*MS Project Proposal*

# Buffalo: An Aspect Oriented Programming Framework for C#

**Wei Liao**

*Committee Chair:* Prof. James E. Heliotis

*Reader:* Prof. Fereydoun Kazemian

*Observer:* Prof. Matthew Fluet

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

October 22, 2012

## Abstract

Aspect Oriented Programming (AOP) is a paradigm that let programmers isolate and separate cross-cutting concerns from the basis of their program. The concept has not been widely adapted by modern languages, support in toolings such as Integrated Development Environment (IDE) is also rare. In this project we will design and implement a framework called Buffalo that provides AOP functionality for C# via IL code weaving, and integrate it with the Visual Studio IDE build system.

# 1 Introduction

Object Oriented Programming (OOP) languages have given programmers a lot of freedom in expressing themselves in Object Oriented Design. However, they are still lacking in some areas when it comes to particular software design decision such as cross-cutting concern [1].

In this project, we will try to solve this type of problem by designing and implementing a framework called "Buffalo" for the C# platform. We will show that by using Buffalo programmers can separate those concerns from the core of the program, and ultimately be more productive.

In section 2 we will explain the background of the problem, show some examples where the current programming paradigms are not efficient. In section 3, we will explore the existing works and show what had been done. In section 5, we will explain what we propose to do with an overview of the architecture of Buffalo, what we want the end result to be, and how to evaluate it. A tentative roadmap will be given in section 6.

# 2 Background

In this section we will explain more about the cross-cutting problem and how AOP can be used to help.

## 2.1 The Problem

Procedural Languages such as C achieve modularity by grouping codes into subroutines, so that they can be reused. Whereas OOP languages, such as JAVA or C# go one step further, by allowing programmers to abstract real world objects into properties and behaviors. Both paradigms give programmers the ability to make their codes cleaner and more reusable.

While OOP languages offer data abstraction and encapsulation in the form of classes and objects, the usage of declared instances of all these classes could still be scattered throughout different modules of the program. Over time, these tangled cross-cutting relations [1] can become difficult and expensive to maintain. One of the often cited example of such concerns is exception handling: the ability for programs to handle errors or terminate gracefully.

To handle an exception in our code, we usually use the follow try..catch block:

```
1 public double Divide(double num1, double num2) {
2     try {
3         return num1 / num2;
4     } catch (Exception e) {
5         //log exception to file, etc
6         Utility.LogToFile(e);
7     }
8 }
```

Listing 1: try..catch pattern

The code snippet in Listing 1 illustrated a few key points. If *num2* is 0, the execution will fail, causing an exception to be thrown, and when that happens, execution control is transferred to line 6, where the exception is logged to a file.

Imagine that if you have 1,000 functions in your program that can potentially throw exceptions, you would have to apply the try..catch block on all of them. And note that the functionality of actually logging the exception is nicely encapsulated in the *Utility* object, but it does not change the fact that the code is still repetitive, because *Utility.LogToFile(e)* still have to be called in 1,000 different places in the source code. Since an OOP program most likely consists of different modules, this repetitive pattern will cut through and appears in all the them.

What if you need to fine tune this try..catch block to catch a specific exception such as the *DivideByZeroException* so your program can act accordingly, or instead of using the *Utility* object you want to use a different object to handle the actual logging? In the worst case, you would have to make the change to all 1,000 of your functions.

The question is how to prevent those cross-cutting concerns from loitering your program. Worst yet, it is not uncommon a program might consist of something like in Listing 2, where exception is not handled at all.

```
1 public double Divide(double num1, double num2) {
2     return num1 / num2;
3 }
```

Listing 2: Unhandle exception

Aspect Oriented Programming techniques [1] can be used to cleanly solve such problems.

## 2.2 Aspect-Oriented Programming

The Aspect Oriented Programming paradigm was first discussed in 1997 [1]. When talking about AOP, the following concepts are worth noting:

*concern* - the repetitive code that cross-cuts into different modules of the program. Usually the code does not conviently fit into the dominant paradigm of design.

*aspect* - the piece of isolated code that can be used to solve the issue of a particluar concern.

*join point* - These are the locations throughout the program where the concern is leaked into; they are also where the aspect will be applied to solve the concern.

The idea of AOP is fairly simple. We have code that is duplicated all over the place, making it difficult to maintain. We need to isolate that duplication into a separate single unit. Then we inject that unit of code into all relevant places in a program either at runtime or at compile time, so that the programmers does not have to do it manually in the source code.

In other words, AOP is about injecting code into a program. This is especially handy when programmers don't have access to the original source code.

Besides exception handling, AOP is commonly used in tracing, profiling, security, etc.

