

# A Comparative Journey Into 3 AOP Implementations with C#.NET

Julie Ann Sakuda  
University of Hawaii  
jsakuda@hawaii.edu

December 13, 2004

## Abstract

Aspect-oriented programming is rearing its head in this object-oriented world. AOP has good intentions, but is a new paradigm really necessary? Some say yes, and it that it is time for change. It is time to change for the goodness of modularity. The focus of this paper is to introduce three AOP implementations that use C#.NET and to compare them to see differences and similarities. Using AOP a simple logging example, which is a common modularity buster, will become fully modularized. This journey into AOP is just an introduction into what AOP is really capable of attaining.

## 1 Introduction

Object-oriented programming (OOP) has made itself THE programming standard with its hierarchical organization. Although widely used, many believe there is room for improvement. OOP's goal is modularity. However, in a number of cases the modularity is easily destroyed. Hence, the belief that full modularity can be achieved through a new design, an aspect-oriented design. While still far from being standard, AOP is slowly evolving through many different implementations. It may very well be the next standard.

## 2 Aspect-Oriented Programming

Aspect-oriented programming is a programming paradigm designed to improve the modularity of a program. Its goal is to separate the core concerns from the crosscutting concerns. A core concern is like the business logic of a

program (ie. credits bank accounts). While a crosscutting concern is something that needs to be done but is not part of the business logic (ie. logging these transactions). AOP focuses on taking the crosscutting concerns and moving their actual code out and away from the core concern code. The program knows when to execute these aspects or pieces code dealing with crosscutting concerns, through pointcuts in the program. When a pointcut is reached, the code at that point is advised with the code (advice) that is associated with that pointcut.

## 2.1 Type-level Aspects vs. Instance-Level Aspects

There are two types of aspects that are used in AOP. It is important to know the difference between these two types of aspects for functionality reasons. Type-level aspects, also known as statically weaved aspects, are weaved into the code at compile time. Once these aspects are weaved into the code they cannot be removed. They are executed on every instance. Instance-level aspects, better known as dynamically weaved aspects, are aspects that are handled at runtime. Decisions on whether or not an aspect should be executed takes place when the program is run. This allows aspects to be removed as needed depending on situations at runtime. While instance-level aspects are very powerful and open up new doors, most AOP frameworks do not or do not yet support this type of aspect. Further readings on instance-level aspects can be found in the Further Readings section.

## 3 .NET

.NET is a development platform made by Microsoft. .NET supports a number of languages, not only C# which is what is mentioned in the title of the paper. C#.NET is just one of the languages. Others include VB.NET (Visual Basic) and J#.NET, which is Microsoft's java variant (it is not the same as Sun Java). C#.NET is strictly an OOP language. While there have been papers on how .NET can be AOP enabled by using things like COM and meta-tags, it is very complicated and difficult to understand. This paper will not be touching upon this AOP style. Refer to the Further Readings section for a link on using COM to emulate AOP. This paper is strictly focusing on third-party AOP add-ons to .NET.

## 4 .NET AOP Implementations

Note that all of these implementations are still under development so features at the time of this paper may change and many more may be added.

### 4.1 EOS

EOS is an extension to the C# language. It heavily resembles AspectJ, which is by far the one of the most popular AOP implementations for Java. EOS, like AspectJ, has its own compiler. EOS's website states that this implementation does the following three things:

- 1) It generalizes aspect instantiation and advice weaving model to eliminate the need for the work-arounds that are unavoidable today when aspects are used to express certain crosscutting concerns.
- 2) It generalizes the join point model.
- 3) It aims to eliminate the distinction between class and aspect constructs in favor of a single conceptual building block that combines the expressive capabilities of current classes and aspects, significantly improving conceptual integrity and uniformity in language design.

Like AspectJ, EOS supports type-level aspects. The aspects are declared in their own .cs file and they are declared "aspect" (normal classes are public class, the aspect file would have public aspect). This is very similar to AspectJ and the syntax is very close to that of AspectJ. However, unlike AspectJ, EOS also supports instance-level aspects. This is done by adding the keyword "instancelevel" in front of the aspect declaration. For more information the creators of EOS have written some excellent papers on instance-level aspects and EOS. See Further Readings for these links.

