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 areas such as unhandled exception catching, tracing and logging, etc.

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 working on this project.

Buffalo comprises around 1,200 line of source codes. A user manual is included in Appendix A, which contains some examples on how the system works. Instructions is also included 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, 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 Post Compilation 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.

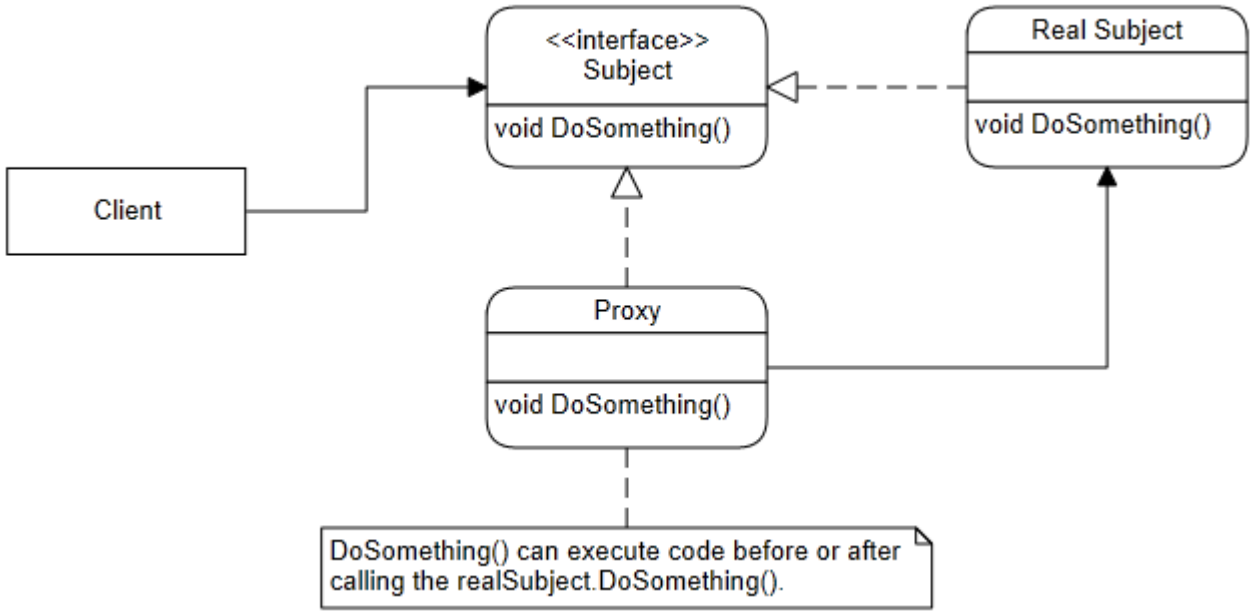

Figure 2.1: AOP Framework Using Proxy Pattern

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 Post Compilation Weaving

The approach Buffalo takes is Post Compilation Weaving. The idea is that after compilation of the source code, Buffalo takes over to disassemble the assembly, and 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. Therefore no messy configuration is required.

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.

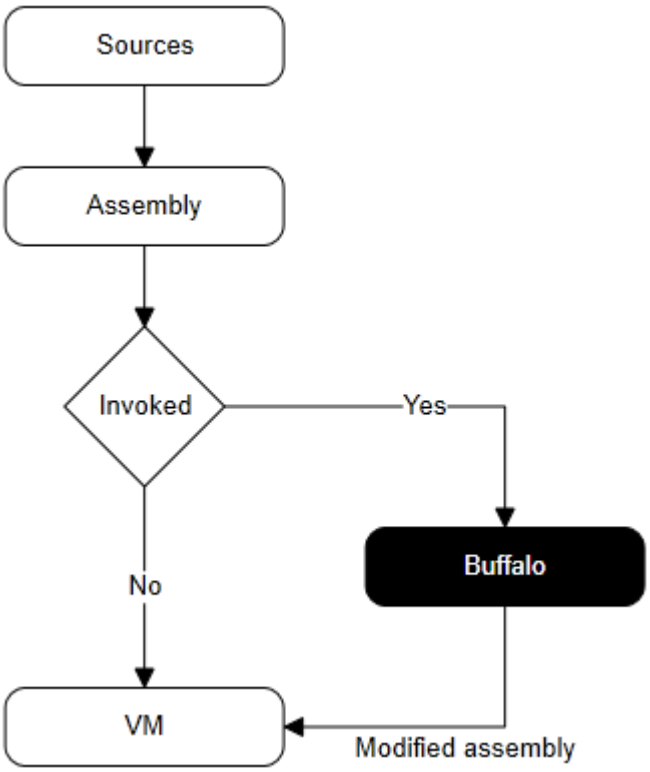

Figure 2.2: Buffalo Model

2.4 A Buffalo Aspect

When performing post compilation weaving, Buffalo has to be able to discover what aspect is applied to what methods 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 inheriting from System.Attribute. It can therefore contain any valid .NET code. But specifically an aspect needs to override various predefined methods in order to do something useful. The next section will discuss the relationship between various aspect types.

