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

July 18, 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 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

High level object oriented programming 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 [3].

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 how 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, we will look at some of the areas that the current programming paradigm are not efficient at solving, and illustrate it by look at some examples. In section 3 we will explore the existing works and what have been done. In section 5 we explain what we propose to do and give an overview of the architecture of Buffalo, what we want the end result to be, and how to evaulate it. A tentative roadmap is given in section 6.

2 Background

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

2.1 The Problem

Procedural Language such as C achieves modularity by grouping codes into subroutines or functions, whereas Object Oriented Programming (OOP) languages such as JAVA or C# go one step further, and they allow programmers to abstract real world object into properties and behaviors. Both paradigms give programmers the ability to make their code cleaner and more reusable.

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

To handle exception in our code, in its most basic form we can have the follow try..catch block:

```
01. public void SomeFunction() {
02.     try {
03.         //this will fail
04.         var result = 1 / zero;
05.         //other operations...
06.     } catch (Exception e) {
07.         //log exception to file, etc
08.         Utility.LogToFile(e);
09.     }
10. }
```

The above code snippet illustrated a few points. First line 4 is going to throw an exception, and when that happens, execution control is transferred to line 8, where the exception is logged to a file. Imagine if you have 10,000 functions in your program that can potentially throw errors, you would have to apply the try..catch block on all of them. Note that the functionality of actually logging the exception is nicely encapsulated in the Utility object, but it does not change the fact that it is still repetitive in this case, because it will still have to be called in 10,000 different places in the source code. Since your program most likely will be consist of many modules, it will mean this repetitive pattern will cut through and appears in all your modules.

Imagine again, wif 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 10,000 of your functions.

The problem is how to isolate those cross-cutting concerns from loitering your program. Aspect Oriented Programming technique [1] can be used to cleanly separate such concerns.

2.2 Aspect-Oriented Programming

The Aspect Oriented Programming paradigm is first discussed in the 1997 paper [3]. When talk about AOP, the following concepts are worth noting.

concern - It is the repetitive code that cross-cut into different modules of the program. It is often code that does not conviently fit into the dominant paradigm of design.

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

join points - Locations through out the program where the concern might be cut into, it is also where the aspect will be applied.

From the above concept we should have a pretty clear idea that first, we have some cross-cutting code that's not easy to maintaint, so we can isolate them into a separate piece of code, then we can injects the aspect into all relevant places in a program.

3 Related Work

AOP can be implemented in a variety of ways, but like other programming paradigms, it is most effective and beneficial to programmers when it is supported on the compiler level, making it a first class citizen like other properties of a language. But as of now, very few languages provide native support, Delphi Prism 2010 is one of them, where the weaving of aspect code happens at compile time [1]. But most languages relies on compiler extension or framework to provide the support.

3.1 Compile Support

Gregor Kiczales from the 1997 paper [3] started and led the Xerox PARC team developed an implementation of AOP for the JAVA platform called AspectJ. AspectJ is a language in and of itself, with its own specific syntax and usages and even compiler, it produces JAVA VM compatible binary. It would have been nice if this is extended directly in the JAVA compiler, which some hopes

it will happen soon. But still, AspectJ integrated nicely with JAVA, especially when used with its own plugin in an Integrated Development Environment (IDE) like Eclipse.

AspectJ can introduce new features to or modify an existing code base, it provides the *aspect* keyword to denote a piece of code as the advice code, this is to be used like the *class* keyword in JAVA.

```
01. public aspect MyAspectJ {
02.     public int Sorter.Count() {
03.         //do something here
04.     }
05.     pointcut doSomething() : call (* * (..));
06.     before() : doSomething() {
07.         //do something before calling the actual function
08.     }
09. }
```

The above is a snippet of how an AspectJ aspect is defined, 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, in AspectJ we can introduce a brand new method to it, as line 2-4 shows. The pointcut doSomething() does a regular expression match to find all matching functions regardless of priviledge, names or parameters, then *before() : doSomething()* will execute the code starting in line 4 before each of those matching functions are executed.

This is just scratching the surface of what AspectJ can do. Behind the scene AspectJ does bytecode weaving, after JAVA compiles the source, it takes the classes and aspects in bytecode form and weaves them together, producing new .class files that can be loaded onto the virtual machine.

