# Strategies in Aspect Oriented Programming with AspectJ

**2002-11-20**

**By Gustav Evertsson, pt99gev@student.bth.se**

# Contents

# Introduction

This paper is done as a part of the course Advanced Software Engineering (pad004) at Blekinge Institute of Technology. The paper is written in a way to try to explain different strategies with Aspect Oriented programming by showing some example written in AspectJ[1]. All code examples in this report comes from the book “Aspect-Oriented Programming with AspectJ” by Ivan Kiselev[2].

# Crosscutting by Design

## *Catalog of Aspects*

Aspects can be divided into three major groups depending on what they are used for.

- First we have the aspects that the final software works well without. These are called development aspects. They are limited to only be useful during development. Examples of these is logging, tracing and profiling.

- An aspect that must be included for the system to work is called product aspects. Examples in this group are Authentication and Exception handling.

- The third group is aspects that make the program work better but they are not required for the program to function. These are called runtime aspects. Example of these is aspects that raise performance like pooling, caching and buffering.

Because it is so easy to make different compile script can the development aspects be excluded from the final version and stress tests can be done to see how much the runtime aspects raise the performance.

## *Design by Contract and Other Coding Improvements*

Design by contract is that all methods in the system must take care of their part. This include that all pre condition are fulfilled such as checking that the input parameters are correct. The most common way to do this is to have a small if statement in the beginning of all method that checks so the parameters are not null. This can instead be moved to an aspect that can check it for many methods with the same code.

```
package aspects;

public aspect NullChecker
{
    pointcut arguments(): execution(* *.*(..));

    before() : arguments()
    {
        Object args[] = thisJoinPoint.getArgs();
        for(int i=0; i<args.length; i++)
        {
            if( null == args[i] )
            {
                throw new IllegalArgumentException("The argument is ” +
                                                   "null.");
            }
        }
    }
}
```

The aspect only contains a small loop that goes through all the arguments and look for null. It will cast an IllegalArgumentException if it found one. The post-condition checking can be conducted similarly using the after advice.

Good design techniques can also be enforsed using AspectJ static crosscutting. One example where this can be a problem is when you designing a multi layered program and you only

want a layer to talk to the layer direct under it. This code below can be used to for example see that no class jumps over the data handler layer and calls the database direct. It checks so all calls to the getConnection method comes within the db package. There are two levels of messages that can be generated; warnings and errors.

```
package aspects;

import java.sql.*;

public aspect CodeSegregation
{

    pointcut dbCode() : call(
                        Connection DriverManager.getConnection(..));
    pointcut badDbCode(): dbCode() && !within(db.*);
    pointcut reallyBadDbCode(): badDbCode() && !within(security.*) &&
                                !within(servlets.*);

    declare warning: badDbCode() : "Database code outside 'db'
package.";
    declare error  : reallyBadDbCode(): "Database code here is not
permitted.";
}
```

# Development Aids

The capability of aspects to affect a lot of code at once can help to devise a set of tools to help the application development. These aspects cleanly and transparently allow us to get a better handle on what is going on with the application [2].

## *Logging*

The easiest is to make an aspect that picks everything, but that is normally too much. You will miss the information you are looking for in the noise of all the information because even a small program will generate rather much outputs. So the solution to this problem is to first make an abstract aspect that takes care of the logging and then make new logging aspects that extend that one. This has the advantage that you can easily add new logging that is more precise than the more simple solution. It may still not be as precise as with logging direct in the code but the code will be cleaner and it is easier to take away the logging for the final release. As with all output will you get some performance penalty but that is only a problem during debugging.

```
package aspects;

abstract public aspect Logger
{
   abstract pointcut logPoint();

   before() : logPoint && !within(Logger+)
   {
       System.out.println(thisJoinPoint.getTarget() + ", " +
                          thisJoinPoint.getThis() + ", " +
                          thisJoinPoint.getSignature());
   }
}
```

This is the main logging aspects, it handle the output, in this case just to the console. Note the additional pointcut at the advice declaration. It is explicitly set not to fire if the execution flow is inside the logger itself to prevent unlimited recursion when the logger advice itself.

```
package aspects;

public aspect myLogger1 extends Logger
{
   abstract pointcut logPoint() : execution(* myClass.*(..));

}

package aspects;

public aspect myLogger2 extends Logger
{
   abstract pointcut logPoint() : initialization(myClass.new(..));

}
```

This two aspects extends the Logger aspect and override the logPoint() pointcut to decide what to log. The first example logs all executions of methods in the myClass class. The second example logs when a new instance of myClass is created. You can develop an unlimited number of logging aspects in the same way as these two examples with more advanced pointcuts to be more accurate in the logging.