2. After compilation, the assembly will now contain the meta-data about all the aspects. Buffalo can inspect the assembly for such 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? What type of weaving does it do? For inspiration existing works such as AspectJ [3] and PostSharp [8] were studied. 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 at any point during the execution. These various points of interception are grouped into the MethodBoundaryAspect.

MethodBoundaryAspect can be cleanly mapped to the try-catch-finally statements of the .NET languages. As far as the runtime 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

When performing CIL modification, the above will be transformed by Buffalo into something shown in Figure 2.2. This clearly captures the spirit of the MethodBoundaryAspect. Regardless of whether the source already contain its own try-catch, or try-catch-finally blocks, the body of the method will be wrapped inside of a new try-catch-finally block by Buffalo.

```
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

The 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 implement, but it turns out to be much more involved than the MethodBoundaryAspect.

Since the option to call back into the original method is needed, 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 stay intact during the CIL modification.

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 instantiates the aspect and makes a call to the Invoke() method, which is the actual code that will be ran as a replacement.

Inside the Invoke method, developer can issue 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 process is illustrated in figure 2.3.

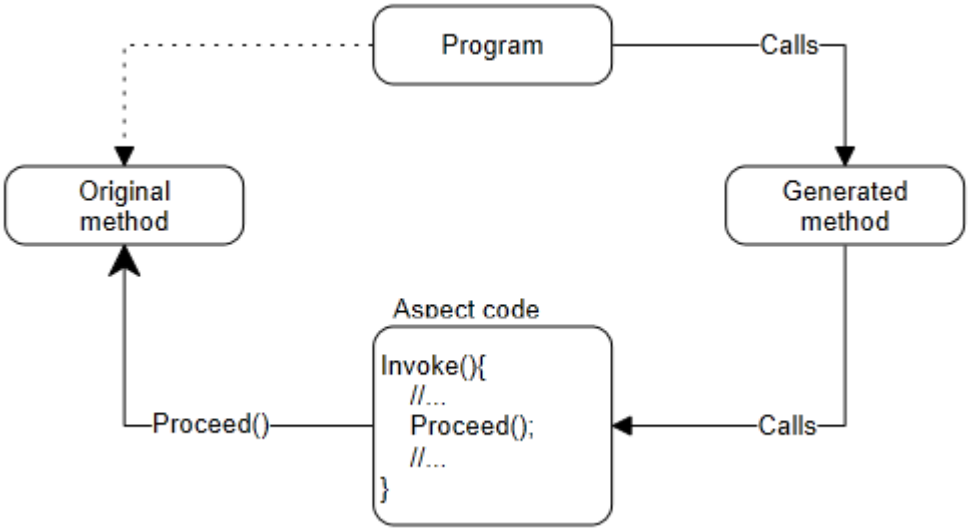

Figure 2.3: Illustration of MethodAroundAspect

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 really 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.

A Buffalo aspect can be applied on three levels, with the following characteristics:

1. Method - apply the aspect to an individual method.
2. Class - if aspect is applied to a class, all public methods including the public properties automatically get applied.
3. Assembly - if aspect is 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.

All aspects have a property named AttributeExclude, if set to true then the annotated target will not be included in the weaving. This exclusion can happen on any levels. For example, if a method contains this annotation [SampleAspect(AttributeExclude=true)], the method will be skipped for the SampleAspect 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.

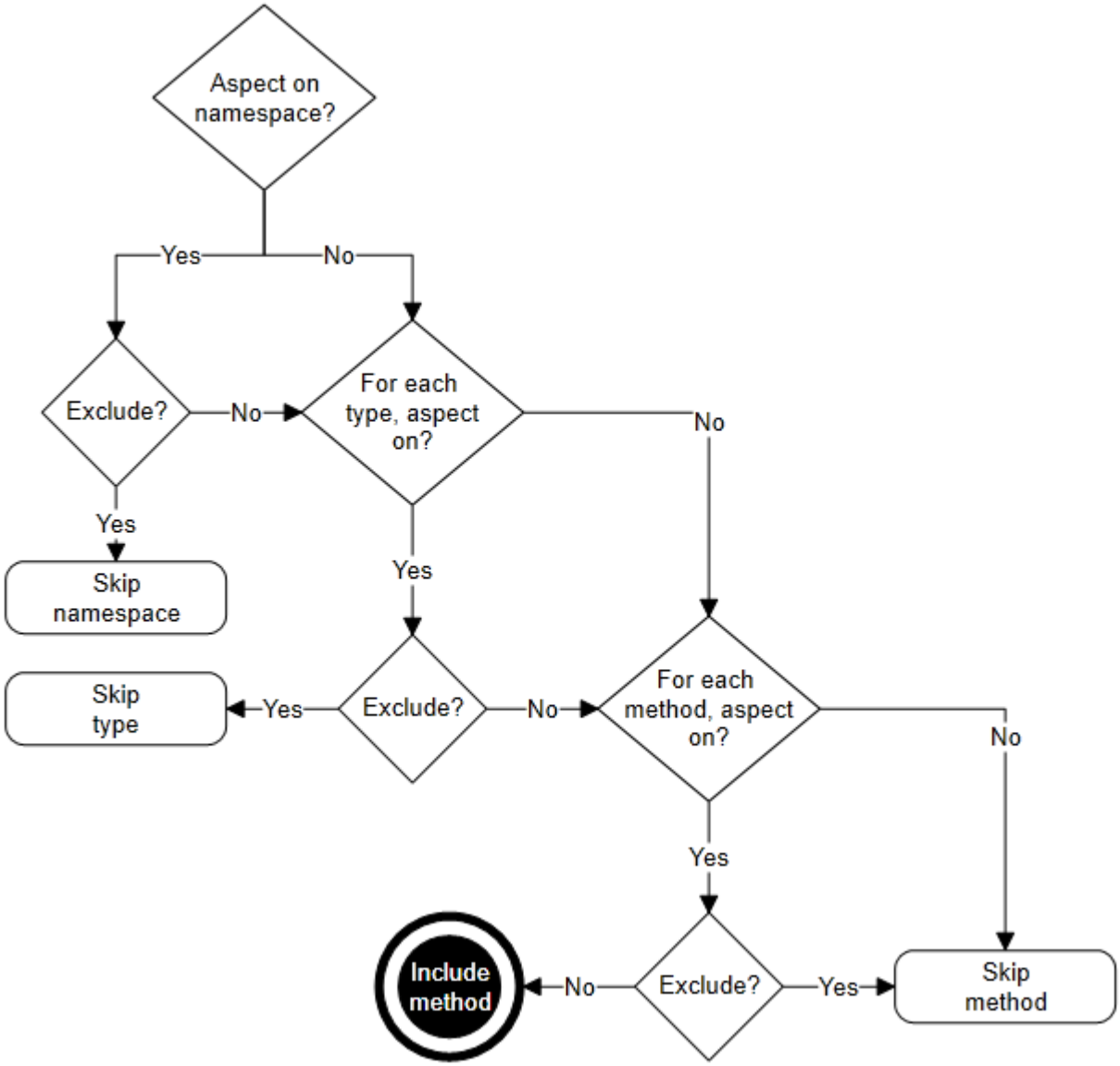

Figure 3.1: Diagram for Finding Eligible Methods

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.

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.

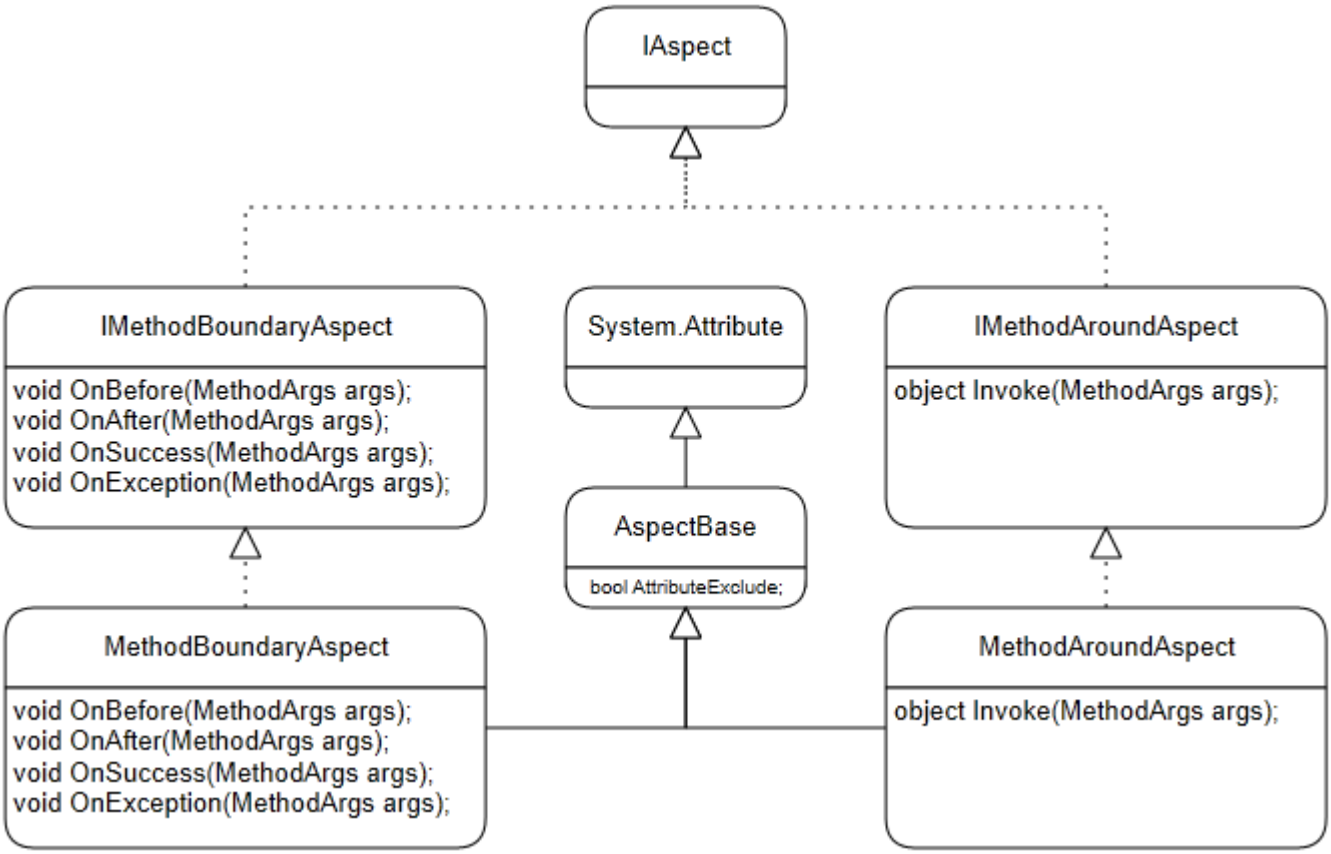

Figure 3.2: Aspect Inheritance

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.

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; that is not currently implemented in Buffalo.

Listing 3.1 shows how a simple aspect is created. It inherits MethodBoundaryAspect and overrides OnBefore and OnAfter.

