Buffalo: An Aspect Oriented Programming Framework for C#

by

Wei Liao

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Prof. James E. Heliotis

Department of Computer Science

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

February 2013

The project "Buffalo: An Aspect Oriented Programming Framework for C#" by Wei Liao has been examined and approved by the following Examination Committee:

Prof. James E. Heliotis
Professor
Project Committee Chair

Prof. Matthew Fluet
Associate Professor

Prof. Fereydoun Kazemian
Assistant Professor

Dedication

To Jackson and Evan

Acknowledgments

I am grateful for my adviser Prof. Heliotis, whose insightful advices, guidance and support from the beginning not only enabled me to complete the project on time, but also to a better understanding of the subject area.

I am also grateful for Prof. Fluet and Prof. Kazemian for their invaluable feed backs.

Last but not the least, I want to thank my wife Michelle, for all the support and encouragement during my years at school.

Abstract

Buffalo: An Aspect Oriented Programming Framework for C#

Wei Liao

Supervising Professor: Prof. James E. Heliotis

Aspect Oriented Programming (AOP) is a paradigm that let programmer isolate and separate crosscutting concerns from their programs. The concept has not been widely adopted by modern languages; support in tooling such as Integrated Development Environment (IDE) is also rare. In this project I designed and implemented Buffalo, an AOP framework to provide this capability for the .NET platform.

Buffalo performs Common Intermediate Language instruction set modification according to the aspects written by developer, with the help of the Mono Cecil library. Buffalo is .NET attribute based, which mean developers with existing .NET skills will have little or no learning curve to get started. Buffalo will help increase developer productivity in many area such as unhandled exception catching, tracing and logging, and many other areas.

Contents

List of Tables

List of Figures

Chapter 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, a framework called ”Buffalo” is designed and implemented to solve this type of problem on the .NET platform. Buffalo makes use of the .NET attribute system to weave aspect code to any targeted methods. The design and rationale of the framework is discussed in section 2. The implementation detail is given in section 3.

The result indicates that by using Buffalo, developers can separate cross-cutting concerns from the core of the program for easy maintenance, and ultimately be more productive. The analysis is discussed in section 4.

The report concludes in section 5 with the current project status. A set of planned future works is also discussed and what is learned from undergoing this project.

Buffalo comprises of around 1,200 line of source codes, which is fully included in Appendix A. A user manual is included in Appendix B, which contains some usage examples and instructions on how to integrate Buffalo with MS-Build.

Chapter 2

Design

2.1 Compiler Support

There are several broad approaches to implementing an AOP framework. The ideal approach would be to extend the compiler of the target language to provide built-in support, thus making AOP the first class citizen. However there are very few languages out there that take this approach, among the few are Delphi Prism [2] and AspectJ [3, 4].

Microsoft is currently in the ”wait and see” mode regarding support of AOP development in the C# compiler [5]. Alternative compiler such as Mono C# [6] is open source, so technically anyone can build AOP support into it. While that would have been a fun challenge, however that would have been a fairly big undertaking, and there is concern that the project might not be finished in the time frame wanted.

That leaves framework support as the other viable option. There are several implementation techniques to provide AOP capabilities [7, 8, 9] via framework.

2.2 Run-time Interception

Early on the implementation approaches were narrowed down to between Run-time Interception and Compile Time Weaving. As its name suggested, run-time interception operates while the program is in execution. It uses the proxy pattern where client communicate with the target object via a proxy, and aspects are injected to the proxy. This enables run-time behavior of the program to be modified. Figure 2.1 illustrates this process.

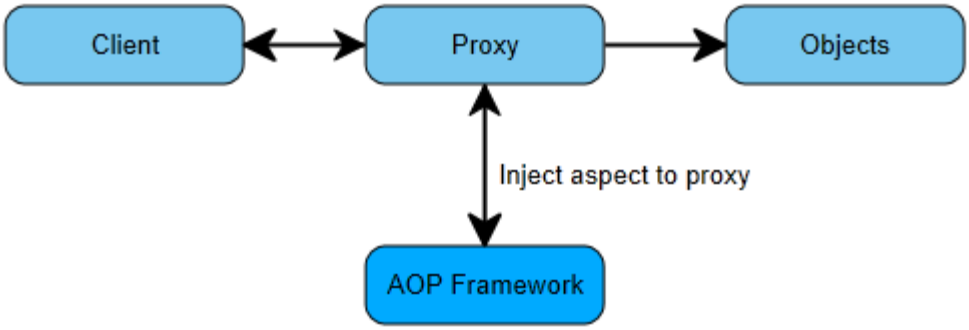

Figure 2.1: AOP Framework Using Proxy Pattern

New functionality can be added to the target object via the proxy. The disadvantage of this approach is that it involves the generation of proxy object at run-time. The run-time performance of the application will be impacted as the result. It is also restricting in that both target object and the proxy must implement a common interface for this to work, and that only virtual methods are exposed for interception.

From the end user's perspective, to use it the developer usually have to provide some type of mapping between the target object and the proxy via a configuration file so the actual proxy generation can occur. This approach although is easier to implement, but not as easy and friendly to use. Buffalo is not taking this approach mainly because one of the goal is to be flexible and simple to use.

2.3 Compile Time Weaving

The approach Buffalo takes is Compile Time Weaving. The idea is that after compilation of the source codes, the framework takes over and disassembles the assembly. Buffalo then weaves in the defined aspect code to all targeted methods. This approach is more difficult to implement as it involves modifying the underlying assembly by changing Common Intermediate Language (CIL) instructions [10]. But the advantage is that no run-time performance of proxy generation will be needed.

Since injection happens post-compilation, the whole process can be integrated into the MS-Build system to have the weaving invoked automatically if needed. This will further

reduce the steps needed from the developer.

Figure 2.2 shows an overview of the compilation process and where Buffalo will fit in.

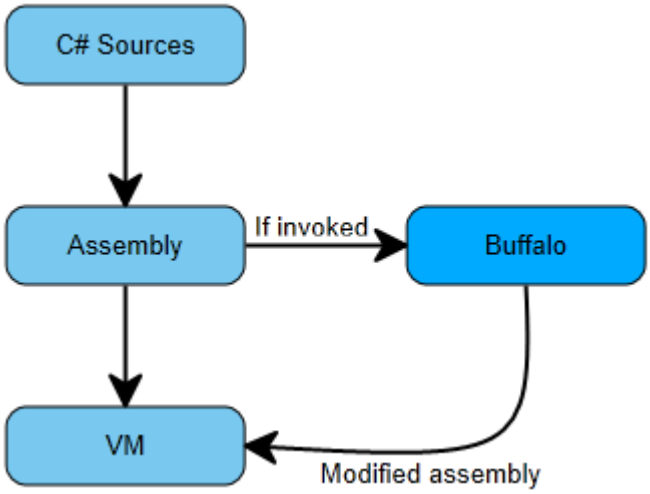

Figure 2.2: Buffalo Model

2.4 What is a Buffalo Aspect

When performing post compilation weaving, Buffalo has to be able to discover what aspect is applied to what method in an assembly. In order to achieve that the target assembly has to carry some identifying meta-data.

A given .NET assembly already carries a great deal of such meta-data for various purpose. .NET has the System.Attribute type that exists primarily for the purpose of inserting meta-data into the assembly during compilation. When the source code is compiled, it is converted into CIL [11] and put inside a portable executable (PE) file, with the meta-data generated by the compiler.

Buffalo takes advantage of this characteristic in two phases.

1. An aspect defined in Buffalo will be in the form of an attribute, by sub-classing System.Attribute. It can contains any valid .NET code. But specifically an aspect needs to override various predefined methods in order to do something useful. Figure 3.2 shows the relationship between various aspect types.

2. After compilation, the assembly will now contain the meta-data about the aspect. Buffalo can inspect the assembly for the information, and perform CIL code injection accordingly.

In other word, a Buffalo aspect is a .NET attribute in disguise.

2.5 MethodBoundaryAspect

What functionality does Buffalo support exactly? What type of weaving does it do? These were the initial questions that needs a definitive answers. For this inspiration were obtained from AspectJ [3] and PostSharp [8], specifically Buffalo will intercept the various point of an executing method. Those points are namely: before a method executes; after a method executes; whether or not the method executed successfully without error; or whether the method throws an exception any point during the execution. These various point of interception is group into the MethodBoundaryAspect.

MethodBoundaryAspect can be cleanly mapped to the try-catch-finally statements of the .NET languages. As far as the run-time is concern [12, 13], try-catch can be used liberally without serious performance degradation. For example, a simple method shown in Figure 2.1:

```
1 public void SomeFunction () {
2   //Perform some action...
3 }
```

Listing 2.1: Sample function

Can be transformed into something shown in Figure 2.2. That clearly captures the spirit of the MethodBoundaryAspect.

```
1 public void SomeFunction () {
2   try {
3      OnBefore();
4      //Perform some action
5      OnSuccess();
6   }
7   catch (Exception e) {
```

```
8       OnException(e);
9    }
10   finally {
11      OnAfter();
12   }
13}
```

Listing 2.2: Sample try-catch-finally

Transformed method in Figure 2.2 still does what the original method intends to do, only now at various point execution are being intercepted to provide more functionality.

2.6 MethodAroundAspect

Another type of aspect that Buffalo supports is the MethodAroundAspect. Rather than intercepting various execution points of a method, the method can be completely replaced by another method defined in an aspect, while preserving the option to call back into the original method if necessary.

At first glance MethodAroundAspect sounds straightforward to do, but it turns out to be much more involved than the MethodBoundaryAspect.

Since the option to call back into the original method is preserved, it is critical that under no circumstance should the original method be modified. If the method body instructions are simply overridden with that of the replacement, the call back to the original method will be meaningless since the method is now changed. The original method must be intact for the call back to happen.

To get around this obstacle, whenever Buffalo encounters the MethodAroundAspect applied to a method, it dynamically generates a replacement method in CIL with the same method signature as the original.

The body of this replacement method is also completely different than the original. It simply instantiate the aspect and makes call to the aspects Invoke() method, which is the actual aspect code that will be ran.

Inside the Invoke method, developer can make a call back to the original method via

a call to the Proceed() method. Then throughout the program, for any calls made to the original method, Buffalo would change them to call the replacement method instead. This is illustrated in figure 2.3.

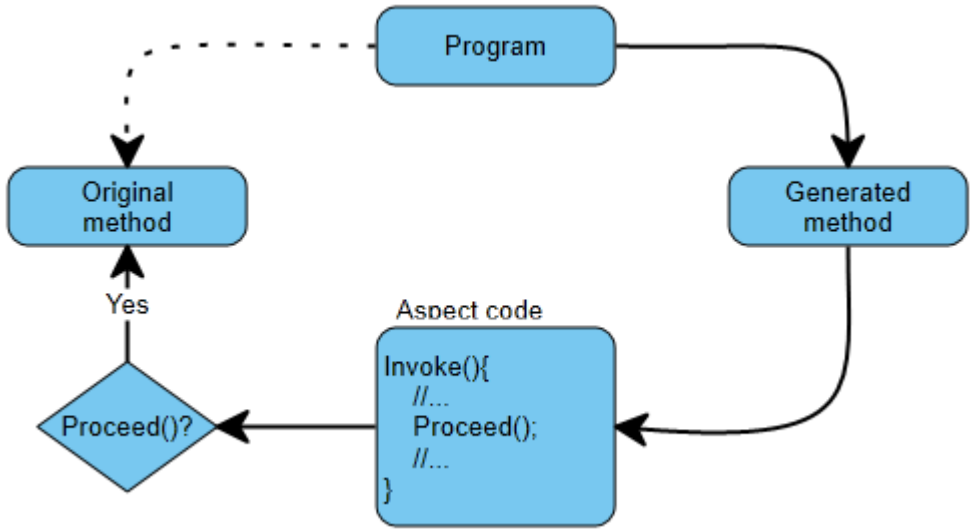

Figure 2.3: Method Around Aspect

The dotted line from Program to the Original method indicates that once the MethodAroundAspect is applied to it, from the perspective of CIL the program cannot directly access that method any more. Access to the original method now has to come from inside the aspect. Also note that the original method is not changed at any given time.