# 3 Related Work

Most modern programming languages already display some AOP-like properties, but full native support is rare. Delphi Prism is one of them, where the weaving of aspect code happens at compile time [2]. AOP can be implemented in a variety of ways. However, just like other programming paradigms, it is most effective and beneficial when it is implemented by the compiler. That makes it a first class citizen like other properties of a language.

## 3.1 Compiler Support

Gregor Kiczales started and led a Xerox PARC team that developed an implementation of AOP for the JAVA platform called AspectJ[1]. AspectJ is an extension to the JAVA compiler. It is a language in and of itself, with its own specific syntax and usages, and even its own compiler. It produces JAVA VM compatible binaries. Some people hope that AspectJ will eventually be merged with the JAVA compiler instead of being just an extension [3]. Still, AspectJ integrates nicely with JAVA, especially when used with its own plugin AJDT in the Eclipse IDE.

A typical AspectJ aspect looks like Listing 3.

```
1 public aspect MyAspectJ {
2    public int Sorter.Count() {
3       //do something here
4    }
5    pointcut doSomething() : call (* * (..));
6    before() : doSomething() {
7       //do something before calling the actual function
8    }
9 }
```

Listing 3: sample AspectJ code

It provides the *aspect* keyword to denote a piece of code as the advice code. It is similar to the *class* keyword in JAVA. Suppose we have a class named *Sorter*, and we want to add a new method to it but we don't have access to the source code. With AspectJ, we can introduce a brand new method *Count* to it, as shown in line 2-4. The pointcut *doSomething* does a match to find all matching functions regardless of access privilege, names or parameters, which means pretty much every function in the program. Line 6-8 means before the matched functions are executed, the block of code in line 7 is executed first. This effectively creates a hook into every single function in a program, allowing programmers to inject custom code into it.

It is important to note that the purpose of AOP is to solve cross-cutting concerns, and not in patching up of code as Listing 3 might have alluded to.

Behind the scene, AspectJ pieces everything together using bytecode weaving. After JAVA compiles the source, AspectJ takes the classes and aspects in bytecode form and weaves them together, producing new .class files that can be loaded onto the virtual machine.

Despite Eclipse' claim that AspectJ is very easy to learn, one of the disadvantages of AspectJ, as Listing 3 shows, is that the syntax is somewhat different from a normal JAVA program, making

the learning curve much deeper. But I agree that once get used to it, AspectJ can be a really powerful tool for programmers [4].

## 3.2 Framework Support

AspectJ is one of the few compilers that does AOP, that is good news for JAVA programmers. For the vast majority of other programming languages out there, AOP support is provided via frameworks. This is especially true on the C# platform [5], where Microsoft has indicated that they will not be integrating AOP support into the C# compiler anytime soon [6]. There was a project called Eos developed at the University of Virginia, which was an aspect oriented extension for C#, but it had been discontinued years ago [7, 8].

There are a number of frameworks available for the C# platform. They come in various flavors and implementation with different techniques. Each has its advantages and disadvantage.

One of the most common implementations involves the usage of a proxy, where the client does not interact with the objects directly, rather everything goes through the proxy.

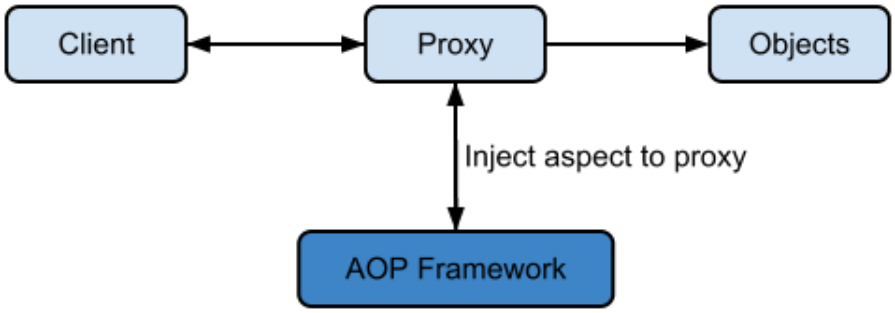


Figure 1: AOP framework using proxy

Using the proxy provides an opportunity for code injection. The advantage of this approach is relative ease of implementation. It is also limiting since in order for the proxy to work, both the proxy and the target object must implement the same interface, therefore the aspect injection point can only occur in the exposed functions for the interface. As this requires reflection at runtime to generate the proxy, it also adds overhead, so performance might not be as good as other approaches.

Another approach is similar to AspectJ, where bytecode weaving is used, but without the extra complexity of a new syntax and language. The commercial product PostSharp [9] is such an example, where aspect weaving happens post compilation by rewriting the MSIL instruction set. PostSharp uses .NET languages with attribute used for advice codes. The advantage is ease of use, as developers already familiar with C# will have little to no learning curve at all. And since aspects are woven in the assembly, the runtime incurs no overhead for reflection, and therefore performance is not compromised. The disadvantage is that, since it has to work with MSIL instruction set, it is very low level and therefore the most difficult to implement.