```
using Buffalo;
using System;

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

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

   void Display(string title, MethodArgs args)
   {
      Console.WriteLine("{0} {1}", title, args.FullName);
      foreach (var p in args.Parameters)
      {
         Console.WriteLine("\t{0} ({1}) = {2}",
                  p.Name, p.Type, p.Value);
      }
   }
}
```

Listing 3.1: Sample TraceAspect

To use the aspect, simply apply it to a method, class or assembly. Once the code is compiled to produce an assembly, BuffaloAOP.exe can be invoked by passing it the path to the assembly. Buffalo will take over and weave in the aspect. This and more examples and details are provided in Appendix A.

```
[TraceAspect]
public class Hello
```

```
3 {
4         //...
5 }
```

Listing 3.2: Apply Aspect on Class Level

3.3 MethodArgs

As mentioned above, when all is said and done, an aspect ultimately gets injected into each *individual* method. When developing an aspect, meta-information about the target method can be accessed. This information is encapsulated via the MethodArgs object passed in as parameter to the aspect. Currently the method name, its full method signature, 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. When using MethodAroundAspect, the parameter values can even be modified and pass back to the original method.

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. 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, one only needs to reference the class library as underlined in figure 3.3.

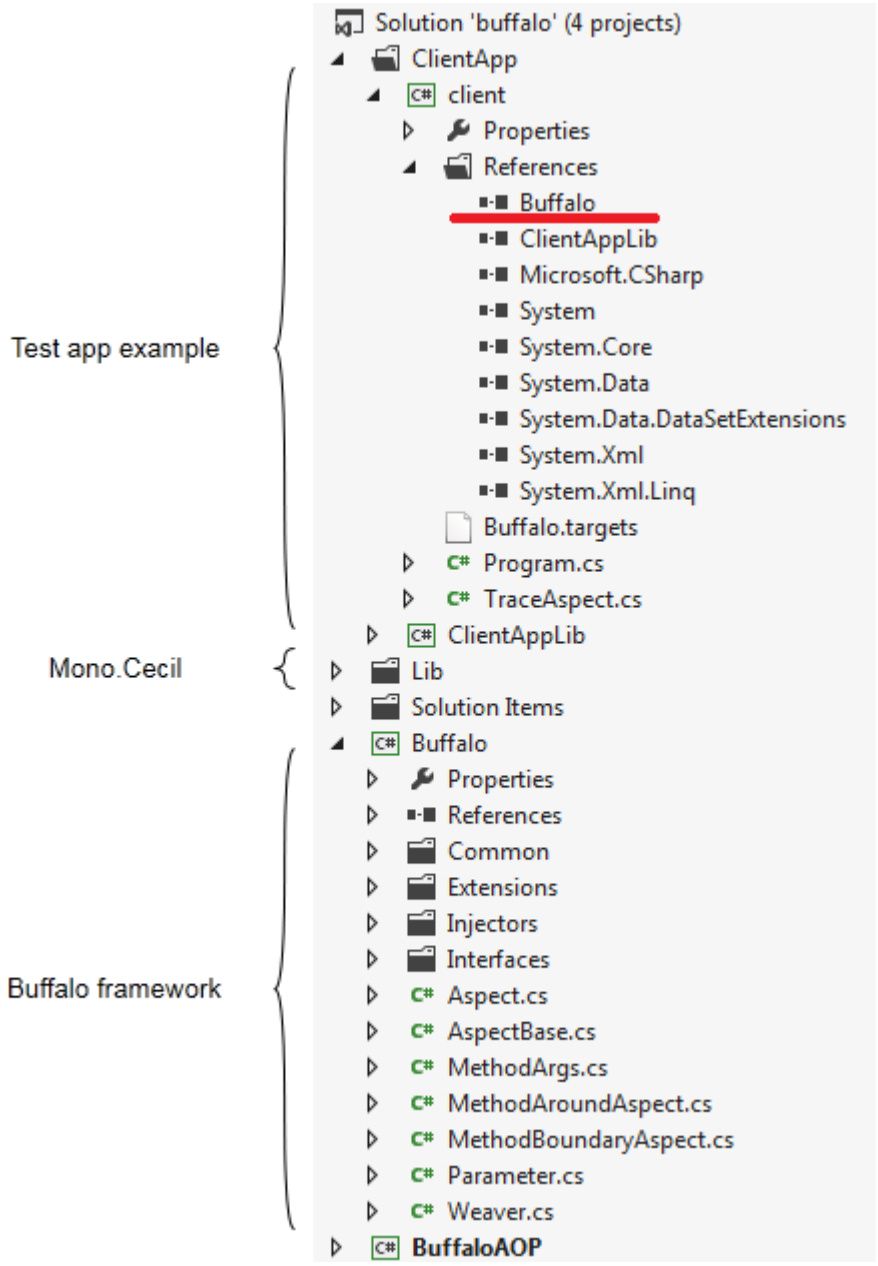

Figure 3.3: Solution Structure

The client project shown above is a simple program included in the solution for testing.

3.5 Implementation Overview

The implementation begins by finding all eligible methods to be injected using the verifying process indicated in figure 3.1. Each eligible method will have one or more aspects applied to it. The actual injection process will loop through each aspects for the eligible method and inject the necessary CIL instructions.

The CIL instructions modification is performed by using the open source Mono.Cecil library. Originally the System.Reflection APIs in the .NET Framework was favored, but it was later determined that Mono.Cecil provides more features and is much easier to work with. Buffalo uses Mono.Cecil heavily to modify CIL instructions and to assemble the final assembly.

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 injects 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:

```
1 public void SayHello()
2 {
3   try{
4      Console.WriteLine(Hello World!);
5   }catch(Exception ex){
6   }finally{
7   }
8 }
```

Listing 3.3: SayHello function

The generated CIL is shown in figure 3.4. For ease of display the CIL has been cleaned up a bit:

```
1 .try
```

```
2  {
3    .try
4    {
5      IL_0002: Ldstr "Hello World!"
6      IL_0007: call void [mscorlib]System.Console::WriteLine(string)
7      IL_000e: leave.s IL0015
8    }
9    catch [mscorlib]System.Exception
10   {
11     IL_0010: stloc.0
12     IL_0013: leave.s IL_0015
13   }
14   IL_0015: leave.s IL_001c
15 }
16 finally
17 {
18   IL_001a: endfinally
19 }
20 IL_001c: ret
```

Listing 3.4: CIL generated for sample C# function

Figure 3.4 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 figure 3.4.

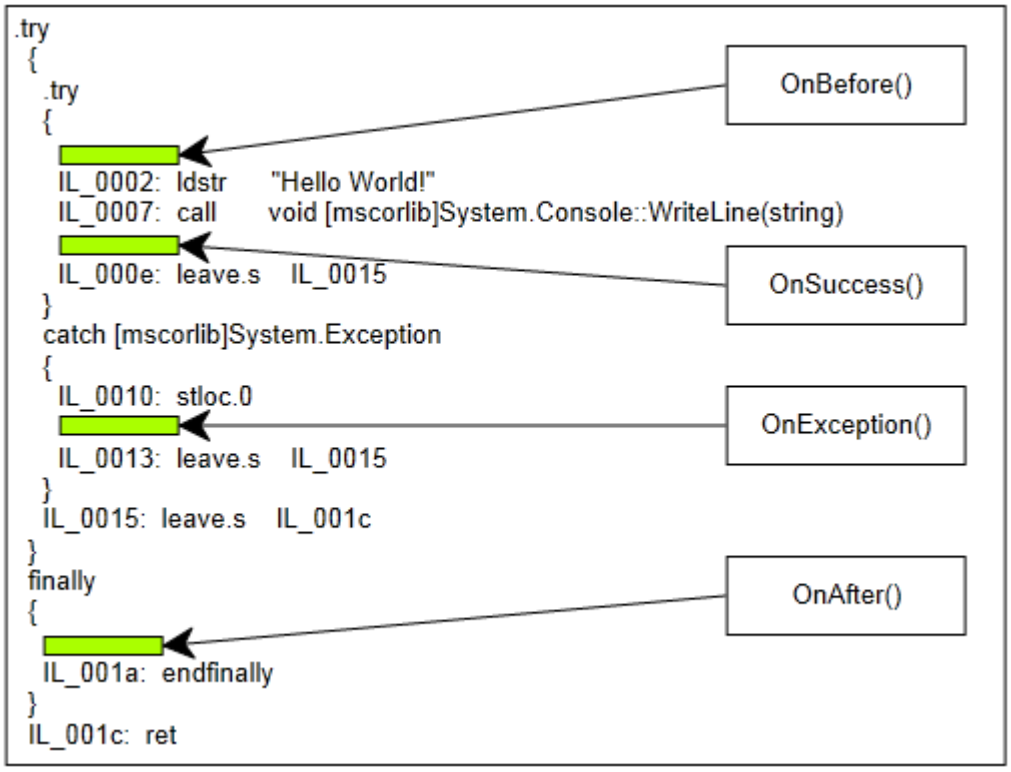

Figure 3.4: CIL Interception Points

3.7 MethodAroundAspect Implementation Detail

The MethodAroundAspect on the other hand is a much more complicated compared to the MethodBoundaryAspect.

MethodAroundAspect implements IMethodAroundAspect which has the following contract:

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

Listing 3.5: IMethodAroundAspect Interface

When developing an aspect a Proceed() can be issued to signal a call back into the original method. The steps taken to implement MethodAroundAspect in CIL are 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 the original method returns non void type, then put the return value back on the stack.
   (b) If the 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!

3.8 MethodArgs Implementation Detail

When developing an aspect, information about the target method can be accessed. This is achieved by using the MethodArgs object. During the weaving, an instance of MethodArgs is created, with all properties assembled dynamically to capture the information of the current executing method. MethodArgs is then passed as parameter into each of the aspect.

Being able to capture some information about the annotated methods will be extremely useful in case of debugging.

A distinct instance of MethodArgs for each boundary aspects was instantiated at an early Buffalo implementation. 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 enables programmers to easily maintain the aspects and modify them as needed.

One 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.

One can argue that since unhandled exceptions will bubble up the chain, a developer can simply catch them in the main method, and this would have achieved similar effect. That approach is limited in that it is very generic. When the main method catches an exception, it has no idea what the internal state of the failed method was. Using Buffalo, the internal state can be inspected by checking the MethodArgs object.

Besides exception handling, Buffalo can be used in scenarios where cross-cutting concerns are common. One problem facing developers when working on database is to ensure data atomicity. Imagine a customer who deposits money to a bank, the amount is saved successfully to the banks own record, but the customers account failed to get updated due to unforeseeable network error. Not only would the database end up with incomplete data, but the bank would have a very unhappy customer. Either all operations are successfully completed or nothing should have been written to the database.

Buffalo can be used to solve this problem by encapsulating transactional scope using the MethodBoundaryAspect. Complete example is available in Appendix A.