Chapter 3

Implementation

3.1 How to Apply an Aspect

Since an aspect is a .NET attribute, it can be used just like any other attribute. But code annotated with an aspect is special in that it can be understood only by Buffalo.

Normally an aspect can be applied in three level:

1. Method level - an individual method.
2. Class level - if applied to a class, all public methods including the public properties automatically get applied.
3. Assembly level - if applied to an assembly, #2 will apply but for all the public classes within the assembly.

An exception to the above rule is the MethodAroundAspect, where it can only be applied on a method level, as will be shown later on.

An aspect can also be excluded on any given level. If excluded the target and its nested children will be skipped during the weaving process.

No matter how the aspect is applied, ultimately it will result in a list of the methods that are annotated. This simply mean if the aspect is applied to a single method, that method is the only one that will get CIL modified. If the aspect is applied on the whole assembly, then all public methods will be CIL modified.

To get the list of the eligible methods for CIL modification, Buffalo attempts various checking according to figure 3.1 to see if it should include a given method.

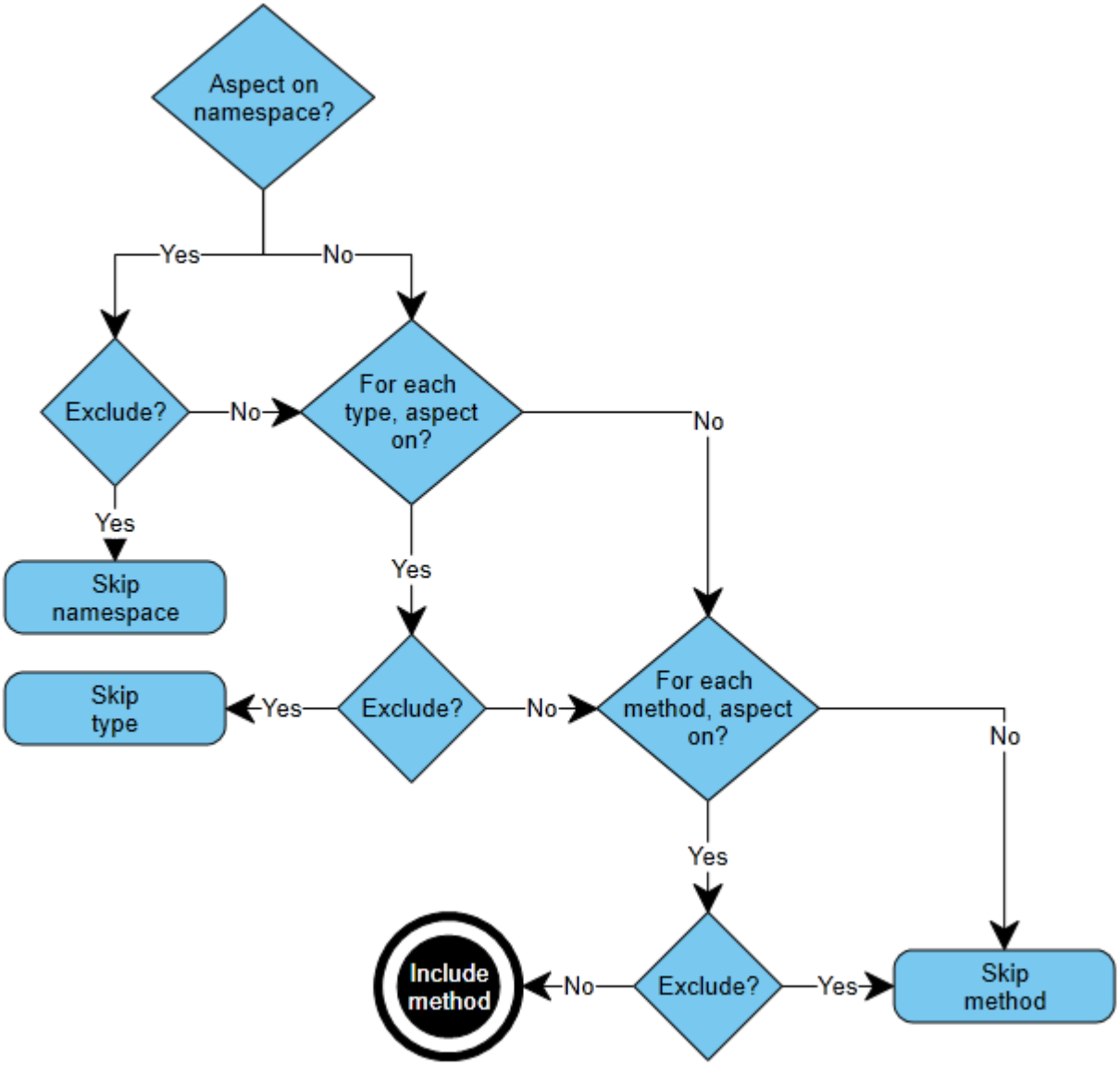

Figure 3.1: Logical Inclusion

If no aspect is applied on the assembly, that does not necessarily mean no aspect is applied anywhere, the aspect might still be applied on any given class or method.

The take away from the above diagram, is that Buffalo first checks if an aspect is applied to the target, then check if it is set to be excluded. At the end it will end up with a list of methods that should be CIL modified.

3.2 Aspect Interface

Figure 3.2 shows the relationship of various aspect types in Buffalo. This is used by Buffalo to identify aspects during reflection.

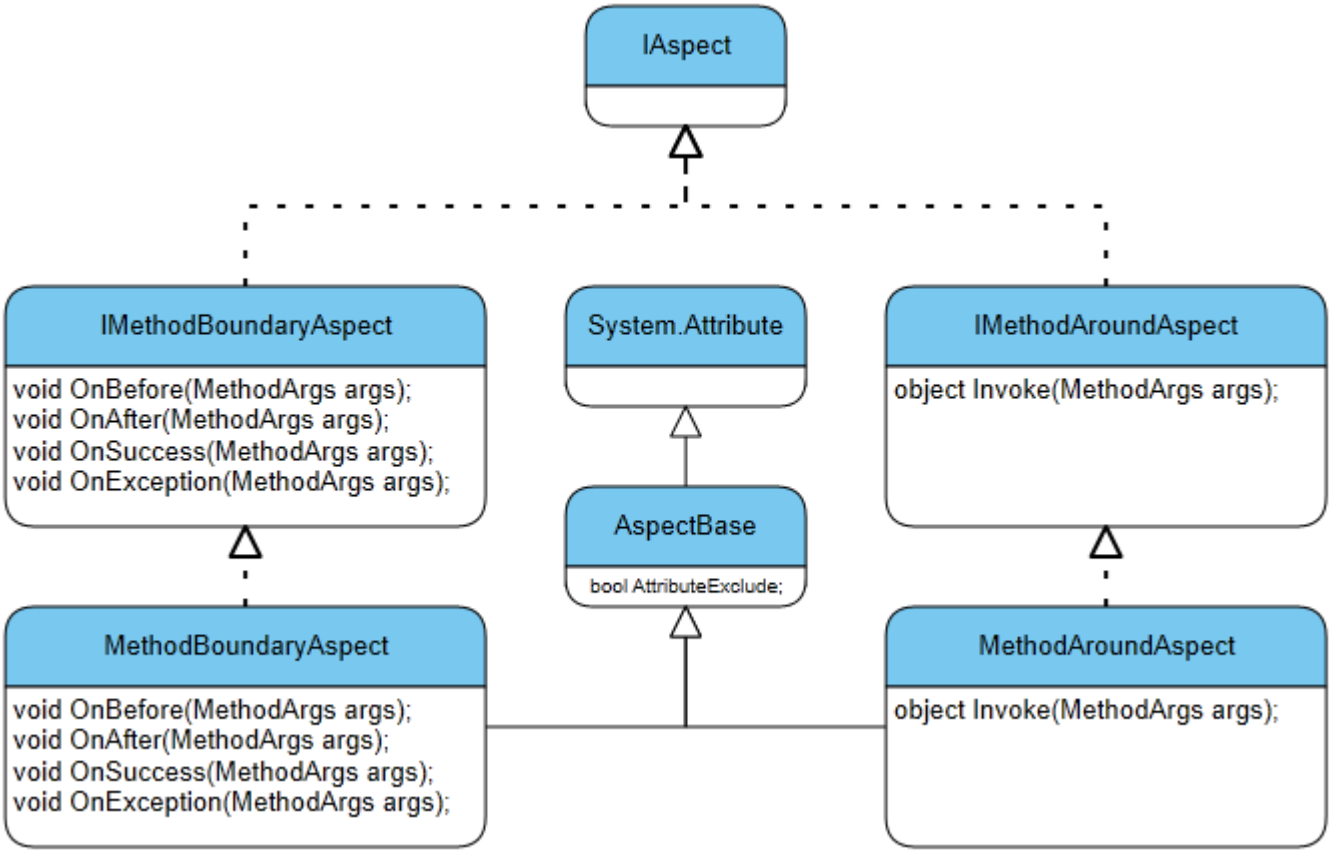

Figure 3.2: Aspect Inheritance

All aspects ultimately implements the IAspect interface, therefore it can be reasoned that for all the public types in an assembly, if it implements IAspect, then it must be an aspect itself.

All aspects have a property named AttributeExclude, if set to true then the annotated target will not be included in the weaving.

Buffalo supports more than one aspect applied at any given level. This will allow developers more flexibility while developing multiple aspects and applying them as needed.

Furthermore, by default, an aspect will be automatically excluded from applying to itself. This is implemented to prevent stack overflow in some cases. Although argument can be made that an aspect should be able to be applied to a different aspect, however that is not currently implemented in Buffalo.

3.3 MethodArgs

As mentioned above, when its all said and done, an aspect ultimately gets injected into each *individual* method. When developing an aspect, a developer can access various information about the target method, via the MethodArgs object passed in as parameter to the aspect. Currently the full method signature, the method name, return type and parameter list including parameter name, type and value are captured for each target method.

The parameter list capturing is especially of interest, it enables developer to peek inside the method that is executing at various point and inspect its parameter values. This will be useful in case such as exception handling, where it will be useful to actually see what the values were at the time of the exception.

3.4 Visual Studio Solution Structure

Originally Buffalo was implemented as one executable, that includes the various aspects and the program that initiates the weaving. It was later on separated into two assemblies. One is a class library that contains the actual implementation. Another is a command line executable that calls into the class library to perform the weaving, as shown in figure 3.3. This separation is necessary so developer can perform weaving from the command line or hook into MS-Build if necessary.

To actually write the aspect, developer only need to reference the class library which is much cleaner than referencing an executable.