### 4.2 Rapier-LOOM.NET

Rapier-Loom.NET is a dynamic weaver for AOP in .NET. It is part of another project called Loom.NET. The difference is that Rapier-Loom.NET is a dynamic weaver and Loom.NET is a static weaver. The focus here will be on the dynamic weaver, Rapier-Loom.NET. Rapier-Loom.NET is language independent, which means it will work with any language that is part of the .NET family. Both the aspects and the target code can be written in any .NET language. Since Rapier-Loom.NET is a dynamic weaver, all weaving is done at runtime. Rapier-Loom.NET is one of the easier implementations

because it requires no new syntax and no external compiler or tools are needed. This is done by having Rapier-Loom.NET realized as a class library. This implementation is pure .NET with the use of metatags.

### 4.3 AOPDotNetAddIn

The AOPDotNetAddIn is a small Visual Studio add-in that implements some of the AOP specifications. It does not require learning a new language. The aspects and the target code can be written in any of the .NET languages. It does not have its own compiler. All of the weaving that is to take place is specified in an XML file. When the add-in is installed it adds two things to Visual Studio:

- 1) Under “Add New Item” there is an option for an Aspect. Using this will create a new file for an aspect and it will add an aspectDescriptor.xml to the project for the weaving specifications.
- 2) Under the Build menu it adds the option to “Weave and Run”. This is what is used instead of the normal build. This weaves the code according to the aspectDescriptor.xml and runs the program.

The library for this implementation is provided in a dll file. A reference to the AspectLib.dll file enables the use of the AOP library for this implementation.

### 4.4 Other

There are a handful of other known implementations of AOP for .NET. One of the implementations not mentioned above is Aspect#. There have been significant changes from version 1 to version 2, which includes the getting rid of having to use an XML file to specify where to weave in aspects. Another implementation is NAop which seems promising but lack of documentation led it to being excluded from this paper. A third implementation not listed above is AspectDNG. This one is interesting because it allows code to be written in different .NET languages (the class can be written in C# and the aspect code in VB).

## 5 Logging Example Without AOP

One of the classic examples to demonstrate AOP is logging. The logging example presented here is converted into C# from the Java logging example

in the paper "I Want My AOP" (see References). Note that all code will not be shown in this paper. See the References for links to the complete code.

Logger Interface (Logger.cs)

```
public interface Logger
{
    void log(string message);
}
```

CreditLogger Class (CreditLogger.cs)

```
public class CreditLogger : Logger
{
    public void log(string message)
    {
        Console.WriteLine(message);
    }
}
```

CreditCardProcessor Class (CreditCardProcessor.cs)

```
public class CreditCardProcessor
{
    Logger logger = new CreditLogger();

    public void debit(CreditCard card, Money amount)
    {
        //Before transaction
        this.logger.log("Starting
CreditCardProcessor.debit(CreditCard,Money)" + "Card: "
+ card.CardNumber + " Amount: " + amount.Amount);

        //Debiting Logic goes here

        //After transaction
        this.logger.log("Completing
CreditCardProcessor.debit(CreditCard,Money)" + "Card: "
+ card.CardNumber + " Amount: " + amount.Amount);
    }

    public void credit(CreditCard card, Money amount)
```

```

    {
        //Before transaction
        this.logger.log("Starting
        CreditCardProcessor.credit(CreditCard,Money)" + "Card: "
        + card.CardNumber + " Amount: " + amount.Amount);

        //Crediting Logic goes here

        //After transaction
        this.logger.log("Completing
        CreditCardProcessor.credit(CreditCard,Money)" + "Card: "
        + card.CardNumber + " Amount: " + amount.Amount);
    }
}

```

The example shown above is a non-AOP implementation of logging. Modularity here is violated because the CreditCardProcessor class should just have the debiting and crediting logic. Instead there must be an instance of the Logging class inside of this class which violates modularity. True modularity should separate these two things completely without losing functionality. Using AOP it will be shown this modularity is attainable.