Some frameworks use static weaving [10], where the source files are pre-processed to include all the relevant aspect code. Then the C# compiler takes over and does the compilation normally. This has the advantage of post compilation weaving but not the complexity of working with MSIL

instruction set. On the other hand, a parser generator has to be developed to efficiently parse the source files.

# 4 Hypothesis

The OOP paradigm cannot efficiently solve the cross-cutting problem. However even though the C# compiler will not support the AOP paradigm, we can still archieve the same goal by performing the separation of concerns via Buffalo.

We can further make the developer's life easier by hooking up Buffalo with the Microsoft's build system in Visual Studio to perform automatic aspect weaving. With zero, developers can just focus on creating the aspects to deal with the problems.

# 5 Approach and Methodology

In this section I will give an overview of the architecture of Buffalo, the tools and approaches I will be using to implement it.

## 5.1 Architecture Overview

The approach we plan to take is to perform compile time weaving. Figure 2 shows an overview of how Buffalo will fit in the overall C# compilation process.

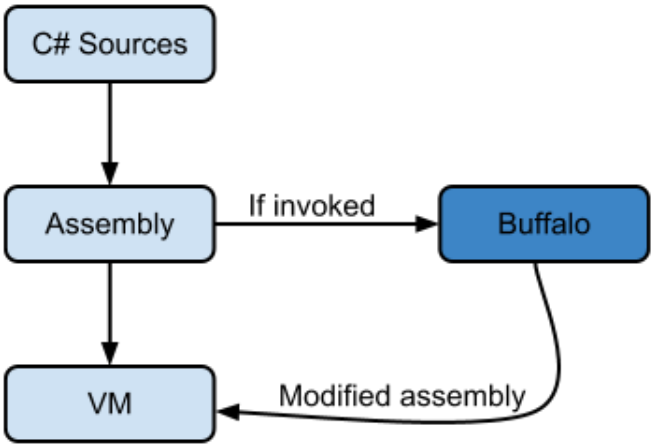


Figure 2: Buffalo model overview

## 5.2 Feature Implementation

For this project, I plan to work on weavering of several aspects. Specifically they are the various interception points of a function:

*Before* - Before the execution of a function.

*After* - After the execution of a function, right before it returns.

*Success* - The function has successfully executed without error.

*Exception* - The function throws an exception during execution.

*Around* - Swap out the target function with one provided by the developer, while preserving the option to call into the original function.

The first four aspects can be nicely group together using a try..catch..finally. As shown in Listing 4, this is how a function would be modified in MSIL when a developer wants to intercep any of those execution points. The try..catch..finally is wrapped around the target function with calls to various custom code supplied by the developer.

```
1  public class SomeFunction {
2      try {
3          Before();
4          //original method body...
5          Success();
6      }catch(Exception e){
7          Exception(e);
8      }finally{
9          After();
10     }
11 }
```

Listing 4: Buffalo aspects

The Around aspect is more involved as we cannot simply replace the target function with something else. The replacement must be able to call into the original function, along with the right number of parameters. For this project I plan to get the basic of Around aspect working, then in the future iteration to make it more flexible.

Contrast from AspectJ, where it has separate getter and setter syntax specifically for Java's properties. In C#, properties are automatically turned into methods when compiled into MSIL. Since Buffalo will be operating on MSIL, all the aspects will apply to both properties and methods.

## 5.3 Compile Time Weaving

As shown in Figure 2, the C# compiler compiles the source files into an assembly, this assembly is then fed into Buffalo. Buffalo takes apart the assembly using reflection to find all the defined *aspects* and possible *pointcuts*. The defined aspects are simply of attribute type implementing the IAspect interface, an example is shown in Listing 5. Buffalo then weave them together at the right places by rewriting the MSIL instruction set, and finally produces a new assembly.

Once C# finishes the compilation, the control is transferred to Buffalo. This is archieved by creating a hook into the MSBuild system. If Buffalo detects any *aspects* defined in the input assembly, and that the *aspects* are being applied, then weaving will take place, otherwise the whole process will simply be ignored.

## 5.4 Intended Usage

From the Visual Studio IDE, developer can use Buffalo by creating a custom C# attribue class; extending a Buffalo interface IAspect (subject to change), for example:

```
1 public class CatchException : System.Attribute, IAspect {
2     public override void Exception(Exception e) {
3         Utility.LogToFile(e);
4     }
5 }
```

Listing 5: Buffalo aspect

The cross-cutting concern is now cleanly separated into a custom attribute *CatchException*. It will be treated as an aspect by Buffalo, any change made to it will be propagated to all the annontated functions. To use this aspect, simply treat it like a regular attribute and apply it to any method, class or assembly.

```
1 [CatchException]
2 public double Divide(double num1, double num2) {
3     return num1 / num2;
4 }
```

Listing 6: applying Buffalo aspect

As the code snippets in Listing 6 shows, by applying this attribute to a function, the repetitive try..catch block is no longer necessary. The target code is much shorter and cleaner.