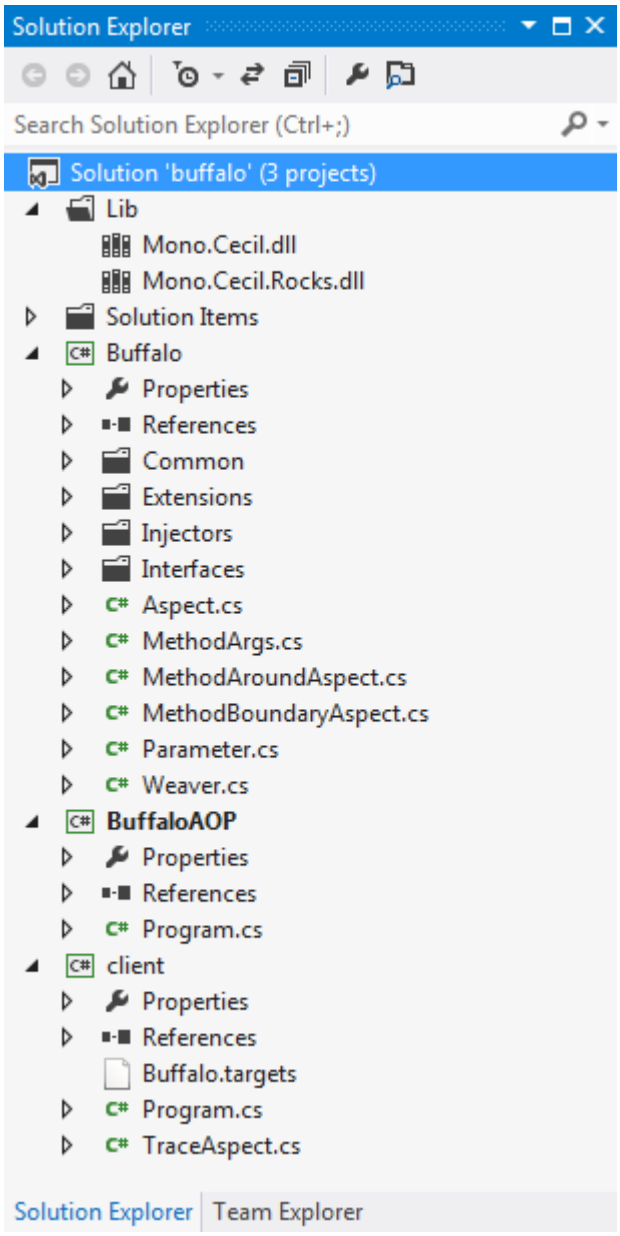

Figure 3.3: Solution Structure

The client project shown above is just a simple program included in the solution for testing. The full source code is included in Appendix A.

3.5 Implementation Overview

The overall implementation process can be illustrated as figure 3.4, where the first step of finding all eligible methods using the logical diagram mentioned in figure 3.1.

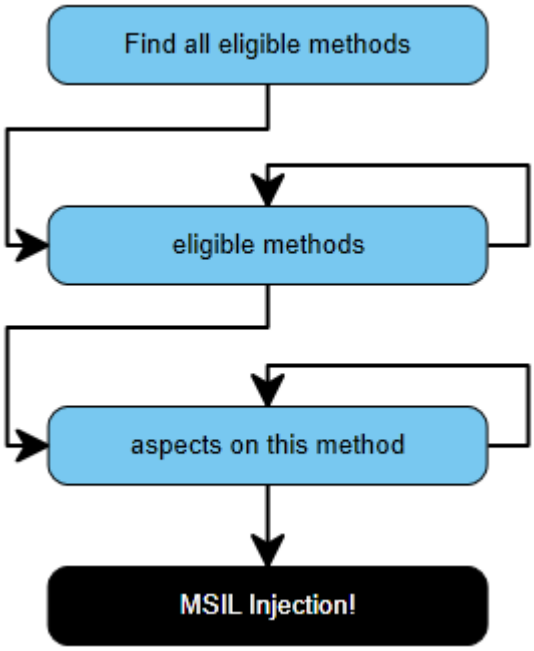

Figure 3.4: Implementation Overview

Then it loops through each eligible method, and for each aspect associated with the eligible method do the CIL injection. The actual injection phrase depends on the type of aspect.

3.6 MethodBoundaryAspect Implementation Detail

Each type of aspect has its own injector that implements the IInjectable interface. This interface contains only one method contract - Inject(..). It takes the list of eligible methods and inject the appropriate aspect to them.

MethodBoundaryAspect is pretty straightforward to implement. Take the following hello world example, wrapped in a try..catch..finally block as mentioned previously:

```
public void SayHello()
{
  try{
     Console.WriteLine(Hello World!);
  }catch(Exception ex){
  }finally{
  }
}
```

Listing 3.1: SayHello function

The generated CIL shown in figure 3.2. For ease of display it has been cleaned up a bit:

```
.try
{
  .try
  {
    IL_0002: Ldstr "Hello World!"
    IL_0007: call void [mscorlib]System.Console::WriteLine(string)
    IL_000e: leave.s IL0015
  }
  catch [mscorlib]System.Exception
  {
    IL_0010: stloc.0
    IL_0013: leave.s IL_0015
  }
  IL_0015: leave.s IL_001c
}
finally
{
  IL_001a: endfinally
}
IL_001c: ret
```

Listing 3.2: CIL generated for sample C# function

Figure 3.2 shows the standard emission of the CLR when it encounters the try-catch-finally statement. In CLR there is a concept of the protected region, where each region is associated with a handler. A try-catch-finally is actually encapsulated in two such regions: a catch and a finally. From here it can be easily figured out where to inject the various

boundary aspects, as shown in the following figure 3.5

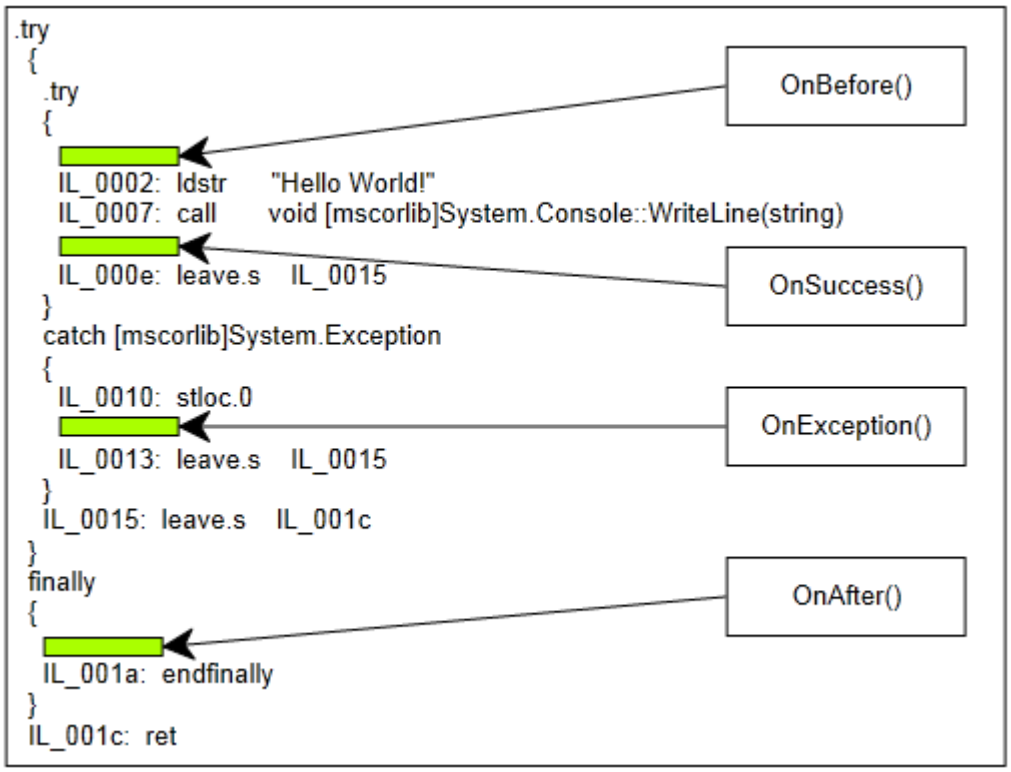

Figure 3.5: CIL Interception Points

The actual implementation is in provided in Listing A.9

3.7 MethodAroundAspect Implementation Detail

The Around aspect on the other hand is a much more complicated compared to the Boundary aspect.

MethodAroundAspect implements IMethodAroundAspect which has the following contract:

```
1 internal interface IMethodAroundAspect : IAspect
2 {
3       void Invoke(MethodArgs args)
4 }
```

Listing 3.3: IMethodAroundAspect Contract

From a developers perspective, when writing an aspect he can issue a Proceed() to signal a call back into the original method. The steps taken to implement MethodAroundAspect in CIL is roughly as follow:

1. Create a replacement for the annotated function with exactly the same method signature.
2. Create and store a variable pointing to the aspect.
3. Copy all parameters from original method to the newly created replacement function.
4. Create a variable to hold MethodArgs.
5. Issue a call to Invoke() from the replacement function, passing in the MethodArgs variable.
6. Handle the return value appropriately.
   (a) If original method returns non void type, put the return value back on the stack.
   (b) If original method returns void, we need to discard the return value from Invoke().
7. Handle Proceed() that might be issued from inside the Invoke().
   (a) Load all the parameters onto the stack.
   (b) Call back into the original method.
   (c) Handle the return value appropriately.
8. Modify all calls from original method to the replacement method.

As figure 2.3 shown, the actual calling of either the original or replacement method is abstracted away. This is also a testament of the saying in Software Engineering that ”anything can be resolved by another layer of abstraction”.

Another important distinction is that MethodAroundAspect currently can be applied only on the method level, and that it should be applied to one method only. This is by design because a replacement method might not be appropriate to replace more than one method. Especially if it is applied on the assembly level, all the public methods will be replaced by a single replacement method!

The actual implementation of MethodAroundAspect is in provided in Listing A.8

3.8 MethodArgs

When developing an aspect, a developer can access information about the annotated method. Details such as the name of the method; its signature and return type. Also the list of parameters that are being passed into the method including type, name and value.

To achieve the above MethodArgs is used. This is the object passed into each aspect. During the weaving, an instance of MethodArgs is injected into each method, with all properties assembled dynamically to capture the information of the current executing method.

Being able to capture some information about the annotated methods will be useful. For example, in case of a profiling aspect, those information about the method at the time it was access will be helpful. Being able to look at the parameter values in case of error will also be extremely useful in case of debugging.

An early implementation instantiated a distinct instance of MethodArgs for each boundary aspects. Later on as an optimization only one instance is instantiated at the beginning of the method body and that instance is used in all the boundary aspects for a target method.

An example of how to use MethodArgs is presented in the user manual.

Chapter 4

Analysis

The project hypothesized that by using Buffalo, programmers can separate the cross-cutting concerns from their applications quickly and easily. Since the concerns are encapsulated in a distinct unit of code, it also enable programmers to easily maintain the aspects and modify them as needed.

One of the analysis performed is to write an aspect to catch unhandled exceptions in test programs. The size of the test programs varies from comprising of 50 methods to 1,000 methods. Suppose that to manually implement the exception handling, a programmer will have to write on average 5 lines of code to catch the exception.

Programs	Lines (Traditions)	Lines (Buffalo)
50	250	0-1
500	2,500	0-1
1,000	5,000	0-1

Table 4.1: Line counts

If exception handling is implemented for every method by hand, more line of the same try-catch block of code will have to be written as the application adds more methods. The number of line of repetitive code would increase linearly.

Lines of code have a direct correlation to the cost of the development as it will take programmers more time. And this will also have a direct impact on application release schedule.

By using a framework like Buffalo, unhandled exception can be centralized in one aspect, and then simply apply it to every method by applying it on the assembly level.

As a result the source code is free from the repetitive try-catch-finally blocks. The line of code we have to write is one line at most, and will stay constant even as more types and methods are added to the application. This will also give developer a peace of mind that every method will be handled automatically.

Buffalo allows developer to quickly create aspects that act as a complete profiler of the application. This is especially useful in case of debugging. If a problem is encountered with the application, it would be helpful if developer knows where in the method it goes wrong, and have a look at the internal state of the method at the time of failure.

A developer can easily access all those information to show the state of each annotated method. Information such as the method signature, and what parameters are being passed in, what their types are, and even record their respective values when they were passed in. This information will be very helpful to aid in bug fixing.

Buffalo has performed well in isolating cross-cutting concerns into single unit of code, that is easily maintained and modified.s

Chapter 5

Conclusions

5.1 Current Status

Buffalo is currently at version 0.2. It contains two types of aspect: MethodBoundaryAspect, where various execution points can be intercepted. And MethodAroundAspect, where a method can be completely replaced while preserving the option to call back into the original method.