Developers that work with UI elements often run into problems updating the UI from a different thread. As UI elements can be updated only by thread that created them, this creates the cross-threading problem. Buffalos MethodAroundAspect can be used to get around this problem by marshaling the update back to the UI thread. For more detail please refer to Appendix A.

Buffalo allows developers to quickly create aspects to solve various problems. Besides the scenarios mentioned above, it can also be used for authorization, where security credential is validated before method execution. It can be used to create a caching mechanism to improve application performance. Buffalo has performed well in isolating cross-cutting concerns into single unit of code, which is easily maintained and modified.

Chapter 5

Conclusions

5.1 Current Status

Buffalo currently contains two types of aspect: MethodBoundaryAspect - where various execution points can be intercepted. MethodAroundAspect - where a method can be wrapped around or completely replaced while preserving the option to call back into the original method. MethodAroundAspect targets individual method, whereas MethodBoundaryAspect can be applied on three levels on a .NET application.

Originally MS-Build integration was planned, that when the Buffalo program was installed via the setup executable, it will modify the relevant .NET configuration files to hook BuffaloAOP.exe into MS-Build. That would trigger the weaving process from within the Visual Studio IDE when the solution is compiled. It was later found that Microsoft has dropped support for the setup project type. As a work around, instruction is provided in appendix A to manually hook into MS-Build.

Originally MethodAroundAspect was intended to be used similar to MethodBoundaryAspect, where it can be applied on three levels. However it was later determined that since the CIL instruction of the actual aspect will also have to be modified, it really does not make sense for it to be applied to more than one method, as it would introduce conflicting changes.

MethodAroundAspect improvement remains to be determined.

5.2 Future Work

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

Functionality wise, when Buffalo performs post compilation weaving it starts fresh each time; it would be interesting to see how incremental weaving can be done here. Another useful functionality is to be able to apply aspects by matching a set of methods.

5.3 Lessons Learned

A 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 in their programming languages.

Developers also have to understand such problems and what solutions are available to better educate themselves.

Only when the concept is widely understood and supported by both developers and vendors can there hope to begin alleviating 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

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. The instructions and examples mentioned in this manual are specifically for the Windows platform. Although preliminary testings show that the project can be compiled under Linux where Mono is installed.

A.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.

A.2 Simple Profiler

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