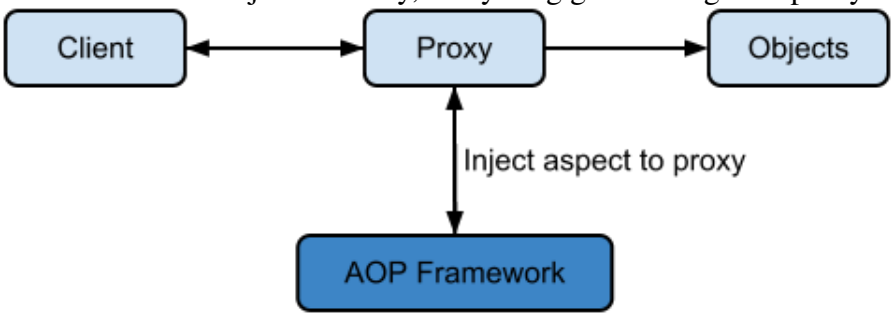
Despite Eclipse's claim that AspectJ is very easy to learn, one of the disadvantage of AspectJ is, as the above example shows, the syntax is somewhat different from a normal JAVA program, making the learning curve much deeper, but we agree that once get used to it AspectJ can be a really powerful tool for a programmer.

3.2 Framework Support

AspectJ is one of the few compiler that does AOP, for the vast programming languages out there, support is provided via some sort of frameworks. This is especially evident on the C# platform [6], as things are not that cosy with AOP. Ander Heijsburg is the Microsoft fellow that lead the team to develop the C# compiler, he has indicated that they will not be integrating support for AOP on the compiler anytime soon. Although their Patterns & Practice team release Policy Injection application block that offer some AOP capability.

There are a number of frameworks available for the C# platform, they come in various flavors and implemented with different technique. There are several approaches and each has its advantages and disadvantage.

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



The injection of code will happen only at the proxy. The advantage of this approach is relatively easy to implement, but it is also limiting in that 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 defined virtual functions [need double check on reference]. Since this require reflection at runtime, the performance is usually not as good.

Another approach is similar to AspectJ, where bytecode weaving is involved, but without the extra complexity of a new syntax and language. The most commercially successful implementation of this is PostSharp [2], where aspect weaving happens post compilation by rewriting MSIL instruction set, PostSharp use just the standard C# language, attribute is used as advice code, the advantage is ease of use, developer already familiar with C# will have no learning curve at all. And since aspect is woven in the assembly, the runtime has no overhead of reflection and therefore performance is good. Disadvantage being, since it has to work with MSIL instruction set, it is very low level and therefore difficult to implement.

Other approach involve static weaving [4], where the source files are pre-processed to include all the relevant aspect code, then just let the C# compile take over and does it work normally. This has the advantage of post compilation weaving but not the MSIL complexity. On the other hand, you do have to develop a parsar generator to efficiently parse the sourc files.

4 Hypothesis

The OOP paradigm cannot efficiently solve the cross-cutting problem, we believe those problems can be solved efficiently at the compiler level, and that is where AOP logically belongs, so that programmer doesn't even have to think about them.

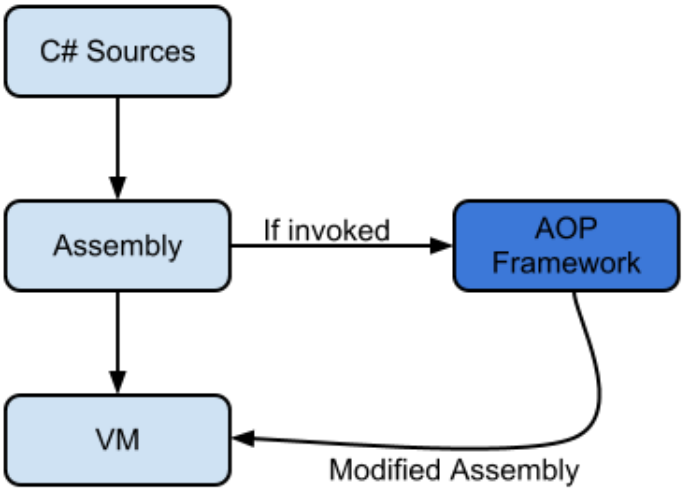
Even-though the C# compiler will not support the AOP paradigm, we can still archieve the same goal by doing the separation of concern for them via a framework, and further ease the developer's life by hooking the framework into the MSBuild system. So developers can focus on just creating the aspects to solve the problems.