5.2 Future Work

There are couple areas where Buffalo can be improved upon. Usability wise, currently there is no automatic setup program that install Buffalo onto users computer and integrate into MS-Build System. Some manual steps are needed in order to provide a more seamless experience.

Functionality wise, when Buffalo does post compilation weaving it starts fresh each time, it would be interesting to see how incremental weaving can be used here.

5.3 Lessons Learned

While framework such as Buffalo mitigates the problem of cross-cutting concerns. Still, to efficiently tackle the root of the problem, compiler vendors have to actively embrace the AOP concept and provide native support it in the their languages.

Developers also have to understand the existent of such problems and solutions available to better educate themselves.

Only when the concept is widely understood and supported by both developers and vendors can there hope to begin eliminating such problems.

Bibliography

[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] MSDN TV. Transcript: Whiteboard with Anders Hejlsberg. `http://msdn.microsoft.com/en-us/cc320411.aspx`. Accessed: 07/19/2012.

[6] Mono. CSharp Compiler - Mono. `http://www.mono-project.com/CSharp_Compiler`. Accessed: 07/19/2012.

[7] 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.

[8] 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.

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

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

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

[12] ECMA International. ECMA-334 - C# Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`, June 2006.

[13] ECMA International. ECMA-335 - Common Language Infrastructure (CLI). `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf`, June 2012.

Appendix A

Code Listing

Buffalo is a fairly compact framework for the current functionality. The full source code is around 1,200 lines, which is fully included here. Note the short listing entitled ”buffalo/BuffaloAOP/Program.cs” at the end is the entire BuffaloAOP.exe program that kicks off the weaving.

```
1
2  namespace Buffalo
3  {
4      [System.AttributeUsage(System.AttributeTargets.Method,
5          AllowMultiple = false)]
6      public abstract class MethodAroundAspect : AspectBase, IMethodAroundAspect
7      {
8          public virtual object Invoke(MethodArgs args) { return null; }
9      }
10 }
```

Listing A.1: ../buffalo/MethodAroundAspect.cs

```
11 using System.Collections.Generic;
12
13 namespace Buffalo
14 {
15     public sealed class MethodArgs
16     {
17         private string name;
18         private string fullName;
19         private string returnTypeStr;
20         private string parameterStr;
21         private List<Parameter> parameters;
22         private object[] parameterArray;
23         private Exception exception;
24         private Object instance;
25
26         public MethodArgs()
27         {
28             this.parameters = new List<Parameter>();
29         }
30
```

```
31          public string Name
32          {
33              get { return this.name; }
34          }
35
36          public string FullName
37          {
38              get { return this.fullName; }
39          }
40
41          public Type ReturnType
42          {
43              get { return Type.GetType(this.returnTypeStr); }
44          }
45
46          public List<Parameter> Parameters
47          {
48              get { return this.parameters; }
49          }
50
51          public Exception Exception
52          {
53              get { return this.exception; }
54          }
55
56          public object Instance
57          {
58              get { return this.instance; }
59          }
60
61          public object[] ParameterArray
62          {
63              get { return this.parameterArray; }
64          }
65
66          public void SetProperties(string name,
67              string fullname,
68              string returnTypeStr,
69              string parameterStr,
70              object[] parameterArray,
71              object instance = null)
72          {
73              this.name = name;
74              this.fullName = fullname;
75              this.returnTypeStr = returnTypeStr;
76              this.parameterStr = parameterStr;
77              this.parameterArray = parameterArray;
78              this.instance = instance;
79
80              var splits = this.parameterStr.Split(new char[] { '|' }, StringSplitOptions.RemoveEmptyEntries);
81              foreach (var split in splits)
82              {
83                  var p = split.Split(new char[] { ':' }, StringSplitOptions.RemoveEmptyEntries);
84                  this.parameters.Add(new Parameter { Name = p[0], Type = Type.GetType(p[1]) });
85              }
86
87              for (int i = 0; i < this.Parameters.Count; ++i)
88              {
```

```
89                      this.Parameters[i].Value = this.parameterArray[i];
90                  }
91              }
92
93              public void SetException(Exception exception)
94              {
95                  this.exception = exception;
96              }
97
98              public object Proceed()
99              {
100                 return null;
101             }
102         }
103 }
```

Listing A.2: ../buffalo/MethodArgs.cs

```
104 using Buffalo.Common;
105 using Mono.Cecil;
106
107 namespace Buffalo
108 {
109     internal class Aspect : IAspect
110     {
111         public Aspect()
112         {
113             this.AssemblyLevelStatus = Enums.Status.NotApplied;
114         }
115
116         public string Name { get; set; }
117
118         public Enums.Status AssemblyLevelStatus { get; set; }
119
120         public TypeDefinition TypeDefinition { get; set; }
121
122         public Buffalo.Common.Enums.BuffaloAspect BuffaloAspect { get; set; }
123
124         public override string ToString()
125         {
126             return this.Name;
127         }
128     }
129 }
```

Listing A.3: ../buffalo/Aspect.cs

```
130
131 namespace Buffalo
132 {
133     public abstract class MethodBoundaryAspect : AspectBase, IMethodBoundaryAspect
134     {
135         public virtual void OnBefore(MethodArgs args) { }
136
137         public virtual void OnAfter(MethodArgs args) { }
138
139         public virtual void OnSuccess(MethodArgs args) { }
```

```
140
141             public virtual void OnException(MethodArgs args) { }
142         }
143     }
```

Listing A.4: ../buffalo/MethodBoundaryAspect.cs

```
144     {
145         [System.AttributeUsage(System.AttributeTargets.All,
146             AllowMultiple = false)]
147         public abstract class AspectBase : System.Attribute
148         {
149             public AspectBase(bool attributeExclude = false)
150             {
151                 this.AttributeExclude = attributeExclude;
152             }
153
154             public bool AttributeExclude { get; set; }
155         }
156     }
```

Listing A.5: ../buffalo/AspectBase.cs

```
157 using Buffalo.Extensions;
158 using Buffalo.Injectors;
159 using Buffalo.Interfaces;
160 using Mono.Cecil;
161 using Mono.Cecil.Cil;
162 using System;
163 using System.Collections.Generic;
164 using System.Collections.Specialized;
165 using System.IO;
166 using System.Linq;
167 using Reflection = System.Reflection;
168 using Mono.Collections.Generic;
169
170 namespace Buffalo
171 {
172     internal class Weaver
173     {
174         public Weaver(string assemblyPath)
175         {
176             ///TODO: Maybe just don't do anything if file not found
177             if (!File.Exists(assemblyPath))
178                 throw new FileNotFoundException();
179
180             AssemblyPath = assemblyPath;
181             this.Init();
182         }
183
184         static internal string AssemblyPath { get; set; }
185
186         static internal List<Aspect> Aspects { get; set; }
187
188         static internal Dictionary<string, Type> UnderlyingAspectTypes { get; set; }
189
190         internal Dictionary<MethodDefinition, List<Aspect>> EligibleMethods { get; set; }
```

```
191
192         internal List<TypeDefinition> TypeDefinitions { get; set; }
193
194         internal AssemblyDefinition AssemblyDefinition { get; set; }
195
196         internal StringCollection NewMethodNames { get; set; }
197
198         internal void Inject()
199         {
200             var injectors = new List<IInjectable>();
201
202             //apply the around aspect if necessary
203             var aroundAspectExist = this.EligibleMethods.Values.Any(x =>
204                 x.Any(y => y.BuffaloAspect == Enums.BuffaloAspect.MethodAroundAspect));
205             if (aroundAspectExist)
206                 injectors.Add(new MethodAroundInjector());
207
208             //apply the boundary aspect if necessary
209             var boundaryAspectExist = this.EligibleMethods.Values.Any(x =>
210                 x.Any(y => y.BuffaloAspect == Enums.BuffaloAspect.MethodBoundaryAspect));
211             if (boundaryAspectExist)
212                 injectors.Add(new MethodBoundaryInjector());
213
214             //inject
215             injectors.ForEach(x => x.Inject(this.AssemblyDefinition, this.EligibleMethods));
216
217             //write out the modified assembly
218             this.AssemblyDefinition.Write2(AssemblyPath);
219             Console.WriteLine("DONE");
220         }
221
222         private void Init()
223         {
224             //initialize the variables
225             Aspects = new List<Aspect>();
226             NewMethodNames = new StringCollection();
227             this.TypeDefinitions = new List<TypeDefinition>();
228             this.EligibleMethods = new Dictionary<MethodDefinition, List<Aspect>>();
229             //set the resolver in case assembly is in different directory
230             var resolver = new DefaultAssemblyResolver();
231             resolver.AddSearchDirectory(new FileInfo(AssemblyPath).Directory.FullName);
232             var parameters = new ReaderParameters { AssemblyResolver = resolver };
233             this.AssemblyDefinition = AssemblyDefinition.ReadAssembly(AssemblyPath, parameters);
234             //populate the type definition first
235             foreach (var m in this.AssemblyDefinition.Modules)
236                 m.Types.ToList().ForEach(x => this.TypeDefinitions.Add(x));
237             //if aspects are defined in a different assembly?
238             var typedefs = this.FindAspectTypeDefinition();
239             this.TypeDefinitions = this.TypeDefinitions.Union(typedefs).ToList();
240             //extract aspects from the type definitions
241             this.TypeDefinitions
242                 .Where(x => x.BaseType != null)
243                 .ToList()
244                 .ForEach(x =>
245                 {
246                     Buffalo.Common.Enums.BuffaloAspect? ba = null;
247                     if (x.BaseType.FullName == typeof(MethodBoundaryAspect).FullName)
248                         ba = Enums.BuffaloAspect.MethodBoundaryAspect;
```

```
249                         else if (x.BaseType.FullName == typeof(MethodAroundAspect).FullName)
250                             ba = Enums.BuffaloAspect.MethodAroundAspect;
251                         if(ba.HasValue)
252                             Aspects.Add(new Aspect { Name = x.FullName, TypeDefinition = x, BuffaloAspect = ba.Value });
253                     });
254                 Aspects.ForEach(x => x.AssemblyLevelStatus = this.CheckAspectStatus(this.AssemblyDefinition, x));
255                 //finally, get all the eligible methods for each aspect
256                 Aspects
257                     .Where(x => x.AssemblyLevelStatus != Enums.Status.Excluded)
258                     .ToList()
259                     .ForEach(x =>
260                     {
261 #if DEBUG
262                         Console.WriteLine("Aspect {0}: {1}", x.Name, x.AssemblyLevelStatus.ToString());
263                         Console.WriteLine("============================================");
264 #endif
265                         this.CheckEligibleMethods(x);
266                         Console.WriteLine("");
267                     });
268             }
269 
270             private List<TypeDefinition> FindAspectTypeDefinition()
271             {
272                 //look for aspect in this assembly, if aspect is defined in a different
273                 //assembly, import it here.
274                 var types = this.AssemblyDefinition.MainModule.Types;
275                 var tdefs = this.FindAspectsFromAttributes(this.AssemblyDefinition.CustomAttributes);
276 
277                 //loop thru the custom attributes of each type, resolve them to find aspects
278                 foreach (var type in types)
279                 {
280                     //if (type.CustomAttributes.Count == 0) continue;
281                     var tmp = this.FindAspectsFromAttributes(type.CustomAttributes);
282                     tdefs.AddRange(tmp);
283                     foreach (var m in type.Methods)
284                     {
285                         tdefs.AddRange(this.FindAspectsFromAttributes(m.CustomAttributes));
286                     }
287                 }
288 
289                 return tdefs;
290             }
291 
292             private List<TypeDefinition> FindAspectsFromAttributes(Collection<CustomAttribute> customAttributes)
293             {
294                 var tdefs = new List<TypeDefinition>();
295                 foreach (var ca in customAttributes)
296                 {
297                     var car = ca.AttributeType.Resolve();
298                     if (car.BaseType.FullName == typeof(MethodBoundaryAspect).FullName
299                         || car.BaseType.FullName == typeof(MethodAroundAspect).FullName)
300                     {
301                         if (!tdefs.Contains(car))
302                             tdefs.Add(car);
303                     }
304                 }
305                 return tdefs;
306             }
```

```
307
308         private void PrintEligibleMethods()
309         {
310             foreach (var de in this.EligibleMethods)
311             {
312                 Console.WriteLine(de.Key.FullName);
313                 foreach (var a in de.Value)
314                     Console.WriteLine("\t" + a.Name);
315             }
316         }
317
318         private void CheckEligibleMethods(Aspect aspect)
319         {
320             foreach (var t in this.TypeDefinitions.Where(x => !x.Name.Equals("<Module>")
321                 && (x.BaseType == null || (x.BaseType.FullName != typeof(MethodBoundaryAspect).FullName
322                     && x.BaseType.FullName != typeof(MethodAroundAspect).FullName))))
323             {
324                 var status = this.CheckAspectStatus(t, aspect);
325 #if DEBUG
326                 Console.WriteLine("\t{0}: {1}", t.Name, status.ToString());
327 #endif
328                 if (status == Enums.Status.Excluded)
329                     continue;
330
331                 var mths = this.GetMethodDefinitions(t, status, aspect);
332                 mths.ForEach(x =>
333                 {
334                     if (!this.EligibleMethods.ContainsKey(x))
335                     {
336                         this.EligibleMethods.Add(x, new List<Aspect>() { aspect });
337                     }
338                     else
339                     {
340                         var aspects = this.EligibleMethods[x];
341                         aspects.Add(aspect);
342                     }
343                 });
344             }
345         }
346
347         private List<MethodDefinition> GetMethodDefinitions(TypeDefinition typeDef, Enums.Status typeStatus, Aspect aspect)
348         {
349             var list = new List<MethodDefinition>();
350             foreach (var method in typeDef.Methods)
351             {
352                 var status = this.CheckAspectStatus(method, aspect);
353                 if (status == Enums.Status.Applied)
354                 {
355                     list.Add(method);
356                 }
357                 else
358                 {
359                     if (typeStatus == Enums.Status.Applied && status != Enums.Status.Excluded)
360                     {
361                         status = Enums.Status.Applied;
362                         list.Add(method);
363                     }
364                 }
```

```
#if DEBUG
                    Console.WriteLine("\t\t{0}: {1}", method.Name, status.ToString());
#endif
                }

                return list;
            }

            /// <summary>
            /// A TypeDefinition and MethodDefinition both implement the
            /// ICustomAttributeProvider interface, so it can be used here
            /// to determined if a method is marked as exclude or not.
            /// </summary>
            private Enums.Status CheckAspectStatus(ICustomAttributeProvider def, Aspect aspect)
            {
                Enums.Status status = aspect.AssemblyLevelStatus;

                bool attrFound = false;
                for (int i = 0; i < def.CustomAttributes.Count; ++i)
                {
                    if (def.CustomAttributes[i].AttributeType.FullName.Equals("System.Runtime.CompilerServices.CompilerGeneratedAttribute"))
                    {
                        status = Enums.Status.Excluded;
                        break;
                    }

                    if (aspect.TypeDefinition != null
                        && (aspect.BuffaloAspect == Enums.BuffaloAspect.MethodBoundaryAspect
                        || aspect.BuffaloAspect == Enums.BuffaloAspect.MethodAroundAspect)
                        && def.CustomAttributes[i].AttributeType.FullName.Equals(aspect.Name))
                    {
                        attrFound = true;
                        if (def.CustomAttributes[i].Properties.Count == 0)
                        {
                            status = Enums.Status.Applied;
                        }
                        else
                        {
                            var exclude = def.CustomAttributes[i].Properties.FirstOrDefault(x => x.Name == "AttributeExclude");
                            if (exclude.Argument.Value != null && (bool)exclude.Argument.Value == true)
                            {
                                status = Enums.Status.Excluded;
                                def.CustomAttributes.RemoveAt(i);
                            }
                        }
                    }
                }

                if (!attrFound && aspect.AssemblyLevelStatus == Enums.Status.Applied)
                {
                    //this aspect is applied on the assembly level and
                    //as a result the type and method might not have the
                    //attributed annotated, this is to programmatically add
                    //in the annotation so IL can be generated correctly.
                    var ctor = aspect.TypeDefinition.Methods.First(x => x.IsConstructor);
                    var ctoref = this.AssemblyDefinition.MainModule.Import(ctor);
                    def.CustomAttributes.Add(new CustomAttribute(ctoref));
                }
```

```
423
424                 return status;
425             }
426         }
427     }
```