## *Tracing*

A tracer can be a very useful tool when it comes to debugging a program. The output is in some ways similar to the logging but have some more distinct functional requirements. First of all is that you only track method calls, and you want to have control over the stack so you can see from where the method calls come from.

```
package aspects;

import java.util.*;

public aspect Tracer
{
   pointcut tracePoint() : execution(* *.*(..)) && !within(Tracer);

   private static Map stackDepth = new HashMap();

   before() : tracePoint()
   {
      Integer depth = (Integer)stackDepths.get(Thread.currentThread());
      if( depth == null )
      {
         depth = new Integer(0);
      }
      System.out.println(depth.intValue() + " >> " +
                         thisJoinPointStaticPart.getSignature());
      stackDepths.put(Thread.currentThread(), new
                      Integer(depth.intValue() + 1));
   }

   after() : tracePoint()
   {
      Integer depth = (Integer)stackDepths.get(Thread.currentThread());
      depth = new Integer(depth.intValue() - 1);
      if(depth.intValue() == 0)
      {
         stackDepths.remove(Thread.currentThread());
      }
      else
      {
         stackDepths.put(Thread.currentThread(), depth);
      }
      System.out.println(depth.intValue() + " >> " +
                         thisJoinPointStativPart.getSignature());
   }

   private static StringBuffer ident(int num)
   {
      StringBuffer ident = new StringBuffer();
      for(int I = 0; I < num; i++)
      {
         ident.append(' ');
      }
      ident.append(Integer.toString(num) + " [" +
                   Thread.currentThread().hashCode() + "]");
      return ident;
   }

}
```

This example is divided into five parts. The first is the pointcut that picks all method calls in the program except within the aspect itself. The next part is the static counter. This is made as a Map so it can work in a multithreaded environment such as JSP pages. So every thread will have its own depth counter and the thread itself is the key in the list. This works this way

because an aspect is singleton as default. The next two parts is the before and after advices that do the actual work. Before creates a new counter if it is the first call and increases the counter. After decrease the counter in the same way and deletes the counter if it reach zero. This is a way of garbage collection for the counters. The last part is made to output the stack depth and thread id so it is easy to follow the execution.

## *Profiling*

After the tracing aspect is it not a long way to make a profiling aspect too. A profiling tool is normally divided into two parts, first data collection and then data analysis. This example here handles only the first part.

```
package aspects;

import java.io.*;
import org.aspectj.lang.SoftException;

public aspect Profiler
{
   pointcut prof(): call(* *.*(..)) && !within(aspects.*);

   FileOutputStream out;

   public Profiler() throws FileNotFoundException
   {
      out = new FileOutputStream("profile.txt");
   }

   before() : prof()
   {
       String record = "+ " +
          Thread.currentThread().hashCode()+" "+
          Long.toString(System.currentTimeMillis())+" "+
          thisJoinPointStaticPart.getSignature()+"\n";
       try
       {
          out.write(record.getBytes());
       }
       catch(IOException e)
       {
           throw new SoftException(e);
       }
   }

   after() : prof()
   {
       String record ="- "+
         Thread.currentThread().hashCode()+" "+
         Long.toString(System.currentTimeMillis())+" "+
         thisJoinPointStaticPart.getSignature()+"\n";
       try
       {
          out.write(record.getBytes());
       }
       catch(IOException e)
       {
          throw new SoftException(e);
       }
   }
}
```

The code is fairly easy. The constructor creates a new file that then before and after writes to. Both write the current time in milliseconds and the signature of the method. The only difference is that before writers a + and after a – in front of the line. This file can then be read by a parser that calculates an average value for every method.

# Runtime Improvements

Here below is some ways to improve the runtime characteristics of your program. They are all problem independent in the way that they can be used in a wide range of different systems.