## 6 Logging With AOP

The following are examples of the logging example shown in Section V. These examples will not be using the Logger interface and the CreditLogger class because it will be the job of AOP to handle the logging. Also, these examples will be using a modified CreditCardProcessor class. It will not contain any instances of a Logger class. The modified code is as follows:

```

CreditCardProcessor Class Revised (CreditCardProcessor.cs)
public class CreditCardProcessor
{
    public void debit(CreditCard card, Money amount)
    {
        //Debiting Logic goes here
    }

    public void credit(CreditCard card, Money amount)

```

```

    {
        //Crediting Logic goes here
        Console.WriteLine("Credit Method");
    }
}

```

Note that the code above is exactly the same as the Section V code without the class instance variable of a Logger and the logger.log statements are gone. A Console.WriteLine has been added in the credit method just to show that this method is actually executed. This can now be considered a proper module. For the remainder of the logging example only implementation for logging the credit will be shown. The code to handle the debit is very similar.

## 6.1 Logging Example with EOS

The following are examples of the logging example shown in Section V. These examples will not be using the Logger interface and the CreditLogger class because it will be the job of AOP to handle the logging. Also, these examples will be using a modified CreditCardProcessor class. It will not contain any instances of a Logger class. The modified code is as follows:

```

LoggingAspect EOS Aspect (LoggingAspect.cs)
public aspect LoggingAspect
{
    pointcut CreditExecution():
        execution(public void any.credit(CreditCard,Money));

    before(): CreditExecution() {
        CreditCard card = (CreditCard)(thisJoinPoint.getArgs()[0]);
        Console.WriteLine("Begin credit of " + card.CardNumber
            + " (" + card.Owner + ")");
    }

    after(): CreditExecution() {
        CreditCard card = (CreditCard)(thisJoinPoint.getArgs()[0]);
        Console.WriteLine("Finishing credit of " + card.CardNumber
            + " (" + card.Owner + ")");
    }
}

```

```
    }  
}
```

First, a pointcut called `CreditExecution` is declared. This pointcut will come into play when the method “`public void any.credit(CreditCard,Money)`” is executed (noted by the “`execution`” in front). The “`any`” keyword is a wildcard that acts like “`*`”. The asterisk cannot be used here because C# uses that symbol for pointers. So the execution statement basically says: Execute advice when any class with a public void `credit` method that takes in the parameters `CreditCard` and `Money` is executed. Just a note that the keyword `any` can be used for other parts other than just the class.

After the pointcut is declared the advice that is to be executed at that pointcut is written. For the pointcut `CreditExecution` there are two pieces of advice. One is declared after the `before()` and the second after the `after()`. This means that the code after the `before()` will be executed before the method is executed and the code in the `after()` will execute after the method has finished its execution. Basically these two pieces of code do the same thing, except one prints out that it is occurring before the `credit` and the other after.

In the advice code there is a `thisJoinPoint.getArgs()[0]` statement. EOS allows the advice to access the parameters being passed to the actual method. The parameters are accessed through an array. They are ordered from left to right. So depending on the array index any of the parameters can be retrieved. It will need to be cast to the type it is supposed to be. After that the parameter can be used just like it could be used in the method it was passed to.

To compile and execute the code a batch file is used. The easiest thing to do is modify a batch file from one of the EOS examples. The batch file compiles the code and creates an executable. The output after running the executable for the complete logging example is:

```
Begin credit of 1234567 (Jane Doe)  
Credit Method  
Finishing credit of 1234567 (Jane Doe)
```

From the output it is visible that the `before()` advice took place before the `credit` method executed, then the `credit` method executed, and lastly the `after()` advice took place after the `credit` method.

## 6.2 Logging Example with Rapier-LOOM.NET

With Rapier-Loom.NET in order to weave in an aspect the method must be called via an interface or it must be a virtual method. This example will use an interface to take care of this requirement. The interface is very short. Its code is as follows:

```
ICreditCardProcessor interface (ICreditCardProcessor.cs)
public interface ICreditCardProcessor
{
    void credit(CreditCard card, Money amount);
    void debit(CreditCard card, Money amount);
}
```

The CreditCardProcessor class will now have to implement this interface or else weaving will not be possible. The code is the same except for one line, the line where it says what interfaces the class implements. The declaration for implementing the ICreditCardProcessor interface is as follows:

```
public class CreditCardProcessor:ICreditCardProcessor
```