Listing A.6: ../buffalo/Weaver.cs

```
428
429 namespace Buffalo
430 {
431     public sealed class Parameter
432     {
433         public Type Type { get; set; }
434
435         public string Name { get; set; }
436
437         public object Value { get; set; }
438     }
439 }
```

Listing A.7: ../buffalo/Parameter.cs

```
440 using Buffalo.Interfaces;
441 using Mono.Cecil;
442 using Mono.Cecil.Cil;
443 using System;
444 using System.Collections.Generic;
445 using System.Collections.Specialized;
446 using System.Linq;
447
448 namespace Buffalo.Injectors
449 {
450     internal class MethodAroundInjector : IInjectable
451     {
452         AssemblyDefinition AssemblyDefinition;
453         Dictionary<MethodDefinition, List<Aspect>> EligibleMethods;
454
455         public void Inject(Mono.Cecil.AssemblyDefinition assemblyDefinition,
456             Dictionary<Mono.Cecil.MethodDefinition, List<Aspect>> eligibleMethods)
457         {
458             /* The around aspect is to intercept all calls to the target method, and
459              * replace those calls with a completely different method. While preserving
460              * the option to call back to the target method when necessary.
461              */
462             this.AssemblyDefinition = assemblyDefinition;
463             this.EligibleMethods = eligibleMethods;
464             var NewMethodNames = new StringCollection();
465             var eligibleAroundMethods = this.EligibleMethods.Where(x => x.Value.Any(y =>
466                     y.BuffaloAspect == Common.Enums.BuffaloAspect.MethodAroundAspect));
467             foreach (var d in eligibleAroundMethods)
468             {
469                 var method = d.Key;
470                 var aspects = d.Value;
471                 var il = method.Body.GetILProcessor();
472                 var methodType = method.DeclaringType;
473                 var maInstructions = new List<Instruction>();
```

```
474                 var aspectVarInstructions = new List<Instruction>();
475                 var aroundInstructions = new List<Instruction>();
476
477                 foreach (var aspec in aspects.Where(x =>
478                     x.BuffaloAspect == Common.Enums.BuffaloAspect.MethodAroundAspect))
479                 {
480                     //if aspect is from a different assembly, need to work from that context
481                     var aspect = aspec;
482                     var writedll = false;
483                     AssemblyDefinition ass = null;
484                     if (!aspect.TypeDefinition.Module.FullyQualifiedName.Equals(
485                         this.AssemblyDefinition.MainModule.FullyQualifiedName))
486                     {
487                         ass = AssemblyDefinition.ReadAssembly(aspect.TypeDefinition.Module.FullyQualifiedName);
488                         var asp = ass.MainModule.Types.FirstOrDefault(x => x.FullName == aspect.Name);
489                         if (asp != null)
490                         {
491                             var newaspect = new Aspect { Name = asp.FullName, TypeDefinition = asp, BuffaloAspect = Common.Enums.BuffaloAspect.Met
492                             aspect = newaspect;
493                             writedll = true;
494                         }
495                     }
496
497                     var varTicks = System.DateTime.Now.Ticks;
498
499                     //create a replacement for the annotated function
500                     var methodName = string.Format("{0}{1}", method.Name, varTicks);
501                     MethodDefinition newmethod =
502                         new MethodDefinition(methodName, method.Attributes, method.ReturnType);
503                     methodType.Methods.Add(newmethod);
504                     NewMethodNames.Add(methodName);
505                     //newmethod.Body.SimplifyMacros();
506                     newmethod.Body.InitLocals = true;
507
508                     //create aspect variable
509                     var varAspectName = "asp" + varTicks;
510                     var varAspectRef = this.AssemblyDefinition.MainModule.Import(aspect.TypeDefinition);
511                     var varAspect = new VariableDefinition(varAspectName, varAspectRef);
512                     newmethod.Body.Variables.Add(varAspect);
513                     var varAspectIdx = newmethod.Body.Variables.Count - 1;
514                     var ctor = aspect.TypeDefinition.Methods.First(x => x.IsConstructor);
515                     var ctoref = this.AssemblyDefinition.MainModule.Import(ctor);
516                     //store the newly created aspect variable
517                     newmethod.Body.Instructions.Add(Instruction.Create(OpCodes.Newobj, ctoref));
518                     newmethod.Body.Instructions.Add(Instruction.Create(OpCodes.Stloc, varAspect));
519                     //copy all the paramters
520                     method.Parameters.ToList().ForEach(x =>
521                         newmethod.Parameters.Add(new ParameterDefinition(x.Name, x.Attributes, x.ParameterType)));
522                     //create a MethodArgs
523                     var var = newmethod.AddMethodArgsVariable(this.AssemblyDefinition);
524
525                     #region Calling MethodArgs.Invoke
526                     newmethod.Body.Instructions.Add(Instruction.Create(OpCodes.Ldloc, varAspect));
527                     newmethod.Body.Instructions.Add(Instruction.Create(OpCodes.Ldloc, var.Var));
528                     var aspectInvoke = aspect.TypeDefinition.Methods.First(x => x.Name.Equals("Invoke"));
529                     var aspectInvokeRef =
530                         this.AssemblyDefinition.MainModule.Import(aspectInvoke, newmethod);
531                     newmethod.Body.Instructions.Add(Instruction.Create(OpCodes.Callvirt, aspectInvokeRef));
```

```
532                 #endregion
533
534                 #region Handling return value
535                 if (!newmethod.ReturnType.FullName.Equals("System.Void"))
536                 {
537                     //create an object variable to hold the return value
538                     var varObj = new VariableDefinition("obj" + varTicks,
539                         this.AssemblyDefinition.MainModule.Import(typeof(object)));
540                     newmethod.Body.Variables.Add(varObj);
541                     newmethod.Body.Instructions.Add(
542                         Instruction.Create(OpCodes.Stloc, varObj));
543                     newmethod.Body.Instructions.Add(
544                         Instruction.Create(OpCodes.Ldloc, varObj));
545                     newmethod.Body.Instructions.Add(
546                         Instruction.Create(OpCodes.Unbox_Any, newmethod.ReturnType));
547                 }
548                 else
549                 {
550                     //pop the return value since it's not used?
551                     newmethod.Body.Instructions.Add(
552                         Instruction.Create(OpCodes.Pop));
553                 }
554                 #endregion
555
556                 #region Handling Proceed()
557                 var invoke = aspect.TypeDefinition.Methods.FirstOrDefault(
558                     x => x.FullName.Contains("::Invoke(Buffalo.MethodArgs)"));
559                 var containProceed = invoke.Body.Instructions.Any(x => x.ToString().Contains("System.Object Buffalo.MethodArgs::Proceed"));
560
561                 while (containProceed)
562                 {
563                     bool found = false;
564                     int instIdx = 0;
565                     for (; instIdx < invoke.Body.Instructions.Count; ++instIdx)
566                     {
567                         if (invoke.Body.Instructions[instIdx].ToString()
568                             .Contains("System.Object Buffalo.MethodArgs::Proceed"))
569                         {
570                             found = true;
571                             break;
572                         }
573                     }
574
575                     if (found)
576                     {
577                         #region Modified the call
578                         var invokeInstructions = new List<Instruction>();
579                         invoke.Body.Instructions.RemoveAt(instIdx);
580                         var startIdx = instIdx;
581
582                         //create a var to hold the original method type instance
583                         var instance = new VariableDefinition("instance" + DateTime.Now.Ticks,
584                             this.AssemblyDefinition.MainModule.Import(typeof(object)));
585                         invoke.Body.Variables.Add(instance);
586                         invoke.Body.InitLocals = true;
587
588                         //get the instance obj from MethodArgs
589                         var getInstance = typeof(MethodArgs).GetMethod("get_Instance");
```

```
590                 var getInstanceRef = this.AssemblyDefinition.MainModule.Import(getInstance);
591                 var getInstanceRef2 = aspect.TypeDefinition.Module.Import(getInstance);
592                 invokeInstructions.Add(Instruction.Create(OpCodes.Callvirt, getInstanceRef2));
593                 invokeInstructions.Add(Instruction.Create(OpCodes.Stloc, instance));
594
595                 //create object array to hold ParameterArray
596                 var objType = this.AssemblyDefinition.MainModule.Import(typeof(object));
597                 var objArray = new ArrayType(objType);
598                 var varArray = new VariableDefinition("va" + DateTime.Now.Ticks,
599                     (TypeReference)objArray);
600                 invoke.Body.Variables.Add(varArray);
601                 var getParameterArray = typeof(MethodArgs).GetMethod("get_ParameterArray");
602                 var getParameterArrayRef = this.AssemblyDefinition.MainModule.Import(getParameterArray);
603                 var getParameterArrayRef2 = aspect.TypeDefinition.Module.Import(getParameterArray);
604                 invokeInstructions.Add(Instruction.Create(OpCodes.Ldarg_1));
605                 invokeInstructions.Add(Instruction.Create(OpCodes.Callvirt, getParameterArrayRef2));
606                 invokeInstructions.Add(Instruction.Create(OpCodes.Stloc, varArray));
607
608                 //modify the Invoke() instruction to make a call to the original method
609                 invokeInstructions.Add(Instruction.Create(OpCodes.Ldloc, instance));
610                 var classref = aspect.TypeDefinition.Module.Import(method.DeclaringType);
611                 invokeInstructions.Add(Instruction.Create(OpCodes.Unbox_Any, classref));
612                 if (method.Parameters.Count > 0)
613                 {
614                     for (int i = 0; i < method.Parameters.Count; ++i)
615                     {
616                         var ptype = aspect.TypeDefinition.Module.Import(method.Parameters[i].ParameterType);
617                         invokeInstructions.Add(Instruction.Create(OpCodes.Ldloc, varArray));
618                         invokeInstructions.Add(il.Create(OpCodes.Ldc_I4, i));
619                         invokeInstructions.Add(il.Create(OpCodes.Ldelem_Ref));
620                         invokeInstructions.Add(Instruction.Create(OpCodes.Unbox_Any, ptype));
621                     }
622                 }
623
624                 //make the call
625                 var methodRef = aspect.TypeDefinition.Module.Import(method);
626                 invokeInstructions.Add(Instruction.Create(OpCodes.Callvirt, methodRef));
627
628                 #region Handling return value
629                 if (!method.ReturnType.FullName.Equals("System.Void"))
630                 {
631                     //create an object variable to hold the return value
632                     var varObj = new VariableDefinition("obj" + DateTime.Now.Ticks,
633                         this.AssemblyDefinition.MainModule.Import(typeof(object)));
634                     var retype = aspect.TypeDefinition.Module.Import(method.ReturnType);
635                     invoke.Body.Variables.Add(varObj);
636                     invokeInstructions.Add(
637                         Instruction.Create(OpCodes.Box, retype));
638                 }
639                 else
640                 {
641                     //method is suppose to return void, but since
642                     //previously it calls Proceed() which returns object type,
643                     //we need to handle that.
644                     invoke.Body.Instructions[instIdx] = Instruction.Create(OpCodes.Nop);
645                 }
646                 #endregion
647
```

```
                                //write out the instruction
                                invokeInstructions.ForEach(
                                    x => invoke.Body.Instructions.Insert(startIdx++, x));

                                #endregion
                            }

                            containProceed = invoke.Body.Instructions.Any(x => x.ToString().Contains("System.Object Buffalo.MethodArgs::Proceed"));
                        }

                        #endregion

                        #region Modify all calls from origin to the generated method
                        foreach (var type in this.AssemblyDefinition.MainModule.Types
                            .Where(x => x.BaseType == null || !x.BaseType.FullName.Equals("Buffalo.MethodAroundAspect")))
                        {
                            var methods = type.Methods.Where(x => !NewMethodNames.Contains(x.Name));
                            foreach (var m in methods)
                            {
                                for (int j = 0; j < m.Body.Instructions.Count; ++j)
                                {
                                    if (m.Body.Instructions[j].ToString().Contains(method.FullName))
                                    {
                                        m.Body.Instructions[j].Operand = newmethod;
                                        //MethodArgs.Invoke returns an object, need to unbox it here to the original type
                                        //However, unboxing is needed only if the original method has a return type
                                        //other than void
                                        if (!newmethod.ReturnType.FullName.Equals("System.Void"))
                                        {
                                            //var unbox = Instruction.Create(OpCodes.Unbox_Any, newmethod.ReturnType);
                                            //var il2 = m.Body.GetILProcessor();
                                            //il2.InsertAfter(m.Body.Instructions[j], unbox);
                                        }
                                    }
                                }
                            }
                        }
                        #endregion

                        newmethod.Body.Instructions.Add(Instruction.Create(OpCodes.Ret));
                        if (writedll)
                        {
                            ass.Write2(ass.MainModule.FullyQualifiedName);
                        }
                    }
                }
            }
        }
    }
}
```

