting from OrderDAO.saveOrder(order) method");

return flag;

}

}

In the code above , the functional requirement implementation and non-functional requirement implementation are mingled in the same place. Logging is placed across the OrderService and OrderDAO classes. At the same time, transaction management is spanned across DAOs.  
  
With this approach we have several issues:

1. The classes need to be changed either to change functional or non-functional requirements. For example: At some point later in the development if the Team decides to log the Method Entry/Exit information along with TimeStamp, we need to change almost all the classes.
2. The Transaction Management code setting the auto-commit to false in the beginning, doing the DB operations, committing/rollbacking the operation logic will be duplicated across all the DAOs.

This kind of requirements which span across the modules/components is referred to as Cross Cutting Concerns. To better design the system we should separate these cross cutting concerns from the actual business logic so that it will be easier to change or enhance or maintain the application at a later point.  
  
Aspect Oriented Programming is a methodology which enables the separation of the cross cutting concerns from the actual business logic. So, let us follow the AOP methodology and redesign the above two classes separating the cross cutting concerns.

public interface IOrderService

{

public boolean placeOrder(Order order);

}

public class OrderService implements IOrderService

{

private OrderDAO orderDAO;

public boolean placeOrder(Order order)

{

return orderDAO.saveOrder(order);

}

}

public class OrderDAO

{

public boolean saveOrder(Order order)

{

boolean flag =false;

Connectoin conn = null;

try

{

conn = getConnection();//get database connection

// insert data into orders\_master table which generates an order\_id

// insert order details into order\_details table with the generated order\_id

// insert shipment details into order\_shipment table

flag = true;

}

catch(Exception e)

{

logger.error(e);

}

return flag;

}

}

Now lets create a LoggingInterceptor implementing how logging should be done and create a Proxy for OrderService which takes the call from the caller, logs the entry/exit entries using LoggingInterceptor and finally delegates to the actual OrderService.  
  
By using Dynamic Proxies we can separate out the implementation of the cross cutting concerns (such as logging) from actual business logic as follows:

public class LoggingInterceptor

{

public void logEntry(Method m)

{

logger.info("Entered into "+m.getName()+" method");

}

public void logExit(Method m)

{

logger.info("Exiting from "+m.getName()+" method");

}

}

public class OrderServiceProxy implements IOrderService extends LoggingInterceptor

{

private OrderService orderService;

public boolean placeOrder(Order order)

{

boolean flag =false;

Method m = getThisMethod();//get OrderService.placeOrder() Method object

logEntry(m);

flag = orderService.placeOrder(order);

logExit(m);

return flag;

}

}

Now the OrderService caller (OrderController) can get the OrderServiceProxy and place the order as:

public class OrderController

{

public void checkout()

{

Order order = new Order();

//set the order details

IOrderService orderService = getOrderServiceProxy();

orderService.placeOrder(order);

}

}

There are several AOP frameworks which can be used in order to separate the implementation from the cross cutting concerns.

* Spring AOP
* AspectJ
* JBoss AOP

Let's see how we can separate out logging from the actual business logic using Spring AOP. Before that, first we need to understand the following terms:

* **JoinPoint:** A joinpoint is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modified.
* **Pointcut:** A pointcut definition matches one or more joinpoints at which advice should be woven. Often you specify these pointcuts using explicit class and method names or through regular expressions that define matching class and method name patterns.
* **Aspect:** An aspect is the merger of advice and pointcuts.
* **Advice:** The job of an aspect is called advice. It is the additional code we apply to the existing model.

SpringAOP supports several types of advices, namely:

* **Before:** This advice weaves the aspect before method call.
* **AfterReturning:** This advice weaves the aspect after method call.
* **AfterThrowing:** This advice weaves the aspect when method throws an Exception.
* **Around:** This advice weaves the aspect before and after method call.