Now that the CreditCardProcessor class can have aspects weaved into it, it is time to write the aspect. Like the EOS example only the advice on the credit example will be shown here. The code for the aspect is as follows:

```
LoggerAspect Rapier-Loom.NET Aspect (LoggerAspect.cs)
using System;
using Loom;
using Loom.ConnectionPoint;

public class LoggerAspect:Aspect
{
    [Include("credit")]
    [Call(Invoke.Before)]
    public void beforeCredit(object[] args)
    {
        CreditCard card = (CreditCard) args[0];
        Console.WriteLine("Begin credit of " + card.CardNumber
            + " (" + card.Owner + ")");
    }
}
```

```

    [Include("credit")]
    [Call(Invoke.After)]
    public void afterCredit(object[] args)
    {
        CreditCard card = (CreditCard)args[0];
        Console.WriteLine("Finishing credit of " + card.CardNumber
            + " (" + card.Owner + ")");
    }
}

```

In Rapier-Loom.NET all aspects must extend off of a class called Aspect. This implementation of AOP provides their own library in a .dll file, which is where the Aspect class can be found. In order to make aspects using Rapier-Loom.NET a reference to RapierLoom.dll must be made.

In the example above there are the two pieces of advice, the one to be executed before the credit method and after the credit method. The [Include("credit")] means that this piece of code will be executed on a method named credit. The next line [Call(Invoke.Before)] tells it when execute this piece of advice. This one in particular will execute before the method executes.

In both beforeCredit and afterCredit there is a parameter, which is an array of objects. In the aspects it is possible to access any of the parameters being passed to the method that the advice is attached to. The parameters are put in an object array and they are in the order they were declared. So by using index numbers on the args array any parameter can be retrieved. In the example above the first parameter is a CreditCard object. After casting it back to a CreditCard it is used to print out information about the number of that CreditCard and the owner of the card.

One interesting thing to note about Rapier-Loom.NET is that since deals with dynamic weaving one must instantiate the aspect. Then one needs to use the Loom.Weaver.CreateInstance() method to instantiate the target class. If the "new" operator is used no weaving will occur. CreateInstance method can take in optional arguments. It uses these arguments to help create the instance of that object by picking a constructor that best matches the parameters given. Once that is done, the instance it can be used like any other variable. An example of this dynamic weaving where one must instantiate both the aspect and the target class is as follows:

Main part of program (Control.cs)

```

class Control:CreditCardProcessor
{
    [STAThread]
    static void Main(string[] args)
    {
        CreditCard card = new CreditCard("Jane Doe", 1234567);
        Money transaction = new Money(45.00);

        LoggerAspect aspect = new LoggerAspect();
        ICreditCardProcessor C = (ICreditCardProcessor
            Loom.Weaver.CreateInstance(typeof
                (CreditCardProcessor), null, aspect));

        C.credit(card, transaction);
    }
}

```

In the Control.cs shown above we see the CreateInstance statement. It is of type CreditCardProcessor, it is not matching up any parameters (there is only a default constructor), and it is using the LoggerAspect. It is possible to specify many parameters and many aspects, therefore widening the things that can be done. The output for this example is as follows:

```

Begin credit of 1234567 (Jane Doe)
Credit Method
Finishing credit of 1234567 (Jane Doe)

```

From the output it is visible that the Invoke.Before and the Invoke.After were executed at their proper times. Notice that this is the same output as the EOS example above.

### 6.3 Logging Example with AOPDotNetAddIn

Please note that the following example does compile properly but it does not run properly. The aspects do not seem to execute. It is believed to be very close to what should work properly. The author has been contacted. If anyone has a fix please email the author of this paper and credit will be given.