Listing A.8: ../buffalo/Injectors/MethodAroundInjector.cs

```
using Buffalo.Extensions;
using Buffalo.Interfaces;
using Mono.Cecil;
using Mono.Cecil.Cil;
using System;
using System.Collections.Generic;
```

```
using System.Linq;

namespace Buffalo.Injectors
{
    internal class MethodBoundaryInjector : IInjectable
    {
        AssemblyDefinition AssemblyDefinition;
        Dictionary<MethodDefinition, List<Aspect>> EligibleMethods;

        public void Inject(AssemblyDefinition assemblyDefinition, Dictionary<MethodDefinition, List<Aspect>> eligibleMethods)
        {
            this.AssemblyDefinition = assemblyDefinition;
            this.EligibleMethods = eligibleMethods;

            var ems = this.EligibleMethods.ToList();
            var eligibleBoundaryMethods = ems.Where(x => x.Value.Any(y =>
                y.BuffaloAspect == Enums.BuffaloAspect.MethodBoundaryAspect));
            foreach (var d in eligibleBoundaryMethods)
            {
                var method = d.Key;
                if (method.Body == null)
                {
#if DEBUG
                    Console.WriteLine(string.Format("{0} has empty body, skipping", method.FullName));
#endif
                    continue;
                }
                var aspects = d.Value;
                var il = method.Body.GetILProcessor();
                var voidType = method.ReturnType.FullName.Equals("System.Void");
                var ret = il.Create(OpCodes.Ret);

                var maInstructions = new List<Instruction>();
                var aspectVarInstructions = new List<Instruction>();
                var beforeInstructions = new List<Instruction>();
                var successInstructions = new List<Instruction>();
                var exceptionInstructions = new List<Instruction>();
                var afterInstructions = new List<Instruction>();

                #region Method detail
                //create a MethodArgs
                var var = method.AddMethodArgsVariable(this.AssemblyDefinition);
                #endregion

                #region Before, Success, Exception, After
                var varExpTypeRef = this.AssemblyDefinition.MainModule.Import(typeof(System.Exception));
                for (int i = 0; i < aspects.Count; ++i)
                {
                    #region Create an aspect variable
                    var varAspectName = "asp" + System.DateTime.Now.Ticks;
                    var varAspectRef = this.AssemblyDefinition.MainModule.Import(aspects[i].TypeDefinition);
                    var varAspect = new VariableDefinition(varAspectName, varAspectRef);
                    method.Body.Variables.Add(varAspect);
                    var varAspectIdx = method.Body.Variables.Count - 1;
                    var ctor = aspects[i].TypeDefinition.Methods.First(x => x.IsConstructor);
                    var ctoref = this.AssemblyDefinition.MainModule.Import(ctor);
                    aspectVarInstructions.Add(Instruction.Create(OpCodes.Newobj, ctoref));
                    aspectVarInstructions.Add(Instruction.Create(OpCodes.Stloc, varAspect));
```

```
761                     #endregion
762
763                     #region Before, success, exception
764                     var before = method.FindMethodReference(aspects[i], Enums.AspectType.OnBefore);
765                     if (before != null)
766                     {
767
768                         beforeInstructions.Add(Instruction.Create(OpCodes.Ldloc, varAspect));
769                         beforeInstructions.Add(Instruction.Create(OpCodes.Ldloc, var.Var));
770                         var aspectBefore = aspects[i].TypeDefinition.Methods.FirstOrDefault(
771                             x => x.Name == Enums.AspectType.OnBefore.ToString());
772                         //this import is needed in case this aspect is defined in different assembly?
773                         var aspectBeforeRef = this.AssemblyDefinition.MainModule.Import(aspectBefore);
774                         beforeInstructions.Add(Instruction.Create(OpCodes.Callvirt, aspectBeforeRef));
775                     }
776
777                     var success = method.FindMethodReference(aspects[i], Enums.AspectType.OnSuccess);
778                     if (success != null)
779                     {
780                         successInstructions.Add(Instruction.Create(OpCodes.Ldloc, varAspect));
781                         successInstructions.Add(Instruction.Create(OpCodes.Ldloc, var.Var));
782                         var aspectSuccess = aspects[i].TypeDefinition.Methods.FirstOrDefault(
783                             x => x.Name == Enums.AspectType.OnSuccess.ToString());
784                         var aspectSuccessRef = this.AssemblyDefinition.MainModule.Import(aspectSuccess);
785                         successInstructions.Add(Instruction.Create(OpCodes.Callvirt, aspectSuccessRef));
786                     }
787
788                     var exception = method.FindMethodReference(aspects[i], Enums.AspectType.OnException);
789                     if (exception != null)
790                     {
791                         var expName = ”exp” + System.DateTime.Now.Ticks;
792                         var varExp = new VariableDefinition(expName, varExpTypeRef);
793                         method.Body.Variables.Add(varExp);
794                         exceptionInstructions.Add(Instruction.Create(OpCodes.Stloc, varExp));
795                         exceptionInstructions.Add(Instruction.Create(OpCodes.Ldloc, var.Var));
796                         exceptionInstructions.Add(Instruction.Create(OpCodes.Ldloc, varExp));
797                         var maSetException = typeof(MethodArgs).GetMethod(”SetException”);
798                         var maSetExceptionRef = this.AssemblyDefinition.MainModule.Import(maSetException, method);
799                         exceptionInstructions.Add(Instruction.Create(OpCodes.Callvirt, maSetExceptionRef));
800
801                         exceptionInstructions.Add(Instruction.Create(OpCodes.Ldloc, varAspect));
802                         exceptionInstructions.Add(Instruction.Create(OpCodes.Ldloc, var.Var));
803                         var aspectException = aspects[i].TypeDefinition.Methods.FirstOrDefault(
804                             x => x.Name == Enums.AspectType.OnException.ToString());
805                         var aspectExceptionRef = this.AssemblyDefinition.MainModule.Import(aspectException);
806                         exceptionInstructions.Add(Instruction.Create(OpCodes.Callvirt, aspectExceptionRef));
807                     }
808
809                     var after = method.FindMethodReference(aspects[i], Enums.AspectType.OnAfter);
810                     if (after != null)
811                     {
812                         afterInstructions.Add(Instruction.Create(OpCodes.Ldloc, varAspect));
813                         afterInstructions.Add(Instruction.Create(OpCodes.Ldloc, var.Var));
814                         var aspectAfter = aspects[i].TypeDefinition.Methods.FirstOrDefault(
815                             x => x.Name == Enums.AspectType.OnAfter.ToString());
816                         var aspectAfterRef = this.AssemblyDefinition.MainModule.Import(aspectAfter);
817                         afterInstructions.Add(Instruction.Create(OpCodes.Callvirt, aspectAfterRef));
818                     }
```

```
819                 #endregion
820             }
821
822             int varIdx = var.VarIdx;
823             //maInstructions.ForEach(x => method.Body.Instructions.Insert(varIdx++, x));
824             aspectVarInstructions.ForEach(x => method.Body.Instructions.Insert(varIdx++, x));
825
826             int beforeIdx = varIdx;
827             //perform this only if user overrides Before() in the aspect
828             if (beforeInstructions.Count > 0)
829             {
830                 beforeInstructions.ForEach(x => method.Body.Instructions.Insert(beforeIdx++, x));
831             }
832
833             /* the last instruction after success should jump to return, or 3 instruction before
834              * return if return type is not void, or as an optimization, maybe we can even skip
835              * directly to last() - 2?
836              */
837             var successRet = voidType ? method.Body.Instructions.Last() :
838                 method.Body.Instructions[method.Body.Instructions.Count - 3];
839             Instruction successLeave = il.Create(OpCodes.Leave, successRet);
840             ///TODO: need to double check the format of the MSIL return
841             ///br.s, ldloc, ret when return type is not void, thereby decrement by 3
842             int successIdx = voidType ? method.Body.Instructions.Count - 1 : method.Body.Instructions.Count - 3;
843             //perform this only if user overrides Success() in the aspect
844             successInstructions.Add(successLeave);
845             if (successInstructions.Count > 0)
846             {
847                 successInstructions.ForEach(x => method.Body.Instructions.Insert(successIdx++, x));
848             }
849
850             int exceptionIdx = voidType ? method.Body.Instructions.Count - 1 : method.Body.Instructions.Count - 3;
851             int exceptionIdxConst = exceptionIdx;
852             var exceptionRet = voidType ? method.Body.Instructions.Last() :
853                 method.Body.Instructions[method.Body.Instructions.Count - 3];
854             Instruction exceptionLeave = il.Create(OpCodes.Leave, exceptionRet);
855             //perform this only if user overrides Exception() in the aspect
856             if (exceptionInstructions.Count > 0)
857             {
858                 exceptionInstructions.Add(exceptionLeave);
859                 exceptionInstructions.ForEach(x => method.Body.Instructions.Insert(exceptionIdx++, x));
860             }
861
862             var afterRet = voidType ? method.Body.Instructions.Last() :
863                 method.Body.Instructions[method.Body.Instructions.Count - 3];
864             var endfinally = il.Create(OpCodes.Endfinally);
865             int afterIdx = voidType ? method.Body.Instructions.Count - 1 : method.Body.Instructions.Count - 3;
866             int afterIdxConst = afterIdx;
867             //perform this only if user overrides After() in the aspect
868             if (afterInstructions.Count > 0)
869             {
870                 afterInstructions.Add(endfinally);
871                 afterInstructions.ForEach(x => method.Body.Instructions.Insert(afterIdx++, x));
872             }
873
874             #endregion
875
876             #region Catch..Finally..
```

```
                    //add the catch block only if user overrides Exception() in the aspect
                    if (exceptionInstructions.Count > 0)
                    {
                        var catchHandler = new ExceptionHandler(ExceptionHandlerType.Catch)
                        {
                            TryStart = method.Body.Instructions[varIdx],
                            TryEnd = successLeave.Next,
                            HandlerStart = method.Body.Instructions[exceptionIdxConst],
                            HandlerEnd = exceptionLeave.Next,
                            CatchType = varExpTypeRef,
                        };
                        method.Body.ExceptionHandlers.Add(catchHandler);
                    }

                    //add the finally block only if user overrides After() in the aspect
                    if (afterInstructions.Count > 0)
                    {
                        var finallyHandler = new ExceptionHandler(ExceptionHandlerType.Finally)
                        {
                            TryStart = method.Body.Instructions[varIdx],
                            TryEnd = method.Body.Instructions[afterIdxConst],
                            HandlerStart = method.Body.Instructions[afterIdxConst],
                            HandlerEnd = afterRet,
                            CatchType = null,
                        };
                        method.Body.ExceptionHandlers.Add(finallyHandler);
                    }
                    #endregion
                }
            }
        }
    }
```