## *Buffering*

This example here below handle buffering for the FileOutputStream class and picks the write(byte[]) method. Of course can buffering be used on other output streams as well but may need some changes in for example buffer size and time between flushing.

```
package aspects;
import java.io.*;
import java.util.*;

public aspect OutputStreamBuffering implements Runnable
{
    private static Thread flushingThread = null;
    private static final int BUFF_SIZE=512;
    public class Buffer
    {
        byte[] buff = null;
        int counter = 0;

        Buffer()
        {
            buff = new byte[BUFF_SIZE];
        }
    }

    private Map buffTable = new HashMap();

    pointcut writeBytes(byte[] bytes):
    call(void FileOutputStream.write(byte[]))
    && args(bytes)
    && !within(OutputStreamBuffering);

    void around(byte[] bytes) throws IOException: writeBytes(bytes)
    {
        FileOutputStream out =
             (FileOutputStream)thisJoinPoint.getTarget();
        Buffer aBuff = (Buffer) buffTable.get(out);
        if( null == aBuff )
        {
            aBuff = new Buffer();
            buffTable.put(out, aBuff);
        }

        if(aBuff.counter + bytes.length > BUFF_SIZE)
        {
            synchronized(out)
            {
                out.write(aBuff.buff, 0, aBuff.counter);
                out.write(bytes);
                out.write(("*** buffer - "+aBuff.counter +
                           "\n").getBytes());
            }

            buffTable.remove(out);
        }
        else
        {
            System.arraycopy(bytes, 0, aBuff.buff, aBuff.counter,
                             bytes.length);
            aBuff.counter += bytes.length;
```

```
            }

            if( null == flushingThread )
            {
                flushingThread = new Thread(this);
                flushingThread.setDaemon(true);
                flushingThread.start();
            }

        }

        before() throws IOException : call(void FileOutputStream.close())
        {
            FileOutputStream out =
                            (FileOutputStream)thisJoinPoint.getTarget();
            Buffer aBuff = (Buffer) buffTable.get(out);
            if( null != aBuff )
            {
                synchronized(out)
                {
                    out.write(aBuff.buff, 0, aBuff.counter);
                }
                buffTable.remove(out);
            }
        }

        public void run()
        {
            while(true)
            {
                try
                {
                    Thread.sleep(3000);
                    flush();
                }
                catch(Throwable e) {}
            }
        }

        void flush() throws IOException
        {
            for(Iterator i=buffTable.keySet().iterator(); i.hasNext(); )
            {
                FileOutputStream out = (FileOutputStream)i.next();
                Buffer aBuff = (Buffer) buffTable.get(out);
                synchronized(out)
                {
                    out.write(aBuff.buff, 0, aBuff.counter);
                    out.write(("*** flushed - "+aBuff.counter +
                               "\n").getBytes());
                }
                i.remove();
            }
        }

        protected void finalize() throws Throwable
        {
            flush();
            super.finalize();
        }
    }
```

The example is fairly big and complicated and there are some reasons for this. First is that you will need to have a buffer for each instance of the FileOutputStream but aspects is as default singleton. The problem you get if you instead change the aspect so it is “pertarget” is that the AspectJ compiler has to change in the FileOutputStream and you don’t have the source for

that file. And even if it could change it is it risky to change something that you don't have the code for. The solution is to have a list of all the buffers and keep the aspect as singleton. The second reason is that the flushing is normally taken care of by the application and must now be done in the aspect. This is done by starting up a thread that flushes the buffer every three second. It will also be flushed if it is explicitly closed.

## Pooling

Pooling is most used to speed up database connections by not connect to the database every time a query is executed. This can save a lot of time if the database is used in many places like it is in many web applications.

```
package aspects;

import java.sql.*;
import java.util.*;

public aspect Pooling
{

    private static Stack pool = new Stack();

    pointcut poolGet(): call(static
                        Connection DriverManager.getConnection(..));
    pointcut poolPut(): call(void Connection.close());

    Connection around() throws SQLException: poolGet()
    {
        synchronized(pool)
        {
            if(pool.empty())
            {
                return proceed();
            }
            return (Connection)pool.pop();
        }
    }

    void around(): poolPut()
    {
        Connection conn = (Connection)thisJoinPoint.getTarget();
        pool.push(conn);
    }
}
```

This example contains a stack of connections that override the getConnection and close methods. So every time the system asks for a connection will the aspect instead see if it already exist one. If it has one will that be returned and if not will it be created in the normal way. And when the system is finished will the connection instead of being disconnected be placed in the pool so it can be reused.