The only things that had to be written for this example was the aspect and the aspectDescriptor.xml file. After making a reference in Visual Studio to the AspectLib.dll file the aspect code is as follows:

```

LoggingAspect with AOPDotNetAddIn (LoggingAspect.cs)
public class LoggingAspect:Aspect
{
    public void logBeforeCredit(CreditCard card, Money amount)
    {
        Console.WriteLine("Starting: Crediting " +
            amount.Amount.ToString() + "to "
            + card.CardNumber.ToString() + " (" +
            card.Owner + ")");
    }
}

```

What this aspect does is take in the arguments of the method being advised. It uses the arguments in the WriteLine statement to display the card number and owner. This is the same as the other two previous examples. The code on when it should be executed is in the following aspectDescriptor.xml file:

```

Weaving Specification file (aspectDescriptor.xml)
<?xml version="1.0" encoding="utf-8" ?>
<aspects>
    <aspect name="LoggingAspect"
        code="LoggingAspect.cs" weave="true">
        <advices>
            <advice type="before">
                <method name="logBeforeCredit" />
                <pointcut>
                    <execution>
                        <method name="credit"
                            class="CreditCardProcessor"
                            access="*" return="*" />
                    </execution>
                </pointcut>
            </advice>
        </advices>
    </aspect>
</aspects>

```

This file lists the aspects inside of the aspects block. An aspect is declared with an aspect block. The name of the aspect is specified, where the code

for it is, and whether or not it should be weaved into the target code. In this case the aspect is called `LoggingAspect` and its code is in `LoggingAspect.cs`. And it should be weaved in. If it is set to `weave=false` then the aspect will not be executed. Inside of the aspect block there are advices. A single advice is declared with an advice block. The advice type is whether the aspect should be executed before, after, or around. In this case it is a before advice. The name of the method containing the advice is `logBeforeCredit`. Next is the pointcut block that specifies where the aspect should be weaved in. The execution block (this can also be call block instead) says that the pointcut is at the credit method in the class `CreditCardProcessor`. It also says that the access can be of any type (private, public, etc) and that the method can return anything (int, float, etc). Since this example has been tested and does not seem to work properly there is no output to be shown. If it were fully functional the output should be identical to that of the EOS and Rapier-LOOM.NET examples. The author claims that a GUI for writing the XML file is being developed. Hopefully, this will fix any errors present in the file shown above.

## 7 Benchmarking & Comparisons

This section provides comparisons of the AOP implementations using a simple test. It also compares the difference in the amount of code needing to be written.

### 7.1 Execution Test

Overhead in a large scale program is always an issue at hand. Product consumers do not like to wait in today's fast paced world. So what all this intercepting method calls and whatnot with AOP cost? Using the two working logging examples done in EOS and Loom.NET a simple test was conducted. The credit method in each example was invoked 100,000,000 times in a row. The examples remain the same except it does not print anything to the screen. Printing to the screen is time consuming and therefore the results would be greatly altered. All other statements in the program have been left untouched. It only prints the time to the screen at the very beginning and at the very end.

Test System:

- Sony Vaio PCG-K13
- Pentium 4 Northwood 2.8GHz (FSB 533Mhz)

- 1GB RAM
- Windows XP Professional SP2

Implementation	Average (seconds)
Non-AOP	less than 1
EOS(Compile-time)	12
EOS(Runtime(Aspect modifier))*	9.5
EOS(Runtime(Advice modifier))*	9
EOS(Runtime(Mixed))*	11
Rapier-Loom.NET(Runtime)	7

Table 1: Results of 100,000,000 Invocations

\* These implementations are instance level, but do not execute advice.

The results above are the averages of the test being run on each implementation 6 times. From the test it is apparent that non-AOP code is much faster. What is surprising is the time for the EOS compile-time. Since the code is weaved in at compile-time one would expect it be much faster at runtime. However, this is not the case. It seems as though Rapier-Loom.NET took less time even though its weaving is done purely at runtime.

The other 3 tests using EOS are various tests that use instance-level weaving. In all 3 of the tests the advice is not executed. Seeing as the times ranged from 9 to 11 seconds without the advice, it is safe to assume that it would take longer if the advice was being executed. This means it would probably take either the same amount of time or perhaps even more time than the EOS compile-time.

## 7.2 Code Size

This section only focuses on the samples presented in section 6.  $\LaTeX$  has inserted the table on the top of the next page. Please see the table.

## 8 Conclusions and Recommendations

From the section on Code Size it is clear that no AOP implementation actually cut down on code. This could be because the examples used in this paper are very simple so AOP ended up adding more lines. However, just

Implementation	Additions	Lines Added
EOS	LoggingAspect	7
Rapier-Loom.NET	Interface, LoggerAspect & modify main() (minus 2 lines, add 3)	7 & 4 meta-tags
AopDotNetAddIn	LoggingAspect & XML file	4 & XML file

Table 2: Code Comparison

because they added around the same number of lines doesn't mean that all AOP implementations are created equal.