```
using System;

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 A.1: Hello program

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

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

Listing A.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 OnBefore(MethodArgs args)
```

```
7       {
8           Display(”ENTERING”, args);
9       }
10
11      public override void OnAfter(MethodArgs args)
12      {
13          Display(”EXITING”, args);
14      }
15
16      public override void OnSuccess(MethodArgs args)
17      {
18          Display(”SUCCESSFULLY EXECUTED”, args);
19      }
20
21      public override void OnException(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 A.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 A.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 A.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 A.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 A.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 2, 11, 14 and 16 show the parameter value passed into each method.

A.3 Transaction Database

It is common to work with code that saves information to a database. And it is typical that the data might have to go into different tables. For example, let us look at an online ordering system. When a customer clicks the buy button during checkout, an order has to be created, and each order item has to be created as well. Order and order items usually go into different tables since they have the logical parent-child relationship. So we might have the following code to save the data:

```
1 public void SaveOrder(string customerName, string item1, string item2)
```

```
2  {
3          //create the linq−to−entity context
4          ModelContainer db = new ModelContainer();
5          //create an order obj
6          Order o = new Order();
7          o.CustomerName = customerName;
8          db.Orders.Add(o);
9          db.SaveChanges(); // <−save the order!
10
11         List<string> items = new List<string>();
12         if (!string.IsNullOrEmpty(item1)) items.Add(item1);
13         if (!string.IsNullOrEmpty(item2)) items.Add(item2);
14         items.ForEach(x =>
15         {
16                 //create the order item
17                 OrderItem item = new OrderItem();
18                 item.ItemName = x;
19                 item.OrderId = −1; //<− intentionally failing the save since there is no such order id
20                 db.OrderItems.Add(item);
21         });
22
23         db.SaveChanges(); //<− saving the order items
24 }
```

Listing A.8: SaveOrder Example

This code is for illustration purpose only. It first saves an order, and then saves each individual order items. The data goes into different tables. Note that we are using the Linq to Entity ORM here, technically it handles transaction if db.SaveChanges() at line 9 is removed and is calls at the end with one call.

Also note that at line 19 we are intentionally failing the OrderItem save. At that point the Order itself has been saved. When OrderItem fails, we ended up with incomplete data in the database.

In case like this, it is important that the atomicity rule is enforced. The save operation should either be completely saved or nothing should be saved at all.

Atomicity rule is usually enforced by using transaction. So we can modify the above code like this:

```
1  public void SaveOrderManualTransaction(string customerName, string item1, string item2)
2  {
3          using (TransactionScope scope = new TransactionScope())
4          {
5                  //same body of code to save..
6          }
7  }
```

Listing A.9: SaveOrderManualTransaction Example

The save operations are now wrapped in a TransactionScope object. If anything bad happens on any of the db.SaveChanges() calls the whole operation is aborted, so we will not ended up with incomplete data in the database.