## Caching

The example below is from a news service there each user has it own list of filtered news stories. So the caching must be divided so each user has its own cached list. The problem with caching is that it is hard to make a domain independent solution that can be used for crosscutting the whole system. But it can still be used to make a clean and transparent performance improvement.

```
package aspects;

import java.sql.*;
import java.util.*;

public aspect ReadCache
{

    private static Map cache = new HashMap();

    pointcut read(String user):
    call(Collection StoriesDb.retrieve(String)) && args(user);

    pointcut dirtyUser(String user):
    call(* StoriesDb.savePreferences(String, ..)) && args(user);

    pointcut dirtyAll():
    call(* StoriesDb.saveStory(..));

    Collection around(String user) throws SQLException: read(user)
    {
        Collection res = (Collection)cache.get(user);
        if(null == res)
        {
            res = proceed();
            cache.put(user, res);
        }
        return res;
    }

    after(String user): dirtyUser(user)
    {
        cache.remove(user);
    }

    after(): dirtyAll()
    {
        cache.clear();
    }
}
```

The aspects keep an item in the cache list per user. The list for a user will be inserted into the list the first time the user tries to get something. Then if he/she changes his preferences so the contents of the list may have changed will it the deleted and reloaded the next time that user tries to access a story again. The cache will be deleted for all users if a new story is saved.

# Coping with Change

It is not unusual that the environment around the system change. And aspect can be used to cope this.

## *New Logging*

This example is used in Servlets. The problem here is that you want to control the build in logging to instead of writing to a file write to the console. This can be done by changing all calls to the log method to instead write it to the console. The drawback is that it can be a lot of places that need to be changed. The aspect can instead pick all calls to the log method and output the message.

```
package aspects;

import javax.servlet.*;

public aspect NewLogging
{
    void around(String message) :
    (
        call(void GenericServlet.log(String))
        ||
        call(void ServletContext.log(String))
    )
    && args(message)
    {
        System.out.println(message);
    }

    void around(String message, Throwable ex) :
    (
        call(void GenericServlet.log(String, Throwable))
        ||
        call(void ServletContext.log(String, Throwable))
    )
    && args(message, ex)
    {
        System.out.println(message);
        ex.printStackTrace(System.out);
    }
}
```

The logging has two log methods that must be pinked, one with only a message and a second with a message and an exception.

## *New Pooling*

The pooling example described before has a problem. I can't handle bad connections. There is many reasons way a connection can't be used any more like timeouts, if the database is restated etc. The solution for this is to check the connection before the Pooling aspects handles them. This is done with the keyword "dominate" that tells the AspectJ compiler that the new aspect will the executed before the Pooling aspect.

```
package aspects;

import java.sql.*;
import java.util.*;

public aspect ConnectionChecking dominates Pooling
{
```

```
    Connection around() throws SQLException: call(static
                          Connection DriverManager.getConnection(..))
    {
        Connection conn;
        do
        {
            conn = proceed();
        }
        while(bad(conn));

        return conn;
    }

    private boolean bad(Connection conn)
    {
        try
        {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT 2+2");

            if(rs.next())
            {
                rs.getString(1);
            }
            rs.close();
            stmt.close();
        }
        catch(SQLException e)
        {
            return true;
        }
        return false;
    }
}
```

It will go through all the connections and test them with a dummy query and if they are bad will they be deleted from the pool.

# Conclusion

Aspects can be a very powerful tool if it is used in the right ways. Some examples have been described here in this report but I think we will see more examples of implementations and new places where AOP can be used in the future. It much depends on how spread the techniques will be among software developers.

The biggest strength with AOP that I see is that it can in an easy way divide the business logic code from other types of code. This makes the Object Oriented design easier to read and understand. The Object can be more specialized in what they are expected to do and the aspects can take care of the rest.

I think we will see much more of AOP in the future in different areas, especially when the bigger companies start supporting it such as Microsoft, Sun, and Borland etc. They have the resources to build good development environments and design tools. And when more developer's starts to use it will it becomes a more mature development method.

# References

1. AspectJ version 1.1a1, http://www.aspectj.org

2. Ivan Kiselev, Aspect-Oriented Programming with AspectJ, 2001, Sams, ISBN: 0-672-32410-5