The real benefit will be evident when an aspect is applied to a large number of functions, or asemblies. For example, if we want every function in every class to be able to catch exception we can apply the attribute to the assembly:

```
1  [assembly:CatchException]
2  namespace MyAssembly {
3      public class MyClass {
4          public double Divide(double num1, double num2) {
5              return num1 / num2;
6          }
7          //other functions...
8      }
9      //other classes...
10 }
```

Listing 7: applying Buffalo aspect on an assembly

By applying the attribute on the assembly level, a single line of code, will effectively allow every function to have exception handling capability. This will be a huge developer productivity gain.

## 5.5 Platform, Languages and Tools

This project will be developed using C# and Visual Studio 2010, on Windows 7. For IL rewriting several options are available. One is to use the Reflection.Emit library that comes with the .NET Framework. However all research indicates that this library represents only a subset of the MSIL instruction set.

Another option is to use the Profiler API by Microsoft, but this API is intended as a debugging feature, therefore is unsuitable for production environment.

While we can opt to invest the time on learning and developing a custom MSIL rewriter, we are afraid that that in itself is a bigger project than Buffalo [11]. Mono is an open source implementation of the C# compiler, Cecil is a project within Mono that provides MSIL rewriting. Cecil provides low level API for working with MSIL. Preliminary evaluation of the tool suggests it is pretty feature complete and flexible enough to satisfy our requirement. However, documentation for Cecil is next to non-existent, so the learning curve is expected to be deep.

The following utility tools will also be heavily used during development: ILSpy, ILDASM and PEVerify. ILSpy is an open source application that dis-assembles C# assembly to show IL instructions. ILDASM does the same thing but comes from Microsoft .NET Framework. PEVerify is a Microsoft Windows SDK tool, it will be used to ensure that the modified assembly we will produce is valid, as this is not verified by Cecil.

## 5.6 Measurement

To evalulate Buffalo, we will show that with Buffalo, code duplication can be reduced. The cross-cutting concern will be separated into single unit of code where it will be easy to maintain. We will use the *Call Hierarchy* feature of Visual Studio to show how many calls are issued before and after IL rewrite with Buffalo.

The line count of code a developer has to write will also be reduced considerably. As a result the code will be cleaner and easier to look at. This can also be translated directly into an estimation of development cost savings. We will compare line counts before and after IL code rewrite by Buffalo using the *Code Analysis* function provided by Visual Studio.

# 6 Roadmap

Table 1 shows the tentative schedule for the major phases of this project.

| Date | Action | Status |
|---|---|---|
| 07/07/2012 | Pre-Proposal | Accepted |
| 07/09/2012 | Proposal | In-progress |
| 07/20/2012 | Begin development of Buffalo | In-progress |
| 11/15/2012 | Finish development, start testing and analysis | - |
| 11/25/2012 | Finish report | - |
| 12/2012 | Defense | - |

Table 1: Timeline

# References


[1] et al. Gregor Kiczales, John Lamping. Aspect-Oriented Programming. In *Proc. of European Conference on Object-Oriented Programming (ECOOP), Finland*, 1997.

[2] Cirrus - The Oxygene Language Wiki. `http://wiki.oxygenelanguage.com/en/Cirrus`. Accessed: 07/16/2012.

[3] The Eclipse Foundation. AspectJ Frequently Asked Questions. `http://www.eclipse.org/aspectj/doc/released/faq.php`. Accessed: 07/19/2012.

[4] Ramnivas Ladda. *Aspectj in Action: Enterprise AOP with Spring Applications*. Manning Publications, second edition, October 2009.

[5] M. Devi Prasad and B. D. Chaudhary. AOP Support for C#. In *Proc. of Second International Conference on Aspect-Oriented Software Development*, pages 49–53, 2003.

[6] Federico Biancuzzi Chromatic. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media, first edition, April 2009.

[7] Kevin Sullivan Hridesh Rajan. Eos. `http://www.cs.virginia.edu/~eos`. Accessed: 07/20/2012.

[8] Kevin Sullivan Hridesh Rajan. Aspect Language Features for Concern Coverage Profiling. In *Proc. of Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005)*, 2005.

[9] Gael Fraiteur. User-friendly aspects with compile-time imperative semantics in .NET: an overview of PostSharp. In *International Conference on Aspect-Oriented Software Development*, 2008.

[10] Howard Kim. AspectC#: An AOSD implementation for C#. Master's thesis, Trinity College Dublin, 2002.

[11] Serge Lidin. *Expert .NET 2.0 IL Assembler*. Apress, first edition, August 2006.