Listing A.9: ../buffalo/Injectors/MethodBoundaryInjector.cs

```
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("Buffalo")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Microsoft")]
[assembly: AssemblyProduct("Buffalo")]
[assembly: AssemblyCopyright("Copyright  Microsoft 2012")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("625c6e79-7034-48cf-8689-b9e44f3bf96d")]

```

```
// Version information for an assembly consists of the following four values:
//
// Major Version
// Minor Version
// Build Number
// Revision
//
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("0.1.0.0")]
[assembly: AssemblyFileVersion("0.1.0.0")]
[assembly: InternalsVisibleTo("BuffaloAOP")]
```

Listing A.10: ../buffalo/Properties/AssemblyInfo.cs

```

namespace Buffalo.Common
{
    internal struct BeginEndMarker
    {
        public int BeginIndex { get; set; }
        public int EndIndex { get; set; }
    }
}
```

Listing A.11: ../buffalo/Common/BeginEndMarker.cs

```
{
    internal class Enums
    {
        internal enum Status
        {
            Applied,
            NotApplied,
            Excluded
        }

        internal enum AspectType
        {
            OnBefore,
            OnAfter,
            OnSuccess,
            OnException,
            Invoke
        }

        internal enum BuffaloAspect
        {
            MethodBoundaryAspect,
            MethodAroundAspect
        }
    }
}
```

Listing A.12: ../buffalo/Common/Enums.cs

```
980 using Mono.Cecil;
981 using Mono.Cecil.Cil;
982 using System;
983 using System.Collections.Generic;
984 using System.IO;
985 using System.Linq;
986 using System.Text;
987
988 namespace Buffalo.Extensions
989 {
990     internal static class Extensions
991     {
992         internal static MethodReference FindMethodReference(this MethodDefinition method, Aspect aspect, Enums.AspectType name)
993         {
994             return aspect
995                 .TypeDefinition
996                 .Methods
997                 .FirstOrDefault(x => x.Name.Equals(name.ToString()));
998         }
999
1000        internal static VariableResult AddMethodArgsVariable(this MethodDefinition method,
1001            AssemblyDefinition assemblyDef)
1002        {
1003            var var = new VariableResult();
1004            var instructions = new List<Instruction>();
1005            method.Body.InitLocals = true;
1006            var il = method.Body.GetILProcessor();
1007            var isValueType = false;
1008
1009            //create var to hold parameter count
1010            var pcVar = new VariableDefinition("pc" + DateTime.Now.Ticks,
1011                    assemblyDef.MainModule.Import(typeof(int)));
1012            method.Body.Variables.Add(pcVar);
1013            instructions.Add(Instruction.Create(OpCodes.Ldc_I4, method.Parameters.Count));
1014            instructions.Add(Instruction.Create(OpCodes.Stloc, pcVar));
1015
1016            //create an object array to hold the parameter values
1017            var objType = assemblyDef.MainModule.Import(typeof(object));
1018            var objArray = new ArrayType(objType);
1019            var varArray = new VariableDefinition("va" + DateTime.Now.Ticks,
1020                (TypeReference)objArray);
1021            method.Body.Variables.Add(varArray);
1022            instructions.Add(Instruction.Create(OpCodes.Ldloc, pcVar));
1023            instructions.Add(Instruction.Create(OpCodes.Newarr, objType));
1024            instructions.Add(Instruction.Create(OpCodes.Stloc, varArray));
1025
1026            #region Handle paramters
1027            //loop thru the parameters, extract the values and
1028            //store them in MethodArgs.ParameterArray
1029            TypeSpecification typeSpec = null;
1030            for (int i = 0; i < method.Parameters.Count; ++i )
1031            {
1032                var metaType = method.Parameters[i].ParameterType.MetadataType;
1033                if (metaType == MetadataType.UIntPtr
1034                    || metaType == MetadataType.FunctionPointer
1035                    || metaType == MetadataType.IntPtr
1036                    || metaType == MetadataType.Pointer)
```

```
1037                    continue;
1038
1039                instructions.Add(Instruction.Create(OpCodes.Ldloc, varArray));
1040                instructions.Add(Instruction.Create(OpCodes.Ldc_I4, i));
1041                if (method.IsStatic)
1042                    instructions.Add(il.Create(OpCodes.Ldarg, i));
1043                else
1044                    instructions.Add(il.Create(OpCodes.Ldarg, i + 1));
1045
1046                isValueType = false;
1047                var pType = method.Parameters[i].ParameterType;
1048                if (pType.IsByReference)
1049                {
1050                    typeSpec = pType as TypeSpecification;
1051                    if (typeSpec != null)
1052                    {
1053                        switch (typeSpec.ElementType.MetadataType)
1054                        {
1055                            case MetadataType.Boolean:
1056                            case MetadataType.SByte:
1057                                instructions.Add(Instruction.Create(OpCodes.Ldind_I1));
1058                                isValueType = true;
1059                                break;
1060                            case MetadataType.Int16:
1061                                instructions.Add(Instruction.Create(OpCodes.Ldind_I2));
1062                                isValueType = true;
1063                                break;
1064                            case MetadataType.Int32:
1065                                instructions.Add(Instruction.Create(OpCodes.Ldind_I4));
1066                                isValueType = true;
1067                                break;
1068                            case MetadataType.Int64:
1069                            case MetadataType.UInt64:
1070                                instructions.Add(Instruction.Create(OpCodes.Ldind_I8));
1071                                isValueType = true;
1072                                break;
1073                            case MetadataType.Byte:
1074                                instructions.Add(Instruction.Create(OpCodes.Ldind_U1));
1075                                isValueType = true;
1076                                break;
1077                            case MetadataType.UInt16:
1078                                instructions.Add(Instruction.Create(OpCodes.Ldind_U2));
1079                                isValueType = true;
1080                                break;
1081                            case MetadataType.UInt32:
1082                                instructions.Add(Instruction.Create(OpCodes.Ldind_U4));
1083                                isValueType = true;
1084                                break;
1085                            case MetadataType.Single:
1086                                instructions.Add(Instruction.Create(OpCodes.Ldind_R4));
1087                                isValueType = true;
1088                                break;
1089                            case MetadataType.Double:
1090                                instructions.Add(Instruction.Create(OpCodes.Ldind_R8));
1091                                isValueType = true;
1092                                break;
1093                            case MetadataType.IntPtr:
1094                            case MetadataType.UIntPtr:
```

```
                                instructions.Add(Instruction.Create(OpCodes.Ldind_I));
                                isValueType = true;
                                break;
                            default:
                                if (typeSpec.ElementType.IsValueType)
                                {
                                    instructions.Add(Instruction.Create(OpCodes.Ldobj, typeSpec.ElementType));
                                    isValueType = true;
                                }
                                else
                                {
                                    instructions.Add(Instruction.Create(OpCodes.Ldind_Ref));
                                    isValueType = false;
                                }
                                break;
                        }
                    }
                }

                if (pType.IsValueType || isValueType)
                {
                    if(isValueType)
                        instructions.Add(Instruction.Create(OpCodes.Box, typeSpec.ElementType));
                    else
                        instructions.Add(Instruction.Create(OpCodes.Box, pType));
                }

                instructions.Add(Instruction.Create(OpCodes.Stelem_Ref));
            }
            #endregion

            StringBuilder sb = new StringBuilder();
            method.Parameters.ToList()
                .ForEach(x =>
                {
                    sb.Append(string.Format("{0}:{1}|", x.Name, x.ParameterType.FullName));
                });

            var maType = typeof(MethodArgs);
            var maName = "ma" + DateTime.Now.Ticks;
            var maSetProperties = maType.GetMethod("SetProperties");
            var varMa = new VariableDefinition(maName, assemblyDef.MainModule.Import(maType));
            method.Body.Variables.Add(varMa);
            var vaMaIdx = method.Body.Variables.Count - 1;
            var maCtr = maType.GetConstructor(new Type[] { });
            MethodReference maCtrRef = assemblyDef.MainModule.Import(maCtr);
            instructions.Add(Instruction.Create(OpCodes.Newobj, maCtrRef));
            instructions.Add(Instruction.Create(OpCodes.Stloc, varMa));
            instructions.Add(Instruction.Create(OpCodes.Ldloc, varMa));
            instructions.Add(Instruction.Create(OpCodes.Ldstr, method.Name));
            instructions.Add(Instruction.Create(OpCodes.Ldstr, method.FullName));
            instructions.Add(Instruction.Create(OpCodes.Ldstr, method.ReturnType.FullName));
            instructions.Add(Instruction.Create(OpCodes.Ldstr, sb.ToString()));
            instructions.Add(Instruction.Create(OpCodes.Ldloc, varArray));
            if (method.IsStatic)
                instructions.Add(Instruction.Create(OpCodes.Ldnull));
            else
                instructions.Add(Instruction.Create(OpCodes.Ldarg_0));
```

```
1153
1154                var maSetPropertiesRef = assemblyDef.MainModule.Import(maSetProperties, method);
1155                instructions.Add(Instruction.Create(OpCodes.Callvirt, maSetPropertiesRef));
1156
1157                int idx = 0;
1158                instructions.ForEach(x => method.Body.Instructions.Insert(idx++, x));
1159
1160                var.Var = varMa;
1161                var.ParamArray = varArray;
1162                var.VarIdx = idx;
1163                return var;
1164            }
1165
1166            internal static void Write2(this AssemblyDefinition assembly, string outpath)
1167            {
1168                //write out the modified assembly
1169                var fi = new FileInfo(outpath);
1170                var folderpath = string.Format(@"{0}\modified", fi.Directory.FullName);
1171                var fn = fi.Name;
1172                if (!Directory.Exists(folderpath)) Directory.CreateDirectory(folderpath);
1173                var n = Path.Combine(folderpath, fn);
1174                assembly.Write(n);
1175            }
1176        }
1177
1178        internal class VariableResult
1179        {
1180            public VariableDefinition Var { get; set; }
1181            public VariableDefinition ParamArray { get; set; }
1182            public int VarIdx { get; set; }
1183        }
1184    }
```