Scenario like this is a suitable candidate for using aspect. If we need to enforce atomicity elsewhere, we would have to do something similar with TransactionScope. We can refactor and extract out the aspect using Buffalo.

```
1  public class RunInTransactionAspect : MethodBoundaryAspect
2  {
3          private TransactionScope scope;
4
5          public override void OnBefore(MethodArgs args)
6          {
7                  this.scope = new TransactionScope(TransactionScopeOption.RequiresNew);
8          }
9
10         public override void OnSuccess(MethodArgs args)
11         {
12                 this.scope.Complete();
13         }
14
15         public override void OnException(MethodArgs args)
16         {
17                 Transaction.Current.Rollback();
18         }
19
20         public override void OnAfter(MethodArgs args)
21         {
22                 this.scope.Dispose();
23         }
24 }
```

Listing A.10: RunInTransactionAspect

Then apply this aspect to the original code:

```
1  [RunInTransactionAspect]
2  public void SaveOrder(string customerName, string item1, string item2)
3  {
4          //same body of code to save..
5  }
```

Listing A.11: SaveOrder with Aspect

This will ensure the method will operate in the transaction safe manner. We can apply the reusable aspect to other operations that need to be transaction scoped.

A.4 Update UI Safely with Thread

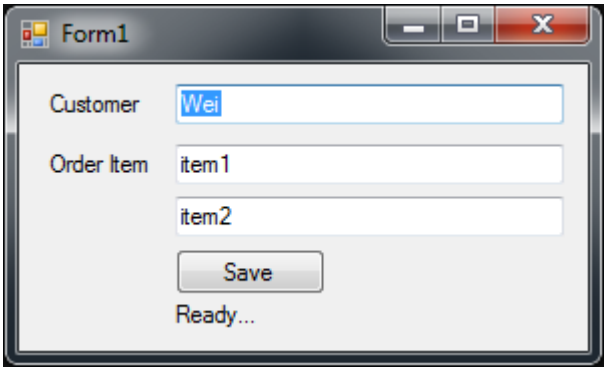

Figure A.1: UI Thread Example

This is a common problem that faces Windows developers. Suppose that when the Save button is clicked, a long running operation is triggered. Here we stimulate the long running operation by putting the thread to sleep. The label that says Ready is updated to OK when the operation finishes.

```
1 public void SaveData ()
2 {
3     //sleep to simulate long running process
4     Thread.Sleep(10000);
5     lblStatus.Text = ”OK”;
6 }
```

Listing A.12: Long running operation

Window application is single thread application. While the long operation is in progress, the whole application will be unresponsive until the operation finishes. This makes a very bad user experience.

One remedy is to put the long running operation is a separate worker thread. And update the UI label when the worker thread finishes.

```
1 public void SaveData ()
2 {
3     new Thread(new ThreadStart(() =>
4     {
5         Thread.Sleep(5000);
6         lblStatus.Text = ”OK”;
7     })).Start();
8 }
```

Listing A.13: Long running operation with thread

But this would cause an exception to be thrown on line 6. UI elements can only be updated from the thread that created them.

```
1  public void SaveData()
2  {
3      Thread.Sleep(5000);
4      Action action = () => lblStatus.Text = ”OK”;
5      this.BeginInvoke(action);
6  }
```

Listing A.14: Update UI from different thread

Listing A.14 will marshal the request to update the UI back to the UI thread. Ensuring the label will be updated without any threading problem. Since this is a common threading problem when working with UI. Buffalo can be used to extract the solution out as an aspect.

```
1  public class ThreadSafeAspect : MethodAroundAspect
2  {
3          public override object Invoke(Buffalo.MethodArgs args)
4          {
5                  Dispatcher.CurrentDispatcher.Invoke(() => args.Proceed());
6                  return null;
7          }
8  }
```

Listing A.15: ThreadSafeAspect

Here we create a new aspect inheriting from MethodAroundAspect, and overriding Invoke. This wraps around the invocation of the method call. Args.Proceed() is Buffalos way of calling back to the original UpdateStatus method.

```
1  public void SaveData()
2  {
3          new Thread(new ThreadStart(() =>
4          {
5                  Thread.Sleep(5000);
6                  this.UpdateStatus();
7          })).Start();
8  }
9  [ThreadSafeAspect]
10 public void UpdateStatus() { lblStatus.Text = ”OK”; }
```

Listing A.16: Apply ThreadSafeAspect

Now when the Save button is clicked, the long running operation is triggered, control is immediately returned. When the operation finishes the label is updated without any

threading issue.

Note that currently MethodAroundAspect supports only method level attribution. This limits its reusability. It will be much more useful when it support three levels as MethodBoundaryAspect does.

A.5 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 references a different configuration file: Microsoft.CSharp.targets. This file in term references Microsoft.Common.targets.

Each .NET version has its own 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 A.2

```
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 A.2: 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 A.17: 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 encoded 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.