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 20, 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 [5].

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 at solving. 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 more about the cross-cutting problem and how AOP can be used to help.

2.1 The Problem

Procedural Language such as C achieves modularity by grouping codes into subroutines, thus those subroutines can be reused. Whereas 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 to maintain. 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, we usually have 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 if you have 1,000 functions in your program that can potentially throw errors, 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 consist 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 problem is how to prevent those cross-cutting concerns from loitering your program. Aspect Oriented Programming technique [5] can be used to cleanly separate such concerns.

2.2 Aspect-Oriented Programming

The Aspect Oriented Programming paradigm was first discussed in the 1997 paper [5]. When talking 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 - These are the locations through out the program where the concern is leaked into, it is also where the aspect will be applied to solve the concern.

The idea of AOP is fairly simple, we have some code that that are duplicated all over the place, making it difficult to maintain, so we need to isolate that duplicate code into a separate single unit of code, then we injects that unit of code into all relevant places in a program either at runtime or compile time, so programmers don't have to do it manually in their source code.

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

Beside exception handling, AOP is commonly used in tracing, profiling and 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 [1]. 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 embraced by the compiler, making it a first class citizen like other properties of a language.

3.1 Compile Support

Gregor Kiczales from the 1997 paper [5] started and led the Xerox PARC team developed an implementation of AOP for the JAVA platform called AspectJ. AspectJ is an extension to the JAVA compiler, but it is a language in and of itself, with its own specific syntax and usages and even compiler. It produces JAVA VM compatible binary. Some hopes that AspectJ will eventually be merged with the JAVA compiler instead of being an extendsion [3]. But still, AspectJ integrated nicely with JAVA, especially when used with its own plugin AJDT in the Eclipse IDE.

AspectJ can introduce new features to a program, as well as modify it. It provides the *aspect* keyword to denote a piece of code as the advice code, the usage is similar to how the *class* keyword is used in JAVA.

```
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 2: sample aspectj code

The example in Listing 2 showed how an aspect is defined in AspectJ. 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 *Count* to it, as line 2-4 shows. The pointcut *doSomething* does a match to find all matching functions regardless of access priviledge, names or parameters, which means pretty much every functions in the program. Line 6-8 means before executing those functions matched, execute the block of code in line 7 first. This effectively created a hook into every single function in a program, allowing programmers to inject custom code into it.

Behind the scene AspectJ pieces everything together by 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 [9].

3.2 Framework Support

AspectJ is one of the few compiler that does AOP, that is good news for JAVA programmer. For the vast programming languages out there, support is provided via frameworks. This is especially

true on the C# platform [11], where Microsoft has indicated that they will not be integrating support for AOP on the C# compiler anytime soon [2]. There was a project called Eos developed at the University of Virginia, it was the aspect oriented extension for C# compiler, but it had been discontinued years ago [6, 7].

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.

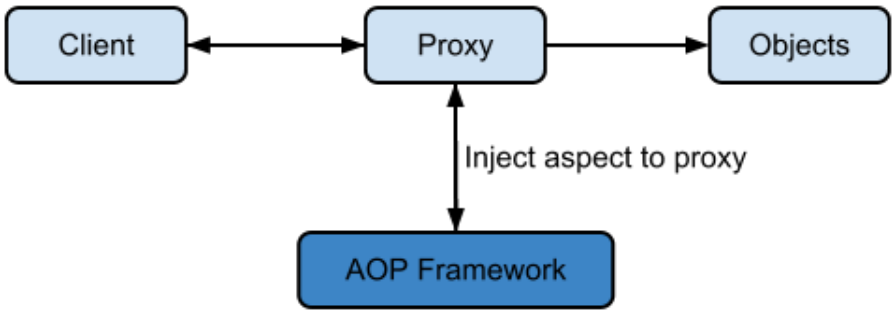

Figure 1: AOP framework using proxy

By using the proxy this provides an opportunity for the injection of code to it. The advantage of this approach is relatively ease of implementation, 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 functions in the interface. Since this require reflection at runtime to generate the proxy, it does create overhead, so performance might not be as good as other approaches.

Another approach is similar to AspectJ, where bytecode weaving is involved, but without the extra complexity of a new syntax and language. The commercial product PostSharp [4] is an example, 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 little or 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 the most difficult to implement.

Some frameworks use static weaving [8], where the source files are pre-processed to include all the relevant aspect code, then just let the C# compiler takes over and does the compilation normally. This has the advantage of post compilation weaving but not the MSIL complexity. On the other hand, parsar generator has to be developed to efficiently parse the source files.

4 Hypothesis

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

We can further make the developer's life easier by hooking Buffalo into the MSBuild system to perform automatic aspect weaving. With zero configuration, 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

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

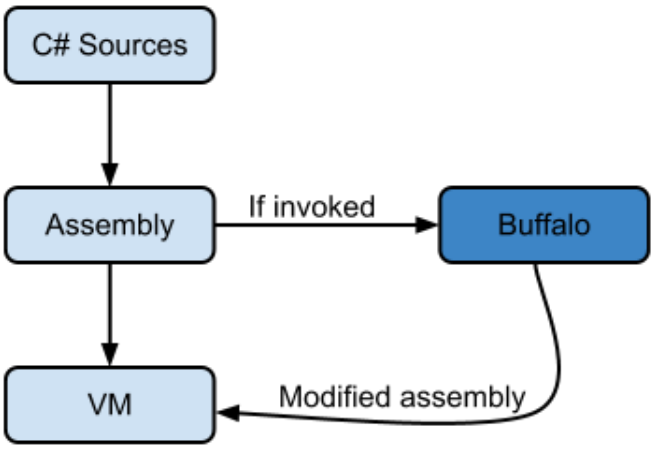

Figure 2: Buffalo model overview

5.2 Post Compilation Weaving

As Figure 2 shows, 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*, then weave them together into the right places by rewriting the MSIL instruction set, finally producing a new assembly.

Once C# finishes the compilation, the control is transfer to Buffalo, this step 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, the weaving will take place, otherwise the whole process will be simply ignored.

5.3 Intended Usage

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

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

Listing 3: Buffalo aspect

The cross-cutting concerns is now cleanly separated into a cutom attribute *CatchException*. It will be treated as an aspect by Buffalo, any change made to it will be propogated to all the annontated functions. To use this aspect, simply use 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 4: applying Buffalo aspect

As the code snippets in Listing 4 shows, by applying this attribute to the function, the repetitive try..block is no longer necessary, the target code is much shorter and cleaner.

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

```
1  [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 5: applying Buffalo aspect on an assembly

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

5.4 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 researches points to that this library represents only a subset of the MSIL instructions.

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

While we can opt to invest the time on learning and rolling a custom MSIL rewriter, we are afraid that in itself is a bigger project than Buffalo [10]. So a more practical option is to use a third party library. Mono is an open source implementation of the C# compiler, Cecil is a project within the Mono that provides MSIL rewriting. Preliminary evaluation of the tool seems to be pretty feature complete and flexible enough to satisfy our requirement. However documentation for Cecil is next to non-existent, as it works with low level MSIL, 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-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, as this is something that Cecil does not verify.

5.5 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 how many calls issued before and after.

The code a developer has to write is also considerable less in terms of number of lines, 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 saved. We will show the before and after using the *Code Analysis* available in Visual Studio.

6 Roadmap

Table 1 shows the 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] Federico Biancuzzi Chromatic. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media, first edition, April 2009.

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

[4] 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.

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

[6] Kevin Sullivan Hridesh Rajan.

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

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

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

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

[11] 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.