Suppose we have the following ArithmeticCalculator interface and implementation classes.

package com.springapp.aop;

public interface ArithmeticCalculator

{

public double add(double a, double b);

public double sub(double a, double b);

public double mul(double a, double b);

public double div(double a, double b);

}

package com.springapp.aop;

import org.springframework.stereotype.Component;

@Component("arithmeticCalculator")

public class ArithmeticCalculatorImpl implements ArithmeticCalculator

{

public double add(double a, double b)

{

double result = a + b;

System.out.println(a + " + " + b + " = " + result);

return result;

}

public double sub(double a, double b)

{

double result = a - b;

System.out.println(a + " - " + b + " = " + result);

return result;

}

public double mul(double a, double b)

{

double result = a \* b;

System.out.println(a + " \* " + b + " = " + result);

return result;

}

public double div(double a, double b)

{

if(b == 0)

{

throw new IllegalArgumentException("b value must not be zero.");

}

double result = a / b;

System.out.println(a + " / " + b + " = " + result);

return result;

}

}

package com.springapp.aop;

import java.util.Arrays;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.core.annotation.Order;

import org.springframework.stereotype.Component;

@Aspect

@Component

public class LoggingAspect

{

private Log log = LogFactory.getLog(this.getClass());

@Pointcut("execution(\* \*.\*(..))")

protected void loggingOperation() {}

@Before("loggingOperation()")

@Order(1)

public void logJoinPoint(JoinPoint joinPoint)

{

log.info("Join point kind : " + joinPoint.getKind());

log.info("Signature declaring type : "+ joinPoint.getSignature().getDeclaringTypeName());

log.info("Signature name : " + joinPoint.getSignature().getName());

log.info("Arguments : " + Arrays.toString(joinPoint.getArgs()));

log.info("Target class : "+ joinPoint.getTarget().getClass().getName());

log.info("This class : " + joinPoint.getThis().getClass().getName());

}

@AfterReturning(pointcut="loggingOperation()", returning = "result")

@Order(2)

public void logAfter(JoinPoint joinPoint, Object result)

{

log.info("Exiting from Method :"+joinPoint.getSignature().getName());

log.info("Return value :"+result);

}

@AfterThrowing(pointcut="execution(\* \*.\*(..))", throwing = "e")

@Order(3)

public void logAfterThrowing(JoinPoint joinPoint, Throwable e)

{

log.error("An exception has been thrown in "+ joinPoint.getSignature().getName() + "()");

log.error("Cause :"+e.getCause());

}

@Around("execution(\* \*.\*(..))")

@Order(4)

public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable

{

log.info("The method " + joinPoint.getSignature().getName()+ "() begins with " + Arrays.toString(joinPoint.getArgs()));

try

{

Object result = joinPoint.proceed();

log.info("The method " + joinPoint.getSignature().getName()+ "() ends with " + result);

return result;

} catch (IllegalArgumentException e)

{

log.error("Illegal argument "+ Arrays.toString(joinPoint.getArgs()) + " in "+ joinPoint.getSignature().getName() + "()");

throw e;

}

}

}

Here is what our applicationContext.xml should include:

<context:annotation-config>

<context:component-scan base-package="com.springapp"></context:component-scan>

<aop:aspectj-autoproxy></aop:aspectj-autoproxy></context:annotation-config>

And here is a standalone test client to test the functionality.

package com.springapp.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringAOPClient

{

public static void main(String[] args)

{

ApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

ArithmeticCalculator calculator =

(ArithmeticCalculator) context.getBean("arithmeticCalculator");

double sum = calculator.add(12, 23);

System.out.println(sum);

double div = calculator.div(1, 10);

System.out.println(div);

}

}

The required libraries are the following:

* Spring.jar (2.5.6 or above)
* commons-logging.jar
* aopalliance.jar
* aspectjrt.jar
* aspectjweaver.jar
* cglib-nodep-2.1\_3.jar

We can define the type of advice using the annotations @Before, @AfterReturning, @Around etc. We can define pointcuts in different ways. For example:  
  
*@Around("execution(\* \*.\*(..))")* means it is an Around advice which will be applied to all classes in all packages and all methods.  
  
Suppose that we want to apply the advice only for all the services that reside in the com.myproj.services package. Then the pointcut declaration would be:  
  
*@Around("execution(\* com.myproj.services.\*.\*(..))")*   
  
In that case, “(..)” means with any type of arguments.   
  
If we want to apply same pointcuts for many advices we can define a pointcut on a method and can refer that later as follows.

@Pointcut("execution(\* \*.\*(..))")

protected void loggingOperation() {}

@Before("loggingOperation()")

public void logJoinPoint(JoinPoint joinPoint){}

If multiple Advices have to be applied on the same pointcut, we can specify the order using the @Order annotation on which advices will be applied. In the previous example, @Before will be applied first. Then, @Around will be applied when the add() method is called.