Listing A.13: ../buffalo/Extensions/Extensions.cs

```
1185    {
1186        internal interface IMethodAroundAspect : IAspect
1187        {
1188            object Invoke(MethodArgs args);
1189        }
1190    }
```

Listing A.14: ../buffalo/Interfaces/IMethodAroundAspect.cs

```
1191    using Mono.Cecil;
1192
1193    namespace Buffalo.Interfaces
1194    {
1195        internal interface IInjectable
1196        {
1197            void Inject(AssemblyDefinition assemblyDefinition, Dictionary<MethodDefinition, List<Aspect>> eligibleMethods);
1198        }
1199    }
```

Listing A.15: ../buffalo/Interfaces/IInjectable.cs

```
{
    internal interface IMethodBoundaryAspect : IAspect
    {
        void OnBefore(MethodArgs args);
        void OnAfter(MethodArgs args);
        void OnSuccess(MethodArgs args);
        void OnException(MethodArgs args);
    }
}
```

Listing A.16: ../buffalo/Interfaces/IMethodBoundaryAspect.cs

```
{
    /// <summary>
    /// This is the base aspect interface
    /// </summary>
    internal interface IAspect
    {
    }
}
```

Listing A.17: ../buffalo/Interfaces/IAspect.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using Buffalo;

namespace BuffaloAOP
{
    class Program
    {
        static string path;
        static void Main(string[] args)
        {
            if (args == null || args.Count() == 0)
            {
                Console.WriteLine("USAGE: BuffaloAOP.exe <assembly_path>");
                Environment.Exit(1);
            }

            path = args[0];
            new Weaver(path).Inject();
        }
    }
}
```

Listing A.18: ../../buffalo/BuffaloAOP/Program.cs

Appendix B

User Manual

It is easy to get start using Buffalo. I assume you are reasonably familiar with the Visual Studio IDE and the general layout of a VS solution. I also assume you know how to compile a solution and where to find the compiled assembly. Buffalo is developed in VS2012, it is recommended you have the same version of the IDE installed.

B.1 Compiling

To begin, first download the full source code from https://github.com/wliao008/buffalo. Open buffalo.sln in VS2012 and compile the source code. This will produce the Buffalo.dll and BuffaloAOP.exe in their respective bin/debug folder.

For this example we will perform the weaving from a command prompt. So create a folder name under the C drive, here I will call it "Buffalo". Copy Buffalo.dll, BuffaloAOP.exe and Mono.Cecil.dll to C:\Buffalo. Note this folder can be located anywhere in your system, I am just putting it on the C drive for simplicity.

Now let us create an aspect.

B.2 Simple Profiler

In this example we will create a profiler for our application. Suppose we have the following simple program.

```
1 using System;
2
3 namespace Hello
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Hello h = new Hello();
            h.SayHello();
            h.Say("Hey Buffalo how's it going!");

            //pause the console
            Console.Read();
        }
    }

    public class Hello
    {
        public void SayHello()
        {
            Console.WriteLine("Hello World!");
        }

        public void Say(string msg)
        {
            Console.WriteLine(msg);
        }
    }
}
```

Listing B.1: Hello program

When the program runs, it will display the following output:

```
Hello World!
Hey Buffalo how's it going!
```

Listing B.2: Hello program output

And suppose that we want to monitor the program, we want to know when a method was accessed and exited. We can easily create a aspect to do such work.

```
using Buffalo;
using System;

public class TraceAspect : MethodBoundaryAspect
{
    public override void Before(MethodArgs args)
    {
        Display("ENTERING", args);
    }

    public override void After(MethodArgs args)
    {
        Display("EXITING", args);
    }

    public override void Success(MethodArgs args)
```

```
17      {
18          Display(”SUCCESSFULLY EXECUTED”, args);
19      }
20
21      public override void Exception(MethodArgs args)
22      {
23          Display(”EXCEPTION ON”, args);
24      }
25
26      void Display(string title, MethodArgs args)
27      {
28          Console.WriteLine(”{0} {1}”, title, args.FullName);
29          foreach (var p in args.Parameters)
30          {
31              Console.WriteLine(”\t{0} ({1}) = {2}”, p.Name, p.Type, p.Value);
32          }
33      }
34  }
```

Listing B.3: TraceAspect

With the aspect defined, now we can apply this aspect on any of the three different levels. Lets apply it to the Hello class for example.

```
1  [TraceAspect]
2  public class Hello
3  {
4          //...
5  }
```

Listing B.4: Apply Aspect to the Hello Class

Now everything is in place. We can now invoke the BuffaloAOP.exe to perform the weaving. Open a command prompt and navigate to C:\Buffalo. And issue this command:

```
1  C:\Buffalo>BuffaloAOP.exe <path_to_the_hello_program_exe>
```

Listing B.5: Invoking BuffaloAOP.exe

Replace path to the hello program exe with the actual complete path to the program assembly. Suppose the program assembly is located at C:\Projects\Hello\bin\Hello.exe, we would issue the command as follow:

```
1  C:\Buffalo>BuffaloAOP.exe C:\Projects\Hello\bin\Hello.exe
```

Listing B.6: Invoking BuffaloAOP.exe Example

If everything goes well BuffaloAOP.exe will perform the injection and put the final assembly in the Modified folder inside the folder of the target assembly. In this case it will

be at C:\Projects\Hello\bin\Modified\Hello.exe. Now when the program runs, it will display the following output:

```
1  ENTERING System.Void Hello.Program::Main(System.String[])
2          args (System.String[]) = System.String[]
3  ENTERING System.Void Hello.Hello::.ctor()
4  SUCCESSFULLY EXECUTED System.Void Hello.Hello::.ctor()
5  EXITING System.Void Hello.Hello::.ctor()
6  ENTERING System.Void Hello.Hello::SayHello()
7  Hello World!
8  SUCCESSFULLY EXECUTED System.Void Hello.Hello::SayHello()
9  EXITING System.Void Hello.Hello::SayHello()
10 ENTERING System.Void Hello.Hello::Say(System.String)
11         msg (System.String) = Hey Buffalo how's it going!
12 Hey Buffalo how's it going!
13 SUCCESSFULLY EXECUTED System.Void Hello.Hello::Say(System.String)
14         msg (System.String) = Hey Buffalo how's it going!
15 EXITING System.Void Hello.Hello::Say(System.String)
16         msg (System.String) = Hey Buffalo how's it going!
```

Listing B.7: TraceAspect output

Line 7 and 12 are the original method output, the rest are the output of the various interception points. Note that line 11 also capture the parameter value passed into each method and is available from the aspect.

B.3 Integrate With MS-Build System

Buffalo can be integrated with MS-Build, so weaving can be invoked automatically when a project is compiled from the Visual Studio IDE. Note that the following instructions are just the bare minimum to get this working, a lot of bell and whistle are omitted.

MS-Build is integrated with Visual Studio IDE via configuration file. For example, a C# project has the associated .csproj, if open in a text editor you will see a line that reference a different configuration file: Microsoft.CSharp.targets. This file in term reference Microsoft.Common.targets.

Each .NET version has a Microsoft.Common.targets file. Depending on the version you are using, open up this file in a text editor. For example, for .NET 4.0, this file is located in C:\Windows\Microsoft.NET\Framework\v4.0.30319\Microsoft.Common.targets.

Under the Compile section, around line 2013, add the line to import the Buffalo.targets file as shown in figure B.1

```
1995    <!--
1996    ===================================================================
1997                                        Compile
1998    ===================================================================
1999    -->
2000    <PropertyGroup>
2001        <CompileDependsOn>
2002            ResolveReferences;
2003            ResolveKeySource;
2004            SetWin32ManifestProperties;
2005            _GenerateCompileInputs;
2006            BeforeCompile;
2007            _TimeStampBeforeCompile;
2008            CoreCompile;
2009            _TimeStampAfterCompile;
2010            AfterCompile
2011        </CompileDependsOn>
2012    </PropertyGroup>
2013    <Import Project="C:\Buffalo\Buffalo.targets"/>
```

Figure B.1: Adding Buffalo.targets

If you open Buffalo.targets in a text editor, it contains the following context:

```
1  <Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
2    <PropertyGroup>
3      <CompileDependsOn>
4        $(CompileDependsOn);
5        Buffalo
6      </CompileDependsOn>
7    </PropertyGroup>
8
9    <Target Name="Buffalo">
10     <Message Text="Hello Buffalo! @(IntermediateAssembly)"/>
11     <Exec Command="&quot;C:\Buffalo\BuffaloAOP.exe&quot; &quot;@(IntermediateAssembly)&quot;"/>
12   </Target>
13 </Project>
```

Listing B.8: Buffalo.targets

This is how Buffalo get hooked into MS-Build, what this mean is that when user compiles a project, everything defined in the CompileDependsOn property group will be performed first, then a new target named "Buffalo" will be called immediately, which will invoke the BuffaloAOP via the Exec Command. Note that for the Exec Command, a complete path to BuffaloAOP.exe must be provided, including the decoded quotation marks as shown.

Make sure to save all the changes.

Now every time a C# project is compiled, Buffalo will be invoked automatically to perform the weaving.