Learning even the simplest things with all of these implementations was difficult because of lack of documentation. The implementation EOS was used primarily because AspectJ had been experimented with prior to this paper. Even though EOS lacks a great deal of documentation, AspectJ has an entire API and many articles. Since AspectJ and EOS are basically new languages, documentation is needed. For the most part the compile-time portion is exactly same, so EOS is probably the way to go because one can always use AspectJ documentation. As far as Rapier-Loom.NET, there was a small API and a reference manual but explanations were not very detailed. After a little experimenting with the meta-tags it becomes easier to use and understand. Lastly, the AopDotNetAddIn is probably one of the harder ones. Not only does it not have very much documentation, it is a great pain to write the XML weaving file. The walkthrough that explains how to write one of these files does not explain each XML attribute individually. It is very easy to get confused as to what is supposed to be filled in. Also, since the XML file's structure gets very long even for short examples it gets very hard to read. Hopefully, with the release of the GUI that writes the XML for users AopDotNetAddIn will become easier to use.

To sum it up, for an implementation with a lot of documentation use EOS. AspectJ documentation is very helpful if one gets stuck. However, even though EOS supports instance-level aspects, Rapier-Loom.NET is the way to go. Rapier-Loom.NET not only seems to be faster, but it is easier to understand than the documentless EOS instance-level. The examples that were timed in section 7 using EOS instance-level aspects do not work because of difficulties with EOS, and not because the intention was for it to not execute the advice. Rapier-Loom.NET is also recommended for those that don't want to learn a whole new language. It is strictly .NET with a reference to a weaving library. With EOS since it uses things like 'aspect' which are

not in the C# language, it cannot be compiled under the popular Visual Studio.NET IDE like Rapier-Loom.NET can. As for AopDotNetAddIn, it makes itself at home in Visual Studio 2003.NET. However, the XML files are a pain in itself so the ease of use is easily destroyed.

Remember that at the time of this paper AOP is still practically unheard of by many. The implementations here are still evolving and will become more advanced. The fate of AOP is still unknown, it may take-off in the future or it may not. While showing it has potential, AOP is a long way from being standard. Until AOP starts to conform to one standard, it will be difficult to understand. AOP is definitely worth keeping up with to see how it evolves over time. While many are true believers in OOP and say AOP is not needed, does one think that OOP was greeted immediately with open arms?

## 9 Further Readings & References

### 9.1 Further Readings

- [1] Rajan, H., and Sullivan, K., *Eos: Instance-Level Aspects for Integrated System Design*. (<http://www.cs.virginia.edu/eos/papers/eseconfse.pdf>).
- [2] Rajan, H., and Sullivan, K., *Need for Instance Level Aspects with Rich Pointcut Language*. (<http://www.cs.virginia.edu/eos/papers/splat.pdf>).
- [3] Shukla, D., Fell, S., and Sells, C., *Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse*. (<http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/>).

### 9.2 References

- [1] AOPDotNetAddIn. ([http://www.geocities.com/m\\_mesalem/aop.html](http://www.geocities.com/m_mesalem/aop.html)).
- [2] *Aspect-oriented programming*. ([http://en.wikipedia.org/wiki/Aspect-oriented\\_programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming)).
- [3] Aspect#. (<http://aspectsharp.sourceforge.net/>).
- [4] AspectDNG. (<http://sourceforge.net/projects/aspectdng/>).
- [5] AspectJ. (<http://www.eclipse.org/AspectJ>).
- [6] EOS. (<http://www.cs.virginia.edu/eos/>).

- [7] Laddad, Ramnivas, *I want my AOP!*  
(<http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html>).
- [8] Rapier-Loom.NET. (<http://www.dcl.hpi.uni-potsdam.de/research/loom/>).
- [9] Mesalem, Mohammad, *An add-in for Aspect Oriented Programming in Visual Studio.NET.*  
(<http://www.thecodeproject.com/macro/AspectOrientedVSNet.asp>).
- [10] NAop. (<http://sourceforge.net/projects/aopnet/>).