5 Approach and Methodology

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

5.1 Architecture Overview

There are many approaches on how to implement an AOP framework. The approach I plan to take is to perform post compilation weaving. Figure 1 shows an overview of how Buffalo will fit in the overall C# compilation process.



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

```
01. public class CatchException : System.Attribute, IAspect() {
02.     public void OnException(Exception e) {
03.         Utility.LogToFile(e);
04.     }
05. }
```

Now CatchException will be treated as an aspect by Buffalo, to use it, simply use it like a regular attribute and annonate it to any method, class or assembly.

```
01. [CatchException]
02. public void SomeFunction() {
03.     var result = 1 / zero;
04.     //other operations...
05. }
```

As the above code snippets shows, the cross-cutting concerns is now cleanly separated into a cutom attribute called CatchException, any change made to the aspect will be propogated to all the

annontated functions, class or assembly. As a result. By applying this attribute to the functions, the repetitive try..block is no longer necessary, the target code is much shorter and cleaner.

The real benefit will be even more evident when an aspect is apply to a large number of functions, or asemblies, for ex, if I want every function within every class to be able to catch exception:

```
01. [CatchException]
02. namespace MyAssembly {
03.    public class MyClass {
04.       public void SomeFunction() {
05.          var result = 1 / zero;
06.          //other operations...
07.       }
08.       //other functions...
09.    }
10.    //other classes...
11. }
```

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

5.2 Compile Time Weaving

Within this method there are a few variants of compile time weaving, one is to preprocess the source files, to perform text transformation to weave all the aspects at the right locations of the source, then just let the compiler take over and compile as normal.

The other type is MSIL rewriting [5], as indicated by figure 1 , the C# compiler performs a compile on the source files to generate the assembly. This assembly is then fed into the Buffalo, which will proceed to rewrite the MSIL instruction calls to weave in the aspects.

There are obvious advantages and disadvantages to each method. [Give some examples of what those are?]

5.3 Platform, Languages and Tools

This project will be developed using C#, on the Windows 7 using Visual Studio 2010. When performing IL rewriting there are a few options available, one is to use the Reflection.Emit library that comes with the .NET Framework, however all my researches points to that this library represents only a subset of the MSIL instructions, the missing instruction might prove to be a problem later on.

Another option is to use the Profiler API by Microsoft, however this API is intended as a debugging feature, and therefore is unsuitable to be used in production environment.

While I can opt to invest my time on learning and rolling my own IL rewriter, a more practical option is probably to use a 3rd party library. Mono is an open source implementation of the C# compiler, Cecil is a project within the Mono project that provides IL rewriting. Preliminary

evaluation of the tool seems to be pretty feature complete and flexible enough to allow me to do what I need to do. However documentation for Cecil is next to non-existent, it works with low level MSIL, so I expect the learning curve is probably deep.

The following utility tools will also be heavily used during development: ILSpy, ILDASM and PEVerify. ILSpy is an open source application that dis-assemble assembly to show IL instructions. ILDASM does the same but comes from Microsofts .NET Framework. PEVerify is a Microsoft Window SDK tool, it will be used to ensure that the modified assembly produced is a correct assembly.

5.4 Measurement

To evalulate Buffalo, we will show that after using Buffalo, code duplication will 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 the before and after.

The code a developer has to write is considerable less in terms of number of lines, as a result the code will be cleaner and easier to look at. We will show the before and after using the *Code Analysis* available in Visual Studio.

6 Roadmap

The following table shows my tentative schedule for the major phases of the 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
09/10/2012	Finish development, start testing and analysis	-
09/27/2012	Finish report	-
10/10/2012	Defense	-

Table 1: Timeline

References

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

[2] 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.

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

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

[5] Aleksandr Mikunov. Rewrite MSIL Code on the Fly with the .NET Framework Proling API. *MSDN Magazine*, 2003.

[6] 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.